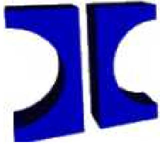


Departamento de Computación
Facultad de Ciencias Exactas y Naturales
UNIVERSIDAD NACIONAL DE BUENOS AIRES

Type Specialisation of Polymorphic Languages

PhD Thesis

to obtain the degree of Doctor of the National University of Buenos Aires, in the area of Computer Sciences

by

Pablo E. Martínez López

Director: Dr. John Hughes
Advisor: Dr. Alfredo Olivero

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad Nacional de Buenos Aires
Argentina

Buenos Aires, November 2005

Director: Dr. John Hughes
Chalmers Tekniska Högskola
Institutionen för Datavetenskap
S-412 96 Göteborg
Sweden - Sverige

Advisor: Dr. Alfredo Olivero
Departamento de Computación, FCEyN
Universidad Nacional de Buenos Aires
Planta Baja, Pabellón I, Ciudad Universitaria
(1428) Buenos Aires, Argentina

The research reported here was supported by a FOMEC scholarship (1998-2001) from Project N° EX 376 from University of Buenos Aires, coordinated by Lic. Irene Loiseau, University of Buenos Aires, and by a scholarship from the ALFA-CORDIAL Project N° ALR/B73011/94.04-5.0348.9 (1998-2005) from the European Union, coordinated by Dr. Fernando Orejas, Universidad de Catalunya.

Type Specialisation
of Polymorphic Languages
Copyright © Pablo E. Martínez López, 2005
E-mail: `fidel@info.unlp.edu.ar`
`http://www-lifia.info.unlp.edu.ar/~fidel`

Para quienes son la razón de estar vivo: Lorena, Aylen y Lautaro.

Contents

Resumen xi

Agradecimientos xiii

Abstract xix

Acknowledgments xxi

1 Introduction 1
- 1.1 Program Specialization 1
- 1.2 Type Specialization 4
- 1.3 Contribution of This Thesis 5
- 1.4 Overview 6

I Type Specialization 9

2 Partial Evaluation 11
- 2.1 Partial Evaluation 11
- 2.2 Binding Time Analysis 15
- 2.3 Compiling by Partial Evaluation 16
- 2.4 Inherited Limit: Types 20

3 Type Specialization 23
- 3.1 Basic Source Language 24
- 3.2 Residual Language 27
- 3.3 Arity Raising 32
- 3.4 Extensions to the Language 33
  - 3.4.1 Booleans 34
  - 3.4.2 Dynamic recursion 36
  - 3.4.3 Static functions 38
  - 3.4.4 Static recursion 39

3.4.5 Sum types . . . 42
3.4.6 Things not included . . . 45
3.4.7 On recursive types . . . 46

4 Examples **49**
4.1 Untyped λ-calculus . . . 50
4.2 Simply Typed λ-calculus . . . 51
4.3 Monomorphizing λ-calculus . . . 53
4.4 Limitations of Type Specialization . . . 55
4.4.1 Lack of principality . . . 55
4.4.2 Failure . . . 56
4.4.3 Interaction between polyvariance and recursion . . . 57
4.5 Summary of Type Specialization . . . 60

II Principal Type Specialization **63**

5 Theory of Qualified Types **65**
5.1 On Notational Conventions . . . 66
5.2 Predicates . . . 67
5.3 An Example: the Haskell Class System . . . 68
5.4 Type Inference with Qualified Types . . . 70
5.5 Coherence and Evidence . . . 75
5.6 Fine Tuning of Predicates . . . 80
5.7 Summary . . . 81

6 Principal Type Specialization **83**
6.1 Residual Language, Revisited . . . 84
6.1.1 Ordering between residual types . . . 87
6.1.2 Typing residual terms . . . 89
6.2 Roadmap Example . . . 91
6.3 Specifying Principal Specialization . . . 92
6.3.1 Source-Residual relationship . . . 93
6.3.2 Specialization rules: the system $\vdash_{P}$ . . . 94
6.3.3 Existence of principal specializations . . . 98
6.4 Examples . . . 99

7 The Algorithm and The Proof **103**
7.1 The Syntax Directed System, S . . . 103
7.2 The Inference Algorithm, W . . . 107
7.2.1 A unification algorithm . . . 107
7.2.2 An entailment algorithm . . . 109
7.2.3 An algorithm for source-residual relationship . . . 109
7.2.4 An algorithm for type specialization . . . 110
7.3 Proof of Principality of P . . . 112
7.4 Examples . . . 113

8 Constraint Solving and Postprocessing **115**
8.1 Simplification and Improvement . . . 116
8.1.1 Motivation . . . 116
8.1.2 Specification . . . 116
8.1.3 Implementing a Simplification . . . 118
8.1.4 Simplification during specialization . . . 119
8.1.5 Discussion . . . 120
8.2 Constraint Solving . . . 121
8.2.1 Motivation . . . 121
8.2.2 Specifying Solutions . . . 122
8.2.3 Solving and Specialization . . . 123
8.2.4 Finding Solutions . . . 125
8.2.5 An Algorithm for Constraint Solving . . . 126
8.2.6 Examples . . . 128
8.2.7 Discussion . . . 130
8.3 Evidence Elimination . . . 131
8.3.1 Extensions to the language of evidence . . . 132
8.3.2 Eliminating Evidence via Constraint Solving . . . 132
8.3.3 Discussion . . . 134

9 Extending the Source Language **139**
9.1 Failure . . . 140
9.2 Booleans . . . 141
9.3 Static Let . . . 146
9.4 Static Functions . . . 146
9.5 Static Recursion . . . 149
9.6 Datatypes . . . 151
9.7 Dynamic Recursion . . . 156
9.8 Other Features . . . 161

10 The Prototype **163**
10.1 A Brief Tutorial . . . 163
10.2 Overall Architecture and Relevant Features . . . 165
10.3 What Have We Learned? . . . 168
10.4 Towards a Proper Implementation . . . 169

III Type Specializing Polymorphism **171**

11 Inherited Limit: Polymorphism **173**
11.1 Monomorphizing Lambda-calculus, Revisited . . . 173
11.2 An Annotation to Generate Polymorphism . . . 174
11.3 Generating Polymorphism . . . 179
11.4 Improving Polyvariance . . . 180

IV Other Issues 183

12 About Jones' Optimality 185
12.1 The Problem . . . 185
12.2 Regaining Jones Optimality . . . 195
12.3 Conclusions . . . 202

13 Related Work 203
13.1 Partial Evaluation . . . 203
13.1.1 Similix . . . 205
13.1.2 Polymorphic and modular partial evaluation . . . 206
13.1.3 Other works on partial evaluation . . . 206
13.2 Type Directed Methods . . . 207
13.2.1 Type Directed Partial Evaluation . . . 207
13.2.2 Tag-elimination . . . 208
13.2.3 Ohori's specialization . . . 208
13.3 Other Approaches . . . 209
13.3.1 Supercompilation . . . 209
13.3.2 Generalized Partial Computation . . . 209
13.3.3 Data specialization . . . 210

14 Future Work 213
14.1 Improving Constraint Solving . . . 213
14.2 Better Implementation . . . 214
14.3 Extensions to the Source Language . . . 215
14.4 Binding Time Assistant . . . 216

15 Conclusions 217

A Proofs 219
A.1 Proof of proposition 6.7 from section 6.1 . . . 219
A.2 Proof of proposition 6.8 from section 6.1 . . . 220
A.3 Proof of proposition 6.9 from section 6.1 . . . 221
A.4 Proof of proposition 6.11 from section 6.1 . . . 222
A.5 Proof of theorem 6.12 from section 6.1 . . . 222
A.6 Proof of proposition 6.13 from section 6.3 . . . 223
A.7 Proof of proposition 6.14 from section 6.3 . . . 223
A.8 Proof of theorem 6.15 from section 6.3 . . . 223
A.9 Proof of theorem 6.19 from section 6.3 . . . 224
A.10 Proof of theorem 6.20 from section 6.3 . . . 224
A.11 Proof of proposition 6.21 from section 6.3 . . . 224
A.12 Proof of proposition 6.22 from section 6.3 . . . 225
A.13 Proof of proposition 7.3 from section 7.1 . . . 225
A.14 Proof of proposition 7.7 from section 7.1 . . . 226
A.15 Proof of proposition 7.8 from section 7.1 . . . 227
A.16 Proof of theorem 7.9 from section 7.1 . . . 227

A.17 Proof of theorem 7.10 from section 7.1 228
A.18 Proof of proposition 7.11 from section 7.2 233
A.19 Proof of proposition 7.12 from section 7.2 235
A.20 Proof of proposition 7.14 from section 7.2 236
A.21 Proof of proposition 7.15 from section 7.2 237
A.22 Proof of theorem 7.17 from section 7.2 241
A.23 Proof of theorem 7.18 from section 7.2 241
A.24 Proof of lemma 8.7 from section 8.1 246
A.25 Proof of theorem 8.8 from section 8.1 247
A.26 Proof of theorem 8.9 from section 8.1 248
A.27 Proof of theorem 8.15 from section 8.2 248
A.28 Proof of lemma 8.16 from section 8.2 249

References **251**

Resumen

. . .
— ¿Qué rayos fue eso?
— Eso fue un comienzo, Matthew. Algo viajó de un estado de existencia a otro. Vino de uno de los más distantes arrecifes del sueño. Observemos las consecuencias.

A Game of You – The Sandman
Neil Gaiman, Shawn Mc Manis

Cuando se consideran las maneras de producir programas, la técnica principal que viene a la mente de cada uno de nosotros es la de escribir el programa a mano. Aunque existen técnicas de derivación y herramientas para producir programas automáticamente, su aplicación está usualmente restringida a cierta clase de problemas, o ciertos dominios (como los generadores de parsers, o las interfaces visuales). En los casos donde tales técnicas se pueden aplicar, la productividad y la confiabilidad se ven ampliamente incrementadas. Por esa razón, nos interesamos en la producción automática de programas en un marco general.

La especialización de programas es una manera particular de producir programas automáticamente. En ella se utiliza un programa fuente general dado para generar diversas versiones particulares, especializadas, del mismo, cada una resolviendo una instancia particular del problema original. La técnica más conocida y más ampliamente estudiada de especialización de programas es llamada evaluación parcial; se la ha utilizado con éxito en varias áreas de aplicación diferentes. Sin embargo, la evaluación parcial tiene problemas cuando se considera la producción automática de *programas con tipos.*

La especialización de tipos es una forma de especialización de programas que puede producir automáticamente programas con tipos a partir de uno fuente. Comprende diversas técnicas muy poderosas, tales como especialización polivariante, especialización de constructores, conversión de clausuras; es la primera de las variantes de especialización de programas que puede generar tipos arbitrarios a partir de un único programa fuente. Creemos que la especialización de tipos puede ser la base sobre la que desarrollar un marco de producción automática de programas.

En esta tesis consideramos la especialización de programas, extendiéndola para producir *programas polimórficos.* Ilustramos eso considerando un intérprete para un lambda cálculo con tipos á la Hindley-Milner, y especializándolo con cualquier programa objeto para producir un programa residual que sea esencialmente igual que el original. En la búsqueda de la generación de polimorfismo, extendemos la especialización de tipos para

que pueda expresar la especialización de programas con información estática incompleta, y probamos que para cada término podemos inferir una especialización particular que puede ser usada para reconstruir cada uno de las otras especializaciones de tal término. Llamamos *especialización de tipos principal* a tal técnica, debido a la analogía de esta propiedad con la noción de tipos principales. Nuestra presentación clarifica muchos de los problemas existentes en la especialización de tipos, lo cual puede ser usado como una guía en la búsqueda de soluciones para ellos.

La presentación se divide en cuatro partes. En la primera Parte, presentamos la Especialización de Tipos en su forma original, junto con material complementario. En la Parte II desarrollamos la presentación de la Especialización de Tipos Principal, explicando todos los detalles técnicos, dando varios ejemplos, y presentando nuestra implementación del prototipo. En la Parte III describimos las posibilidades de la formulación nueva, proveyendo una extensión de la Especialización de Tipos para generar programas polimórficos. Finalmente, la última parte describe trabajos relacionados, trabajos futuros, y concluye.

Este trabajo es el resultado de siete años de investigación, realizados durante mis estudios de doctorado.

Palabras clave: especialización principal de tipos, especialización de tipos, especialización de programas, producción automática de programas, programas polimórficos.

Agradecimientos

Para Shevek el retorno siempre sería tan importante como la partida. Partir no era suficiente, o lo era sólo a medias: necesitaba volver. En aquella tendencia asomaba ya, tal vez, la naturaleza de la inmensa exploración que un día habría de emprender hasta más allá de los confines de lo inteligible. De no haber tenido la profunda certeza de que era posible volver (aun cuando no fuese él quien volviera), y de que en verdad, como en un periplo alrededor del globo, el retorno estaba implícito en la naturaleza misma del viaje, tal vez nunca se hubiera embarcado en aquella larga aventura.

Los Desposeídos
Úrsula K. Le Guin

Es una tarea ciclópea escribir agradecimientos. Porque tenés que decidir: o bien escribís algo simple, como "¡¡Gracias a todos!!", que es, por supuesto, muy vago y general, o bien enumerás a todas y cada una de las personas que te ayudaron, te animaron y apoyaron durante el largo período que te toma ir desde ser un recién graduado a ser un doctor, y tratar de justificar por qué merecen el agradecimiento. Pero esta segunda tarea es imposible. Hay, simplemente, demasiadas personas y demasiadas razones para agradecerles. Por eso, como en muchos casos en la vida, uno tiene que llegar a un acuerdo, y terminar con un pequeño 'capítulo' de agradecimientos que es incompleto, y por ello injusto con los que fueron olvidados, y sin embargo demasiado largo. Así que, si vos, que sabés que estuviste allí conmigo cuando lo necesité, no te encontrás en estas líneas, no creas que no aprecio lo que hiciste, sino, simplemente que mi mente se sobrecarga a veces, y el viaje fue muy largo.

Porque mis estudios de doctorado fueron un viaje, lleno a su vez de pequeños viajes. Y en todos los casos, como Úrsula Le Guin ha expresado tan bien, "el verdadero viaje es el retorno", yo regresé… a un nuevo comienzo. Fue un viaje en conocimiento, porque aprendí tantas cosas nuevas (y no sólo académicas), que me siento verdaderamente perfeccionado como persona. Fue un viaje en sentimientos, en el que aprendí acerca de mí mismo, y acerca de otros, y de cómo amarlos mejor, pelearlos, descubrirlos, divertirme con ellos, sondear nuestros límites, y después de todo, sobrevivir juntos, más ricos que antes. Fue un viaje a mundos imaginarios, Norstrilia, el Japón del período Edo, el Londres de Jack el Destripador, las tierras del Sandman, el mundo de Lyra, y muchos otros que no hubiera conocido si no hubiera visitado a mi 'guía de turistas' en Gotemburgo. Fue un viaje en el tiempo, aprendiendo a pararme frente a un futuro impredecible, y a descubrir cómo el pasado se revela en un nuevo comienzo con cada sorpresa, en lugar de estar cerrado y terminado. Y contuvo varios viajes en el espacio, cruzando casi la mitad del globo una vez al año para encontrarme con mi director, y con esa increíble comunidad de excelentes personas en Chalmers. Y siempre retorné,

aunque nunca al mismo 'lugar' donde comenzó el viaje, y cada vez para empezar un nuevo viaje.

Siempre dejé, en todos mis anteriores agradecimientos, a las personas más importantes para el final, con la certeza de que el lugar de "cerrar la procesión" era importante; pero no esta vez, porque ellos aguantaron cuando yo estaba por ahí, viajando y embarcándome en aventuras, y también en las ocasiones en las que perdí mi rumbo, y por eso, si no sólo por ser las personas más importantes en toda mi vida, merecen estar primero. Así que, Lorena, Aylen, Lautaro, deben saber que realmente les agradezco que estén ahí, porque los amo más que a mi propia vida, y le dan verdadero sentido a la misma, y porque cada segundo que estuve lejos de ustedes estuve deseando volver, y muriendo lentamente por adentro, mi alma marchitádose por la falta de su amor vital. Sé que saben como me sentí y como me siento, porque sé que ustedes sienten lo mismo por mí. Soy muy afortunado.

En segundo lugar quiero agradecer a mis padres, porque ellos me atrajeron a este camino, me cuidaron y me ayudaron a seguir, y los amo. También a mis hermanas, porque me dí cuenta que son una parte importante de mi vida, y también las quiero. Las cosas no van a volver a ser igual ahora que ya no podemos estar todos juntos de nuevo. Pá, má, Sole y especialmente Ceci (que estoy seguro que sabés, aunque no puedas leer esto), gracias.

A continuación quiero agradecer a esas personas que fueron mi guía, y que me dieron un marco de referencia en cómo ser un excelente profesional y una mejor persona. Hay varios de ellos, pero los tres más importantes son Gabriel Baum, John Hughes, y Roberto Di Cosmo. Gabriel fue mi director como estudiante de grado, y mi jefe y amigo por los últimos doce años, y en cada ocasión en la que hablamos, me maravillé descubriendo la increíble fuente de sabiduría que es; mis estudiantes han sufrido mis intentos de aprender de él cómo ser un buen líder y una mejor persona. John fue mi director de doctorado, y una fuente importante de ideas, guía y apoyo; luego de cada una de las reuniones con él me sentí intimidado y estimulado a límites que pensaba imposibles para mí. John, tengo que agradecerte especialmente el haberme aceptado como tu estudiante en condiciones lejos de las óptimas (¡con un océano entre medio de nosotros!) y sin embargo te las arreglaste para hacerlo bien, a pesar de mis esfuerzos por lo contrario. Ah! No debo olvidar las lecciones de ski! Y Roberto fue mi director en el Master, y también, por un tiempo, mi director de doctorado, antes de que John entrase en escena; me enseñó y me guió y me alentó tan bien y con tanta gentileza, que todavía estoy preguntándome cómo lo hizo (¡para imitarlo!). A vos, Roberto, tengo que agradecerte especialmente que me dejaras libre cuando estaba coqueteando con la especialización de tipos. A los tres, mi más sincera gratitud. Sepan que todavía voy a beber de sus manantiales por un tiempo.

Hay también dos docentes ejemplares, a quienes debo mi vocación como docente. La Profesora Lía Oubiña, porque es la docente más increíble que conocí, y si sólo consiguiese durante mi carrera una pequeña fracción de lo que ella me mostró posible, estaría satisfecho de ser un excelente docente. Y el Profesor Michael Christie, porque él también es un excelente docente (tu curso es algo memorable; ¡gracias!), y fue él el que provocó mi gran descubrimiento: enseñar es mi vida.

Continuando con modelos y guías, debo mencionar a diversas personas: Peter Thieman, Mark Jones y Marcelo Frías, los jurados de mi tesis, y Olivier Danvy, quién re-

conoció el valor de algunas ideas en mi presentación. A Peter y Olivier, especialmente, les debo diversas discusiones e iluminaciones.

Estoy completamente en deuda con diversos amigos a ambos lados del Océano Atlántico. Quiero mencionar a seis en particular. En el lado americano, mis amigos (y estudiantes y colegas) Esteban y Hernán, y mi colega Eduardo, por su apoyo, aliento e interés en mi trabajo, y también porque se permitieron involucrarse en el mismo, enriqueciendo esta tesis (Hernán, te debo especialmente varias partes del Capítulo 8, y el autógrafro de mi admirada Úrsula); y también por ser parte del ambiente directo en mi trabajo diario, el cual es mucho más animado y divertido gracias a ellos. En el lado europeo, Ana, Carlos y Verónica, porque fueron mi familia durante mis largas estadías en Gotemburgo, y mis viajes no hubieran sido tan fáciles, interesantes y productivos sin su ayuda, apoyo, amabilidad y amistad permanentes. Los tres me abrieron las puertas de sus casas y me permitieron hacerlas la mía. (¡Ah! También tengo que mencionar el refugio que Eduardo me dio cuando él también estaba de viaje por Europa, por las mismas razones que yo.) Además me siento en la obligación de mencionar la cantidad de veces que tanto Ana como Verónica me alimentaron; si hubiesen escuchado los consejos de mi abuela, que solía decir que era más barato vestirme que alimentarme, se hubieran ahorrado bastante. ¡Mis amigos, espero ser bendecido con vuestra amistad por un largo tiempo!

Hubo otros varios modelos para mi trabajo, y ellos también han contribuido con mi tesis en formas directas e indirectas. Quiero mencionar a Juan Vicente Echagüe (siempre Juanvi para mí, aunque a él no le guste del todo…), Delia Kesner, Alfredo Olivero, Gustavo Rossi e Irene Loiseau, porque me ayudaron con sus consejos, su aliento y su trabajo, y, para nada lo menos importante, ¡con mis asuntos administrativos! A Delia, en particular, quiero agradecerle todas las veces que me dio alojamiento en París, sus fantásticas cenas, y por ser quién me señaló la beca que hizo todo esto posible.

Volviendo nuevamente mi atención al oeste del océano, llega el momento de mencionar a mis estudiantes, aunque nombrándolos de esa manera los estoy presentando como menos de lo que realmente son. He sido bendecido con la amistad de la gente que empezó aprendiendo de mí, pero que se transformó en mucho más. Así que, Alejandro, Pablo P., Bocha, Andrés, Walter, Daniel, Diego Y., Pablo B., Guillermo, Juan C., Diego P. y Laura se merecen un lugar acá. Sin embargo, siento que debo enfrentar también la difícil parte de mencionar algunas de las cosas que hicieron por mí. ¡Va a ser injusto, seguro, porque todos hicieron muchísimo! A Andrés y Ale, por ayudarme con el código del prototipo; a Diego P. y Laura por la cuidadosa lectura (¡ni siquiera una prima en un font diferente escapó a su aguda vista!); a Juan y Guille, por permitirme descansar en sus casas en los tiempos más difíciles (¡acá está tu regalo, Guille!); y a Pablo B., por respaldarme cada vez que lo necesité. En particular, sobre el final debo volver a mencionar a Pablo B. y a Laura, porque me ayudaron a que la semana de mi defensa fuese organizada y tranquila; ¡gracias, amigos!

También merece mi gratitud el grupo de docentes y estudiantes que fueron parte de esa fantástica aventura que fue enseñar Programación Funcional (a los que aún no mencioné, al menos… :-). Ellos creyeron en mí, aprendieron de mí, me enseñaron, me enfrentaron y me asistieron en el trabajo más gratificante del mundo: enseñar. Mencionaré sólo a unos pocos: Matías M., Marcelo C., el Colo, Nachi, Majo, Vero,

Wanda, Vanesa, Agustina, Mariana, Pepe, Valeria, Hernán C., Mara.

Shevek imaginó y realizó el Sindicato de Iniciativas, inspirándome a iniciar uno similar. No voy a decir mucho más que esto aquí, pero el tiempo que tomé de mi tesis para pensarlos, alentarlos y ayudarlos fue realmente una buena inversión. Ustedes hacen que muchas cosas sean más valiosas; no abandonen — nunca.

Cada persona que menciono aquí vió mis viajes como algo diferente. Pero hay cuatro, mis amigotes de siempre, que los 'vistieron' con sus fantasías. Luis, Federico, José y Eduardo K., si mis viajes hubiesen sido sólo la mitad de las cosas que ustedes imaginaron, ¡mi vida hubiese tomado una dirección completamente diferente! Valoro su amistad como una de las joyas más importantes en mi tesoro.

Ahora es momento de volver al este del Atlántico. Hice un montón de amigos en Chalmers, y es difícil ordenarlos ahora; ¡otra vez hay que llegar a un acuerdo! En primer lugar quiero mencionar a Koen; me maravilla su inteligencia, nuestra pasión compartida por los juegos, y la facilidad con la que me aceptó como uno de sus amigos. ¡Koen, no me voy a olvidar nunca tu expresión cuando aparecí por sorpresa en tu defensa de doctorado, y cómo hiciste que todos se volvieran a mirarme! Luego tengo que mencionar a Josef; nunca se rehusó a una charla conmigo, o a discutir cualquier tema, y quizás sea eso, o su imborrable sonrisa, la que me hace sentir tan a gusto con él. Y a ambos debo agradecerles por ser los anfitriones de los Multi-meetings, e inspirarme para tener los míos propios en Argentina. En tercer lugar quier mencionar a Birgit, porque tiene un buen humor permanente, siempre sonriente, y su alegría es contagiosa, y me ofreció su amistad tan abiertamente. En particular, no voy a olvidar que fuiste vos la que empezó con la loca idea de una fiesta sorpresa para mi 34to. cumpleaños; te debo por eso. Y luego, todo el resto: Jörgen, Elin, Tuomo, Rogardt, Ilona, Mia, Andrei, Walid, Makoto, Per B., Gordon, Boris, David R., Björn, Dave, Peter, Magnus, Thomas, y muchos otros. Todos ustedes juntos han hecho que cada vez que dejé Gotemburgo, un pedazo mío se quedara atrás.

Hay un lugar especial en mi corazón para aquellos con los que compartí tantas discusiones (¡incluso en versos!), complots y batallas. Juan G., Pablo S., Pite, Marcos, y nuevamente Steve y Hernán, jugar con ustedes a nuestra propia versión del 1914 fue genial. Fue muy divertido y entretenido, y también me enseñó varias cosas útiles.

A lo largo de mis cursos de doctorado, hice sufrir a diversos docentes. Alejandro R., Laurance, Thomas, Thierry, Philippas, Gonzalo, Martín F.C., Daniel F. (Frito), gracias por su paciencia, sus enseñanzas y su tiempo.

Las instituciones no pueden vivir sin la gente que las hace posible. Pero sobreviven a cualquier individuo particular, y por ello, merecen un lugar por sí mismas acá. Estoy en deuda con diversas instituciones, pero especialmente con el LIFIA de la Facultad de Informática de la Universidad Nacional de La Plata, en Argentina, por ser mi lugar de trabajo y por proveerme con todas las cosas necesarias para hacer buena investigación; la ESLAI, porque sus esporas están creciendo y floreciendo en cientos de lugares alrededor del mundo, y cada una de ellas es un hogar para mí; el Departamento de Ciencias de la Computación de la Facultad de Ciencias Exactas de la Universidad Nacional de Buenos Aires, en Argentina, porque es mi *alma mater*, y la financiación que me dieron fue un ingrediente fundamental en esta receta; el Departamento de Ciencias de la Computación de la Universidad Tecnológica de Chalmers, en Suecia, por el ambiente estimulante que

provee, y porque me aceptó como uno más de sus habitantes; y a la institución detrás de `http://citeseer.nj.nec.com`, por el servicio invaluable que le proveen a la comunidad académica.

En los últimos 'minutos' de mi viaje, conocí algunas personas que constituyen una promesa para el futuro. Marjan, Tomaz, Simon, Andrea, Pedro, María, espero que podamos continuar trabajando y divirtiéndonos juntos.

Un lugar especial debe reservarse para mi escritora favorita, Úrsula Le Guin. Mi vida es mucho más rica gracias a ella; sus palabras inundan mi mente, ayudándome a soportar el dolor cuando viene, y así haciendo que no me sienta sólo, de forma que puede empezar la fraternidad con otros, como enunció Shevek. Valoro tu autógrafo como un regalo muy especial.

Finalmente, para aquellos que pueden entender las vueltas de mi retorcida mente, mencionaré a mis propios Atro, Chifoilisk, Saio Pae, y el resto de los Urrasti. Yo no hubiese estado tan afilado, enfocado y lleno de recursos si no hubieran estado allí.

Y así llega el final de este 'capítulo'. Traté de hacer una breve descripción de todas las maravillas que viví, ví y sufrí en estos años. Pero hacerlo con justicia me hubiese tomado todas las páginas de esta tesis. Entonces, para terminar, haré mías una vez más las palabras de Úrsula.

> *Uno puede volver nuevamente al hogar (...), siempre y cuando entienda que el hogar es un lugar en el que nunca se estuvo.*
>
> Los Desposeídos
> Úrsula K. Le Guin

Abstract

...
— *What the hell was that?*
— *That was a beginning, Matthew. Something traveled from one state of existence to another. It came from one of the more distant skerries of dream. Let us observe the consequences.*

A Game of You – The Sandman
Neil Gaiman, Shawn Mc Manis

When considering the ways in which programs are produced, the main technique that comes to everybody's mind is writing by hand. Although derivation techniques and tools for producing programs exist, their application is usually restricted to certain kind of problems, or certain domains (such as parsing generators, or visual interfaces). In those cases where such techniques can be applied, productivity and reliability are highly boosted. For that reason, we are concerned with the automatic production of programs in a general setting.

Program specialization is a particular way to produce programs automatically. A given, general source program is used to generate several particular, specialized versions of it, each one solving a particular instance of the original problem. The best-known and thoroughly studied technique for program specialization is called partial evaluation; it has been successfully used in several different application areas. But partial evaluation falls short when automatic production of *typed* programs is considered.

Type specialization is a form of program specialization that can automatically produce typed programs from some general source one. It comprises several powerful techniques, such as polyvariant specialization, constructor specialization, and closure conversion, and it is the first variant of program specialization that can generate arbitrary types from a single source program. We believe that type specialization can be the basis in which to develop a framework for automatic program production.

In this thesis we consider type specialization, extending it to produce *polymorphic programs*. We illustrate that by considering an interpreter for Hindley-Milner typed lambda-calculus, and specializing it to any given object program, producing a residual program that is essentially the same as the original one. In achieving the generation of polymorphism, we extend type specialization to be able to express specialization of programs with incomplete static information, and prove that for each term we can infer a particular specialization that can be used to reconstruct every other for that term. We call that *principal type specialization* because of the analogy this property has with the notion of principal types. Our presentation clarifies some of the problems existing in

type specialization, clarification that can be used as a guide in the search for solutions to them.

The presentation is divided in four parts. In the first Part we present Type Specialization in its original form, together with some background material. In Part II we develop the presentation of Principal Type Specialization, explaining all the technical details, offering several examples, and presenting our prototype implementation. In Part III we describe the possibilities of the new formulation, by providing an extension of Type Specialization to generate polymorphic programs. And finally, in the last Part we describe related and future work and conclude.

This work is the result of seven years of research, performed during my PhD studies.

Keywords: principal type specialization, type specialization, program specialization, automatic program production, polymorphic programs.

Acknowledgments

> *He would always be one for whom the return was as important as the voyage out. To go was not enough for him, only half enough; he must come back. In such a tendency was already foreshadowed, perhaps, the nature of the immense exploration he was to undertake into the extremes of the comprehensible. He would most likely not have embarked on that years-long enterprise had he not had profound assurance that return was possible, even though he himself might not return; that indeed the very nature of the voyage, like a circumnavigation of the globe, implied return.*
>
> The Dispossessed
> Úrsula K. Le Guin

It's a cyclopean task to write acknowledgements. Because you have to come to a decision: you can either write something simple, such as "Thank you, all!!", which is, of course, too vague and general, or, you can enumerate each and every one of the persons that helped, encouraged, and supported you during the long period that takes you from a graduate student to a doctor, and attempt to justify why they deserved to be acknowledged. But this second task is impossible. There are, simply, too many people, and too many reasons for them to be thanked. So, as in many cases in life, one should compromise, and finish with a small 'chapter' of acknowledgments, that is incomplete, and thus unfair with the forgotten ones, but nonetheless too long. So, if you who know how you were there for me when I needed you, do not found yourself among these lines, do not think I do not appreciate that, but simply that my mind is crammed to overflowing sometimes, and the voyage was too long.

My PhD studies were a voyage, full of many small voyages inside. And in all cases — as Úrsula Le Guin has said so well, "true voyage is return" — I have returned... to a new beginning. It was a voyage in knowledge, because I have learned so many new things (and not only academic ones), that I really feel improved as a person. It was a voyage in feelings, in which I have learned about myself and about others, how to better love them, fight them, discover them, enjoy with them, probe our limits, and above all, survive together, richer than before. It was a voyage into imaginary worlds, Norstrilia, the Japan of the Edo-period, the London of Jack the Ripper, the lands of the Sandman, Lyra's world, and many others, which I wouldn't have known had I not visited my 'tourist-guide' in Göteborg. It was a voyage in time, learning to stand before an unpredictable future, and to discover how the past reveals a new beginning in itself with every surprise, instead of being closed and finished. And it contained several voyages in space, crossing half the globe almost once a year to meet my supervisor and that incredible community of excellent people at Chalmers. And the return always happened, although never to the same 'place' from where the voyage started, and every

time to start a new voyage.

I have always left, in all my acknowledgements previous to this one, the most important persons to the very end, believing that it was an important place to "close the procession"; but not this time, because they have put up with me when I was somewhere else voyaging and adventuring, and on those occasions when I have lost my way, and for that, if not only for being the most important persons in my whole life, they deserve to be the first. So, Lorena, Aylen, Lautaro, you should know that I am really thankful to you for being there, because I love you more than my life itself, and you give real meaning to it, and because every second I was away from you I was longing for the return, and slowly dying inside, my soul withering because of the lack of your vital love. I know you know how I felt and how I feel, because I know that you feel just the same for me. I am very lucky.

In second place I want to thank my parents, because they have brought me into this path, they have looked after me, and helped me to keep going, and I loved them. And also to my sisters, because I have realized that they were and are an important part of my life, and I love them too. Things will not be the same now that we cannot be all together anymore. Dad, Mom, Sole, and specially Ceci (even though you will not read this, I am sure you will know), thank you.

Next I want to thank those persons that were my guide and who provided me with a frame of reference on how to be an excellent professional and a better person. There are several ones, but the three most important are Gabriel Baum, John Hughes, and Roberto Di Cosmo. Gabriel has been my supervisor as undergraduate, and my boss and friend for the last twelve years, and on every occasion we have talked I marveled at the incredible source of wisdom he is; my students have suffered my attempts to learn from him how to be a good leader and a better person. John has been my PhD supervisor, and an important source of ideas, guidance, and support; after every single meeting with him I have felt awed and stimulated to limits I thought were impossible for me. John, I have to specially thank you because you accepted me as your student in conditions far from optimal (with an ocean between us!), and nonetheless you managed to do it properly despite my best efforts to the contrary. Ah! I should not forget the lessons on skiing! And Roberto has been my Master's supervisor, and also, for a while, my PhD supervisor, before John entered the scene; he has taught me and guided me and encouraged me so well and so smoothly, that I am still wondering how he managed to do it (to copy it!). To you, Roberto, I am specially grateful from setting me free when I was flirting with type specialization. To the three of you, my most sincere gratitude. You should know that I will be drinking from your springs for a while, yet.

There are also two exemplary teachers to whom I owe my vocation as a teacher. Professor Lía Oubiña, because she is the most incredible teacher I have ever met, and if I only achieve during my career a tiny fraction of what she showed me to be possible, I will be satisfied at being an excellent teacher. And Professor Michael Christie, because he is also an excellent teacher (your course is a memorable thing; thanks!), and he was the one that brought about my breakthrough: teaching is my life.

I am completely in debt to several friends on both sides of the Atlantic Ocean. I want to mention six in particular. On the American side, my friends (and students and colleagues) Esteban and Hernán, and my colleague Eduardo, for their support, encour-

agement, and interest in my work, and also because they let themselves be involved in it, enriching this thesis (Hernán, I specially owe you several parts of Chapter 8, and the autograph of my admired Úrsula); and also for being the immediate environment in my daily work, which thanks to them is much more lively and funny. On the European side, Ana, Carlos, and Verónica, because they have been my family during my long stays in Göteborg, and my trips would not have been so easy, interesting, and productive without their permanent help, support, kindness, and friendship. The three of them opened the doors of their houses to me, and they let me bring them mine. (Ah! I have also to mention the shelter Eduardo gave me in Paris when he was also journeying to Europe, for much the same reasons as me!) Besides, I must mention the number of times that both Ana and Verónica have fed me — had they listened to my grandmother's counsels, who said that it is cheaper to clothe me than to feed me, they would have been saved a lot. My friends, I hope I can be blessed with your friendship for a long time!

There have been several other models for my work, and they have also contributed to my thesis in direct or indirect ways. I want to mention Juan Vicente Echagüe (always Juanvi to me, although he doesn't completely like it...), Delia Kesner, Alfredo Olivero, Gustavo Rossi, and Irene Loiseau, because they have helped me with their counsels, their encouragement, and their work, and, not at all least, with my administrative stuff! Delia, in particular, I want to thank for those times when she provided me with a house in Paris, those fantastic dinners, and for being the one to point out the scholarship that made all this possible.

Continuing with models and guides, I have to mention several people: Peter Thiemann, Mark Jones, and Marcelo Frías, the reviewers of my thesis, and Olivier Danvy, whom recognize the value of some ideas in my presentation. To Peter and Olivier, specially, I owe several discussions and enlightenments.

Turning my attention once more West of the ocean, it is now time to mention my students, although by naming them that way I am presenting them as less than they really are. I am blessed with the friendship of the people that started learning with me, and then become much more. So, Alejandro, Pablo P., Bocha, Andrés, Walter, Daniel, Diego Y., Pablo B., Guillermo, Juan C., Diego P., and Laura, you deserve a place here. However, I feel that I have to face also the difficult task of mentioning some of the things they have done for me — it will be unfair, for sure, because all of them have done a lot! To Andrés y Ale, for helping me with the code of the prototype; to Diego P. and Laura, for the careful reading (even a prime not in the same font as in other places did not escape his keen sight!); to Juan and Guille, for allowing me to rest at their places in the most difficult times (here you have your present, Guille!!); and to Pablo B., for backing me up every time I needed it. In particular, reaching the end I have to mention again Pablo B. and Laura, because they have helped me so that the week of my defense was organized and smooth; thanks, friends!

Also the group of teachers and students that were part of that fantastic adventure that was teaching Functional Programming (those not already mentioned, at least... :-), deserves my gratitude. They believed in me, learned from me, taught me, confronted me, and assisted me in the most rewarding work in the world: to teach. I will mention but a few: Matías M., Marcelo C., el Colo, Nachi, Majo, Vero, Wanda, Vanesa, Agustina, Mariana, Pepe, Valeria, Hernán C., Mara.

Shevek imagined and realized the Syndicate of Initiative, inspiring me to start a similar one. I will not say much more than that here, but the time I took away from my thesis to conceive of them, encourage them, and help them was really a good investment. You make many things more valuable; do not give up — ever.

Every one of the people I mention here has seen my voyages as something different. But there are four persons, my buddies from long, that have 'dressed' them with their fantasies. Luis, Federico, José, and Eduardo K., had only one of my trips been half the things you have imagined, I would have taken a completely different direction for my life! I value your friendship as one of the most important jewels in my treasure.

Now it's the turn for the East of the Atlantic. I have made a lot of friends in Chalmers, and it is difficult to order them now — again the trade-off! In first place I want to mention Koen; I marvel at his intelligence, our shared passion for gaming, and the easy way he accepted me as one of his friends. Koen, I will never forget your expression when I appeared by surprise during your PhD defense, and how you made everyone turn to look at me! Then I have to mention Josef; he never refused to chat with me, or to discuss any topic, and perhaps it was that, or his indelible smile, that makes me feel so comfortable with him. And I have to thank both of them for hosting the Multi-meetings, and inspiring me to have my own ones in Argentina. In third place I want to mention Birgit, because she has a permanent good mood, always smiling, and her joy is contagious, and she offered me her friendship so openly. In particular, I will not forget that you were the one that started that crazy idea of a surprise party for my 34th birthday; I owe you for that. And then, all the rest: Jörgen, Elin, Tuomo, Rogardt, Ilona, Mia, Andrei, Walid, Makoto, Per B., Gordon, Boris, David R., Björn, Dave, Peter, Magnus, Thomas, and many more. All of you together meant that every time I was leaving Göteborg, some piece of me was left behind.

There is a special place in my heart for those with whom I shared so many discussions (even in verses!), plots, and battles. Juan G., Pablo S., Pite, Marcos, and again Steve and Hernán, playing our own version of 1914 with you was great. It was very amusing and entertaining, and it has also taught me several useful things.

Through my PhD courses I have made several teachers suffer. Alejandro R., Laurance, Thomas, Thierry, Philippas, Gonzalo, Martin F.C., Daniel F. (Frito), thank you for your patience, your teachings, and your time.

Institutions cannot live without the people that make them possible. But they outlive any particular individual, and thus, they have a place of their own here. I am in debt to several institutions, but specially to LIFIA at the Faculty of Informatics in the University of La Plata, in Argentina, for being my working place, and providing me with all the things needed to do good research; ESLAI, because its spores are growing and blossoming in hundreds of places around the world, and every one of them is a home for me; the Department of Computer Science from the Faculty of Exact Sciences at University of Buenos Aires, in Argentina, because it was my alma mater, and the funding they gave me was a key ingredient in this recipe; the Department of Computing Science from Chalmers University of Technology, in Sweden, because of the stimulating environment it provides, and because I was accepted as one more of their inhabitants; the European Community, because of the Alfa projects: one of them made possible my visits to Göteborg; and to the insitution behind `http://citeseer.nj.nec.com`, because

of the invaluable service they provide to the academic community.

In the last ‘minutes’ of my trip, I have met some people who constitute a promise for the future. Marjan, Tomaz, Simon, Andrea, Pedro, María, I hope we can continue working and enjoying things together.

A special place should be reserved for my favorite writer, Úrsula Le Guin. My life is much richer thanks to her; her words flood my mind, helping me to bear the pain when it comes, and thus making me feel not alone, so brotherhood with others can start, as Shevek has stated. I treasure your autograph as a very special gift.

Finally, for those who can understand the turns of my twisted mind, I will mention my own Atro, Chifoilisk, Saio Pae, and the rest of the Urrasti. I would not have been so sharp, focused, and resourceful, had you not been there.

And thus we come to the end of this ‘chapter’. I tried to made a brief description of all the wonders I have lived through, seen, and suffered during these years. But it would have taken me all the pages of this thesis to be fair. So, to finish, I will once again make mine the words of Úrsula.

> *You can go home again (. . .), so long as you understand that home is a place where you have never been.*
>
> The Dispossessed
> Úrsula K. Le Guin

Chapter 1

Introduction

Programming is a handcraft!!

Course on "Program Specialization"
John Hughes
Montevideo, 1997

1.1 Program Specialization

"Programming is a handcraft!!" was the first sentence John Hughes said during my first course on automatic program generation. And the explanation followed: almost every program is hand-written by some human programmer, and almost every line of code is checked by a human programmer.

The situation is comparable with the task of carpet-weaving: if every carpet were woven by a human weaver, the production of carpets would be limited by the speed of the weavers, and their number; also the quality and complexity of the carpets produced would depend heavily on the ability of the artisans, and on their physical and mental state during the weaving process: many mistakes might be made because of the tiredness of the weaver.

In the case of carpets, the solution to go from an artistic handcraft to an industrial discipline of carpet-making was the invention of the weaver-machine. The machine performs the repetitive tasks needed to weave a carpet, and the old craftsman is replaced by designers, programmers, and operators for the weaver-machine. The old error-prone, slow task of hand weaving is replaced by a highly efficient, fast process performed by a machine, and production can be improved to limits that were impossible with the old model.

The equivalent of the weaver-machine in the case of programming would be a program that produces programs automatically. There are several well-known examples of programs generating programs, but most of them are restricted in some way, the typical case being the restriction to a certain domain (for example, parser generators); thus we can talk of domain-oriented program generation. But to turn programming from a handcraft to an industrial discipline, we need a general way to produce programs automatically.

Automatic program generation is the field treating the problem of automatic production of programs from a general point of view. There are several different ways to produce a program automatically. *Program specialization* — the form of program generation that we consider in this thesis — is perhaps the most successful: a given program

is used as input to produce one or more particular versions of it as output, each specialized to particular data. The program used as input is called the *source program*, and those produced as output are called the *residual programs*.

The motivation for this approach comes from the following scenario: when solving several similar problems, a programmer can choose between either writing several small efficient programs by hand and then running them to solve each problem, or writing a big, general, and usually inefficient program with many parameters, changing the data on each run to solve each problem. In the former case, the programs will be very efficient, but a lot of effort of programming is required; in the latter case, the programming effort is better spent, but the resulting program will repeat many computations that depend on the additional parameters used to distinguish among instances. The idea of program specialization is to take the best of both worlds, writing a general source program and then *specializing* it to each situation, producing the set of small residual program for each instance of the class of problems. The classic example is the recursive `power` function calculating x^n

```
power n x = if n == 1
            then x
            else x * power (n-1) x
```

whose computation involves several comparisons and recursive calls, but that, when the input parameter `n` is known — for example let's say it is `3` — can be specialized to a non-recursive residual version that can only computes powers of that particular `n` — the function

```
power3 x = x * (x * x)
```

in our example. It is clear that the residual version is much more efficient than the source version when computing cubes.

Program specialization has been studied from several different approaches; among them, *Partial Evaluation* [Jones *et al.*, 1993; Consel and Danvy, 1993] is, by far, the most popular and well-known. And because the work of John Hughes [Hughes, 1996b] that has inspired the present thesis, has in its turn been inspired by the partial evaluation approach, we begin by describing its basics.

Partial evaluation is a technique that produces residual programs by using a generalized form of reduction: subexpressions with known arguments are replaced by the result of their evaluation, and combined with those computations that cannot be performed. That is, a partial evaluator works with the *text* of the source program by fixing some of the input data (the *static data*) and performing a mixture of computation and code generation to produce a new program. The programs produced, when run on the remaining data — called *dynamic data* because it is only known during run-time — yield the same result as the original program run on all the data. Partial evaluation may sound like a sophisticated form of constant folding, but in fact a wide variety of powerful techniques are needed to do it successfully, and these may completely transform the structure of the original program.

An area where partial evaluation is particularly successful is the automatic production of compilers: compilation is obtained by specializing an interpreter for a language to a given program [Futamura, 1971; Jones *et al.*, 1985; Jones *et al.*, 1989; Wand, 1982; Hannan and Miller, 1992]. In this case, the interpreter is used as the

source program, the object program is used as the static data, and then the residual program is the compiled version of the object program; so, the specialization of the interpreter yields compilation. Another layer of complexity can be added when the partial evaluator is written in the language it specializes: self-application becomes possible, and thus compilers can be generated as well. The (code of the) partial evaluator is the source program and the interpreter is the static data; the resulting residual program performs specialization of the interpreter mentioned above: a compiler! This is very useful in the area of domain-specific languages [Thibault *et al.*, 1998], where the cost of generating a compiler must be kept to a minimum. Other areas where partial evaluation has been applied successfully include networking [Muller *et al.*, 1998], software architectures [Marlet *et al.*, 1999], hardware design and verification [Hogg, 1996; Singh and McKay, 1998], virtual worlds [Beshers and Feiner, 1997], numerical computation [Lawall, 1998], and aircraft crew planning [Augustsson, 1997].

The partial evaluation community chooses to regard a partial evaluator as a "black box" that can be used to optimize existing programs by totally automatic means [Jones, 1996] — that is, their primary goal is *efficiency* obtained by automatic means. The main goal guiding our work, instead, is to obtain increased productivity by means of a specializer. When writing the source program, a programmer can afford a *vast amount* of static computations just to spare a small amount of dynamic ones; being aware of how the specializer works is the key for the programmer to increase his abilities. While *efficiency* and *productivity* are two sides of the same coin, we argue that the latter is much more relevant. This is our *rule of thumb*: we regard a program specializer as a tool that *amplifies* the programming abilities of programmers, and so, it should be very flexible.

An important notion in the program specialization approach is that of *inherited limit* [Mogensen, 1996; Mogensen, 1998a]. An inherited limit is some limitation in the residual program imposed by the structure of the source program and the specialization method; that is, the *form* of obtainable generated programs is limited by the form of the program to be specialized. Inherited limits imply that residual programs cannot use the full potential of the language. For example, the number of functions may be an inherited limit: if every function in the source program must be specialized in a unique way — this is called *monovariant specialization* — then the number of functions in the residual program cannot be more than that in the source one, and then it will constitute an inherited limit. For that reason, all good partial evaluators are polyvariant: *polyvariance* is the ability of an expression to specialize to more than one residual expression. In the majority of existing partial evaluators, polyvariance is the default for let-bounded declarations, thus allowing the declaration of functions that can be specialized to multiple different functions.

Other inherited limits we can encounter in a specializer are the size of the program, the number of variables, the nesting depth of some structures, the number of types, etc. Mogensen [1996] has argued that historical developments in program specialization gradually remove inherited limits, and suggests how this principle can be used as a guideline for further development. Our work made use of this guideline to further develop program specialization.

One good way to detect the presence or absence of inherited limits is to specialize

a self-interpreter and compare the residual programs with the source one: if they are essentially the same, then we can be confident that no inherited limits exist. We then say that the specialization was *optimal* (or *Jones-optimal*, after Neil Jones [Jones, 1988b]).

Partial evaluation, in the case of self-interpreters written in untyped languages, can obtain optimality; but for typed interpreters things are different. As partial evaluation works by reduction, the type of the residual program is necessarily the same as that of the source one; thus, the residual code will contain type information coming from the representation of programs in the interpreter: optimality cannot be achieved, and the inherited limit of types is exposed. This problem was stated by Neil Jones in 1987 as one of the open problems in the partial evaluation field [Jones, 1988b].

1.2 Type Specialization

A different form of program specialization is called *Type Specialization* [Hughes, 1996b; Hughes, 1996a; Hughes, 1998b]; it was introduced by John Hughes in 1996 as a solution for optimal specialization of typed interpreters — that is, it removed the inherited limit of types — but it has also proved to be a rich approach to program specialization. Type specialization resembles partial evaluation in many respects — it is also a technique for program specialization, after all — but the reader familiar with partial evaluation should be warned: it is a *different* approach to program specialization. For example, using the same interpreter for a given object language L, a compiler for both untyped and typed versions of L can be automatically produced, just by varying the annotations. This is possible because type checking is an integral part of type specialization, and so it is embedded in the process of generating the residual program; partial evaluation falls short in this respect.

The basic idea behind type specialization is to move static information to the type, thus specializing *both* the source term and source type to residual term and residual type. Types can be regarded as a static property of the code, approximating the known facts about it; for example, in a functional language based on the simply typed lambda-calculus, when some expression has type *Int*, we statically know that if its evaluation produces some value, it will be an integer. But if the expression is known to be the constant 42, for example, then a better approximation can be obtained by having a *type representing the property* of being the integer 42 — let's call this type $\hat{42}$, and allow the residual type system to have types like this one. Having all the information in the type, there is no need to execute the program anymore, and thus, we can replace the integer constant by a dummy constant having type $\hat{42}$ — that is, the source expression $42 : \mathit{Int}$ can be specialized to $\bullet : \hat{42}$, where $\bullet$ is the dummy constant. This specialization of types into ones expressing more detailed facts about expressions — the type *Int* specialized to the type $\hat{42}$ in this example — needs a more powerful residual type system, which is the key fact allowing optimal specialization for typed interpreters.

Polymorphism is an important property of modern typed languages, because it allows reuse of code while keeping all the good properties of strong typing: one function is reused at several different instances without change. It is important that a formalism for program specialization can produce polymorphic programs for two reasons: firstly, because we want as few inherited limits as possible in a specializer, and currently poly-

morphism is one of them, and secondly, because polymorphism is a way to reduce the size of code, avoiding the duplication of a piece of code when only the type of both instances is different. Unfortunately, in the original formulation of type specialization [Hughes, 1996b] both the source and residual type systems were monomorphic, and this imposes an inherited limit: the residual programs cannot be polymorphic, they cannot have more polymorphism than the source program.

Type specialization is specified by a system of rules that is very similar to those used to specify type systems — indeed, type specialization can be seen as a generalized form of type inference. In this way, the specification of the specialization procedure is modular: the specialization of new constructs in the source language can be specified by the addition of new rules, without changing the rest of them. The monomorphic nature of the rules used in the original formulation produces some problems: there are rules that are not completely syntax directed and so, for some source terms, several different *unrelated* specializations can be produced. This is very similar to the problem appearing in simply typed λ-calculus when typing an expression like $\lambda x.x$: the type of x is determined by the context of use, and different types for this expression have no relation between them expressible in the system. The solution for this last problem is to extend the type system to a polymorphic one, where a principal type expresses all the possible ways of assigning a type to a given expression. For this reason, we say that the original formulation of type specialization lacks the property of *principality*.

The lack of principality has important consequences. Firstly, the extension to produce polymorphic residual code or specializing polymorphic source code is very difficult to treat. Secondly, an algorithm for specialization will fail for terms with more than one specialization when the context does not indicate which one to choose, or even worse, will choose one of them arbitrarily. Thirdly, although the context provides enough information for the choice, it is too restrictive for the *whole* process of specializing a term to depend on the *whole* context; with this kind of restriction, specialization of program modules is very hard to achieve, because in order to specialize a module, the specializer may need to know *all* the program — in general, information about other modules on which the current one has no syntactic dependencies.

Our long term goal is to extend the power of type specialization to treat other features of modern languages, such as polymorphism, type classes and lazy evaluation — to be able, for example, to type specialize Haskell programs.

1.3 Contribution of This Thesis

The main contribution of this thesis is the formulation of a system for type specialization that can produce *principal specializations* as proved in Theorem 6.26. A principal specialization is one that is more general than any other a given source term can have, and from which all those can be obtained by a suitable notion of instantiation.

We express the type specialization system in a different way from the original formulation of Hughes, by using constraints to express specialization, and separating the specialization in two phases: constraint generation and constraint solving. In this way we can express specialization with different degrees of detail, thus providing a better understanding of the information flow during specialization, and enabling the appli-

cation of different heuristics to the process of calculating the right residual program. On the other hand, it makes it possible to define modular specialization: each module can be specialized in a principal manner, and to link the residual code to the residual main program, the right instantiation should be produced. Finally, the improved understanding of the information flow can be helpful in the treatment of other problems related with type specialization, such as the interaction of recursion with features of type specialization such as polyvariance.

The possibilities that our new formulation allows are shown by considering the removal of the inherited limit of polymorphism: our approach can generate polymorphic programs from monomorphic source ones. This is important because it increases the abilities of program specialization, and makes the generation of more powerful and general programs possible. However, the development presented here is just a basic one, and our purpose when including it was to show the potential of our formulation.

Finally, from a slightly different perspective, we show that the type specialization problem posed by Neil Jones in 1987 is not really there. We propose to read the problem in a different way, showing that optimality for typed interpreters can indeed be obtained with a Mix-style partial evaluator. This insight has been discovered during our work with type specialization, and, while it does not invalidate 20 years of good research, it shows that sometimes researchers' inspiration can come from very strange places, indeed.

1.4 Overview

The thesis is divided into four parts.

In Part I we present Type Specialization. We begin in Chapter 2 describing partial evaluation, and discussing the features of it relevant to our work. In Chapter 3 we describe the work of John Hughes on type specialization [Hughes, 1996b; Hughes, 1998a; Dussart *et al.*, 1997b; Hughes, 2000], which is the starting point of this thesis; we explain the pragmatics of type specialization, and present the specification given by Hughes. Finally in Chapter 4 we provide examples motivating the use of type specialization and discuss some of its limitations.

Then in Part II we present the new formulation of type specialization, which we call *Principal Type Specialization* (PTS). We start in Chapter 5 by reviewing the theory of qualified types developed by Mark Jones [Jones, 1994a], because it is the technical foundation of our development. Then in Chapter 6 we explain the problems of the original formulation and present the new formulation for a small subset of the language. After that, in Chapter 7 we develop an algorithm calculating principal specializations, and we prove the principality of the system introduced in the previous chapter. Constraint solving and postprocessing phases are considered in Chapter 8. We describe, then, in Chapter 9, how to extend the principal specialization to a full language including static functions, datatypes and recursion; the problems posed by the dynamic version of recursion are also discussed. This part ends in Chapter 10, with a quick glance at the details of the prototype implementation we have made in the functional language Haskell [Peyton Jones and Hughes (editors), 1999].

The PTS system is the basis on which we built a type specialization framework that

handles polymorphism. It is presented in Part III. We consider only the production of polymorphic residual programs from monomorphic ones (*residual polymorphism*), showing the possibilities of our formulation to remove this inherited limit. The full treatment of this topic, as well as the specialization of polymorphic programs (*source polymorphism*) is left for future work.

Part IV presents related and future work and concludes. We review some of the different approaches to program specialization in Chapter 13, and compare them with ours, providing an appropriate context for this work. Then, in Chapter 14, we discuss some of the possible continuations for the work presented here. Finally, Chapter 15 presents the insights gained during the development of this thesis.

Part I

Type Specialization

Chapter 2

Partial Evaluation

Ahead at warp 9. Engage!

Captain Picard
The Best of Both Worlds
Star Trek, Next Generation

A one-argument function can be obtained from one with two arguments by fixing one input. This is called 'projection' in mathematical analysis, and 'currying' in logic. Program specialization, however, deals with *programs* rather than functions. The first formulation of the idea of program specialization was Kleene's *s-m-n* theorem, 50 years ago [Kleene, 1952], also an important building block of the theory of recursive functions. Basically, what the theorem says is that a specialized version of a recursive function is itself an effectively constructible recursive function — that is, there exists a recursive function which acts as a general specializer.

Partial evaluation is a technique to specialize a program by fixing some of its inputs and performing the computations that depend on them [Jones *et al.*, 1993; Consel and Danvy, 1993]. So, residual programs are produced by a *generalized form of reduction*: subexpressions with known arguments are replaced by the result of their evaluation, and combined with those computations that cannot be performed. But it should be kept in mind that a partial evaluator works with the *text* of the source program by fixing the known input data (the *static data*) and performing a mixture of computation and code generation to produce a new program — that's why Ershov [1982] called it "mixed computation", hence the historical name `mix` used for a partial evaluator. The programs produced, when run on the remaining data — called *dynamic data* because it is only known during run-time — yield the same result as the original program run on all the data.

In this chapter we briefly revisit the main notions of partial evaluation, mention some of its successful applications, and present the notion of compilation by specialization, and show a particular problem that is difficult to solve with this approach: optimal compilation of interpreters written in typed languages.

2.1 Partial Evaluation

The idea of partial evaluation can be clearly expressed with equations. We write $[\![\texttt{p}]\!]_L$ for the meaning, as a function from inputs to output, of a program `p` written in language L. When dealing with partial evaluators, several languages are used; we denote them symbolically by

L, the language in which the partial evaluator is implemented,

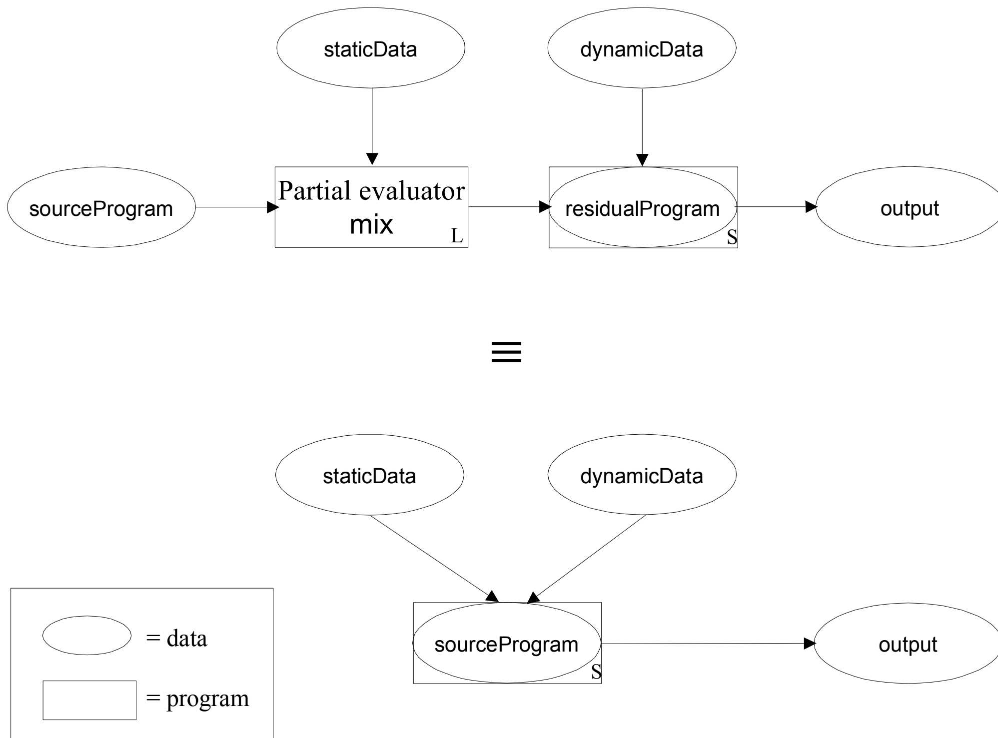


Figure 2.1: Graphical description of `mix`

S, the language specialized by the partial evaluator, and

O, an object language under treatment.

We use the name `mix` for program written in a language L that is the partial evaluator for a language S, and whose behaviour is specified by:

$$[\![\texttt{mix}]\!]_L\ \texttt{sourceProgram staticData} = \texttt{residualProgram}$$

such that

$$[\![\texttt{residualProgram}]\!]_S\ \texttt{dynamicData} = [\![\texttt{sourceProgram}]\!]_S\ \texttt{staticData dynamicData}$$

(occasionally, instead of a single language S, there can be two different languages: a source language and a target language for source and residual programs; this is important, for example, when the code of `mix` is to be used as input to itself in order to specialize it [Jones *et al.*, 1993]; the rationales for variations of a specializers target language are discussed by **?** [?]). The equations are also clarified in the diagram on Figure 2.1.

Several techniques known from the area of program transformation [Burstall and Darlington, 1977] are used in partial evaluation; some of them are *constant folding* (replacing variables with constant values by the actual values), *folding and unfolding of function calls* (the replacement of a definition by a function call, or the call by the corresponding instance of the body of the function), *symbolic computation* (the manipulation of information in symbolic form, e.g. not considering the actual values of

variables), and *program point specialization* (a combination of definition, folding, and memoization such that a single function — or point — in a program can be specialized to several different versions, each corresponding to different static data). Although partial evaluation may sound like a sophisticated form of constant folding, a wide variety of powerful techniques are needed to do it successfully, and these may completely transform the structure of the original program.

In the case of the `power` function mentioned in Chapter 1, the residual program can be obtained by partial evaluation by computing all the expressions involving `n`, reducing the `if-then-else`, unfolding the function calls, and repeating these steps until the base case is reached:

$$[\![\texttt{mix}]\!]_L \ \texttt{power 3} \ = \texttt{power3}$$

such that

$$[\![\texttt{power3}]\!]_S \ \texttt{x} \ = [\![\texttt{power}]\!]_S \ \texttt{3 x}$$

We want to draw attention to the fact that the main emphasis of partial evaluation has been to achieve *efficiency by automatic means* while keeping *correctness* of the code. In the words of Neil Jones *et al.* [1993]:

> *"The chief motivation for doing partial evaluation is speed..."* [Jones *et al.*, 1993, pp.5]
>
> *"By this we mean more than just a tool to help humans write compilers. Given a specification of a programming language, for example a formal semantics or an interpreter, our goal is* automatically *and* correctly *to transform it into a compiler..."* [Jones *et al.*, 1993, pp.9]
>
> *"Further, users shouldn't need to understand* how the partial evaluator works. *If partial evaluation is to be used by non-specialists in the field, it is essential that the user thinks as much as possible about the problem he or she is trying to solve, and as little as possible about the tool being used to aid its solution."* [Jones *et al.*, 1993, pp.15]

This amounts to regarding a partial evaluator as a sophisticated form of a compiler, instead of as a tool for automatic program generation. We prefer to put the emphasis in the increased ability a programmer has when using a program specializer; while efficiency is still an issue, in this thesis we use *expressiveness* as the main guide.

Partial evaluation has been studied in several programming paradigms: functional programming [Gomard and Jones, 1991; Mogensen, 1989; Consel, 1990b], imperative programming [Andersen, 1992; Andersen, 1993], logic programming [Lloyd and Shepherdson, 1991; Fuller and Abramsky, 1988], and object oriented programming [Marquard and Steensgaard, 1992], including compiler generation for object oriented languages [Khoo and Sundaresh, 1991]. However, historically the innovations were first studied in the functional programming paradigm, because of its simplicity, and they were incorporated into the other paradigms when they were understood. We follow the same principle, and restrict our study to functional languages.

Partial evaluation has been applied successfully in an important number of areas; among them, we can mention the following ones.

Compilation

[Wand, 1982; Hannan and Miller, 1992; Hannan and Miller, 1990; Penello, 1986]
Compilers can be mechanically constructed from a description of the semantics of the language, either an interpreter (as described in the next section), from an operational semantics, or from other descriptions (LR parsing tables, etc.). The resulting compilers are guaranteed by construction to be correct with respect to the semantics.

Domain-specific languages

[Thibault *et al.*, 1998; Thibault *et al.*, 1999]
A domain-specific language is a language that is expressive uniquely over the specific features of programs in a given problem domain. Partial evaluation has been used in the development of application generators (translators from a domain-specific to a general purpose language).

Networking

[Muller *et al.*, 1998]
Remote procedure call (RPC) is a layer in the communication stack on some operating systems. Partial evaluation has been applied to Sun RPC code, improving it to run up to 1.5 times faster; these optimizations have been applied to mature, commercial, representative system code.

Software architectures

[Marlet *et al.*, 1999; Marlet *et al.*, 1997]
Software architectures express how systems should be built from various components and how those components should interact. Implementing the mechanisms of flexible software architectures can lead to efficiency problems; those problems can be solved by the careful use of a partial evaluator.

Hardware design and verification

[Hogg, 1996; Singh and McKay, 1998; Au *et al.*, 1991]
Partial evaluation provides a systematic way to manage the complexity of dynamic reconfiguration in the case where a general circuit is specialized with respect to a slowly changing input. For example, circuits for encryption algorithms with specific keys can be designed using partial evaluation techniques, reducing the number of circuit resources needed.

Virtual worlds

[Beshers and Feiner, 1997]
Partial evaluation and dynamic compilation have been used in the implementation of flexible and efficient interactive visualization tools (e.g., information visualization, pictorial explanations, and multimedia explanations).

Numerical computation

[Lawall, 1998; Berlin, 1990; Berlin and Weise, 1990; Berlin and Surati, 1994]

For an important class of numerical programs, partial evaluation can provide marked performance improvements: speedups over conventionally compiled code that range from seven times faster to 91 times faster have been measured; for example, efficient implementations of the Fast Fourier Transform have been produced automatically and reliably by partial evaluation. By coupling partial evaluation with parallel scheduling techniques, the low-level parallelism inherent in a computation can be exploited on heavily pipelined or parallel architectures.

Aircraft crew planning

[Augustsson, 1998]
The scheduling of aircraft and crews in large airlines is a very complex problem, using a very complex system of rules to specify restrictions, but with several of the values of variables fixed (such as the number and type of aircrafts); partial evaluation is especially well-suited to this problem.

2.2 Binding Time Analysis

A key question in any partial evaluator is this: which expressions should be evaluated, and which should be built into the residual program? Partial evaluation can be done in either of two ways: using the values of the static data to determine which operations should be computed, or performing an a priori analysis of the program to divide its operations, based only on the knowledge of the division of the inputs *but not on their actual values*. These two strategies are usually called *online* partial evaluation and *offline* partial evaluation, respectively. The terminology used in this section is the one used by Jones *et al.* [1993].

An online partial evaluator uses the concrete values computed during program specialization to guide further specialization decisions; an offline partial evaluator does not use the values, but only the information of which of the data are known, to produce a congruent division, ensuring that there is sufficient static information to do static computations. For computability reasons, offline methods will inevitably classify some expressions as dynamic even though they may sometimes assume values computable by the specializer. An example of this is a conditional expression `e = if` e_1 e_2 e_3 where e_1 and e_2 are static but e_3 is dynamic. The expression has to be classified as dynamic, but an online test would discover that it may be computable if e_1 evaluates to `True`. If expression `e` is the argument of a function `f`, an online partial evaluator could apply the function whenever e_1 was `True`; but it would take more work to make an offline specializer to do that (for example, converting the program to continuation passing style [Consel and Danvy, 1991]).

All early partial evaluators were online, but offline techniques are good for self-application and the generation of program generators. Both approaches have their benefits and their drawbacks. While online techniques can exploit more static information during specialization, thus producing better residual programs, their behaviour is less predictable. And the opposite is true for offline techniques: it is very clear what the division of the program constructs is, but less improvement is possible.

The process of taking a program and the knowledge of which of its input variables

are static (but *not* their values!) and producing a division of all program variables and operations into static or dynamic is called *Binding Time Analysis* (also known as BTA), because it calculates at what time the value of a variable can be computed, that is, the time when the value can be bound to the variable [Jones, 1988a; Jones *et al.*, 1993]. The success of BTA was due to the fact that by using it, self-application becomes feasible, producing residual program generators that were reasonably small and efficient. Indeed BTA was introduced to make self-application possible [Jones *et al.*, 1985; Sestoft, 1985; Jones *et al.*, 1989].

One of the key features allowing BTA is the restrictions imposed by the use of reduction as the mechanism for passing information. The most common example is that of a dynamic function: as it will not be reduced at specialization-time, the argument of such a function will not be known before run-time, and thus is forced to be dynamic; conversely, if the argument is static, the function must be made static. A general principle establishes that any variable that depends on a dynamic value must itself be classified as dynamic [Jones, 1988a]. This principle has different results according on how the notion of dependency is defined; in general, the semantics of the specializer affects the way in which BTA behaves, by changing the notion of dependency. For example, a specializer written in continuation passing style (CPS) allows its BTA to annotate as static the computations in the branches of a dynamic conditional, because it specializes the context of it twice, once in each branch [Lawall and Danvy, 1994].

There has been a vast amount of research in the area of BTA, improving the way in which different constructs are annotated. To cite just a few advances, BTA was extended to handle partially static data structures [Mogensen, 1988], higher-order functions both untyped and typed [Consel, 1990a; Henglein, 1991], pointers and memory allocation [Andersen, 1993], polyvariant expansions [Thiemann and Sperber, 1996], and polymorphic languages [Mogensen, 1989; **?**; Heldal and Hughes, 2001].

2.3 Compiling by Partial Evaluation

An area where partial evaluation is particularly successful is the automatic production of compilers: compilation is obtained by specializing an interpreter for a language to a given program [Wand, 1982; Hannan and Miller, 1992; Hannan and Miller, 1990].

Consider a program `interpreter`, written in S, which is an interpreter for some language O; that is, the result of running `interpreter` on a `program` written in O is the same as the result of running that `program` on the `input-data`:

$$[\![\texttt{interpreter}]\!]_S\ \texttt{program input-data} \ = [\![\texttt{program}]\!]_O\ \texttt{input-data}$$

Then, using the `mix` equations, we can easily verify that the following equations hold:

$$\begin{aligned}
[\![\texttt{mix}]\!]_L\ \texttt{interpreter program} &= \texttt{program'}\\
[\![\texttt{program'}]\!]_S\ \texttt{input-data} &= [\![\texttt{interpreter}]\!]_S\ \texttt{program input-data}\\
&= [\![\texttt{program}]\!]_O\ \texttt{input-data}
\end{aligned}$$

This tells us that by specializing the O-interpreter to a program, we can obtain an equivalent program in another language (the diagram in Figure 2.2 illustrates the equations).

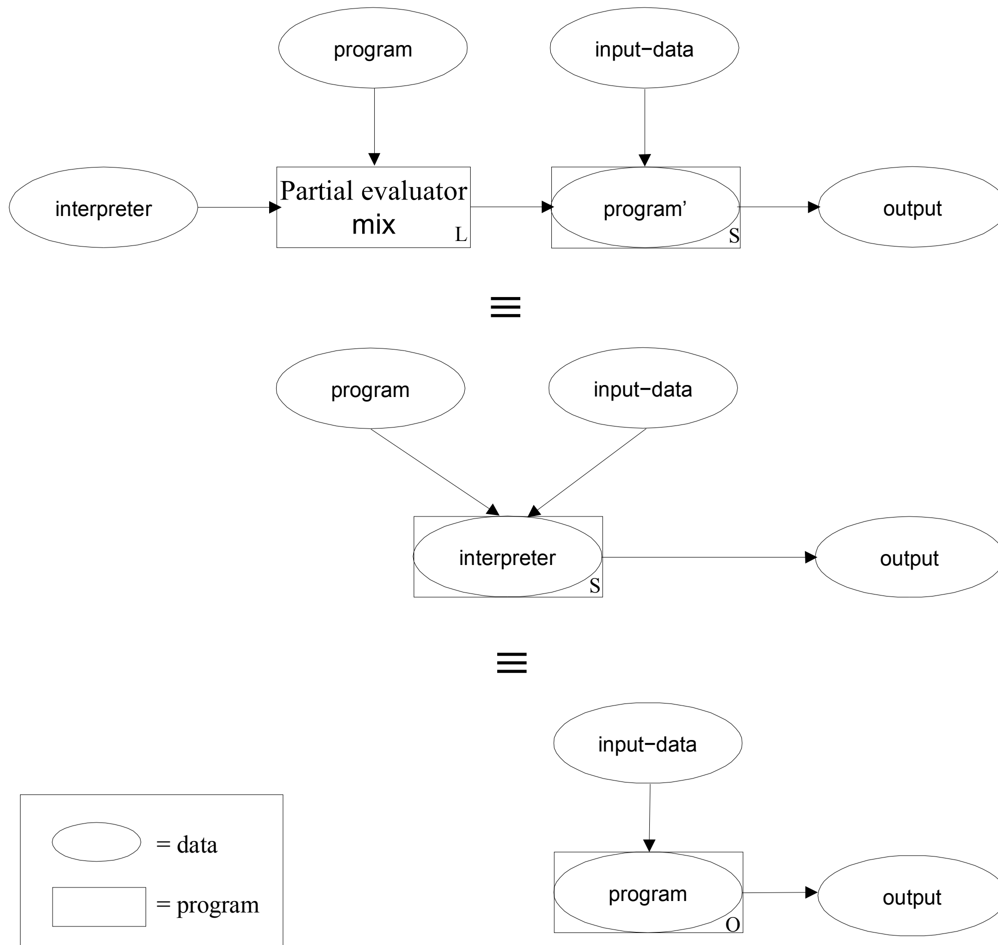


Figure 2.2: Compilation by means of `mix`

Thus, ($[\![$`mix`$]\!]_L$ `interpreter`) compiles O-programs to S-programs, and the only programming effort was writing the O-interpreter — as it is easier to write an interpreter than a compiler, the gains are evident. This result is known as the *first Futamura projection* [Futamura, 1971]; there are two more Futamura projections, related to automatic production of a compiler (*second Futamura projection*), and automatic production of a compiler generator (*third Futamura projection*) — Ershov [1988] provides an account of their discovery.

To illustrate in more detail the notion of compilation by partial evaluation, we consider a simple (self-)interpreter for untyped lambda calculus. We use the one presented by Gomard and Jones [1991] (Figure 2.3), which is written in the untyped (or dynamically typed) functional language Scheme [Abelson *et al.*, 1998]. To be able to compare it with the more complex versions used later, we also offer it in a "sugared" version written in an untyped Haskell-like language in Figure 2.4. We have taken some liberties when sugaring Scheme. Firstly, we are using constructors to represent S-expressions whose `car` is a particular atom; later on we turn this into a proper datatype, when using Haskell to express the example in a typed setting (Section 2.4). In the case of Scheme, S-expressions can be viewed both as programs or data, depending on how they are used; but when types are used, this amounts to encoding the program in the datatype, and

```
(fix (lambda (sint)
(lambda (exp) (lambda (env)
  (if (atom? exp)
      (env exp)
    (((lambda (hdexp) (lambda (tlexp)
       (if ((eq? hdexp) (const const))
           (lift (car tlexp))
       (if ((eq? hdexp) (const lam))
           (lam-r value
             ((sint (take-body exp))
              (lambda (y) (if ((eq? y) (take-var exp)) value (env y)))))
       (if ((eq? hdexp) (const @))
            (@-r ((sint (car tlexp)) env)
                 ((sint (cadr tlexp)) env))
       (if ((eq? hdexp) (const if))
           (if-r ((sint (car tlexp)) env)
                 ((sint (cadr tlexp)) env)
                 ((sint (caddr tlexp)) env))
       (if ((eq? hdexp) (const fix))
           (fix-r ((sint (car tlexp)) env))
       (@-r (@-r error-r exp)
            (lift (const "Wrong syntax")) ))))))))
      (car exp)) (cdr exp)))))))
```

Figure 2.3: Scheme code for a self-interpreter of lambda calculus, `sint`.

$$
\begin{array}{lll}
\mathit{sint} = \mathbf{fix}\,(\lambda \mathit{sint}.\lambda \mathit{exp}.\lambda \mathit{env}. & & \\
\quad \mathbf{case}\ \mathit{exp}\ \mathbf{of} & & \\
\qquad \mathit{Var}\ x & \rightarrow & \mathit{env}@x \\
\qquad \mathit{Const}\ n & \rightarrow & \mathbf{lift}\ n \\
\qquad \mathit{Lam}\ x\ e & \rightarrow & (\lambda^{D} v.\,(\mathit{sint}@e@\,(\lambda y.\,\mathbf{if}\ \ x == y \\
 & & \qquad\qquad\qquad\qquad \mathbf{then}\ v \\
 & & \qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{env}@\,y))) \\
\qquad \mathit{App}\ e_1\ e_2 & \rightarrow & (\mathit{sint}@\,e_1@\,\mathit{env})\ \ @^{D}\ (\mathit{sint}@\,e_2@\,\mathit{env}) \\
\qquad \mathit{If}\ \ e\ e_1\ e_2 & \rightarrow & \mathbf{if}^{\,D}\ \mathit{sint}@\,e@\,\mathit{env} \\
 & & \quad \mathbf{then}\ \mathit{sint}@\,e_1@\,\mathit{env} \\
 & & \quad \mathbf{else}\ \mathit{sint}@\,e_2@\,\mathit{env} \\
\qquad \mathit{Fix}\ e & \rightarrow & \mathbf{fix}^{D}\ (\mathit{sint}@\,e@\,\mathit{env}) \\
\qquad _ & \rightarrow & \mathbf{error}\ \ \mathit{exp}\ \ \text{“Wrong syntax”}\ )
\end{array}
$$

Figure 2.4: “Sugared” version of `sint` in a Haskell-like syntax.

that is the reason we use double-quotes in the sugared version: it means that the string should be encoded with the proper constructors. Secondly, the program is clearly untyped (or dynamically typed), because the result of the case expression is a number in one branch, but a function in another; also the tag `Const` is used for numbers, booleans and operations. Thirdly, the binding time annotations indicated by the suffix `-r` in the Scheme version are indicated in the sugared one by ^{D}; static code is not marked because the Scheme program, when applied to a particular lambda expression, will produce the residual program — it is a *generating extension* of `sint`, a name coined by Andrei Ershov for a program that generates specialized versions of another, in this case, `sint`.

The self-interpreter takes two arguments: an expression to evaluate, and an environment relating free variables to values. It analyzes the form of the expression, and combines the evaluation of the subexpressions in an appropriate way. The interesting case is that of lambda abstractions: the result of evaluating an lambda expression is a function, and in the recursive call the environment is extended with the binding of `x` to the value `v`, argument to this function. This is a standard translation from denotational semantics to Scheme.

To illustrate the function of the partial evaluator, Gomard and Jones [1991] use the Fibonacci function, and partially evaluate `sint` to it. The Scheme version of `fib` used there is:

```
(fix (lambda (fib) (lambda (x)
  (if ((< x) (const 2))
      (const 1)
      ((+ (fib ((- x) (const 1))))
          (fib ((- x) (const 2))))))))
```

Looking at the code generated by $[\![$`mix`$]\!]_L$ `sint fib`, for a `mix` specializing Scheme, we can observe that it is structurally equal to the original object program; this means that the partial evaluator was optimal: it removed a complete layer of interpretation (that of `sint`).

```
(fix (lambda (value-6) (lambda (value-7)
  (if ((< value-7) (const 2))
      (const 1)
      ((+ (value-6 ((- value-7) (const 1))))
          (value-6 ((- value-7) (const 2))))))))
```

We present the same example using the Haskell-like syntax and the encoding as type tagged expression in Figure 2.5. Observe that the encoding of the small function is complex, full of parenthesis, and similar in structure to the Scheme program; it is for that reason that we choose to express them as strings to be appropriately encoded (we discuss the consequences of this way of looking at object programs in Chapter 12).

The result of $[\![$`mix'`$]\!]_L$ `sint fib` for a `mix'` specializing the Haskell-like language is the following one.

$$\begin{array}{l} \mathbf{fix}\,(\lambda v.\lambda v'.\,\mathbf{if}\ \ (v' < 2) \\ \qquad\qquad\quad \mathbf{then}\ 1 \\ \qquad\qquad\quad \mathbf{else}\ v@\,(v'-1)\ +\ v@\,(v'-2)) \end{array}$$

$$
\begin{aligned}
&\mathit{fib} = \text{“}\mathbf{fix}\ (\lambda \mathit{fib}.\lambda x.\\
&\qquad\qquad\quad \mathbf{if}\ \ (x < 2)\\
&\qquad\qquad\quad\ \mathbf{then}\ 1\\
&\qquad\qquad\quad\ \mathbf{else}\ \mathit{fib}@\,(x-1)\ +\ \mathit{fib}@\,(x-2))\text{”}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Fix}\ (\mathit{Lam}\ \text{“fib”}\ (\mathit{Lam}\ \text{“x”}\\
&\quad (\mathit{If}\ (\mathit{App}\ (\mathit{App}\ (\mathit{Const}\ (<))\ (\mathit{Var}\ \text{“x”}))\ \ (\mathit{Const}\ 2))\\
&\qquad (\mathit{Const}\ 1)\\
&\qquad (\mathit{App}\ (\mathit{App}\ (\mathit{Const}\ (+))\\
&\qquad\ \ (\mathit{App}\,(\mathit{Var}\ \text{“fib”})\ (\mathit{App}\ (\mathit{App}\\
&\qquad\qquad (\mathit{Const}\ (-))\ (\mathit{Var}\ \text{“x”}))\ (\mathit{Const}\ 1))))\\
&\qquad\ \ (\mathit{App}\,(\mathit{Var}\ \text{“fib”})\ (\mathit{App}\ (\mathit{App}\\
&\qquad\qquad (\mathit{Const}\ (-))\ (\mathit{Var}\ \text{“x”}))\ (\mathit{Const}\ 2)))))))
\end{aligned}
$$

Figure 2.5: “Sugared” `fib`, both as text and encoded as a datatype.

We also present a simpler example for the purpose of comparison: a function that applies its argument twice to the number 0.

$$\mathit{ex} = \text{“}(\lambda f.\,f@\,(f@\,0))\text{”}$$

$$\mathit{Lam}\ \text{“f”}\ (\mathit{App}\ \ (\mathit{Var}\ \text{“f”})\ \ (\mathit{App}\ \ (\mathit{Var}\ \text{“f”})\ \ (\mathit{Const}\ \ 0))$$

The result of $[\![\texttt{mix'}]\!]_L$ `sint ex` is

$$(\lambda v.\ v@\,(v@\,0))$$

and again we can see that the result is structurally equal to the original object program, and so, optimal.

But what happens when you try to use a typed language instead?

2.4 Inherited Limit: Types

Consider a version of `sint` written in a typed language, as presented in Figure 2.6 As `sint` does not manage types, this version is no more a self-interpreter, so we have changed its name to *eval*. Self-interpretation is much more difficult to achieve in the presence of types, and we do not attempt to do it.

There are some differences between this typed version, and the untyped *sint* presented in the previous section.

- The order of parameters is changed: Gomard and Jones [1991] put `exp` as the first parameter because the partial evaluator assumed that the first parameter is the static one, but this one is preferred.

$$
\begin{array}{l}
\textbf{data}\ \mathit{LExp}^S = \mathit{Var}\ \mathit{Char}^S \mid \mathit{Const}\ \mathit{Int}^S \\
\qquad\qquad \mid\ \mathit{Lam}\ \mathit{Char}^S\ \mathit{LExp}^S \mid \mathit{App}\ \mathit{LExp}^S\ \mathit{LExp}^S \\
\textbf{data}\ \mathit{Value}^D = \mathit{Num}\ \mathit{Int}^D \mid \mathit{Fun}\ (\mathit{Value}^D \to^D \mathit{Value}^D) \mid \mathit{Wrong} \\
\textbf{let}^S\ \mathit{bind} = \lambda^S x.\lambda^S v.\lambda^S \mathit{env}. \\
\qquad\qquad \lambda^S y.\textbf{if}^S\ x == y\ \textbf{then}\ v\ \textbf{else}\ \mathit{env}\ @^S\ y \\
\ \textbf{in}\ \textbf{let}^S\ \mathit{preeval} = \\
\qquad\quad \textbf{fix}^S\ (\lambda^S \mathit{eval}.\lambda^S \mathit{env}.\lambda^S \mathit{expr}. \\
\qquad\qquad\quad \textbf{case}^S\ \mathit{expr}\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Var}\ x \quad\ \to \mathit{env}\ @^S\ x \\
\qquad\qquad\qquad \mathit{Const}\ n \ \to \mathit{Num}^D\ (\textbf{lift}\ n) \\
\qquad\qquad\qquad \mathit{Lam}\ x\ e \ \to \mathit{Fun}^D\ (\lambda^D v. \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}^S\ \mathit{env}' = \mathit{bind}\ @^S\ x\ @^S\ v\ @^S\ \mathit{env} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \mathit{eval}\ @^S\ \mathit{env}'\ @^S\ e) \\
\qquad\qquad\qquad \mathit{App}\ e_1\ e_2 \to \textbf{case}^D\ (\mathit{eval}\ @^S\ \mathit{env}\ @^S\ e_1)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Fun}\ f \to\ f\ @^D\ (\mathit{eval}\ @^S\ \mathit{env}\ @^S\ e_2)) \\
\quad \textbf{in}\ \textbf{let}^S\ \mathit{eval} = \mathit{preeval}\ @^S\ (\lambda^S x.\mathit{Wrong}^D) \\
\qquad \textbf{in}\ \langle \ldots \rangle
\end{array}
$$

Figure 2.6: The interpreter for lambda-calculus written in a typed language.

- Type tags are added to the result of eval, corresponding to the data declarations. Because of types, there is no need for the branch corresponding to a syntax error — that kind of error is detected by the type system. Additionally, the branch for application has to use a case construct to pattern match the result of evaluating the first argument.
- Primitive operations can be no longer treated in the same way as before, because their types have to be taken into account.
- To keep the example simple, we have removed if-then-else, recursion, and primitive operations (which can be easily added again in a full example).
- We also work with texts of programs, rather than their encodings — although those encodings are needed for specialization, they can be produced by a suitable parser.

It is important to notice that the constructors for the type *Value* must be marked as dynamic, because the type of the residual program must be the same as that of the source — the specialization mechanism used by partial evaluation is reduction, and in a (good) typed language, reduction does not change types. This also forces us to mark as dynamic the `case` construct in the branch for application. But this is an important restriction. Consider again the expression $ex = (\lambda f.f@(f@0))$; the partial evaluation of *eval* with respect to `ex` is

$$
\begin{array}{l}
\mathit{Fun}\ (\lambda v.\ \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad \mathit{Fun}\ f \to f\ (\textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Fun}\ f\ \to\ f@\ (\mathit{Const}\ 0)))
\end{array}
$$

The result is no longer optimal! The tags and their corresponding untag operations appear in the residual program, and thus it is not essentially the same as the original one.

The problem we have encountered — type tags appearing in the residual program because of some characteristic of the specializer — is a particular instance of a more general phenomenon. Whenever a feature of the source program imposes a restriction on the occurrences of the same feature in the residual program, we say that we have encountered an *inherited limit*: it is a limitation on the form of obtainable residual programs that is imposed by the source program and the specialization method. The presence of an inherited limit shows a weakness of the specialization method: the particular feature imposing the limit is treated in a poor way by the specializer. So, it is very important that we detect inherited limits and try to remove them. Mogensen has argued that the historical development of program specialization can be viewed as successive achievements in removing specific inherited limits [Mogensen, 1996]; he suggests that this principle can be used as a guideline for future development.

The case of the inherited limit of types presented here is an instance of how the guideline of inherited limits has been effectively used: several ways to solve this particular limit have been proposed. Some include the use of type directed methods for partial evaluation, such as Type Directed Partial Evaluation [Danvy, 1996] or Tag Elimination [Taha and Makholm, 2000] (which are discussed in Section 13.2.1 and Section 13.2.2, respectively), or the use of a system with dependent types [Thiemann, 1999c] or additional techniques in partial evaluation, such as first-class polyvariance and co-arity raising [Thiemann, 2000a] (both discussed in Section 13.1). Another one is Type Specialization, the subject of the rest of this work.

Chapter 3

Type Specialization

The person who takes the banal and ordinary and illuminates it in a new way can terrify. We do not want our ideas changed. We feel threatened by such demands. ≪I already know the important things!≫ *we say. Then Changer comes and throws our old ideas away.*
The Zensufi Master

Chapterhouse: Dune
Frank Herbert

Type Specialization is an approach to program specialization introduced by John Hughes [1996b]. The main idea of type specialization is to specialize *both* the source program and its type to a residual program and residual type. To do this, instead of a generalized form of evaluation, type specialization uses a generalized form of type inference.

The key question behind type specialization is

> "How can we improve the static information provided by the type of an expression?"

An obvious first step is to have a more powerful type system. But we also want to *remove* the static information expressed by this new type from the code, to obtain a simpler and (hopefully) more efficient code. So, we work with two typed languages: the source language, in which we code the programs we want to specialize, and the residual language, which may contain additional constructs to express specialization features.

The rest of the chapter is organized as follows. In Section 3.1 we describe a basic source language and its type system; the particular feature of this language is that it is a two level language [Nielson and Nielson, 1992; Gomard and Jones, 1991], and the type system reflects that; we provide a system of rules to derive valid typing judgements and explain the main differences between annotations as used here in comparison to how they are used in partial evaluation. In Section 3.2 we describe the residual language corresponding to the source language presented earlier, and the rules performing the specialization from the source to the residual language; we discuss several simple examples of the different features in the language. This section is based almost completely on the original article where type specialization was presented [Hughes, 1996b]. We defer the presentation of a system to derive residual typing judgements to Chapter 6, because in Hughes' work the residual type system is implicit in the specialization rules. Finally, in Section 3.4, we present several extensions to the basic source language, the corresponding extensions in the residual language, and the rules used to specialize the new constructs; we also discuss some features of Hughes' work that we do not consider in the rest of this work.

3.1 Basic Source Language

When considering the source language for type specialization, we can ask ourselves about how to know which information should be moved into the type. In the best scenario, some process will analyze our program and mark those parts containing static information — a *binding time analyzer*. But, as shown by Hughes [1996b], no completely automatic binding time analysis can be performed for type specialization, because the only difference between an interpreter for a typed language and an interpreter for an untyped one is their binding times: by annotating an interpreter, we *decide* the static semantics of the object language — and we cannot expect a program to decide whether our object language is typed or untyped, can we? See the programs in Figures 4.1 and 4.2 and Examples 4.1 and 4.3, and the discussion in Chapter 4 for an example of such a situation.

We assume that the programmer has to add the corresponding binding time annotations by hand, and define the source language as a two level language: expressions with information we want to move from the code into the type will be marked *static* (with a $_^S$ superscript), and those we want to keep in the residual code will be marked *dynamic* (with a $_^D$ superscript). Annotation $_^S$ is interpreted as the requirement to remove an expression from the source program, by computing it and moving its result into the residual type, and annotation $_^D$ as the requirement to keep the expression in the residual code. Additionally, we want some extra features, such as casting a static computation into a dynamic one, or allowing a single expression to produce several different residual expressions in the residual code; these features can be obtained by some extra annotations (**lift**, **poly**, and **spec**), whose effect is explained with examples in the next section.

Definition 3.1. Let x denote a *source term variable* from a countably infinite set of variables, and let n denote an integer number. A *source term*, written using the symbol e, is an element of the language defined by the following grammar:

$$\begin{array}{rlll} e ::= & x & \mid n^D & \mid e +^D e \\ \mid & \mathbf{lift}\ e & \mid n^S & \mid e +^S e \\ \mid & \lambda^D x.e & \mid e \mathbin{@^D} e & \mid \mathbf{let}^D\ x = e\ \mathbf{in}\ e \\ \mid & (e, \ldots, e)^D & \mid \pi^D_{n,n}\, e & \\ \mid & \mathbf{poly}\ e & \mid \mathbf{spec}\ e & \end{array}$$

Note that this language includes finite tuples of the form $(e_1, \ldots, e_n)^D$ for every possible arity n. The projections $\pi^D_{1,2}\, e$ and $\pi^D_{2,2}\, e$ may be abbreviated $\mathbf{fst}^D\ e$ and $\mathbf{snd}^D\ e$, respectively.

Source types also reflect the static or dynamic nature of expressions — the types of constants, functions and operators are consistent with the types of arguments. For example, the constant 42^D has type Int^D and the constant 42^S has type Int^S, and a dynamic function can only be dynamically applied, that is, the $_^D$'s in the following expression correspond to each other: $(\lambda^D x.x) \mathbin{@^D} y$. Additionally, expressions annotated with **poly** will have a corresponding **poly** in their type.

Definition 3.2. A *source type*, written using the symbol τ, is an element of the language defined by the following grammar:

$$\tau ::= \mathit{Int}^{D} \mid \mathit{Int}^{S} \mid (\tau, \ldots, \tau)^{D} \mid \tau \rightarrow^{D} \tau \mid \mathbf{poly}\ \tau$$

This language has finite tuples of the form $(\tau_1, \ldots, \tau_n)^{D}$ for every possible arity n.

This language is a small subset of the language of the type specializer from [Hughes, 1996b], but contains enough constructs to illustrate the basic notions. We expand it to the full language in Section 3.4. One thing should be remarked: the interpretation for types is coinductive, that is, as greatest fixpoint of some generating function (values of types are both finite and infinite valid combinations of constructors and values). This is important for the way they are used during specialization. We discuss this feature in Section 3.4.7.

The source language is *simply typed*: source terms may be given a monomorphic type (that is, if they receive a type, then it is monomorphic), according to a typing system we call ST — it is given below. A source term that cannot be given a type by the ST system is considered to be invalid. Source type inference is straightforward, but with a major difference from partial evaluation: no restrictions are imposed on annotations other than consistency formation ones — i.e. introduction and elimination constructs must have the same annotation — and thus constraints that a BTA uses to infer a "best" type do not exist — see Example 3.9-2. It is important to keep in mind that annotating a program involves taking decisions about the meaning of what we are specializing; so, annotations need to be very flexible.

Definition 3.3. A *source typing assignment*, written using the symbol Γ_{ST}, is a (finite) list of *source typing statements* of the form $x : \tau$, where no variable x appears more than once.

Judgements in the ST system have the form $\Gamma_{\mathrm{ST}} \vdash_{\mathrm{ST}} e : \tau$ which can be read as "if the free variables of e have types as indicated in Γ_{ST}, then expression e has type τ". We use the standard notation $\vdash_{\mathrm{ST}} e : \tau$ when Γ_{ST} is the empty list. The rules for ST are given in Figure 3.1.

The only non-standard rules are (o-lift), (o-poly), and (o-spec). The first one captures the notion that **lift** is a coercion from static integers to dynamic ones. The second one permits to annotate the type of a polyvariant expression as polyvariant. And the last one eliminates a polyvariant expression (known because the type is annotated as polyvariant).

The only surprising thing about the typing rules for the source language is that they are so unsurprising. Other authors using two-level languages restrict the ways in which static and dynamic type formers are mixed — for example, Nielson and Nielson [1992] do so by distinguishing *compile-time* from *run-time* . In contrast, we can allow completely free type formation. Type-based binding-time analysis, such as the one presented by Henglein [1991], must usually handle 'dependency constraints' which force the residual type of a conditional expression to be dynamic if the condition is. In contrast, we can allow static constructs under dynamic ones. Our 'binding time-checker' is simply an ordinary type-checker (note that we use a *checker* and not an *analyser*!).

$$\text{(ST-VAR)}\quad \frac{x : \tau \in \Gamma_{\text{ST}}}{\Gamma_{\text{ST}} \vdash_{\text{ST}} x : \tau}$$

$$\text{(ST-DINT)}\quad \Gamma_{\text{ST}} \vdash_{\text{ST}} n^D : Int^D$$

$$\text{(ST-D+)}\quad \frac{(\Gamma_{\text{ST}} \vdash_{\text{ST}} e_i : Int^D)_{i=1,2}}{\Gamma_{\text{ST}} \vdash_{\text{ST}} e_1 +^D e_2 : Int^D}$$

$$\text{(ST-LIFT)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e : Int^S}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \textbf{lift}\ e : Int^D}$$

$$\text{(ST-SINT)}\quad \Gamma_{\text{ST}} \vdash_{\text{ST}} n^S : Int^S$$

$$\text{(ST-S+)}\quad \frac{(\Gamma_{\text{ST}} \vdash_{\text{ST}} e_i : Int^S)_{i=1,2}}{\Gamma_{\text{ST}} \vdash_{\text{ST}} e_1 +^S e_2 : Int^S}$$

$$\text{(ST-DTUPLE)}\quad \frac{(\Gamma_{\text{ST}} \vdash_{\text{ST}} e_i : \tau_i)_{i=1,..,n}}{\Gamma_{\text{ST}} \vdash_{\text{ST}} (e_1, \ldots, e_n)^D : (\tau_1, \ldots, \tau_n)^D}$$

$$\text{(ST-DPRJ)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e : (\tau_1, \ldots, \tau_n)^D}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \pi^D_{i,n}\, e : \tau_i}$$

$$\text{(ST-DLAM)}\quad \frac{\Gamma_{\text{ST}}, x : \tau_2 \vdash_{\text{ST}} e : \tau_1}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \lambda^D x.e : \tau_2 \rightarrow^D \tau_1}$$

$$\text{(ST-DAPP)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e_1 : \tau_2 \rightarrow^D \tau_1 \qquad \Gamma_{\text{ST}} \vdash_{\text{ST}} e_2 : \tau_2}{\Gamma_{\text{ST}} \vdash_{\text{ST}} e_1 @^D e_2 : \tau_1}$$

$$\text{(ST-DLET)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e_2 : \tau_2 \qquad \Gamma_{\text{ST}}, x : \tau_2 \vdash_{\text{ST}} e_1 : \tau_1}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \textbf{let}^D\ x = e_2\ \textbf{in}\ e_1 : \tau_1}$$

$$\text{(ST-POLY)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e : \tau}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \textbf{poly}\ e : \textbf{poly}\ \tau}$$

$$\text{(ST-SPEC)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e : \textbf{poly}\ \tau}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \textbf{spec}\ e : \tau}$$

Figure 3.1: Typing rules for the source language.

The next example illustrates the absence of restrictions other than consistency formation ones. This lack of restrictions is possible because of the enhanced propagation of information obtained by using types.

EXAMPLE 3.4. Observe that only consistency formation restrictions are imposed, i.e. static (resp. dynamic) integers can only be statically (resp. dynamic) added (items 1 and 2), dynamic functions can only be dynamically applied (item 3), and polyvariant expressions need to be **spec**'ed before they can be used (item 4). Also observe that a dynamic function can have static and/or polyvariant arguments (item 5), showing that there are no restrictions such as those needed in partial evaluation, and that polyvariance is first class (items 5 and 6).

1. $\vdash_{\text{ST}} \lambda^D x.x +^S 1^S \ : Int^S \rightarrow^D Int^S$
2. $\vdash_{\text{ST}} \lambda^D x.\textbf{lift}\ x +^D 1^D \ : Int^S \rightarrow^D Int^D$
3. $\vdash_{\text{ST}} \textbf{let}^D\ f = \lambda^D x.x +^S 1^S \ \textbf{in}\ f\ @^D\ 2^S \ : Int^S$
4. $\vdash_{\text{ST}} \textbf{let}^D\ f = \textbf{poly}\ (\lambda^D x.x +^S 1^S)$
   $\textbf{in}\ (\textbf{spec}\ f\ @^D\ 2^S, \textbf{spec}\ f\ @^D\ 3^S)^D$
   $: (Int^S, Int^S)^D$
5. $\vdash_{\text{ST}} \lambda^D f.\textbf{spec}\ f\ @^D\ 2^S \ : \textbf{poly}\ (Int^S \rightarrow^D Int^S) \rightarrow^D Int^S$
6. $\vdash_{\text{ST}} (\lambda^D f.\textbf{spec}\ f\ @^D\ 2^S)\ @^D\ \textbf{poly}\ (\lambda^D x.x +^S 1^S)$
   $: Int^S$

You may have noticed that in the first items, the addition under the lambda can actually be resolved by the specializer (because its arguments can be both known), so it seems preferable to annotate it static, as in item 1, rather than dynamic, as in item 2. However, we want to keep the possibility to choose between both annotations — if we restrict the possibility to annotate it dynamic, it implies that we restrict the range of residual programs that can be produced, and then our framework may not be able to produce arbitrary programs.

The freedom in choosing the way to annotate programs is important to allow flexibility in the generation of programs; it is the programmer's responsibility to control the annotations (perhaps with the help of a binding time assistant — see Chapter 14, Section 14.4).

3.2 Residual Language

The residual language has constructs and types corresponding to all the dynamic constructs and types in the source language, plus additional ones used to express the result of specializing static constructs. In the original formulation, these additional constructs are, in the term language, the dummy constant ($\bullet$) and tuples and projections used for polyvariance, and, in the type language, singleton types (or *one-point* types, e.g. $\hat{42}$) and tuple types.

DEFINITION 3.5. Let x' denote a *residual term variable* from a countably infinite set of variables. A *residual term*, written using the symbol e', is an element of the language defined by the following grammar:

$$\begin{array}{rcllll} e' & ::= & x' & \mid n & \mid e' + e' & \mid \bullet \\ & \mid & \lambda x'.e' & \mid e'@e' & \mid \mathbf{let}\ x' = e'\ \mathbf{in}\ e' & \\ & \mid & (e'_1, \ldots, e'_n) & & \mid \pi_{n,n}\, e' & \end{array}$$

As in the source language, this language has finite tuples of the form $(e'_1, \ldots, e'_n)$ for every possible arity n, and $\pi_{1,2}\, e'$ and $\pi_{2,2}\, e'$ may be abbreviated $\mathbf{fst}\ e'$ and $\mathbf{snd}\ e'$ respectively.

The expression $\bullet$ corresponds to the residual of static numbers, the numbers n to the residual of dynamic numbers, lambda abstraction and application and the let construct are the residual of the corresponding dynamic ones, and finally, tuples and projections correspond to both the residual of tuples and the residual of polyvariant expressions and their specializations. It is important to mention that Hughes makes no distinction between static and dynamic tuples, so both the residual of dynamic tuples and the tuples introduced by polyvariance will be eliminated in a postprocessing phase called *arity raising* — see Section 3.3.

Residual types reflect the definition of source types.

DEFINITION 3.6. A *residual type*, written using the symbol τ', is an element of the language defined by the grammar:

$$\tau' ::= \mathit{Int} \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau')$$

This language has finite tuples of the form $(\tau'_1, \ldots, \tau'_n)$ for every possible arity n.

The novel feature of this language is the use of an infinite number of one-point types — the one-point type $\hat{n}$ being the residual type corresponding to some static integer whose value is known to be n. This feature makes this language more powerful than traditional ones. It may seem at first glance that one-point types are subtypes of the type *Int*, but this is not the case, as the information they carry is different: the fact that an expression e' has type *Int* means that if the computation of e' terminates, it will produce some number; but the fact that it has type $\hat{n}$ means that its value, $\bullet$, represents the number n without the need to perform any computation.

To express the result of the specialization procedure, Hughes introduced a new kind of judgement and a system of rules to infer valid judgements. These judgements, like typing judgements in the source language, make use of assignments to determine the specialization of free variables.

DEFINITION 3.7. A *specialization assignment*, written using the symbol Γ, is a (finite) list of *specialization statements* of the form $x : \tau \hookrightarrow e' : \tau'$, where no source variable appears more than once.

We write $\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$ which can be read as "if the free variables in e specialize to the expressions indicated in Γ, then source expression e of source type

$$\text{(O-VAR)}\quad \frac{x : \tau \hookrightarrow e' : \tau' \in \Gamma}{\Gamma \vdash x : \tau \hookrightarrow e' : \tau'}$$

$$\text{(O-DINT)}\quad \Gamma \vdash n^{D} : Int^{D} \hookrightarrow n : Int$$

$$\text{(O-D+)}\quad \frac{(\Gamma \vdash e_i : Int^{D} \hookrightarrow e'_i : Int)_{i=1,2}}{\Gamma \vdash e_1 +^{D} e_2 : Int^{D} \hookrightarrow e'_1 + e'_2 : Int}$$

$$\text{(O-LIFT)}\quad \frac{\Gamma \vdash e : Int^{S} \hookrightarrow e' : \hat{n}}{\Gamma \vdash \mathbf{lift}\ e : Int^{D} \hookrightarrow n : Int}$$

$$\text{(O-SINT)}\quad \Gamma \vdash n^{S} : Int^{S} \hookrightarrow \bullet : \hat{n}$$

$$\text{(O-S+)}\quad \frac{(\Gamma \vdash e_i : Int^{S} \hookrightarrow e'_i : \hat{n}_i)_{i=1,2}}{\Gamma \vdash e_1 +^{S} e_2 : Int^{S} \hookrightarrow \bullet : \hat{n}}\ (n_1+n_2=n)$$

$$\text{(O-DTUPLE)}\quad \frac{(\Gamma \vdash e_i : \tau_i \hookrightarrow e'_i : \tau'_i)_{i=1,..,n}}{\Gamma \vdash (e_1,\ldots,e_n)^{D} : (\tau_1,\ldots,\tau_n)^{D} \hookrightarrow (e'_1,\ldots,e'_n) : (\tau'_1,\ldots,\tau'_n)}$$

$$\text{(O-DPRJ)}\quad \frac{\Gamma \vdash e : (\tau_1,\ldots,\tau_n)^{D} \hookrightarrow e' : (\tau'_1,\ldots,\tau'_n)}{\Gamma \vdash \pi^{D}_{i,n}\, e : \tau_i \hookrightarrow \pi_{i,n}\, e' : \tau'_i}$$

$$\text{(O-DLAM)}\quad \frac{\Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash e : \tau_1 \hookrightarrow e' : \tau'_1}{\Gamma \vdash \lambda^{D} x.e : \tau_2 \rightarrow^{D} \tau_1 \hookrightarrow \lambda x'.e' : \tau'_2 \rightarrow \tau'_1}\ (x'\ \textit{fresh})$$

$$\text{(O-DAPP)}\quad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow^{D} \tau_1 \hookrightarrow e'_1 : \tau'_2 \rightarrow \tau'_1 \qquad \Gamma \vdash e_2 : \tau_2 \hookrightarrow e'_2 : \tau'_2}{\Gamma \vdash e_1 \mathbin{@^{D}} e_2 : \tau_1 \hookrightarrow e'_1 @ e'_2 : \tau'_1}$$

$$\text{(O-DLET)}\quad \frac{\Gamma \vdash e_2 : \tau_2 \hookrightarrow e'_2 : \tau'_2 \qquad \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash e_1 : \tau_1 \hookrightarrow e'_1 : \tau'_1}{\Gamma \vdash \mathbf{let}^{D}\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \hookrightarrow \mathbf{let}\ x' = e'_2\ \mathbf{in}\ e'_1 : \tau'_1}\ (x'\ \textit{fresh})$$

$$\text{(O-POLY)}\quad \frac{(\forall i)(\exists e'\tau')(\Gamma \vdash e : \tau \hookrightarrow e' : \tau' \quad \vec{\tau}'!i = \tau' \quad \vec{e}'!i = e')}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow \vec{e}' : \vec{\tau}'}$$

$$\text{(O-SPEC)}\quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : \vec{\tau}' \qquad \vec{\tau}'!i = \tau'}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_i e' : \tau'}$$

Figure 3.2: Type specialization rules from the original formulation.

τ specializes to residual expression e' and residual type τ'". As usual, we use the form $\vdash e : \tau \hookrightarrow e' : \tau'$ when Γ is empty. The original rules for type specialization are presented in Figure 3.2. In those rules, $\vec{\tau}'$ stands for a tuple of residual types $(\tau'_1, \ldots, \tau'_n)$, and $\vec{\tau}'!i$ for its i-th component, and similarly for $\vec{e}'$. Observe that there is a correspondence between specialization rules and source typing rules — that is, there are the same number, corresponding to each of the source language constructs, and they have a similar structure. This changes when if-then-else or case constructs are introduced, because the typing rules type-check all the branches, but the original form of type specialization specializes only the one that is necessary; we return to this issue in Section 3.4.

EXAMPLE 3.8. Observe that every expression annotated as dynamic appears in the residual term (in fact, we have that a fully dynamic expression, that is one in which every annotation is ^{D}, specializes to a copy of itself with the annotations removed).

1. $\vdash 42^D : Int^D \hookrightarrow 42 : Int$
2. $\vdash 42^S : Int^S \hookrightarrow \bullet : \hat{42}$
3. $\vdash (2^D +^D 1^D) +^D 1^D : Int^D \hookrightarrow (2+1)+1 : Int$
4. $\vdash (2^S +^S 1^S) +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
5. $\vdash \textbf{lift}\ (2^S +^S 1^S) +^D 1^D : Int^D \hookrightarrow 3+1 : Int$

Also observe in item 5 how the use of **lift** allows us to cast a static integer into a dynamic one, thus inserting the result of the static computation back into the residual term.

To derive the judgements in Example 3.8, the rules (O-DINT), (O-D+), (O-SINT), (O-S+), and (O-LIFT)[1] are used. It is important to note that both (O-S+) and (O-LIFT) used to derive specialization of static operations and lifts, force the choice of a suitable one-point type for the subexpressions, n in the former, and n_1, n_2 in the latter — this is not (always) syntax directed (for example, when a variable appears as a subexpression).

EXAMPLE 3.9. Assumptions provide the information for the specialization of free variables, which allows the specialization of functions.

1. $x : Int^S \hookrightarrow \bullet : \hat{3} \vdash x +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
2. $\vdash (\lambda^D x.x +^S 1^S) @^D (2^S +^S 1^S) : Int^S \hookrightarrow (\lambda x'.\bullet)@\bullet : \hat{4}$
3. $\vdash (\lambda^D x.\textbf{lift}\ x +^D 1^D) @^D (2^S +^S 1^S) : Int^D \hookrightarrow (\lambda x'.3+1)@\bullet : Int$

The function appearing in item 3 will be used in additional examples in this and the next chapters, and so, to keep details in those cases simple, we have chosen to mark the addition as dynamic. As we have mentioned, this is possible because of the enhanced propagation of information, and desirable to have flexibility in the range of residual programs that can be generated.

[1]The O- prefix distinguishes the Original rules from the ones presented later.

Observe in Example 3.9-2 and 3.9-3 that a dynamic function is allowed to have a static argument, because residual type inference will provide the value needed for the computation; for example, in Example 3.9-2, the residual function $(\lambda x'.\bullet)$ has residual type $\hat{3} \rightarrow \hat{4}$ thus providing information about x' to the function body even when the function is not reduced. It is not possible for dynamic functions to have static arguments with the partial evaluation approach, as the only way to propagate information is by reduction, and a dynamic function is never reduced. The examples in Chapter 4 show the additional expressiveness that this lack of restrictions implies.

The rules to derive judgements involving free variables and functions are (O-VAR), (O-DLAM), and (O-DAPP). Observe that in the rule (O-DAPP) there is the expected flow of information, because in a specialization algorithm the antecedent of the residual function type should be unified with the residual type of the argument. Also observe that the residual type τ_2' assigned to the residual of the λ-bound variable in rule (O-DLAM) is not restricted in any way, allowing *any* type to be chosen at this stage (this choice will be further restricted by application, as noted above).

To see how type specialization works with higher order functions, we present the following examples.

EXAMPLE 3.10. Observe how the static information can be moved from the body of the higher-order function to the place where **lift** will reinsert it into the residual term.

$$\vdash (\lambda^D f. f \mathbin{@^D} 42^S) \mathbin{@^D} (\lambda^D x.\mathbf{lift}\ x +^D 1^D) : Int^D \hookrightarrow (\lambda f. f @ \bullet) @ (\lambda x. 42 + 1) : Int$$

This is possible because the residual type of f is $\hat{42} \rightarrow Int$.

EXAMPLE 3.11. Observe again that the value of f's function is propagated by using the residual type of f; in this example, also the result of f is propagated, and reinserted in the residual code by the **lift**.

$$\vdash (\lambda^D f.\mathbf{lift}\ (f \mathbin{@^D} 42^S)) \mathbin{@^D} (\lambda^D x. x +^S 1^S) : Int^D \hookrightarrow (\lambda f. 43) @ (\lambda x.\bullet) : Int$$

In this case, the residual type of f is $\hat{42} \rightarrow \hat{43}$.

One important feature of type specialization is that there exist correctly annotated terms that cannot be specialized. Consider

$$\begin{array}{l}\mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x +^D 1^D \\ \ \mathbf{in}\ (f \mathbin{@^D} 42^S, f \mathbin{@^D} 17^S)^D\ : (Int^D, Int^D)^D.\end{array}$$

What should the specialization for this term be? As we have seen in Example 3.9-3, the body of the function is specialized according to the parameter, but f has two *different* parameters! We discuss why this is an important feature in Examples 4.3 and 4.5 of Chapter 4.

But what to do if we do not want an error in this case? The solution is to allow f to specialize in more than one way in the same program. *Polyvariance* is the ability of an expression to specialize to more than one residual expression. The use of polyvariance is indicated in the source language by using the annotations **poly** and **spec**, the former to produce a polyvariant expression, and the latter to choose the corresponding specialization of it.

EXAMPLE 3.12. Observe the use of **poly** in the definition of f (and how that annotation produces a tuple for the definition of f' in each of the residual codes), and the use of **spec** in every application of f to an argument (and how that produces the corresponding projections).

1. $\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)$
   $\mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D : (Int^D, Int^D)^D \hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.42 + 1, \lambda x'.17 + 1)$
   $\mathbf{in}\ (\mathbf{fst}\ f'@\bullet, \mathbf{snd}\ f'@\bullet) : (Int, Int)$

2. $\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)$
   $\mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D : (Int^D, Int^D)^D \hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.17 + 1, \lambda x'.55 + 1, \lambda x'.42 + 1)$
   $\mathbf{in}\ (\pi_{3,3}\ f'@\bullet, \pi_{1,3}\ f'@\bullet) : (Int, Int)$

The size and order of the residual tuple is arbitrary, provided that it has at least two elements ($\lambda x'.42 + 1$ and $\lambda x'.17 + 1$), and that the projections select the appropriate element, as can be seen when contrasting item 2 to item 1.

The rules (O-POLY) and (O-SPEC) used to specialize polyvariance reflect the freedom to choose the size and order of the residual tuples. This freedom is the reason why the rules are not syntax directed, because it is the context where the expression appears which imposes the minimal restrictions the tuples must satisfy.

3.3 Arity Raising

The residual programs produced by the described specialization process contain many subexpressions without computational content (that is, that are $\bullet$ or can only take that value) — see Examples 3.9 and 3.10 — and also big tuples resulting from polyvariant expressions — see Example 3.12. To produce efficient residual programs that do not work with $\bullet$'s or construct unnecessary tuples, we use a postprocessing phase called *arity raising*, that removes subexpressions without computational content, and splits tuples into their components, increasing the arity of functions. This phase works by inspecting the types of subexpressions, detecting tuples and types whose only value is void, and replacing them by equivalent types. As a consequence, arity raising can only be applied as a postprocessing phase, because the residual types will only be completely known at the end of the specialization phase. It is important to note that the language produced by arity raising is not the same as the source or residual ones — in particular, it has a let that can have multiple definitions.

In the original paper [Hughes, 1996b], this phase was called *void erasure*, because it only removes $\bullet$'s; but as voids are similar to empty tuples, its removal can be obtained by a general treatment of tuples. Arity raising for type specialization was introduced by Hughes [1998a]. The original idea of arity raising is due to Romanenko [1990], and it was further developed by Hannan and Hicks [1998].

To illustrate what arity raising does, consider the following examples.

EXAMPLE 3.13. Observe that the second component of the argument in residual function f can only take the value $\bullet$; to see that, look at the residual type of f, $(\mathit{Int}, \hat{3}) \to \mathit{Int}$.

$$\begin{array}{l} \vdash \mathbf{let}^{D}\ f = \lambda^{D} p.\mathbf{fst}^{D}\ p +^{D} \mathbf{lift}\ (\mathbf{snd}^{D}\ p) \\ \quad \mathbf{in}\ f\ @^{D}\ (2^{D}, 3^{S})^{D} \\ \ \ : \mathit{Int}^{D} \\ \qquad \hookrightarrow \mathbf{let}\ f = \lambda p.\mathbf{fst}\ p + 3 \\ \qquad\qquad \mathbf{in}\ f @ (2, \bullet) \\ \qquad : \mathit{Int} \end{array}$$

The arity raiser will remove the second component of the tuple, and all the corresponding references, producing

$$\mathbf{let}\ f = \lambda p_1.p_1 + 3\ \mathbf{in}\ f @ 2 : \mathit{Int}$$

EXAMPLE 3.14. Observe the tuples expressing the residual of the polyvariant expressions, and how the arity raiser removes them in favour of several definitions/arguments. Also observe that the residual of x is removed as well.

$$\begin{array}{l} \vdash \mathbf{let}^{D}\ f = \mathbf{poly}\ (\lambda^{D} g.(\mathbf{spec}\ g\ @^{D}\ 2^{S}) +^{D} (\mathbf{spec}\ g\ @^{D}\ 3^{S})) \\ \quad \mathbf{in}\ \mathbf{spec}\ f\ @^{D}\ \mathbf{poly}\ (\lambda^{D} x.\mathbf{lift}\ x) +^{D} \mathbf{spec}\ f\ @^{D}\ \mathbf{poly}\ (\lambda^{D} x.\mathbf{lift}\ (x +^{S} 2^{S})) \\ \ \ : \mathit{Int}^{D} \\ \qquad \hookrightarrow \mathbf{let}\ f = (\lambda g.(\mathbf{fst}\ g @ \bullet) + (\mathbf{snd}\ g @ \bullet), \lambda g.(\mathbf{fst}\ g @ \bullet) + (\mathbf{snd}\ g @ \bullet)) \\ \qquad\qquad \mathbf{in}\ \mathbf{fst}\ f @ (\lambda x.2, \lambda x.3) + \mathbf{snd}\ f @ (\lambda x.4, \lambda x.5) \\ \qquad : \mathit{Int} \end{array}$$

The result of arity raising the previous term is

$$\begin{array}{l} \mathbf{let}\ f_1 = \lambda g_1.\lambda g_2.g_1 + g_2 \\ \qquad f_2 = \lambda g_1.\lambda g_2.g_1 + g_2 \\ \ \mathbf{in}\ f_1 @ 2 @ 3 + f_2 @ 4 @ 5 \end{array}$$

Observe that all the tupling has been removed, and the final term is, thus, more efficient.

Arity raising is important because it removes the last traces of static computations, allowing optimal specialization. Inspired by Hughes' work, Thiemann [2000a] used a variation of this phase — dual to arity raising, thus named *co-arity raising* by him — in combination with first-class polyvariance to obtain optimal specialization for a typed interpreter of the lambda-calculus, thus showing the central role that arity raising plays in the elimination of the inherited limit of types.

3.4 Extensions to the Language

The source language considered in Section 3.1 is quite small, and only a limited number of programs can be written. To be able to specialize real programs, some additional constructs must be considered. In most cases the static version of the construct is the really interesting one, while the dynamic version is very simple; but there are some cases, such as dynamic datatypes and dynamic recursion, that pose their own problems.

Among the simplest things that we can add are characters and other base types; their treatment is essentially the same as that of numbers: dynamic versions just copy the values in the residual term, while static versions need the enriching of the residual language with suitable residual types to express the specialization of static constants (that is, additional one-point types). Additional binary operations on numbers (such as equality or multiplication), and operations on other base types follow the same treatment as addition.

Another simple extension is that of a static let: the only difference with the dynamic version is the degree of unfolding in the residual term, as can be seen comparing rule (O-DLET) with (O-SLET):

$$\text{(O-SLET)}\quad \frac{\Gamma \vdash e_2 : \tau_2 \hookrightarrow e_2' : \tau_2' \qquad \Gamma, x : \tau_2 \hookrightarrow e_2' : \tau_2' \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'}{\Gamma \vdash \mathbf{let}^S\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'}$$

Observe that, instead of introducing a new residual variable, we make e_2' the residual of x, thus producing the unfolding of the let.

We can also add booleans, static functions, datatypes, polyvariant sums, imperative features (in monadic form [Dussart *et al.*, 1997b]), etc. We present each of these extensions in a separate subsection.

3.4.1 Booleans

In the original paper [Hughes, 1996b], booleans were included as a particular case of a datatype. But as booleans are not a recursive type, we present them separately, because their treatment is simpler, and allow us to illustrate the techniques when reformulating the presentation. First we consider the dynamic version of booleans, and then the static one.

Dynamic booleans are almost as simple as dynamic numbers, with a small addition: both branches of the if-then-else construct are forced to have the *same* residual type, in order to be able to generate the residual if-then-else — see rule (O-DIF) below. This introduces another way of forcing two residuals to be the same (we have already seen that applying a monovariant function produces the same effect before Example 3.12).

The source and residual term languages are extended with straightforward constructs: True^D, False^D, $\mathbf{if}^D\, b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ in the source language, and *True*, *False*, $\mathbf{if}\ b\ \mathbf{then}\ e_1'\ \mathbf{else}\ e_2'$ in the residual one. In the type languages, Bool^D and *Bool* are added. Source typing rules are standard. The rules to specialize dynamic booleans are the following:

$$\text{(O-DTRUE)}\quad \Gamma \vdash \mathit{True}^D : \mathit{Bool}^D \hookrightarrow \mathit{True} : \mathit{Bool}$$

$$\text{(O-DFALSE)}\quad \Gamma \vdash \mathit{False}^D : \mathit{Bool}^D \hookrightarrow \mathit{False} : \mathit{Bool}$$

$$\text{(O-DIF)}\quad \frac{\Gamma \vdash e : \mathit{Bool}^D \hookrightarrow e' : \mathit{Bool} \qquad \Gamma \vdash e_1 : \tau \hookrightarrow e_1' : \tau' \qquad \Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'}{\Gamma \vdash \mathbf{if}^D\, e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau \hookrightarrow \mathbf{if}\ e'\ \mathbf{then}\ e_1'\ \mathbf{else}\ e_2' : \tau'}$$

Static information can be used under each of the branches of a dynamic if-then-else, in much the same way as under a dynamic λ.

$$\begin{array}{l}\vdash (\lambda^D x.\lambda^D b.\textbf{if}^D\, b \textbf{ then lift } (x +^S 1^S) \textbf{ else lift } x) @^D 42^S : \mathit{Int}^D \\ \quad \hookrightarrow (\lambda x.\lambda b.\textbf{if } b \textbf{ then } 43 \textbf{ else } 42)@\bullet : \mathit{Int}\end{array}$$

As a more interesting case, we can have a dynamic conditional with a static result. However, the residual type of both branches must be the same, producing a failure when this is not the case — see Example 3.15; this restriction can be relaxed by using polyvariant sums, as discussed in Section 3.4.6 — see Example 3.23.

EXAMPLE 3.15. We present two conditionals: one with the same static information on each branch, and the second one with different ones. In the first conditional, specialization is possible, giving

$$\begin{array}{l}\vdash \lambda^D b.\textbf{let}^D\ f = \lambda^D x.\textbf{if}^D\, b \\ \qquad\qquad\qquad\qquad \textbf{then } (2^S, \textbf{lift } x)^D \\ \qquad\qquad\qquad\qquad \textbf{else } (2^S, 51^D)^D \\ \qquad \textbf{in let}^D\ y = (f\ @^D 42^S) \textbf{ in snd}^D\ y +^D \textbf{lift } (\textbf{fst}^D\ y) \\ \quad : \mathit{Int}^D \\ \qquad \hookrightarrow \lambda b.\textbf{let } f = \lambda x.\ \textbf{if } b \\ \qquad\qquad\qquad\qquad\qquad \textbf{then } (\bullet, 42) \\ \qquad\qquad\qquad\qquad\qquad \textbf{else } (\bullet, 51) \\ \qquad\qquad \textbf{in let}^D\ y = (f@\bullet) \textbf{ in snd } y + 2 \\ \qquad\quad : \mathit{Int}\end{array}$$

However, if we change the static information in one of the branches, thus having different residual types for each of them, specialization is no longer possible.

$$\begin{array}{l}\lambda^D b.\textbf{let}^D\ f = \lambda^D x.\textbf{if}^D\, b \\ \qquad\qquad\qquad\qquad \textbf{then } (2^S, \textbf{lift } x)^D \\ \qquad\qquad\qquad\qquad \textbf{else } (3^S, 51^D)^D \\ \qquad \textbf{in let}^D\ y = (f\ @^D 42^S) \textbf{ in snd}^D\ y +^D \textbf{lift } (\textbf{fst}^D\ y) \\ : \mathit{Int}^D\end{array}$$

In this case, the error will be that $\hat{2}$ and $\hat{3}$ are not equal.

This variation of static information on each branch is truly useful when used in combination with static sum types and static functions — again, see Section 3.4.6.

For static booleans, True^S, False^S, and $\textbf{if}^S\, b \textbf{ then } e_1 \textbf{ else } e_2$ on terms, Bool^S on types, are added to the source language, and $\widehat{\mathit{True}}$ and $\widehat{\mathit{False}}$ are added to the residual types (the constant $\bullet$ is used for the residual of static boolean constants). Static booleans are treated similarly to static numbers, the main difference being that only one of the branches is actually specialized, while the other is simply ignored. This fact is realized by having two rules for static if-then-else:

$$\text{(O-STRUE)} \quad \Gamma \vdash \mathit{True}^S : \mathit{Bool}^S \hookrightarrow \bullet : \widehat{\mathit{True}}$$

$$\text{(O-SFALSE)} \quad \Gamma \vdash \mathit{False}^{S} : \mathit{Bool}^{S} \hookrightarrow \bullet : \hat{\mathit{False}}$$

$$\text{(O-SIF-TRUE)} \quad \frac{\Gamma \vdash e : \mathit{Bool}^{S} \hookrightarrow e' : \hat{\mathit{True}} \qquad \Gamma \vdash e_1 : \tau \hookrightarrow e'_1 : \tau'_1}{\Gamma \vdash \mathbf{if}^{S}\, e \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 : \tau \hookrightarrow e'_1 : \tau'_1}$$

$$\text{(O-SIF-FALSE)} \quad \frac{\Gamma \vdash e : \mathit{Bool}^{S} \hookrightarrow e' : \hat{\mathit{False}} \qquad \Gamma \vdash e_2 : \tau \hookrightarrow e'_2 : \tau'_2}{\Gamma \vdash \mathbf{if}^{S}\, e \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 : \tau \hookrightarrow e'_2 : \tau'_2}$$

As an example, we can consider a static if-then-else such that a specialization error is located in the branch which is ignored, and see that the result still can be obtained — the error will not be detected.

$$\begin{array}{l}\vdash (\lambda^{D} x.\lambda^{D} b.\mathbf{if}^{S}\, x ==^{S} 42^{S} \;\mathbf{then\ lift}\; x \;\mathbf{else\ lift}\; (\mathbf{if}^{D}\, b \;\mathbf{then}\; x \;\mathbf{else}\; 0^{S})) \;@^{D}\, 42^{S} : \mathit{Int}^{D} \\ \quad \hookrightarrow (\lambda x.\lambda b.42)@\bullet : \mathit{Int}\end{array}$$

If we change the argument of the function from 42^{S} to some other number different from zero, the specialization will fail with an error, because the residual type of x would not be equal to the residual type of 0^{S} (remember that the residual types of the two branches of a dynamic if-then-else must be the same!).

3.4.2 Dynamic recursion

Dynamic recursion is added in the source language by the construct $\mathbf{fix}^{D}\ e$ representing a fixpoint operator; no new type is required. The rule for typing a $\mathbf{fix}^{D}$ construct is the classic one. In the residual language, a corresponding $\mathbf{fix}\ e'$ is added, together with the rule to specialize it:

$$\text{(O-DFIX)} \quad \frac{\Gamma \vdash e : \tau \rightarrow^{D} \tau \hookrightarrow e' : \tau' \rightarrow \tau'}{\Gamma \vdash \mathbf{fix}^{D}\ e : \tau \hookrightarrow \mathbf{fix}\ e' : \tau'}$$

While this seems a minor addition, its combination with other features in the language, such as polyvariance, poses the most challenging problems. In addition to all the non-termination problems common to partial evaluation — see upcoming Example 3.17 — there is a problem when a recursive function is polyvariant: the recursive calls will be specializations of the polyvariant expression, and so knowing all the necessary elements for the residual tuple is very involved.

How often will dynamic recursion and polyvariance may appear together? The answer is: too often! In fact, this scenario is *the* prime motivation for introducing and using **poly**.

If the (dynamic) recursive function has static arguments, then making it monovariant forces it to be used only with one single value for each one of those arguments, even in recursive calls! And in that case, why those values are used as arguments at all? It is better considering them to be global variables. Let's see an example.

Example 3.16. We consider again the *power* function mentioned in Chapter 1, but with dynamic recursion (to obtain the same result as that presented in the introduction, we have to use static recursion — see Example 3.19 below).

The source expression is the following one.

$$
\begin{array}{l}
\mathbf{let}^{D}\ power = \mathbf{fix}^{D}\ (\lambda^{D} p.\mathbf{poly}\ (\lambda^{D} n.\lambda^{D} x.\mathbf{if}^{S}\ n ==^{S} 1^{S} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ x *^{D} \mathbf{spec}\ p\ @^{D} (n -^{S} 1)\ @^{D} x)) \\
\ \mathbf{in\ spec}\ power\ @^{D} 3^{S}
\end{array}
$$

Observe that *power* has the first argument static, and thus it has to be polyvariant (because the recursive call will be applied to a different number!). To make it polyvariant, the recursive argument p and the result of the **fix** body are annotated polyvariant (the former by using it inside a **spec**, and the latter with an explicit **poly**.)

The specialization of this example is the following one.

$$
\begin{array}{l}
\mathbf{let}\ power = \mathbf{fix}\ (\lambda p.(\lambda n.\lambda x.x * \pi_{2,3}\, p@\bullet @x, \\
\qquad\qquad\qquad\qquad \lambda n.\lambda x.x * \pi_{3,3}\, p@\bullet @x, \\
\qquad\qquad\qquad\qquad \lambda n.\lambda x.x)) \\
\ \mathbf{in}\ \pi_{1,3}\, power@\bullet
\end{array}
$$

After arity raising, it becomes the following (final) term.

$$
\begin{array}{l}
\mathbf{let}\ power_3 = \lambda x.x * power_2 @x, \\
\qquad power_2 = \lambda x.x * power_1 @x, \\
\qquad power_1 = \lambda x.x \\
\ \mathbf{in}\ power_3
\end{array}
$$

It can be observed that one function for every call was generated.

Non-termination is an issue, because there exist programs that cannot be specialized without an infinite number of variants. For example, consider the following case.

EXAMPLE 3.17. Observe that the variable x controlling the termination of the recursive function is dynamic; so, there is potentially an infinite number of static values for y, which implies that the specializer must attempt to generate a tuple with an infinite number of variants to express the residual of f.

$$
\begin{array}{l}
\mathbf{let}^{D}\ f = \mathbf{fix}^{D}\ \lambda^{D} f.\mathbf{poly}\ (\lambda^{D} x.\lambda^{D} y.\mathbf{if}^{D}\ x ==^{D} 0^{D} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ y \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ spec}\ f\ @^{D} (x -^{D} 1^{D})\ @^{D} (y +^{S} 1^{S})) \\
\ \mathbf{in\ spec}\ f\ @^{D} 3^{D}\ @^{D} 0^{S}
\end{array}
$$

Any specializer will loop when attempting to specialize this example.

However, there are several programs for which (finite) specializations exist, but that may cause the specializer to loop if it does not proceed with care. We leave the discussion of these problems for Section 4.4, where some problematic cases will be considered.

3.4.3 Static functions

If only dynamic functions can be defined in the source language, then the specialization process would be very weak because no function call will ever be unfolded. For that reason, every good specializer must provide static functions. The additions to the source language are simple: just lambda-abstractions, applications, and function types with static annotations. The rules for typing static functions are just copies of those for dynamic ones, but with the annotations changed; this is so simple because there are no restrictions on annotations, as we have explained.

The problem with static functions in this formulation of type specialization is that, as the specialization of an expression may depend on the context, and one expects the context to be different at every application (a static function that can only be applied to a unique element could not be considered a function at all!), then the body of the function cannot be specialized until the function is applied. The solution chosen by Hughes was to have residual types representing *closures* of a source function and its environment, so that the body can be specialized every time there is the need to unfold an application.

But there is yet another problem! The body of a static function is likely to contain free variables whose residuals may contain some dynamic information, so when unfolding the function, residual variables may escape their scopes. The solution to this is to make the residual term of a static function be a tuple of the residuals of the free variables of the function, which is very close to the solution to the same problem adopted in Similix [Bondorf, 1991]; in such a way, residual variables still appear in the same scope as in the source term, and they are accessed on unfoldings by using projections.

The rule for static λ reflects that:

$$\text{(O-SLAM)} \quad [z_i : \tau_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n} \vdash \lambda^S x.e : \tau_2 \rightarrow^S \tau_1 \hookrightarrow (e'_1, \ldots, e'_n) : \mathbf{clos} \ll [(z_i, \tau_i, \tau'_i)]_{i=1}^{n}, x, e \gg$$

Observe that the closure contains the source code of the function's body (e) and the name and source and residual type of each of the variables in the specialization assignment. As the only variables that are really needed are those that appear free in the body, a rule to restrict the variables of an assignment to those appearing in the term can be used; the rule is the following one:

$$\text{(O-WEAK)} \quad \frac{\Gamma_1, \Gamma_2 \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma_1, x : \tau_x \hookrightarrow e'_x : \tau'_x, \Gamma_2 \vdash e : \tau \hookrightarrow e' : \tau'} \quad (x \notin FV(e))$$

An efficient implementation should apply (O-WEAK) as much as possible.

It is important to remark that no specialization at all is performed to a static function when it is declared. This allows the body to contain specialization errors that will not be detected unless the function is applied. The application of a static function is the real place where the specialization process takes place.

$$\text{(O-SAPP)} \quad \frac{\Gamma \vdash e_1 : \tau_a \rightarrow^S \tau_r \hookrightarrow e'_1 : \mathbf{clos} \ll [(z_i, \tau_i, \tau'_i)]_{i=1}^{n}, x, e \gg \quad \Gamma \vdash e_2 : \tau_a \hookrightarrow e'_2 : \tau'_a \quad [z_i : \tau_i \hookrightarrow \pi_{i,n}\, e'_1 : \tau'_i]_{i=1}^{n}, \ldots}{\Gamma \vdash e_1 \mathbin{@^S} e_2 : \tau_r \hookrightarrow e' : \tau'_r}$$

Observe how the body of the static function (e) is specialized once for every application (in the third premise), with a different specialization assignment each time: the residual term and type of x is different. Also observe how the free variables in e are specialized to projections of the residual tuple; this guarantees that no variable escapes its scope.

An example shows what the residual program looks like after unfolding a static function with free variables.

EXAMPLE 3.18. Note that the variable x appears free in the body of the function.

$$\begin{array}{l}\vdash \textbf{let}^{D}\ f = \textbf{let}^{D}\ x = (5^{S}, 6^{D})^{D} \\ \qquad\qquad\quad \textbf{in}\ \lambda^{S}y.\textbf{lift}\ (\textbf{fst}^{D}\ x +^{S} y) +^{D} \textbf{snd}^{D}\ x \\ \quad \textbf{in}\ f\ @^{S}\ 2^{S} \\ \quad : \mathit{Int}^{D} \\ \quad \hookrightarrow \textbf{let}\ f = \textbf{let}\ x = (\bullet, 6) \\ \qquad\qquad\qquad \textbf{in}\ x \\ \qquad\quad \textbf{in}\ 7 + \textbf{snd}\ f \\ \qquad\quad : \mathit{Int}\end{array}$$

However, the residual of the static function keeps the scope for the residual of x; in the unfolded code, where x would have appeared, f is used (in general, it is a projection of f). The residual type of f is

$$\textbf{clos} \ll [(x, (\mathit{Int}^{S}, \mathit{Int}^{D})^{D}, (\hat{5}, \mathit{Int}))], y, \textbf{lift}\ (\textbf{fst}^{D}\ x +^{S} y) +^{D} \textbf{snd}^{D}\ x \gg$$

reflecting that x appears free.

After arity raising, the result will be

$$\begin{array}{l}\textbf{let}\ f = \textbf{let}\ x = 6\ \textbf{in}\ x \\ \ \textbf{in}\ 7 + f \\ \ \ : \mathit{Int}\end{array}$$

3.4.4 Static recursion

Static recursion is added in the source language by a fixpoint operator $\textbf{fix}^{S}\ e$, which is unfolded during the specialization process. The problem of unfolding recursion during specialization is non-termination: we need some way to control the termination of the unfolding because, if the specialization does not terminate, then it will either produce an infinite answer or none at all. A sensible restriction to ensure termination is to allow $\textbf{fix}^{S}$ to be used *only* to produce static functions, delaying the unfolding until an application of the function is performed; in this way, only non-terminating static functions will produce non-terminating specializations. This restriction is the same adopted in strict programming languages.

To represent the residual of a static fixpoint operator we need a new residual type capturing the fact that the name of the recursive function appears free in its body. It is similar to the static closures considered above, but with the addition of the name of the function:

$$\textbf{rec} \ll [(z_i, \tau_i, \tau'_i)]_{i=1}^{m}, f, x, e \gg$$

This new form of types will be the residual type of a source expression of the form $\mathbf{fix}^S\ (\lambda^S f.\lambda^S x.e)$, with free variables among the $\{z_i\}$s — observe, in particular, that is has the same information appearing in the static closure, with the addition of the recursive variable.

The rule to specialize a static recursive function works in two steps: it first specializes the argument of the $\mathbf{fix}^S$ construct to a function from a static function to itself, and then specializes the body of this function *without* binding the name of the recursive function (this will produce no problems, because if the specialized version of the recursive function is needed to obtain the specialized version of the recursive function, then no specialization can be done!). The result is a recursive closure built from the closure of the body and the name of the function:

$$\frac{\Gamma \vdash e_f : (\tau_a \to^S \tau_r) \to^S (\tau_a \to^S \tau_r) \hookrightarrow e'_f : \mathbf{clos} \ll [(z_i, \tau_{f_i}, \tau'_{f_i})]_{i=1}^n, x, e_b \gg \qquad \begin{array}{r}[z_i : \tau_{f_i} \hookrightarrow \pi_{i,n}\, e'_f : \tau'_{f_i}]_{i=1}^n \vdash \\ e_b : \tau_a \to^S \tau_r \hookrightarrow e'_b : \mathbf{clos}\end{array}}{\Gamma \vdash \mathbf{fix}^S\ e_f : \tau_a \to^S \tau_r \hookrightarrow e'_b : \mathbf{rec} \ll [(w_i, \tau_{b_i}, \tau'_{b_i})]_{i=1}^m, f, x, e_r \gg}\ \text{(O-SFIX)}$$

An interesting observation is that, as presented, the rule does not take into account the case when the e_f expression is itself the result of a fixpoint (that is, its residual e'_f is of type **rec** instead of **clos**). This is an oversight in Hughes' original work, but easily fixed by η-conversion.

The details can be better understood with an example. Consider the program

$$\begin{array}{l}\mathbf{let}^D\ n = 35^D \\ \ \mathbf{in\ let}^D\ f = \mathbf{fix}^S\ (\lambda^S g.\lambda^S x.1^D +^D \mathbf{if}^S\, x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S)) \\ \quad\ \mathbf{in}\ f\ @^S 2^S\end{array}$$

The argument of $\mathbf{fix}^S$ specializes as follows (in the first premise of rule (O-SFIX))

$$\begin{array}{l} n : Int^D \hookrightarrow 35 : Int \\ \vdash \lambda^S g.\lambda^S x.1^D +^D \mathbf{if}^S\, x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S) \\ \quad : (Int^S \to^S Int^D) \to^S (Int^S \to^S Int^D) \\ \qquad \hookrightarrow 35 : \mathbf{clos} \ll [(n, Int^D, Int)], g, \\ \qquad\qquad\qquad\qquad \lambda^S x.1^D +^D \mathbf{if}^S\, x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S) \gg \end{array}$$

Then (on the second premise of (O-SFIX)), the body of the function is specialized *without* a binding for g, thus resulting in

$$\begin{array}{l} n : Int^D \hookrightarrow 35 : Int \\ \vdash \lambda^S x.1^D +^D \mathbf{if}^S\, x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S) \\ \quad : Int^S \to^S Int^D \\ \qquad \hookrightarrow 35 : \mathbf{clos} \ll [(n, Int^D, Int)], x, \\ \qquad\qquad\qquad\qquad 1^D +^D \mathbf{if}^S\, x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S) \gg \end{array}$$

in which g appears as a free variable. The residual of f can be obtained applying (O-SFIX):

$$35 : \mathbf{rec} \ll [(n, Int^D, Int)], g, x, 1^D +^D \mathbf{if}^S\, x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S) \gg$$

It is important to note that the body of the recursive function was not specialized at all; it was simply stored in the closure, waiting for an application.

The application of a recursive function needs a special rule, because unfolding of recursive calls must be performed; it is obtained by binding the name of the recursive variable to the very function being applied.

$$\frac{\Gamma \vdash e_1 : \tau_a \to^S \tau_r \hookrightarrow e_1' : \textbf{rec} \ll [(z_i, \tau_i, \tau_i')]_{i=1}^n, f, x, e \gg \quad \Gamma \vdash e_2 : \tau_a \hookrightarrow e_2' : \tau_a' \qquad \Gamma_{\text{rec}} \vdash e : \tau_r \hookrightarrow e' : \tau_r'}{\Gamma \vdash e_1 \mathbin{@^S} e_2 : \tau_r \hookrightarrow e' : \tau_r'} \text{(O-RSAPP)}$$

where

$$\begin{aligned}\Gamma_{\text{rec}} = [&z_i : \tau_i \hookrightarrow \pi_{i,n}\, e_1' : \tau_i']_{i=1}^n,\\ &f : \tau_a \to^S \tau_r \hookrightarrow e_1' : \textbf{clos} \ll [(z_i, \tau_i, \tau_i')]_{i=1}^n, x, e \gg,\\ &x : \tau_a \hookrightarrow e_2' : \tau_a'\end{aligned}$$

Concluding the example of the static recursive function f, we may apply the rule (O-RSAPP) obtaining the following residual code:

$$\textbf{let}\ n = 35\ \textbf{in let}\ f = n\ \textbf{in}\ 1 + (1 + (1 + f))$$

The recursive function has been unfolded three times, and the name of the function is bound to a (one-)tuple of its free variables, used to interpret the reference of n in the unfolded code.

Had we used dynamic recursion instead, the resulting code would be very similar, except that the residual of f would be a tuple with a circular (recursive) reference to itself, and projections would be used to extract the correct component.

$$\begin{array}{l}\vdash \textbf{let}^D\ n = 35^D\\ \quad \textbf{in let}^D\ f = \textbf{fix}^D\ (\lambda^D g.\textbf{poly}\ (\lambda^D x.1^D +^D \textbf{if}^S\ x ==^S 0^S\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else spec}\ g \mathbin{@^D} (x -^S 1^S)))\\ \qquad \textbf{in spec}\ f \mathbin{@^D} 2^S\\ \ : \mathit{Int}^D\\ \quad \hookrightarrow \textbf{let}\ n = 35\\ \qquad \textbf{in let}\ f = \textbf{fix}\ \lambda g.\lambda x.(1 + \pi_{2,3}\, g@\bullet, 1 + \pi_{3,3}\, g@\bullet, 1 + n)\\ \qquad\quad \textbf{in}\ \pi_{1,3}\, f@\bullet\\ \qquad : \mathit{Int}\end{array}$$

The similarity is much more apparent after arity raising, when the residual code becomes the following final term.

$$\begin{array}{l}\textbf{let}\ n = 35\ \textbf{in let}\ f_1 = 1 + f_2\\ \qquad\qquad\qquad\qquad\ f_2 = 1 + f_3\\ \qquad\qquad\qquad\qquad\ f_3 = 1 + n\\ \qquad\qquad\quad\ \ \textbf{in}\ f_1\end{array}$$

The usefulness of static recursion can be seen in the following example.

EXAMPLE 3.19. We consider again the *power* function mentioned in Chapter 1.

$$\begin{array}{l} \mathbf{let}^S\ power = \mathbf{fix}^S\ (\lambda^S p.(\lambda^S n.\lambda^S x.\mathbf{if}^S\ n ==^S 1^S \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ x \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ x *^D p\ @^S (n -^S 1)\ @^S x)) \\ \ \mathbf{in}\ \lambda^D z.power\ @^S 3^S\ @^S z \end{array}$$

Observe that *power* has the first argument static, but it is also a static function — thus, it will be unfolded, and there is no need for polyvariance. When using *power*, we provide a dynamic function as context to its application, to be able to observe the result.

The specialization of this example is $\lambda z.z * (z * z)$, as expected.

3.4.5 Sum types

A very important extension to the language is the addition and treatment of sum types. Hughes defined sum types as anonymous, tagged, n-ary sums of the form $C_1\ \tau_1\ \mid\ \ldots\ \mid\ C_n\ \tau_n$, in which every constructor, distinguished lexically using an initial capital letter, takes only one argument. The order of constructors in a sum is irrelevant, that is, sum types are identified by the set of constructors. Sum types are usually abbreviated $\sum_{i=1}^{n} C_i\ \tau_i$. For example, the type *Bool* of booleans can be expressed as the sum *True* () | *False* (), where we identify *True* : *Bool* and *True* () (and similarly for *False*).

The construct to operate with sums is the **case** construct: **case** e **of** $C_1\ x_1\ \rightarrow\ e_1; \ldots; C_n\ x_n\ \rightarrow\ e_n$. It has a branch for each constructor in the type, and performs a pattern matching on the value of the expression e, selecting the ith branch when it begins with constructor C_i, and returning the value of e_i with x_i bound to the value of the argument of that constructor.

The decision of representing sum types as anonymous sums requires the introduction of recursive types to have the possibility to define inductive sets. This results in a type system that is more powerful than Haskell's, introducing some minor complications in the treatment; in later sections we use the more familiar definition of datatypes à la Haskell, so we add those definitions at the beginning of our examples.

Sum types are divided into static and dynamic, just like every other construct in the language. The specialization of each variant is different, and so we discuss each one in its own subsection. Examples of the use of static and dynamic sum types in practical situations are given in Chapter 4.

Static sums

Following the basic idea of "move the static information into the type", the specialization of a value of a static sum has to move the constructor tag into the residual type. To do that, we introduce one residual type constructor for every static sum constructor, and let the residual type of a tagged value live in the type resulting from applying the corresponding constructor to the residual type of the argument.

$$\text{(O-SCON)}\quad \frac{\Gamma \vdash e : \tau_k \hookrightarrow e' : \tau'_k}{\Gamma \vdash C_k^S\ e : \sum_{i=1}^{n} C_i^S\ \tau_i \hookrightarrow e' : C_k\ \tau'_k}$$

The original system uses a variation of rows [Rèmy, 1989; **?**; Pottier, 2000; Shields and Meijer, 2001] to calculate the resulting sum type, but we have chosen to present it here in a simpler form, because the treatment of rows is just an implementation detail not affecting the specialization.

Observe that, in the rule (O-SCON), the residual type $C_k\ \tau'_k$ is a specialized version of the source type $\sum_{i=1}^{n} C_i^S\ \ \tau_i$, corresponding to a specialization of the k-th constructor; the information that e lies in summand C_k^S is recorded in the residual type, thus removing the tag in the residual code. This is very important in the optimal specialization of lambda calculus presented in Chapter 4.

Static case expressions are specialized by choosing the branch corresponding to the constructor used in the scrutinized expression. Determining which branch to choose is possible because the residual type of the control expression has information of which constructor was used; this can be seen in the first premise of the rule, by observing that the residual type of the expression is $C_k\ \tau'_k$.

$$\text{(O-SCASE)}\quad \frac{\Gamma \vdash e : \sum_{i=1}^{n} C_i^S\ \ \tau_i \hookrightarrow e' : C_k\ \tau'_k \qquad \Gamma, x_k : \tau_k \hookrightarrow e' : \tau'_k \vdash e_k : \tau \hookrightarrow e'_k : \tau'}{\Gamma \vdash \mathbf{case}^S\, e\ \mathbf{of}\ [C_i^S\ \ x_i\ \rightarrow\ e_i]_{i=1}^{n} : \tau \hookrightarrow e'_k : \tau'}$$

Observe that the second premise only specializes the k-th branch; all the other branches are not specialized at all.

It is interesting to see some examples using static sums.

Example 3.20. Observe how the static case in the body of the function can be eliminated, because the type of the argument gives the information about which branch to choose, even when the function is not reduced. Also observe that the static number 17^S is propagated as expected.

$$\begin{array}{l}
\vdash (\lambda^D x.\mathbf{case}^S\ x\ \mathbf{of} \\
\qquad\qquad \mathit{Left}^S\ f\ \rightarrow\ f\ @^D\, 17^S \\
\qquad\qquad \mathit{Right}^S\ n\ \rightarrow\ n) \\
\quad @^D\, (\mathit{Left}^S\ \ (\lambda^D y.\mathbf{lift}\ y +^D 1^D\,)) \\
\ : \mathit{Int}^D \\
\ \hookrightarrow (\lambda x.x@\bullet)@(\lambda y.17 + 1) : \mathit{Int}
\end{array}$$

Example 3.21. Observe how the static case in the body of the polyvariant function is resolved differently in each specialization.

$$\begin{array}{l}
\vdash \mathbf{let}^D\ \ f = \mathbf{poly}\ (\lambda^D x.\mathbf{case}^S\, x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Left}^S\ n\ \rightarrow\ n +^D 1^D \\
\qquad\qquad\qquad\qquad\qquad \mathit{Right}^S\ n\ \rightarrow\ n +^D n) \\
\quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\, (\mathit{Left}^S\ \ 2^D\,), \mathbf{spec}\ f\ @^D\, (\mathit{Right}^S\ \ 3^D\,))^D \\
\ : (\mathit{Int}^D\,, \mathit{Int}^D\,)^D \\
\ \hookrightarrow \mathbf{let}\ f = (\lambda x.x +^D 1^D\,, \lambda x.x +^D x) \\
\qquad \mathbf{in}\ (\mathbf{fst}\ f@2, \mathbf{snd}\ f@3) \\
\ : (\mathit{Int}, \mathit{Int})
\end{array}$$

Dynamic sums

In the case of dynamic sums, the tag must remain in the residual code, and the residual type must be a sum type. That is reflected in the rule for dynamic sum constructors.

$$\text{(O-DCON)}\quad \frac{\Gamma \vdash e : \tau_k \hookrightarrow e' : \tau'_k}{\Gamma \vdash C_k^D\ e : \sum_{i=1}^{n} C_i^D\ \tau_i \hookrightarrow C_k\ e' : C_k\ \tau'_k \mid \phi'}$$

This rule restricted the residual sum type to have a summand with the right tag, C_k, but left the rest unspecified (as expressed by the metavariable ϕ); several rules taken together will provide all the constructors that will be part of the residual type. In this way, new sum types will be created depending on how they are used.

The static information that was moved into the residual type argument of a constructor is used in a dynamic case to specialize each branch accordingly.

$$\text{(O-DCASE)}\quad \frac{\Gamma \vdash e : \sum_{i=1}^{n} C_i^D\ \tau_i \hookrightarrow e' : \sum_{i=1}^{n} C_i\ \tau'_i \quad [\Gamma, x_i : \tau_i \hookrightarrow x'_i : \tau'_i \vdash e_i : \tau \hookrightarrow e'_i : \tau']_{i=1}^{n}}{\begin{array}{c}\Gamma \vdash \mathbf{case}^D\, e\ \mathbf{of}\ [C_i^D\ \tau_i\ \rightarrow\ e_i]_{i=1}^{n} : \tau \\ \hookrightarrow \mathbf{case}\ e'\ \mathbf{of}\ [C_i\ \tau'_i\ \rightarrow\ e'_i]_{i=1}^{n} : \tau' \\ \left(x'_i \text{ fresh}, i \in 1 \ldots n\right)\end{array}}$$

It is interesting to observe that, while the rule for constructors allows the residual sum type to have fewer constructors than the source one, the rule for case forces the same number to appear on both. This is a small inconsistency in Hughes' presentation that can be fixed, either by relaxing the rule for case, or by forcing also the same number of constructors in the other rule.

An example of the use of dynamic sums shows that a dynamic case produces a case in the residual program; but, even more importantly, it also shows that static information under a constructor can be used in the proper branch of that case (but it has to be the same on all the uses of the constructor!)

Example 3.22. Observe how each branch of the case uses static information appearing under a dynamic if.

$$\begin{array}{l}
\vdash \lambda^D b.(\lambda^D x.\mathbf{case}^D\ x\ \mathbf{of} \\
\qquad\qquad \mathit{Left}^D\ n\ \rightarrow\ \mathbf{lift}\ (n +^S 1^S) \\
\qquad\qquad \mathit{Right}^D\ b'\ \rightarrow\ \mathbf{if}^S\, b'\ \mathbf{then}\ 1^D\ \mathbf{else}\ 0^D) \\
\qquad @^D\, (\mathbf{if}^D\, b\ \mathbf{then}\ \mathit{Left}^D\ 42^S\ \mathbf{else}\ \mathit{Right}^D\ \mathit{True}^S) : \mathit{Bool}^D \rightarrow^D \mathit{Int}^D \\
\quad \hookrightarrow \lambda b.(\lambda x.\mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\quad \mathit{Left}\ n\ \rightarrow\ 43 \\
\qquad\qquad\quad \mathit{Right}\ b'\ \rightarrow\ 1) \\
\qquad\quad @\ (\mathbf{if}\ b\ \mathbf{then}\ \mathit{Left}\ \bullet\ \mathbf{else}\ \mathit{Right}\ \bullet) : \mathit{Bool} \rightarrow \mathit{Int}
\end{array}$$

This propagation of static information is possible because it is carried on the residual type of x: $\mathit{Left}\ \hat{42} \mid \mathit{Right}\ \widehat{\mathit{True}}$; then, the residual type of n in the left branch is $\hat{42}$ and the residual type of b' in the right branch is $\widehat{\mathit{True}}$, allowing the specialization of the branch expressions.

3.4.6 Things not included

The specializer implemented by Hughes has some additional features that we do not consider in this thesis, but that are important for expressiveness. One such feature is polyvariant sums, that are the dual of the polyvariance we have described. Another one is the treatment of imperative features, as presented by Dussart *et al.* [1997b].

The polyvariance presented here creates n-ary products in the residual program. The dual notion is a polyvariant construct that creates polyvariant sums. It is introduced by a new source type **sum** τ with a single data constructor **In**, and eliminated using a special form of **case** expression. For example,

$$
\begin{array}{l}
\vdash \mathbf{let}^{D}\ f = \lambda^{D}x.\mathbf{case}^{D}\, x\ \mathbf{of}\ \mathbf{In}\ y\ \rightarrow\ \mathbf{lift}\ y +^{D} 1^{D} \\
\quad \mathbf{in}\ (f\ @^{D}\,(\mathbf{In}\ 17^{S}), f\ @^{D}\,(\mathbf{In}\ 42^{S}))^{D} \\
\quad : (Int^{D}, Int^{D})^{D} \\
\qquad \hookrightarrow \mathbf{let}\ f = \lambda x.\mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathbf{In}_0\ y\ \rightarrow\ 17+1 \\
\qquad\qquad\qquad\qquad \mathbf{In}_1\ y\ \rightarrow\ 42+1 \\
\qquad\quad \mathbf{in}\ (f@(\mathbf{In}_0\ \bullet), f@(\mathbf{In}_1\ \bullet)) \\
\qquad\quad : (Int, Int)
\end{array}
$$

Polyvariant sums and polyvariant products are used for similar purposes, that is, to allow multiple specializations of some expression; polyvariant sums produce case expressions that take apart the different specializations of **In**, but can be used to produce some residual codes that cannot be obtained using polyvariant products, as can be observed in the following example, or in the firstifying interpreter presented by Hughes [1996b].

EXAMPLE 3.23. The expression presented here is similar to that in Example 3.15, except that it uses polyvariant sums to allow the variation of static information in the branches of the conditional.

$$
\begin{array}{l}
\vdash \lambda^{D}b.\mathbf{let}^{D}\ f = \lambda^{D}x.\mathbf{if}^{D}\, b \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathbf{In}\ (2^{S}, \mathbf{lift}\ x)^{D} \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathbf{In}\ (3^{S}, 51^{D})^{D} \\
\qquad \mathbf{in}\ \mathbf{case}^{D}\,(f\ @^{D}\,42^{S})\ \mathbf{of}\ \mathbf{In}\ y\ \rightarrow\ \mathbf{snd}^{D}\ y +^{D} \mathbf{lift}\ (\mathbf{fst}^{D}\ y) \\
\quad : Int^{D} \\
\qquad \hookrightarrow \lambda b.\mathbf{let}\ f = \lambda x.\ \mathbf{if}\ p \\
\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \mathbf{In}_0\ (\bullet, 42) \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{In}_1\ (\bullet, 51) \\
\qquad\qquad \mathbf{in}\ \mathbf{case}\ (f@\bullet)\ \mathbf{of} \\
\qquad\qquad\quad \mathbf{In}_0\ y\ \rightarrow\ \mathbf{snd}\ y+2 \\
\qquad\qquad\quad \mathbf{In}_1\ y\ \rightarrow\ \mathbf{snd}\ y+3 \\
\qquad\quad : Int
\end{array}
$$

Observe how the dynamic conditional can have different static information on each branch, because in the residual that will give rise to a new version of constructor **In**, and compare that with Example 3.15. The same idea is used in the firstifying interpreter of Hughes: to produce firstification, the universal type to represent expressions of the object language will use static functions; but to allow those functions to have different

static information, a polyvariant sum is used in a way similar to the one showed here — more details were given by Hughes [1996b]. A natural question to ask is whether a firstifying interpreter can be constructed using only polyvariant products. There are several alternatives, but none of them produce a suitable firstifying interpreter. One of those alternatives is to use CPS to represent the sums (i.e. **poly** $(\tau \to \text{ans}) \to \text{ans}$ representing the polyvariant sum **sum** τ), but then the resulting program will still contain functions coming from the representation of continuations, and so it will not be first order; additionally, it is not clear that one type of answers is sufficient (ans in the previous definition). Another alternative is to use only polyvariant products. But then the resulting program will contain a function for every function call with a different type, even for higher order functions in the object program — it is expected that the first order version of a higher order function is a single function receiving closures, and not a set of different functions. So, the best choice to produce a firstifying interpreter is to use polyvariant sums.

Computational effects expressed as monadic operations can be type specialized, as explained by Dussart *et al.* [1997b]. The rules to specialize monadic primitives and operations on state are modularly added to the ones presented here, showing the advantages of presenting the specialization by a system of rules.ting the specialization by a system of rules. Optimal specialization for a language with first-class references is presented, and also the specialization of a lazy interpreter using references to store closures that can be updated. One of the distinguishing features of this specialization is that some operations on references can be performed statically. Another one is that the specializer is independent of the evaluation order: only the sequence of monadic operations is fixed.

3.4.7 On recursive types

We have said, when presenting the syntax of source types in Section 3.1, that their interpretation is coinductive. This is different with respect to the classical interpretation of types as the initial algebra (that is, inductive types), and implies that these types are any finite or infinite expressions conforming to this grammar. In this way, recursive types can be represented by their infinite unfolding — which is called *equi-recursive types* by **?** [?]. We use the abbreviation $\mu X.T(X)$ to represent the infinite unfolding of T — that is, $T(T(T(\ldots)))$.

This feature is important because it is our goal that residual recursive types be generated from an interpreter with no indication of where the recursion takes place. That is, if we are using a universal type to represent values of the object language, then the tags of this type will be used to generate residual types for the program — even recursive ones. Then, the interpretation for that universal type has to include infinite terms, corresponding to the expansion of the recursive type to be generated. Let's see that with an example.

EXAMPLE 3.24. We consider a fragment of a universal datatype to represent values, `V`, containing numbers, constructors, unit, and pairs. The notation $\mathbf{data}\,V^S$ is used to indicate that the sum represented by V is a static sum at top level — that is, all its constructors are static.

$$\begin{array}{l}\textbf{data}\ V^S = N\ Int^D \mid C\ (String^S, V^S)^D \\ \qquad\quad \mid\ U\ ()^D \quad \mid P\ (V^S, V^S)^D\end{array}$$

We define a program that produces an object list of seventeens from a given number, and annotate it so the residual program will produce such a list. The residual type must then be a recursive type representing lists.

$$\begin{array}{l}\vdash \textbf{fix}^D\,(\lambda^D f.\lambda^D x.\textbf{if}^D\, x ==^D 0^D \\ \qquad\qquad\qquad\quad \textbf{then In}\ (C^S\ (\text{“Nil”}^S, U^S\ ()^D)^D) \\ \qquad\qquad\qquad\quad \textbf{else In}\ (C^S\ (\text{“Cons”}^S, P^S\ (N^S\ 17^D, f\ @^D\,(x -^D 1^D))^D)^D) \\ \qquad)\ @^D\,3^D \\ \quad : \textbf{sum}\ V^S \\ \quad \hookrightarrow \textbf{fix}(\lambda f.\lambda x.\ \textbf{if}\ x == 0 \\ \qquad\qquad\qquad\quad \textbf{then In}_0\ (\bullet, ()) \\ \qquad\qquad\qquad\quad \textbf{else In}_1\ (\bullet, (17, f@(x-1))) \\ \qquad\quad)@3 \\ \qquad : \mu X.\textbf{In}_0\ (C(\text{“Nil”}, U())) + \textbf{In}_1\ (C(\text{“Cons”}, P(N Int, X)))\end{array}$$

After arity raising, we obtain the following program, as expected.

$$\begin{array}{l}\textbf{fix}(\lambda f.\lambda x.\ \textbf{if}\ x == 0 \\ \qquad\qquad\quad \textbf{then In}_0\ () \\ \qquad\qquad\quad \textbf{else In}_1\ (17, f@(x-1)) \\ \quad)@3 \\ \quad : \mu X.\textbf{In}_0\ + \textbf{In}_1\ Int\ X\end{array}$$

where $\textbf{In}_0$ represents the constructor *Nil* and $\textbf{In}_1$, the constructor *Cons*. Observe that the residual type is recursive, although the source type does not indicate where the recursion has to be introduced — the source type *is* recursive, but static, thus indicating that the infinite unfolding must be used. For this reason the interpretation of the source type has to be coinductive.

Chapter 4

Examples

> *And in a day or two more he was bold enough to ask his master, 'When will my apprenticeship begin, Sir?'*
> *'It has begun', said Ogion.*
> *There was a silence, as if Ged was keeping back something he had to say. Then he said it: 'But I haven't learned anything, yet!'*
> *'Because you haven't found out what I am teaching', replied the mage...*
>
> A Wizard of Earthsea
> Úrsula K. Le Guin

In this chapter we give three variations of the interpreter for the lambda-calculus appearing in Chapter 2; the purpose is to illustrate the usefulness of the techniques presented. We also discuss some of the limitations and problems in type specialization, and summarize the main ideas.

The three examples are based on the same basic code for an interpreter of the lambda-calculus, but they are annotated differently, obtaining compilers with different static semantics. The first example, in Section 4.1, is annotated in such a way that the resulting lambda-calculus is untyped. The only difference between the interpreter presented in Section 4.2 and the previous one is in the annotation of the sum type representing lambda terms; however, that makes it an interpreter for the simply typed lambda-calculus. Then, in Section 4.3, we show a variation of the previous interpreter to deal with let-bound polymorphism in the object language; polyvariance is used to express polymorphism, but the resulting residual program is monomorphic, so, in that sense, the compilation obtained by specializing this interpreter is a monomorphizer: a polymorphic function in the object language is expanded to one copy for each use with a different monomorphic type.

In Section 4.4 we discuss some limitations of type specialization providing examples of the kind of problems they produce. Finally in Section 4.5 we summarize the strengths and problems of type specialization.

The examples considered here show that the ability to choose the binding times of the source program is a great advantage when designing typed languages. It is completely unreasonable to expect that a program, such as a BTA, can determine the annotations in our stead: that would mean that the BTA is choosing the static semantics of our object language! It is for this reason that we argue that Jones' claim that the goal is to automatically transform an interpreter into a compiler is unrealistic, and therefore we put the stress on the increased power that a specializer can give to a programmer (in this case, the ability to choose between untyped, simply typed, and Hindley-Milner typed versions of the object language!). Additionally, the understanding of how the specializer works helps to choose where to place polyvariance, giving more flexibility to

$$
\begin{array}{l}
\textbf{data}\ \mathit{LExp}^S = \mathit{Var}\ \mathit{Char}^S \mid \mathit{Const}\ \mathit{Int}^S \\
\qquad\qquad \mid \mathit{Lam}\ \mathit{Char}^S\ \mathit{LExp}^S \mid \mathit{App}\ \mathit{LExp}^S\ \mathit{LExp}^S \\
\qquad\qquad \mid \mathit{Let}\ \mathit{Char}^S\ \mathit{LExp}^S\ \mathit{LExp}^S \\
\textbf{data}\ \mathit{Value}^D = \mathit{Num}\ \mathit{Int}^D \mid \mathit{Fun}\ (\mathit{Value}^D \to^D \mathit{Value}^D) \mid \mathit{Wrong} \\
\textbf{let}^S\ \mathit{bind} = \lambda^S x.\lambda^S v.\lambda^S \mathit{env}. \\
\qquad\qquad \lambda^S y.\textbf{if}^S\ x == y\ \textbf{then}\ v\ \textbf{else}\ \mathit{env}\ @^S\ y \\
\ \textbf{in} \\
\textbf{let}^S\ \mathit{preeval} = \\
\quad \textbf{fix}^S\ (\lambda^S \mathit{eval}.\lambda^S \mathit{env}.\lambda^S \mathit{expr}. \\
\qquad \textbf{case}^S\ \mathit{expr}\ \textbf{of} \\
\qquad\quad \mathit{Var}\ x \to \mathit{env}\ @^S\ x \\
\qquad\quad \mathit{Const}\ n \to \mathit{Num}^D\ (\textbf{lift}\ n) \\
\qquad\quad \mathit{Lam}\ x\ e \to \mathit{Fun}^D\ (\lambda^D v. \\
\qquad\qquad\qquad \textbf{let}^S\ \mathit{env}' = \mathit{bind}\ @^S\ x\ @^S\ v\ @^S\ \mathit{env} \\
\qquad\qquad\qquad \textbf{in}\ \mathit{eval}\ @^S\ \mathit{env}'\ @^S\ e) \\
\qquad\quad \mathit{App}\ e_1\ e_2 \to \textbf{case}^D\ (\mathit{eval}\ @^S\ \mathit{env}\ @^S\ e_1)\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Fun}\ f \to f\ @^D\ (\mathit{eval}\ @^S\ \mathit{env}\ @^S\ e_2) \\
\qquad\quad \mathit{Let}\ x\ e_1\ e_2 \to \textbf{let}^D\ v = \mathit{eval}\ @^S\ \mathit{env}\ @^S\ e_1 \\
\qquad\qquad\qquad \textbf{in let}^S\ \mathit{env}' = \mathit{bind}\ @^S\ x\ @^S\ v\ @^S\ \mathit{env} \\
\qquad\qquad\qquad\quad \textbf{in}\ \mathit{eval}\ @^S\ \mathit{env}'\ @^S\ e_2) \\
\ \textbf{in} \\
\textbf{let}^S\ \mathit{ueval} = \mathit{preeval}\ @^S\ (\lambda^S x.\mathit{Wrong}^D) \\
\ \textbf{in}\ \langle \ldots \rangle
\end{array}
$$

Figure 4.1: An evaluator for untyped lambda-calculus.

the programmer. For that reason we think that users (in this case, *programmers*!) need to understand the details of the specialization process.

4.1 Untyped λ-calculus

Our first example, presented in Figure 4.1, is an interpreter for lambda-calculus with the same annotations as the one given in Chapter 2. The only differences are that we have made a function to add a binding variable-value to an environment (*bind*), and we have added an initial environment $(\lambda^S x \to \mathit{Wrong}^D)$[1]. Observe that the constructors of the result of *ueval* (*Num* and *Fun*) have been annotated dynamic; this amounts to keeping the tags in the residual code, and then it is during runtime that type errors are detected — that is, the language is untyped. See how that happens in the following example.

[1]We are also taking some liberties in the use of datatype syntax to express sum types. The names LExp^S and Value^D must be considered as abbreviations for the anonymous sum of their components. Additionally, we show constructors with several arguments, but the sum types introduced can only handle single arguments. This can be considered syntactic sugar, because it can be expressed using a tuple as argument for the constructor, and pattern matching as the use of the corresponding projection.

EXAMPLE 4.1. The following specialization is possible.

$$\vdash ueval\ @^S\,(App^S\ (Const^S\ 2^S)\ (Const^S\ 3^S)) : Value^D$$
$$\hookrightarrow \textbf{case}\ (Num\ 2)\ \textbf{of}\ Fun\ f\ \rightarrow\ f@(Num\ 3) : Value$$

The App^S produces the residual **case** and the residual application, and every $Const^S$ produces a residual *Num*. This program will fail when evaluated, exactly as the object program would have failed, because there is no branch for *Num* in the residual **case**.

We are, again, compiling by specialization, and the fact that the residual program is produced indicates that the resulting compilation is for untyped lambda-calculus — that is, specialization is analogous to compile-time, and, as the error is produced at run-time, we can conclude that no type checking was performed.

As the language interpreted is untyped, the generation of runtime type checks has to be done even when the program is correctly typed; as we have seen this is a problem for partial evaluation, because in that approach, this is the best annotation possible.

EXAMPLE 4.2. The result of specializing the expression

$$\begin{array}{l} ueval\ @^S\,(Lam^S\ \texttt{'f'}^S \\ \qquad\qquad (App^S\,(Var^S\ \texttt{'f'}^S) \\ \qquad\qquad\qquad (App^S\ (Var^S\ \texttt{'f'}^S)\ (Const^S\ 0^S)))) \end{array}$$

is the following residual code

$$\begin{array}{l} Fun\ (\lambda v.\textbf{case}\ v\ \textbf{of} \\ \qquad\quad Fun\ f\ \rightarrow\ f@(\textbf{case}\ v\ \textbf{of} \\ \qquad\qquad\qquad\qquad Fun\ f\ \rightarrow\ f@(Num\ 0))) \\ \ : Value \end{array}$$

Every function requires a *Fun* tag, every application requires a *case*, and every number requires a *Num* tag.

4.2 Simply Typed λ-calculus

But with type specialization, we can go further. In Figure 4.2 the same evaluator is annotated in a different way: observe that the constructors in the result of *teval* are annotated as static, but their arguments are still dynamic; this amounts to removing the tags, specializing the residual type. The object language is now *simply typed* lambda-calculus, because the specialization process (and therefore, the compilation of the object program) will succeed only when the object program is well-typed, and it will fail for ill-typed ones. If we are to compile by specializing an interpreter, then the specialization method must fail on those inputs. This is achieved by the failure of an equality test on residual types, something that seemed a not-so-good idea when we introduced polyvariance, but that is an essential feature of the system. We illustrate this feature with the following examples.

$$
\begin{array}{l}
\mathbf{data}\ LExp^S = Var\ Char^S \mid Const\ Int^S \\
\qquad\qquad \mid Lam\ Char^S\ LExp^S \mid App\ LExp^S\ LExp^S \\
\qquad\qquad \mid Let\ Char^S\ LExp^S\ LExp^S \\
\mathbf{data}\ Value^S = Num\ Int^D \mid Fun\ (Value^S \rightarrow^D Value^S) \mid Wrong \\
\mathbf{let}^S\ bind = \lambda^S x.\lambda^S v.\lambda^S env. \\
\qquad\qquad \lambda^S y.\mathbf{if}^S\ x == y\ \mathbf{then}\ v\ \mathbf{else}\ env\ @^S\ y \\
\ \mathbf{in} \\
\mathbf{let}^S\ preeval = \\
\qquad \mathbf{fix}^S\ (\lambda^S eval.\lambda^S env.\lambda^S expr. \\
\qquad\qquad \mathbf{case}^S\ expr\ \mathbf{of} \\
\qquad\qquad\quad Var\ x \rightarrow env\ @^S\ x \\
\qquad\qquad\quad Const\ n \rightarrow Num^S\ (\mathbf{lift}\ n) \\
\qquad\qquad\quad Lam\ x\ e \rightarrow Fun^S\ (\lambda^D v. \\
\qquad\qquad\qquad\qquad \mathbf{let}^S\ env' = bind\ @^S\ x\ @^S\ v\ @^S\ env \\
\qquad\qquad\qquad\qquad \mathbf{in}\ eval\ @^S\ env'\ @^S\ e) \\
\qquad\qquad\quad App\ e_1\ e_2 \rightarrow \mathbf{case}^S\ (eval\ @^S\ env\ @^S\ e_1)\ \mathbf{of} \\
\qquad\qquad\qquad\qquad Fun\ f \rightarrow f\ @^D\ (eval\ @^S\ env\ @^S\ e_2) \\
\qquad\qquad\quad Let\ x\ e_1\ e_2 \rightarrow \mathbf{let}^D\ v = eval\ @^S\ env\ @^S\ e_1 \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \mathbf{let}^S\ env' = bind\ @^S\ x\ @^S\ v\ @^S\ env \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ eval\ @^S\ env'\ @^S\ e_2) \\
\ \mathbf{in} \\
\mathbf{let}^S\ teval = preeval\ @^S\ (\lambda^S x.Wrong^S) \\
\ \mathbf{in}\ \langle \ldots \rangle
\end{array}
$$

Figure 4.2: An evaluator for simply-typed lambda-calculus.

EXAMPLE 4.3. The expression $teval\ @^S (App^S\ (Const^S\ 2^S)\ (Const^S\ 3^S))$ has no specialization, thus showing that the compilation process obtained by specialization treats the object language as typed. The failure is caused by the annotations: type tags Fun^S and Num^S are moved into the type, so the residual type of $(Const^S\ 2^S)$ is $\bullet :: Num\ \hat{2}$; *but* the static case in the evaluator has no branch for the *Num* tag, and then it will fail when trying to generate a residual term for this expression. Observe that the specialization failure is indeed the desired behaviour!

EXAMPLE 4.4. The result of specializing the expression

$$\begin{aligned} teval\ @^S (&Lam^S\ \texttt{'f'}^S \\ &(App^S\ (Var^S\ \texttt{'f'}^S) \\ &\quad (App^S\ (Var^S\ \texttt{'f'}^S)\ (Const^S\ 0^S)))) \end{aligned}$$

is the following residual code

$$\begin{aligned} \lambda v.\ v\ @\ (v\ @\ 0)\ :: Fun\ (&Fun\ (Num\ Int \to\ Num\ Int) \\ &\to Num\ Int) \end{aligned}$$

Observe that the residual type reflects the typing of the object program! Type specialization is performing type inference using only the text of the interpreter appropriately annotated.

EXAMPLE 4.5. The result of specializing the expression

$$\begin{aligned} teval\ @^S (&Let^S\ \texttt{'i'}^S\ (Lam^S\ \texttt{'x'}^S\ (Var^S\ \texttt{'x'}^S)) \\ &(App^S\ (App^S\ (Var^S\ \texttt{'i'}^S)\ (Var^S\ \texttt{'i'}^S)) \\ &\quad (Const^S\ 0^S))) \end{aligned}$$

is a failure, because the two occurrences of $(Var^S\ \texttt{'i'}^S)$ should have different residual types: the first should have residual type

$$Fun\ ((Fun\ (Num\ Int \to\ Num\ Int)) \to\ (Fun\ (Num\ Int \to\ Num\ Int)))$$

and the second one, $Fun\ (Num\ Int \to Num\ Int)$. This is showing that the lambda calculus interpreted is *simply* typed, and once more the importance of failure when residual types do not match.

4.3 Monomorphizing λ-calculus

There is still one more useful alternative we can take when annotating the interpreter: by using polyvariance, we can make our interpreter accept let-bound polymorphic programs and monomorphize them — that is, produce an equivalent monomorphic program where the polymorphic functions are expanded to monomorphic copies of themselves. We present the variation in Figure 4.3. The only change to the evaluator presented in previous sections is in the elements that can be stored in the environment: they can be monovariant or polyvariant, reflecting the fact that the object elements associated with the corresponding variables are monomorphic or polymorphic. To be able to return either monovariant or polyvariant expressions, a new datatype, MP^S is introduced. Polyvariant expressions are created by the *Let* rule, and eliminated by the *Var* rule.

$$
\begin{array}{l}
\textbf{data}\, \mathit{LExp}^{S} = \mathit{Var}\ \mathit{Char}^{S} \mid \mathit{Const}\ \mathit{Int}^{S} \\
\qquad\qquad \mid \mathit{Lam}\ \mathit{Char}^{S}\ \mathit{LExp}^{S} \mid \mathit{App}\ \mathit{LExp}^{S}\ \mathit{LExp}^{S} \\
\qquad\qquad \mid \mathit{Let}\ \mathit{Char}^{S}\ \mathit{LExp}^{S}\ \mathit{LExp}^{S} \\
\textbf{data}\, \mathit{Value}^{S} = \mathit{Num}\ \mathit{Int}^{D} \mid \mathit{Fun}\ (\mathit{Value}^{S} \rightarrow^{D} \mathit{Value}^{S}) \mid \mathit{Wrong} \\
\textbf{data}\, \mathit{MP}^{S} = M\ \mathit{Value}^{S} \mid P\ (\textbf{poly}\ \mathit{Value}^{S}) \\
\textbf{let}^{S}\ \mathit{bind} = \lambda^{S} x.\lambda^{S} v.\lambda^{S} \mathit{env}. \\
\qquad\qquad \lambda^{S} y.\textbf{if}^{S}\ x == y\ \textbf{then}\ v\ \textbf{else}\ \mathit{env}\ @^{S}\ y \\
\textbf{in} \\
\textbf{let}^{S}\ \mathit{preeval} = \\
\quad \textbf{fix}^{S}\ (\lambda^{S} \mathit{eval}.\lambda^{S} \mathit{env}.\lambda^{S} \mathit{expr}. \\
\qquad \textbf{case}^{S}\ \mathit{expr}\ \textbf{of} \\
\qquad\quad \mathit{Var}\ x \rightarrow \textbf{case}^{S}\ (\mathit{env}\ @^{S}\ x)\ \textbf{of} \\
\qquad\qquad\qquad M\ v \rightarrow v \\
\qquad\qquad\qquad P\ v \rightarrow \textbf{spec}\ v \\
\qquad\quad \mathit{Const}\ n \rightarrow \mathit{Num}^{S}\ (\textbf{lift}\ n) \\
\qquad\quad \mathit{Lam}\ x\ e \rightarrow \mathit{Fun}^{S}\ (\lambda^{D} v. \\
\qquad\qquad\qquad \textbf{let}^{S}\ \mathit{env}' = \mathit{bind}\ @^{S}\ x\ @^{S}\ (M^{S}\ v)\ @^{S}\ \mathit{env} \\
\qquad\qquad\qquad \textbf{in}\ \mathit{eval}\ @^{S}\ \mathit{env}'\ @^{S}\ e) \\
\qquad\quad \mathit{App}\ e_1\ e_2 \rightarrow \textbf{case}^{S}\ (\mathit{eval}\ @^{S}\ \mathit{env}\ @^{S}\ e_1)\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Fun}\ f \rightarrow f\ @^{D}\ (\mathit{eval}\ @^{S}\ \mathit{env}\ @^{S}\ e_2) \\
\qquad\quad \mathit{Let}\ x\ e_1\ e_2 \rightarrow \textbf{let}^{D}\ v = P^{S}\ (\textbf{poly}\ \mathit{eval}\ @^{S}\ \mathit{env}\ @^{S}\ e_1) \\
\qquad\qquad\qquad \textbf{in}\ \textbf{let}^{S}\ \mathit{env}' = \mathit{bind}\ @^{S}\ x\ @^{S}\ v\ @^{S}\ \mathit{env} \\
\qquad\qquad\qquad\quad \textbf{in}\ \mathit{eval}\ @^{S}\ \mathit{env}'\ @^{S}\ e_2) \\
\textbf{in} \\
\textbf{let}^{S}\ \mathit{meval} = \mathit{preeval}\ @^{S}\ (\lambda^{S} x.M^{S}\ \mathit{Wrong}^{S}) \\
\textbf{in}\ \langle \ldots \rangle
\end{array}
$$

Figure 4.3: A monomorphizer for lambda-calculus.

EXAMPLE 4.6. The result of specializing the expression

$$\begin{array}{l} meval\ @^{S}(Let^{S}\ \texttt{'i'}^{S}\ (Lam^{S}\ \texttt{'x'}^{S}\ (Var^{S}\ \texttt{'x'}^{S})) \\ \qquad (App^{S}(App^{S}\ (Var^{S}\ \texttt{'i'}^{S})\ (Var^{S}\ \texttt{'i'}^{S})) \\ \qquad\quad (Const^{S}\ 0^{S}))) \end{array}$$

is the expression

$$\begin{array}{l} \mathbf{let}\ v = (\lambda v'.v', \lambda v'.v') \\ \ \mathbf{in}\ (\mathbf{fst}\ v\ (\mathbf{snd}\ v))\ 0 \\ \ ::\ Num\ Int \end{array}$$

Observe that the residual of the let-bound identity function is a pair of functions: the type of the first one is

$$Fun\ (\,(Fun\ (Num\ Int \to Num\ Int))\ \to\ (Fun\ (Num\ Int \to Num\ Int))$$

and that of the second one is $Fun\ (Num\ Int \to Num\ Int)$. Also observe the projections appearing on every use of the residual of the identity function (v), choosing the corresponding monomorphic version.

This is a very powerful transformation, not possible with other program specialization techniques.

4.4 Limitations of Type Specialization

However, Type Specialization, like every other method, has limitations and problems. We review some of them here, explaining them through examples.

4.4.1 Lack of principality

We remarked that when the context provides enough information, the specialization can proceed with no problems (Example 3.9-2, Example 3.9-3, Example 3.12). But, what happens when specializing any of the expressions in the following example?

EXAMPLE 4.7. Observe that in all cases there is some static information missing.

1. $\lambda^{D}x.x +^{S} 1^{S}\ : Int^{S} \to^{D} Int^{S}$
2. $\mathbf{poly}\ (\lambda^{D}x.\mathbf{lift}\ x +^{D} 1^{D}) : \mathbf{poly}\ (Int^{S} \to^{D} Int^{D})$
3. $\lambda^{D}f.\mathbf{spec}\ f\ @^{D}\ 13^{S}\ : \mathbf{poly}\ (Int^{S} \to^{D} Int^{D}) \to^{D} Int^{D}$

All have many different *unrelated* specializations! For example, the function in Example 4.7-1 has one specialization for each possible value for x — in particular, $\lambda x'.\bullet : \hat{n} \to \hat{n}'$, for every value of n and n' such that $n' = n + 1$. If this function appears in one module, but is applied in another one, then the specialization should wait until the value n of the argument is known, in order to decide the residual type. The same

problem appears in the case of polyvariance, as can be observed in Example 4.7-2 and Example 4.7-3: the generation of the tuple or the selection of the right projection should be deferred until all the information is available — in the second case, if the expression is annotated as polyvariant, the right projection can even be a different one on each application!

Another example with a similar problem is the following one.

EXAMPLE 4.8. Consider the term

$$\begin{array}{rl} e_f = & \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) \\ & \ \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S, \mathbf{spec}\ f)^D \end{array}$$

Observe that there is not enough information to determine which is the right specialization for the third component of the tuple! As a consequence, we have several different specializations for this term; in particular, all the three specializations shown differ in the residual assigned to that component.

1. $\vdash e_f : (Int^D, Int^D, Int^S \to^D Int^D)^D$
   $\hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.42 + 1, \lambda x'.17 + 1)$
   $\ \mathbf{in}\ (\mathbf{fst}\ f'@\bullet, \mathbf{snd}\ f'@\bullet, \mathbf{snd}\ f') : (Int, Int, \hat{17} \to Int)$

2. $\vdash e_f : (Int^D, Int^D, Int^S \to^D Int^D)^D$
   $\hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.42 + 1, \lambda x'.17 + 1)$
   $\ \mathbf{in}\ (\mathbf{fst}\ f'@\bullet, \mathbf{snd}\ f'@\bullet, \mathbf{fst}\ f') : (Int, Int, \hat{42} \to Int)$

3. $\vdash e_f : (Int^D, Int^D, Int^S \to^D Int^D)^D$
   $\hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.42 + 1, \lambda x'.17 + 1, \lambda x'.n + 1)$
   $\ \mathbf{in}\ (\pi_{1,3}\ f'@\bullet, \pi_{2,3}\ f'@\bullet, \pi_{3,3}\ f') : (Int, Int, \hat{n} \to Int)$

The first two are not general enough, because they force the unknown number to be either 42 or 17, respectively. And the third, while general, is not optimal in the case when the value of n is 42 or 17, because two copies of the same function have to be generated.

Fixing this problem requires a big change in the residual language.

4.4.2 Failure

We have seen that the specialization of a term fails when two different residual types have to be unified, and that this is an important feature that allows us to express the failure of typechecking by failure of the specialization.

When the failure is inside a polyvariant expression, it cannot be detected until some **spec** of it is calculated. Consider the following example.

EXAMPLE 4.9. Observe that the **poly** is never specialized (its residual is a 0-tuple), and thus the specialization succeeds.

$$\begin{array}{l}\vdash \textbf{let}^{D}\ f = \textbf{poly}\ (\textbf{let}^{D}\ id = \lambda^{D}x.x\ \textbf{in}\ (id\ @^{D}\ 1^{S}, id\ @^{D}\ 2^{S})^{D})\\ \quad \textbf{in}\ 2^{D}\\ \ : Int^{D} \hookrightarrow \textbf{let}\ f = ()\ \textbf{in}\ 2 : Int\end{array}$$

But if we try to specialize the body of the **poly**, a failure will occur because $\hat{1}$ is not equal to $\hat{2}$.

In the interpreters of Figures 4.2 and 4.3 the case of unbound variables in object programs was handled with a special constructor $Wrong^{S}$, which specializes to $\bullet$: $Wrong$. The consequence of this can be seen in the following example.

EXAMPLE 4.10. Observe that the argument of *teval* represents the term $\lambda x.y$.

$$\vdash teval\ @^{S}(Lam^{S}\ \texttt{'x'}^{S}\ (Var^{S}\ \texttt{'y'}^{S})) : Value^{S} \hookrightarrow \lambda v.\bullet : Fun\ (Num\ Int \to Wrong)$$

The residual program is not a failure, but a function delivering an element statically known to be $Wrong$! To make the type specializer fail in this case, we can change the $Value^{S}$ element $Wrong^{S}$ in the initial environment for some computation with no specialization; but in that case the specialization failure will have a very strange error message!

To have the possibility of producing a failure in specialization with a proper error message, a simple extension may be the addition of a (specialization) failure primitive. This can be represented using a static failing computation — see Section 9.1.

4.4.3 Interaction between polyvariance and recursion

We have said that the interaction between dynamic recursion and polyvariance introduces the most challenging problems; we explain what kind of problems are involved with some examples.

The first example shows that when specializing a dynamic recursive function, some care is needed to avoid a loop during specialization. In this case, the argument of f can only take a finite number of static values, and thus a recursive program can be generated.

EXAMPLE 4.11. The source program

$$\begin{array}{l}\textbf{let}^{S}\ not = \lambda^{S}\ b.\textbf{if}^{S}\ b\ \textbf{then}\ False^{S}\ \textbf{else}\ True^{S}\\ \ \textbf{in}\\ \ \textbf{let}^{D}\ f = \textbf{fix}^{D}\ (\lambda^{D} f.\textbf{poly}\ (\lambda^{D}\ b.\textbf{spec}\ f\ (not\ @^{S}\ b)))\\ \quad \textbf{in spec}\ f\ True^{S}\end{array}$$

specializes to

$$\begin{array}{l}\textbf{let}\ f = \textbf{fix}\ (\lambda f.(\lambda b.\textbf{snd}\ f\ \bullet, \lambda b.\textbf{fst}\ f\ \bullet))\\ \ \textbf{in fst}\ f\ \bullet\end{array}$$

It can be argued that the program is non-terminating, but in a lazy language a program producing an infinite list of alternating values will have the same structure, so it is worth considering it.

To find the correct specialization in Example 4.11, an algorithm should perform some kind of memoization: when calculating new polyvariant versions it is important not to calculate one already seen or else the specialization will loop! This is much clearer to see with the formulation presented in Chapters 6 and 9 — see Example 9.15.

The second example shows that identifying when two recursive calls are the same — which we have shown is very important to achieve termination — may imply comparing types with missing information.

EXAMPLE 4.12. The source program

$$
\begin{array}{l}
\mathbf{let}^{D}\ f = \mathbf{fix}^{D}\ (\lambda^{D} f.\mathbf{poly}\ (\lambda^{D} x.\lambda^{D} y. \\
\qquad\qquad\qquad\qquad \mathbf{if}^{D}\ \mathbf{lift}\ x ==^{D} 0^{D} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ x +^{S} 1^{S} \\
\qquad\qquad\qquad\qquad \mathbf{else\ spec}\ f\ @^{D}\ x\ @^{D}\ y +^{S} 0^{S}\,)) \\
\ \mathbf{in}\ \lambda^{D} z.\mathbf{spec}\ f\ @^{D}\ 3^{S}\ @^{D}\ z \\
\ : Int^{S} \to^{D} Int^{S}
\end{array}
$$

specializes to

$$
\begin{array}{l}
\mathbf{let}\ f = \mathbf{fix}\ (\lambda f.(\lambda x.\lambda y. \\
\qquad\qquad\qquad \mathbf{if}\ \ 3 == 0 \\
\qquad\qquad\qquad \mathbf{then}\ \bullet \\
\qquad\qquad\qquad \mathbf{else}\ f @ x @ \bullet)) \\
\ \mathbf{in}\ \lambda z.f @ \bullet @ z \\
\ : \hat{n} \to \hat{4}
\end{array}
$$

for all values of n. Observe that the recursive call to f is invoked with the same values, and so there is no need for more than one specialization of the polyvariant recursive function; however, realizing that this is the case can only be done by comparing the residual type of y with the residual type of $y +^{S} 0^{S}$, which depends on an unknown number. In this case it is easy to see that both are the same, but this is not decidable in general.

Another version of the same problem is presented in the next example.

EXAMPLE 4.13. The source program

$$
\begin{array}{l}
\mathbf{let}^{D}\ f = \mathbf{fix}^{D}\ (\lambda^{D} f.\mathbf{poly}\ (\lambda^{D} x.\lambda^{D} ys. \\
\qquad\qquad\qquad\qquad \mathbf{if}^{D}\ \mathbf{lift}\ x ==^{D} 0^{D} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ x +^{S} 1^{S} \\
\qquad\qquad\qquad\qquad \mathbf{else\ spec}\ f\ @^{D}\ x\ @^{D}\ \mathbf{snd}^{D}\ ys)) \\
\ \mathbf{in}\ \lambda^{D} zs.\mathbf{spec}\ f\ @^{D}\ 3^{S}\ @^{D}\ zs \\
\ ::\ \alpha \to^{D} Int^{S} \\
\quad \text{where}\ \ \alpha = (Int^{S}, \alpha)^{D}
\end{array}
$$

specializes to

$$
\begin{array}{l}
\textbf{let}\ f = \textbf{fix}\ (\lambda f.(\lambda x.\lambda ys. \\
\qquad\qquad\qquad \textbf{if}\ \ 3 == 0 \\
\qquad\qquad\qquad\ \textbf{then}\ \bullet \\
\qquad\qquad\qquad\ \textbf{else}\ f@x@\textbf{snd}\ ys)) \\
\ \textbf{in}\ \lambda zs.f@\bullet@zs \\
\ ::\ \alpha \rightarrow \hat{4} \\
\quad \text{where}\ \ \alpha = (\hat{n}, \alpha)
\end{array}
$$

for all values of n. In this case, the recursive call to f invokes itself with almost the same arguments: it takes the 'tail' of the argument 'list' (represented as a recursive pair). Again there is no need for more than one specialization of the polyvariant recursive function; however, identifying that this is the case involves taking the decision that the infinite 'list' has all identical elements.

There are also other possible solutions; for example, the following term is also obtainable by specialization from the same source code.

$$
\begin{array}{l}
\textbf{let}\ f = \textbf{fix}\ (\lambda f.(\lambda x.\lambda ys.\textbf{if}\ 3 == 0\ \textbf{then}\ \bullet\ \textbf{else}\ \textbf{snd}\ f@x@\textbf{snd}\ ys), \\
\qquad\qquad\qquad \lambda x.\lambda ys.\textbf{if}\ 3 == 0\ \textbf{then}\ \bullet\ \textbf{else}\ \textbf{fst}\ f@x@\textbf{snd}\ ys)) \\
\ \textbf{in}\ \lambda zs.\textbf{fst}\ f@\bullet@zs \\
\ ::\ \alpha \rightarrow \hat{4} \\
\quad \text{where}\ \ \alpha = (\hat{n}, \beta), \\
\qquad\qquad\quad \beta = (\hat{m}, \alpha)
\end{array}
$$

for any value of n and m. In this case, instead of all identical elements, we have decided that the infinite list has two alternating values (observe that the only difference between both components of the residual of the recursive function's body is in their types: one uses α and the other uses β).

Other combinations are also possible: any choice of values that makes ys have a finite number of static values in a repeated pattern is a possible specialization for this code. For that reason, it is difficult to decide what the correct behaviour of the algorithm should be in this case.

Our final example fails to specialize. Its purpose is to illustrate new problems introduced by possible solutions to the previous problems, as discussed after the example.

EXAMPLE 4.14. The source program

$$
\begin{array}{l}
\textbf{let}^{D}\ id = \lambda^{D} z.z \\
\ \textbf{in} \\
\ \textbf{let}^{D}\ f = \textbf{fix}^{D}\ (\lambda^{D} f. \\
\qquad\qquad\qquad \textbf{poly}\ (\lambda^{D} b.\lambda^{D} x.\lambda^{D} y. \\
\qquad\qquad\qquad\qquad \textbf{if}^{D}\ \textbf{lift}\ b \\
\qquad\qquad\qquad\qquad\ \textbf{then}\ \textbf{lift}\ (id\ @^{D} x +^{S} id\ @^{D} y) \\
\qquad\qquad\qquad\qquad\ \textbf{else}\ \textbf{spec}\ f\ @^{D} b\ @^{D} x\ @^{D} y)) \\
\quad \textbf{in}\ \lambda^{D} b'.(\textbf{spec}\ f\ @^{D} b'\ @^{D} 2^{S}\ @^{D} 2^{S}, \textbf{spec}\ f\ @^{D} b'\ @^{D} 3^{S}\ @^{D} 3^{S})
\end{array}
$$

cannot be specialized. The problem is the use of the monovariant function id: as it is applied to both arguments of the function f, they have both to be the same *in every call to* f! As f is invoked with 2 and 3 as arguments, the code cannot be specialized.

All these examples present some kind of problem to an algorithm performing specialization. The one presented by Hughes [1996b] solved some of these problems with different degrees of success. In the case of Example 4.7, it simply fails with an error message; after all, it is assumed that all the static information is present, and so it is sensible to fail if this is not the case. For the other cases, it uses an aggressive technique for memoization: two specializations of the same polyvariant expression are identified if it is possible to do so, even when it may not be the case that they are equal; if later an error is encountered, the algorithm backtracks and tries with those two specializations as different. This involves choosing one particular specialization between all the possible ones for a given source term.

In Hughes' implementation, the body of a **poly** is not specialized at all until a **spec** is encountered. In the cases of Examples 4.12 and 4.13 it would mean that to determine if different **spec**s are the same, both should be specialized! The technique of memoization used helps in this case, identifying the second **spec** with the first one before proceeding with its specialization.

The aggressive identification of **spec**s also helps in Examples 4.8 and 4.11: the algorithm attempts a solution guessing that the unknown **spec**s are one of the known ones, and succeeds. That means choosing one of the first two alternatives in Example 4.8; in case a failure occurs later, the backtracking mechanism is used to calculate the correct solution.

But for Example 4.14, backtracking constitutes a problem. In fact, the "guessing" that identifies **spec**s is correct, but when the error unifying $\hat{2}$ with $\hat{3}$ is found, backtracking causes a loop, generating more and more different specializations of the recursive function. Hughes [1996b] argued that better control of the backtracking mechanism (some kind of dependency-directed backtracking) is necessary to solve this problem.

4.5 Summary of Type Specialization

We have presented type specialization, a technique for program specialization that produces specialized versions of both programs and types. Every good specializer is able to produce arbitrary programs, but the novel feature of type specialization is that it is also able to produce arbitrary *types*. This allows more expressiveness, and enables solving the problem of optimal specialization for interpreters written in typed languages — as shown in Figure 4.2 — through performing type inference by the specialization process.

Type specialization combines several powerful features to program specialization. For example, traditional partial evaluation, polyvariance, constructor specialization, firstification (or closure conversion), some forms of data specialization, arity raising, and all these for a higher-order language — it is remarkable that type specialization is the first program specialization approach that supports both constructor specialization and higher-order functions.

There is still a long way to go before type specialization can be as successfully applied as partial evaluation because several problems remain to be solved: the generation of polymorphic programs; the treatment of polymorphic source code; non-termination produced by combining polyvariance and recursion; self-application; a semi-automatic tool to help annotating a program with binding-times suitable for type specialization;

modular specialization; and the elimination of dead code and code duplication.

In the rest of this thesis we contribute to the development of type specialization by solving the first of them, and by providing a richer way of presenting type specialization, such that the rest of the problems can be stated and treated more easily.

Part II

Principal Type Specialization

Chapter 5

Theory of Qualified Types

> **Q:** *How do you kill a purple elephant?*
>
> **A:** *With a purple elephant gun.*
>
> ...
>
> Popular joke (first part)

The theory of qualified types [Jones, 1994a] is a framework that allows the development of constrained type systems in an intermediate level between monomorphic and polymorphic type disciplines. We have made a thorough use of the techniques developed by Mark Jones, by reformulating type specialization as a system with qualified types (Chapter 6). For that reason, we consider it valuable to describe the basics of this theory first.

Qualified types can be seen in two ways: either as a restricted form of polymorphism, or as an extension of the use of monotypes (commonly described as *overloading*, in which a function may have different interpretations according to the types of its arguments or its result). Predicates are used to restrict the use of type variables, or, using the second point of view, to express several possible different instances with one single type (but without the full generality of parametric polymorphism). The theory explains how to enrich types with predicates, how to perform type inference using the enriched types, and which are the minimal properties predicates must satisfy in order for the resulting type system to have similar properties as in the Hindley-Milner one. In particular, it has been shown that any well typed program has a *principal type* that can be calculated by an extended version of Milner's algorithm.

The material presented in this chapter is based completely on the presentation of the theory given by Jones [1994a], with the only exception that the names of the syntactic categories were changed to fit in the type specialization system. In particular, we use similar notational conventions to those used by Mark Jones. Those conventions can be a bit confusing at first glance, but they are extremely useful once mastered. After describing the notational conventions (Section 5.1), we review the use of predicates (Section 5.2) and basic type inference for qualified types (Section 5.4). One of the most important technical concepts in the theory of qualified types is that of *evidence*, and the associated notion of *conversion*; in Section 5.5 we discuss the use of *evidence* as a technique to provide coherence to terms with qualified types, and also how type inference can be enriched with evidence. We conclude with a discussion about simplification of predicates generated by the inference process (Section 5.6) and a summary of the ideas.

5.1 On Notational Conventions

In technical reasoning, it is sometimes convenient to blur some notions behind the notation. Jones [1994a] uses several of these conventions, in particular regarding lists of elements, lists of pairs, the usual operations on lists, and their use on type expressions. Despite the possible source of confusion that these conventions may be for a casual reader, they can be easily mastered with very little practice.

The first convention blurs the notions of sets, lists, and individual elements. Using the conventions in Notation 5.1, the use of $\emptyset$ and $_,_$ can be reformulated to either sets or lists without needing to change the definition of rules that use them.

NOTATION 5.1. Let L and L' be any kind of (finite) lists or sets, and l an element.

We write L, L' for the result of the *union* of sets L and L' or the *append* of lists L and L'. We write l, L for the result of the inclusion of element l to set L or the *cons* of l to list L. Finally, we write $\emptyset$ for the empty set or list, and assume that $\emptyset, L = L, \emptyset = L$. Observe that l can be used as a singleton list, when the context requires a list.

Another point where blurring is convenient is when working with different kinds of quantification; there are two possible ways to add quantifiers to the basic syntax: one by one, or all together in a set. For example, in the case of type schemes, this amounts to define either that $\sigma ::= \forall\alpha.\sigma \mid \tau$ or $\sigma ::= \forall\{\alpha_i\}_{i\in 1\ldots n}.\tau$, respectively. We use the first form as the definition and the second as an abbreviation.

NOTATION 5.2. Let $\sigma = \forall\{\alpha_i\}_{i\in 1\ldots n}.\tau$ be a type scheme. We use the shorter form $\forall\{\alpha_i\}.\tau$ or even $\forall\alpha_i.\tau$ for σ. We also use the notation $\forall\beta.\sigma$ as an abbreviation for $\forall\beta, \alpha_i.\tau$. Finally, we identify $\forall\emptyset.\tau$ with τ.

This is consistent with the blurring between lists and sets defined by Notation 5.1.

The convention for quantification is used for several kinds of expressions. The table in Notation 5.3 summarizes them (the syntax for each kind of expression is introduced in Chapter 6).

NOTATION 5.3. Assuming the following equalities, $\Delta = \delta_1, \cdots, \delta_m$, $\alpha = \alpha_1, \cdots, \alpha_m$, $h = h_1, \cdots, h_m$, and $v = v_1, \cdots, v_m$, we use the abbreviations:

Object	Expression	Abbreviation(s)
Qualified type	$\delta_1 \Rightarrow \cdots \delta_m \Rightarrow \tau'$	$\Delta \Rightarrow \tau'$
Type scheme	$\forall\alpha_1.\cdots\forall\alpha_m.\rho$	$\forall\alpha.\rho$
Evidence abstr.	$\Lambda h_1.\cdots\Lambda h_m.e'$	$\Lambda h.e'$
Evidence app.	$((e'(\!(v_1)\!))\cdots)(\!(v_m)\!)$	$e'(\!(v)\!)$

In the special case when $m = 0$, all the sequences are empty, and then the abbreviations stand for the enclosed element (e.g. $e'(\!(v)\!)$ represents e'). This implies, for example, that a type τ can be understood as a qualified type ($\emptyset \Rightarrow \tau$) or a type scheme ($\forall\emptyset.\emptyset \Rightarrow \tau$) depending on the context of use.

Another convention is concerned with lists of pairs.

NOTATION 5.4. Lists of pairs may be abbreviated by a pair of lists in the following way. If $h = h_1, \ldots, h_n$ and $\Delta = \delta_1, \ldots, \delta_n$, the list $h_1 : \delta_1, \ldots, h_n : \delta_n$ may be abbreviated as $h : \Delta$ or as Δ depending on the context. The latter is also used for a list of predicates — no explicit removal of the variables (first components of pairs) will be used.

The union (concatenation) of two sets (lists) of pairs $h : \Delta$ and $h' : \Delta'$ will be written $h : \Delta, h' : \Delta'$ (as an alternative to $h, h' : \Delta, \Delta'$, which may also be used).

Again this is consistent with the blurring between lists and individual elements.

Finally, we should mention substitutions. Substitutions are functions from type (scheme) variables to types (type schemes). We use two notations for substitutions. The first one, used for substitutions changing only a finite number of variables, emphasizes the individual replacement of variables, while the second one emphasises the functional nature of substitutions.

NOTATION 5.5. If $\alpha = \alpha_1, \cdots, \alpha_n$ are variables, and $\tau = \tau_1, \cdots, \tau_n$ are types, then $S = _[\alpha_i/\tau_i]$ is the substitution mapping each of the α_i to the corresponding τ_i. When applied to a particular type τ', it will be written $S\,\tau'$ or $\tau'[\alpha_i/\tau_i]$ (and sometimes $\tau'[\alpha/\tau]$ or $\tau'[\alpha_i/\tau_i]$ identifying the lists with elements).

5.2 Predicates

Polymorphism is the ability to treat some terms as having many different types. We can express a polymorphic type by means of a *type scheme* [Damas and Milner, 1982], using universal quantification to abstract those parts of a type that may vary. That is, if $f(t)$ is a type for every possible value of type variable t, then giving the type scheme $\forall t.f(t)$ to a term means that that term can receive any of the types in the set

$$\{f(\tau) \text{ s.t. } \tau \text{ is a type.}\}$$

But sometimes that is not enough: not all the types can replace t and still express a possible type for the term. For those cases, a form of restricted quantification can be used. If $P(t)$ is a predicate on types, we use the type scheme $\forall t.P(t) \Rightarrow f(t)$ to represent the set of types

$$\{f(\tau) \text{ s.t. } \tau \text{ is a type such that } P(\tau) \text{ holds.}\}$$

and accurately reflect the desired types for a given term.

One important property of a type system is that of type inference: given a term without any information of typing, infer a type for it that reflects the ways in which the term may be used. The *theory of qualified types* [Jones, 1994a] describes how to perform type inference in presence of type schemes with restricted quantification (*qualified types*).

The key feature in the theory is the use of a language of *predicates* to describe sets of types (or, more generally, relations between types). The exact set of predicates may vary from one application to another, but the theory effectively captures the minimum required properties by using an entailment relation ($\Vdash$) between (finite) sets of predicates satisfying a few simple laws. If Δ is a set of predicates, then $\Delta \Vdash \{\delta\}$ (also written as

$$\text{(Fst)} \quad \Delta, \Delta' \Vdash \Delta$$

$$\text{(Snd)} \quad \Delta, \Delta' \Vdash \Delta'$$

$$\text{(Univ)} \quad \frac{\Delta \Vdash \Delta' \qquad \Delta \Vdash \Delta''}{\Delta \Vdash \Delta', \Delta''}$$

$$\text{(Trans)} \quad \frac{\Delta \Vdash \Delta' \qquad \Delta' \Vdash \Delta''}{\Delta \Vdash \Delta''}$$

$$\text{(Close)} \quad \frac{\Delta \Vdash \Delta'}{S\,\Delta \Vdash S\,\Delta'}$$

Figure 5.1: Structural laws satisfied by entailment.

$\Delta \Vdash \delta$ by virtue of Notation 5.1) indicates that the predicate δ can be inferred from the predicates in Δ. This can be generalized to bigger sets by using the following property:

$$\Delta \Vdash \Delta' \text{ iff for all } \delta \in \Delta',\ \Delta \Vdash \delta$$

which will be used implicitly in the definitions; in this way, it is only necessary to describe the rules for entailments of the form $\Delta \Vdash \delta$.

The basic properties that entailment must satisfy are:

Monotonicity: $\Delta \Vdash \Delta'$ whenever $\Delta \supseteq \Delta'$

Transitivity: if $\Delta \Vdash \Delta'$ and $\Delta' \Vdash \Delta''$, then $\Delta \Vdash \Delta''$

Closure property: if $\Delta \Vdash \Delta'$, then $S\Delta \Vdash S\Delta'$.

The last condition is needed to ensure that the system of predicates is compatible with the use of parametric polymorphism. This properties can be expressed by a system containing the rules in Figure 5.1; every particular use of the theory should provide additional rules to capture the exact relation between predicates.

5.3 An Example: the Haskell Class System

The technical aspects of the theory of qualified types will be of central importance in the next chapter, where we present our approach to type specialization. For that reason, it is important to see an example in order to understand the role of predicates.

Mark Jones' main motivation to study qualified types was formalizing the Haskell class system, so new developments could be done, and several alternatives could be tested. Additionally, Haskell classes are pretty well understood nowadays by the functional programming community, and we assume that the reader is familiar with the concept — although we provide a short description of it, it will be far from optimal if lacking some familiarity. For those reasons, we think that presenting how to express classes by using predicates is a good way to clarify this theory.

In Haskell, the programmer has the possibility to declare *type classes*, that can be understood as families of types — the *instances* of a given class — having particular *member functions*. The declarations in Haskell allow to declare also the dependence of a given type class from certain others — the *superclasses*. Type inference will use this information to restrict the use of polymorphism when a function that is member of certain class is used (an overloaded use of it), and these restrictions will be propagated to other functions using the overloaded one, making them also overloaded. Let's see a brief example taken from the Haskell prelude: the classes for equality and ordering (although we simplify them a bit for the purposes of this presentation).

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<) :: a -> a -> Bool
```

The first of these declarations establishes that to belong to the family denoted by the class `Eq` — written in the predicate form `Eq a` — a type `a` must have a function named `(==)`, with the correct type. The second one establishes that to belong to the class `Ord`, a type must belong first to the `Eq` class, and must have a function named `(<)`, with the correct type. In the latter case, we say that `Eq` is a superclass of `Ord`.

Instance declarations are used to declare that a particular type belongs to a class. Continuing with our example,

```
instance Eq Int where
  (==) = primEqInt

instance Eq Char where
  c == c' = ord c == ord c'

instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x==y && xs==ys
  _      == _      = False
```

The first two declarations establish that `Int` and `Char` are instances of `Eq`, and define the implementation of the corresponding functions `(==) :: Int -> Int -> Bool`, and `(==) :: Char -> Char -> Bool` — the former as a built-in primitive, and the latter based on that of `Int`. The third one does the same for lists `[a]`, but also asks the condition that the type `a` belongs to the `Eq` class to provide the instance of `[a]`.

In the formalization of this system using qualified types, the assertion that a given type belongs to a class is used as a predicate — for example, `Eq a` is a predicate — and the information provided by the class and instance declarations will be captured by a global context D, called a *type class environment*, containing two kind of terms:

- *Class* $(\Delta \Rightarrow \delta)$, corresponding to class declarations, where each of the classes in Δ is a superclass of δ.

- *Inst* $(\Delta \Rightarrow \delta)$, corresponding to instance declarations, where if there is an instance for every predicate in Δ, then there is an instance for δ.

For example, all the Haskell declarations given previously can be expressed by the type class environment

$$\begin{array}{l}\{Class\ (\{\} \Rightarrow Eq\ a),\\ \ Class\ (\{Eq\ a\} \Rightarrow Ord\ a),\\ \ Inst\ (\{\} \Rightarrow Eq\ Int),\\ \ Inst\ (\{\} \Rightarrow Eq\ Char),\\ \ Inst\ (\{Eq\ a\} \Rightarrow Eq\ [a])\}\end{array}$$

Once the type class environment D for a given program has been determined, it is possible to define the particular rules for entailment — these rules, together with those in Figure 5.1 will establish the relation.

$$\text{(Super)}\ \frac{\Delta \Vdash \delta \qquad Class\ (\Delta' \Rightarrow \delta) \in D \qquad \delta' \in \Delta'}{\Delta \Vdash \delta'}$$

$$\text{(Inst)}\ \frac{\Delta \Vdash \Delta' \qquad Inst\ (\Delta' \Rightarrow \delta) \in D}{\Delta \Vdash \delta}$$

The first of the rules establishes the conditions to obtain information from superclass declarations — basically, it says that if a particular class declaration δ can be deduced in a context Δ, then all its superclasses δ' are also deducible in that context. This is in accordance with the superclasses declarations: to declare a particular class, all the superclasses have to be declared first. The second rule is used to obtain information for instance declarations — it establishes that if all the conditions Δ' needed to prove a particular instance δ can be proved in a given context Δ, then the instance δ is provable as well. This is in accordance with instance declarations: if all the member functions for the required classes have been constructed, then the member functions for the particular instance can be constructed as well.

With this entailment relation, type inference in Haskell proceeds as described in the following sections.

5.4 Type Inference with Qualified Types

In the theory of qualified types, the language of types and type schemes is stratified in a similar way as in the Hindley-Milner system, where the most important restriction is that qualified or polymorphic types cannot be arguments of functions. That is, types (written using the symbol τ) are defined by a grammar with at least these productions $\tau\ ::=\ t \mid \tau \rightarrow \tau$; on top of types are constructed qualified types of the form $\Delta \Rightarrow \tau$ (written using the symbol ρ), and then type schemes of the form $\forall\{\alpha_i\}.\rho$ (written using the symbol σ). We use freely the conventions defined in Notation 5.3. Using that notation, any type scheme can be written in the form $\forall\alpha_i.\Delta \Rightarrow \tau$, representing the set of qualified types

$$\{\Delta[\alpha_i/\tau_i] \Rightarrow \tau[\alpha_i/\tau_i]\ s.t.\ \tau_i \text{ is a type}\}$$

$$\text{(VAR)}\quad \frac{x : \tau \in \Gamma}{\Delta \mid \Gamma \vdash x : \tau}$$

$$\text{(LAM)}\quad \frac{\Delta \mid \Gamma_x, x : \tau_2 \vdash e : \tau_1}{\Delta \mid \Gamma \vdash \lambda x.e : \tau_2 \to \tau_1}$$

$$\text{(APP)}\quad \frac{\Delta \mid \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Delta \mid \Gamma \vdash e_2 : \tau_2}{\Delta \mid \Gamma \vdash e_1 @ e_2 : \tau_1}$$

$$\text{(LET)}\quad \frac{\Delta \mid \Gamma \vdash e_2 : \sigma \qquad \Delta \mid \Gamma, x : \sigma \vdash e_1 : \tau}{\Delta \mid \Gamma \vdash \mathbf{let}\ x = e_2\ \mathbf{in}\ e_1 : \tau}$$

$$\text{(QIN)}\quad \frac{\Delta, \delta \mid \Gamma \vdash e : \rho}{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \mid \Gamma \vdash e : \rho}$$

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash e : \sigma}{\Delta \mid \Gamma \vdash e : \forall \alpha.\sigma}\ (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

$$\text{(INST)}\quad \frac{\Delta \mid \Gamma \vdash e : \forall \alpha.\sigma}{\Delta \mid \Gamma \vdash e : S\,\sigma}\ (\mathrm{dom}(S) = \alpha)$$

Figure 5.2: Typing rules for OML.

Observe that here Δ is taken as a list of predicates — this will be important later, when evidence is considered (see Section 5.5).

The language of terms — written using the symbol e — is based on the untyped λ-calculus (it has, at least, variables, applications, abstractions, and the **let** construct); it is called OML, abbreviating 'Overloaded ML'. Type inference uses judgements extended with a context of predicates

$$\Delta \mid \Gamma \vdash e : \sigma$$

representing the fact that when the predicates in Δ are satisfied, and the types of the free variables of e are as specified by Γ, then the term e has type σ. Valid typing judgements can be derived using a system of rules specifying typing derivations — see Figure 5.2. The notation Γ_x is used to indicate the environment Γ without the association for x. The interesting rules (those actually involving the predicate set Δ) are (QIN) and (QOUT), that move predicates in to or out of the type of an object, and (GEN), that allows polymorphism. In particular, rule (QIN) can be used to maximize the opportunities to use rule (GEN).

For the examples concerning Haskell in the rest of this chapter, we consider the system as defined in Section 5.3.

EXAMPLE 5.6. Consider the Haskell declaration

```
member x [] = False
member x (y:ys) = x == y || member x ys
```

The following type inference judgement will hold

$$\emptyset \mid \emptyset \vdash \texttt{member} : \forall a. Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Observe how the use of the overloaded function `(==)` in the body of the function `member` is reflected in the predicate qualifying the variable a in the resulting type.

To find all the ways in which a particular e can be used within a given Γ, one should consider all the judgements of the form $\Delta \mid \Gamma \vdash e : \sigma$, that is, all the pairs of Δ and σ such that the typing judgement holds — observe that some predicates may have been introduced in the type scheme σ while some others may remain in the context Δ. This motivates the definition and study of *constrained type schemes*, written $(\Delta \mid \sigma)$.

Definition 5.7. A *constrained type scheme* is an expression of the form $(\Delta \mid \sigma)$ where Δ is a set of predicates and σ is a type scheme.

Thus, given an expression e and and assignment Γ, the theory has to deal with sets of the form

$$\{(\Delta \mid \sigma) \text{ s.t. } (\Delta \mid \Gamma \vdash e : \sigma)\}$$

The main tool used to deal with these sets is a preorder $\geq$ — pronounced *more general* — defined on pairs of constrained type schemes, and whose intended meaning is that if $(\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ then it is possible to use an object that can be treated as having type σ in an environment satisfying the predicates in Δ whenever an object of type σ' is required in an environment satisfying the predicates in Δ'. To define it formally, the notion of *generic instance* is needed.

Definition 5.8. A *qualified type* $\Delta_\tau \Rightarrow \tau$ is a *generic instance* of the constrained type scheme $(\Delta \mid \forall \alpha_i.\Delta' \Rightarrow \tau')$ if there are types τ_i such that

$$\Delta_\tau \Vdash \Delta, \Delta'[\alpha_i/\tau_i] \text{ and } \tau = \tau'[\alpha_i/\tau_i]$$

In particular, a qualified type $\Delta \Rightarrow \tau$ is instance of another qualified type $\Delta' \Rightarrow \tau'$ if and only if $\Delta \Vdash \Delta'$ and $\tau = \tau'$. Also, any constrained type scheme has at least one generic instance: if the constrained type scheme is $(\Delta \mid \forall \alpha_i.\Delta' \Rightarrow \tau)$ then for any types τ_i it holds that $\Delta, \Delta'[\alpha_i/\tau_i] \Rightarrow \tau'[\alpha_i/\tau_i]$ is a generic instance of it.

Now we are in position to define the "more general" ordering $(\geq)$ on constrained type schemes.

Definition 5.9. The constrained type scheme $(\Delta \mid \sigma)$ is said to be *more general* than the constrained type scheme $(\Delta' \mid \sigma')$, written $(\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, if every generic instance of $(\Delta' \mid \sigma')$ is a generic instance of $(\Delta \mid \sigma)$.

Because every type scheme σ is equivalent to a constrained type scheme of the form $(\emptyset \mid \sigma)$ and every qualified type ρ is equivalent to a type scheme of the form $\forall\emptyset.\rho$, the ordering defined can be used to compare type schemes and qualified types as well as

constrained type schemes. For example, $\sigma \geq \sigma'$ indicates that σ is more general than σ' in any environment.

Definition 5.9 implies that $(\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ if and only if the set of generic instances of $(\Delta' \mid \sigma')$ is a subset of the set of generic instances of $(\Delta \mid \sigma)$, and so it is straightforward to show that $\geq$ is a preorder on constrained type schemes, and that a qualified type ρ is a generic instance of a type scheme σ if and only if $\sigma \geq \rho$.

Studying the properties of $\geq$ is important for the study of the typing of terms in the theory of qualified types. The first step is to define an equivalence relation between constrained type schemes:

$$(\Delta \mid \sigma) \;\simeq\; (\Delta' \mid \sigma') \text{ iff } (\Delta \mid \sigma) \;\geq\; (\Delta' \mid \sigma') \text{ and } (\Delta' \mid \sigma') \;\geq\; (\Delta \mid \sigma)$$

Observe that two constrained type schemes are equivalent when they have the same set of generic instances. Note in particular that, if $\sigma = \forall \alpha_i.\Delta \Rightarrow \tau$, then $\sigma \simeq \forall \beta_i.(\Delta \Rightarrow \tau)[\alpha_i/\beta_i]$ for any distinct variables β_i which do not appear free in σ.

For example, in the Haskell system, it is true that

$$(\mathit{Eq}\ a, \mathit{Eq}\ [a] \mid a \to [a] \to \mathit{Bool}) \;\simeq\; (\mathit{Eq}\ a \mid a \to [a] \to \mathit{Bool})$$

because $\mathit{Eq}\ a \Vdash \mathit{Eq}\ [a]$ by (Inst), and also that

$$(\mathit{Eq}\ [a] \mid \mathit{Ord}\ a \Rightarrow [a] \to [a] \to \mathit{Bool}) \;\simeq\; (\mathit{Ord}\ a \mid [a] \to [a] \to \mathit{Bool})$$

because $\mathit{Ord}\ a \Vdash \mathit{Eq}\ a$ by (Super). Equivalences like these will be exploited in Section 5.6 to produce a set of predicates as simple as possible, equivalent to a given one.

The following properties are easily established:

- $(\Delta \mid \rho) \;\simeq\; \Delta \Rightarrow \rho$
- $\sigma \;\geq\; (\Delta \mid \sigma)$
- if $\sigma \;\geq\; \sigma'$ and $\Delta' \Vdash \Delta$, then $(\Delta \mid \sigma) \;\geq\; (\Delta' \mid \sigma')$
- if none of the variables α_i appears in Δ, then $\forall \alpha_i.\Delta \Rightarrow \rho \;\simeq\; (\Delta \mid \forall \alpha_i.\rho)$

The definition of $\geq$ is an extension of the ordering relation described by Damas and Milner [1982]. In the latter, a simple syntactic characterization of the relation is given, which is useful for working in algorithms. The equivalent characterization in this framework for $\geq$ is given by the following proposition.

PROPOSITION 5.10. *Let $\sigma = \forall \alpha_i.\Delta_\tau \Rightarrow \tau$ and $\sigma' = \forall \beta_i.\Delta'_\tau \Rightarrow \tau'$ be two type schemes, and suppose that none of the β_i appears free in σ, Δ, or Δ'. Then $(\Delta \mid \sigma) \;\geq\; (\Delta' \mid \sigma')$ if and only if there are types τ_i such that:*

$$\tau' = \tau[\alpha_i/\tau_i] \textit{ and } \Delta', \Delta'_\tau \Vdash \Delta, \Delta_\tau[\alpha_i/\tau_i]$$

Another property that the ordering relation on constrained type schemes has is that it is preserved by substitution. This is particularly important for the treatment of polymorphism. The application of a substitution S to a constrained type scheme $(\Delta \mid \sigma)$ is defined by $S\,(\Delta \mid \sigma) = (S\,\Delta \mid S\,\sigma)$; with this, it is easy to see that the following proposition holds.

$$\text{(W-VAR)} \quad \frac{x : \forall\alpha_i.\Delta \Rightarrow \tau \in \Gamma \qquad \beta_i \text{ new}}{\boldsymbol{\Delta[\alpha_i/\beta_i]} \mid \mathbf{Id}\,\Gamma \vdash_{\mathrm{W}} x : \boldsymbol{\tau[\alpha_i/\beta_i]}}$$

$$\text{(W-LAM)} \quad \frac{\boldsymbol{\Delta} \mid \boldsymbol{S}\,(\Gamma_x, x : t) \vdash_{\mathrm{W}} e : \boldsymbol{\tau_1}}{\boldsymbol{\Delta} \mid \boldsymbol{S}\,\Gamma \vdash_{\mathrm{W}} \lambda x.e : \boldsymbol{S\,t \to \tau_1}} \; (t \textit{ fresh})$$

$$\text{(W-APP)} \quad \frac{\boldsymbol{\Delta_1} \mid \boldsymbol{S_1}\,\Gamma \vdash_{\mathrm{W}} e_1 : \boldsymbol{\tau} \quad \boldsymbol{\Delta_2} \mid \boldsymbol{S_2}\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \boldsymbol{\tau_2} \qquad S_2\,\tau \sim^{\boldsymbol{U}} \tau_2 \to t}{\boldsymbol{U\,S_2\,\Delta_1, U\,\Delta_2} \mid \boldsymbol{(US_2S_1)}\,\Gamma \vdash_{\mathrm{W}} e_1@e_2 : \boldsymbol{U\,t}} \; (t \textit{ fresh})$$

$$\text{(W-LET)} \quad \frac{\boldsymbol{\Delta_2} \mid \boldsymbol{S_2}\,\Gamma \vdash_{\mathrm{W}} e_2 : \boldsymbol{\tau_2} \quad \boldsymbol{\Delta_1} \mid \boldsymbol{S_1}\,(S_2\,\Gamma, x : \sigma) \vdash_{\mathrm{W}} e_1 : \boldsymbol{\tau_1}}{\boldsymbol{\Delta_1} \mid \boldsymbol{(S_1S_2)}\,\Gamma \vdash_{\mathrm{W}} \mathbf{let}\; x = e_2 \;\mathbf{in}\; e_1 : \boldsymbol{\tau_1}} \; \big(\sigma = \mathrm{Gen}_{S_2\,\Gamma}(\Delta_2 \Rightarrow \tau_2)\big)$$

Figure 5.3: Type Inference Algorithm for OML[1].

PROPOSITION 5.11. *For any substitution S and constrained type schemes $(\Delta \mid \sigma)$ and $(\Delta' \mid \sigma')$:*

$$(\Delta \mid \sigma) \;\geq\; (\Delta' \mid \sigma') \textit{ implies } S\,(\Delta \mid \sigma) \;\geq\; S\,(\Delta' \mid \sigma')$$

All these elements are used to formally compare the system of rules specifying OML typings with an algorithm calculating a particular typing for a given expression, and to prove that the typing produced by this algorithm has the important property of expressing all the others — i.e. it is *principal.* The algorithm is specified by a system of rules given in Figure 5.3 deriving judgements of the form $\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau$ where Γ, and e are the type assignment and OML expression provided as inputs, and Δ, S and τ are a predicate set, a substitution, and a type, respectively, produced as results (marked in bold in the figure). We also use a unification judgement, $\tau \sim^S \tau'$, where the two types τ and τ' are inputs, and the substitution S is output. The notation $\mathrm{Gen}_\Gamma(\rho)$ used in rule (W-LET) indicates the *generalization* of ρ with respect to Γ, defined as $\forall\alpha_i.\rho$ where $\{\alpha_i\}$ indicates the set of type variables $FV(\rho)/FV(\Gamma)$. Observe that rules (QIN), (QOUT), (GEN), and (INST) have been incorporated into the other rules; in particular, (QIN) and (GEN) are expressed by the use of $\mathrm{Gen}_{_}(_)$ in rule (W-LET), and (QOUT) and (INST) by taking new type variables in rule (W-VAR). This is the standard procedure in Milner's algorithm extended to incorporate predicates.

The following proposition is proved.

PROPOSITION 5.12. *If $\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau$, then $\sigma_p = \mathrm{Gen}_{S\Gamma}(\Delta \Rightarrow \tau)$ is a principal type scheme for e under $S\,\Gamma$, that is, $\Delta \mid \Gamma \vdash e : \sigma$ iff $\sigma_p \;\geq\; (\Delta \mid \sigma)$*

The concept of *principal type scheme* corresponds to the most general derivable typing with respect to $\geq$ under a given type assignment. Notice that it is also possible for the type inference algorithm to fail, either because e contains a free variable not bound in Γ, or because the calculation of a most general unifier, described by the notation $\tau \sim^U \tau'$ fails as a result of a mismatch between the expected and actual type of a function argument. In this case, there are no derivable typings of the form $\Delta \mid \Gamma \vdash e : \sigma$, established by the completeness of the property.

[1]Outputs from the algorithm are marked in bold.

$$\text{(Fst)}\quad h:\Delta, h':\Delta' \Vdash h:\Delta$$

$$\text{(Snd)}\quad h:\Delta, h':\Delta' \Vdash h':\Delta'$$

$$\text{(Univ)}\quad \frac{h:\Delta \Vdash v':\Delta' \qquad h:\Delta \Vdash v'':\Delta''}{h:\Delta \Vdash v':\Delta', v'':\Delta''}$$

$$\text{(Trans)}\quad \frac{h:\Delta \Vdash v':\Delta' \qquad h':\Delta' \Vdash v'':\Delta''}{h:\Delta \Vdash v''[h'/v']:\Delta''}$$

$$\text{(Close)}\quad \frac{h:\Delta \Vdash v':\Delta'}{h:S\,\Delta \Vdash v':S\,\Delta'}$$

Figure 5.4: Structural laws satisfied by entailment.

5.5 Coherence and Evidence

To give semantics to the terms in the system, [Jones, 1994a] introduces the notion of *evidence*, and provides a translation from the original language of terms, OML, to one manipulating evidence explicitly — called OP, for 'Overloaded Polymorphic λ-calculus. The essential idea is that an object of type $\Delta \Rightarrow \tau$ can only be used if supplied with suitable evidence that the predicates in Δ do indeed hold. The treatment of evidence can be ignored in the basic typing algorithm, but is essential to provide *coherence*, which means that the meaning of a term does not depend on the way it is typechecked [Breazu-Tannen *et al.*, 1991]. The properties of predicate entailment must be extended to deal with predicate assignments and evidence expressions — the rules given in Figure 5.4 are the same as those of Figure 5.1 but extended with evidence; h denotes an evidence variable, and v denotes an evidence expression. Observe that we are using the conventions introduced in Notation 5.4, so predicate assignments are written as $h : \Delta$ meaning $h_1 : \delta_1, \ldots, h_n : \delta_n$, and similarly for $v : \Delta$.

One important property of the theory of qualified types is that it is abstract, admitting different realizations of the notions of predicates and evidence. In particular, when choosing what evidence should be, there is a great freedom in the decision of which particular things should be resolved during type checking, and which ones should be deferred to run-time. Following a distinction made by Thatte [1992], we observe that implementations may vary from a *prescriptive* type system (that is, where meaning and well-typing can be treated independently), to a *descriptive* type system (that is, where meaning and well-typing are inseparable).

In the case of the Haskell type system, the treatment of evidence suggests an implementation of overloading using *dictionaries* — a special record containing the implementation of the member functions — with the formal treatment described below establishing how these dictionaries have to be manipulated in overloaded terms. For example, in the case of the class `Eq` from Section 5.3, the corresponding dictionary declaration will be

```
data EqD a = EqDict (a -> a -> Bool)
```

```
eq :: EqD a -> (a -> a -> Bool)
eq (EqDict e) = e
```

and the instance declarations for `Int` and `Char` will then produce the following dictionaries:

```
eqDInt :: EqD Int
eqDInt = EqDict primEqInt

eqDChar :: EqD Char
eqDChar = EqDict (\c -> \c' -> eq eqDInt (ord c) (ord c'))
```

The dictionaries `eqDInt` and `eqDChar` will be used as the evidence proving the predicates *Eq Int* and *Eq Char*. The use of dictionaries in particular functions is presented in Example 5.13.

Terms in the language OP are extended with evidence variables, and corresponding constructs to abstract and instantiate evidence. We will here use the name h to denote evidence variables, and the constructs $\Lambda h.e'$ and $e'(\!(v)\!)$ to denote evidence abstraction and evidence instantiation, respectively. The typing rules for OP terms allow the introduction and elimination of those constructs, and also unrestricted use of polymorphism — although this makes type inference undecidable, OP will be used as a target language for the translation of OML terms, and then that is not an issue. This feature will also allow treating constrained type schemes as types, by means of the equivalence $(\Delta \mid \rho) \;\simeq\; \Delta \Rightarrow \rho$. Any OML typing can be treated as an OP typing, and can be obtained from it by erasing all the evidence information; in particular, an OP term that can be erased to obtain the corresponding OML term is the *translation* of this term. A more direct approach is to define a relation $\Delta \mid \Gamma \vdash e \hookrightarrow e' : \sigma$, that, given a OML term e, calculates an OP term e' and type scheme σ, such that erasing evidence from e' gives e again — see Figure 5.5. Observe the way in which evidence is abstracted and instantiated in rules (QIN) and (QOUT), respectively.

To illustrate how this translation works with an actual program, we consider again the function from Example 5.6, and its translation with evidence.

EXAMPLE 5.13. The function `member` from Example 5.6 is translated to the following term dealing with dictionaries:

```
member :: EqD a -> a -> [a] -> Bool
member d x [] = False
member d x (y:ys) = eq d x y || member d x ys
```

In the particular language used in the formalization this will be written `member` $=$ $\Lambda d.\lambda x.\lambda x_s.\ldots$, showing clearly that the variable d is evidence abstracted by means of the new kind of abstraction. When this function is used with particular data, as in `member 2 [1]`, the translation will provide the right evidence: `member eqDInt 2 [1]` — in the language of evidence used in this chapter, this last term will be written `member`$(\!($`eqDInt`$)\!)$@2@[1].

$$\text{(VAR)}\quad \frac{x : \tau \in \Gamma}{\Delta \mid \Gamma \vdash x \hookrightarrow x : \tau}$$

$$\text{(LAM)}\quad \frac{\Delta \mid \Gamma_x, x : \tau_2 \vdash e \hookrightarrow e' : \tau_1}{\Delta \mid \Gamma \vdash \lambda x.e \hookrightarrow \lambda x.e' : \tau_2 \rightarrow \tau_1}$$

$$\text{(APP)}\quad \frac{\Delta \mid \Gamma \vdash e_1 \hookrightarrow e_1' : \tau_2 \rightarrow \tau_1 \qquad \Delta \mid \Gamma \vdash e_2 \hookrightarrow e_2' : \tau_2}{\Delta \mid \Gamma \vdash e_1 @ e_2 \hookrightarrow e_1' @ e_2' : \tau_1}$$

$$\text{(LET)}\quad \frac{\Delta \mid \Gamma \vdash e_2 \hookrightarrow e_2' : \sigma \qquad \Delta \mid \Gamma, x : \sigma \vdash e_1 \hookrightarrow e_1' : \tau}{\Delta \mid \Gamma \vdash \mathbf{let}\ x = e_2\ \mathbf{in}\ e_1 \hookrightarrow \mathbf{let}\ x = e_2'\ \mathbf{in}\ e_1' : \tau}$$

$$\text{(QIN)}\quad \frac{\Delta, h : \delta \mid \Gamma \vdash e \hookrightarrow e' : \rho}{\Delta \mid \Gamma \vdash e \hookrightarrow \Lambda h.e' : \delta \Rightarrow \rho}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash e \hookrightarrow e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v : \delta}{\Delta \mid \Gamma \vdash e \hookrightarrow e'(\!(v)\!) : \rho}$$

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash e \hookrightarrow e' : \sigma}{\Delta \mid \Gamma \vdash e \hookrightarrow e' : \forall \alpha.\sigma}\ (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

$$\text{(INST)}\quad \frac{\Delta \mid \Gamma \vdash e \hookrightarrow e' : \forall \alpha.\sigma}{\Delta \mid \Gamma \vdash e \hookrightarrow e' : S\,\sigma}\ (\mathrm{dom}(S) = \alpha)$$

Figure 5.5: Translation from OML to OP.

Unfortunately, there exist OML terms for which the translation gives two or more non-equivalent terms, showing that the meaning of OML terms depends in the way they are typed. To characterize those terms with a unique meaning, OP typings have to be studied; thus, reduction and equality of OP terms are defined, and then, the central notion of conversion. A *conversion* from σ to σ' is a collection of OP terms that allow the transformation of any OP term of type σ into an OP term of type σ' by manipulating evidence; this is an extension of the notion of $\geq$ defined before. The motivation for using this notion is that an important property of the ordering relation $\geq$ used to compare types in OML breaks down in OP, due to the presence of evidence: a term with a general type can be used as having an instance of that type only after adjusting the evidence it uses.

The definition of conversions extends the definition of $\geq$ (Definition 5.9) with the treatment of evidence, following the characterization given by Proposition 5.10.

DEFINITION 5.14. Let $\sigma = \forall\alpha_i.\Delta_\tau \Rightarrow \tau$ and $\sigma' = \forall\beta_i.\Delta'_\tau \Rightarrow \tau'$ be two type schemes, and suppose that none of the β_i appears free in σ, Δ, or Δ'. A closed OP term C of type $(\Delta \mid \sigma) \to (\Delta' \mid \sigma')$[1], such that erasing all evidence from it returns the identity function, is called a *conversion* from $(\Delta \mid \sigma)$ to $(\Delta' \mid \sigma')$, written $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, if there are types τ_i, evidence variables h' and h'_τ, and evidence expressions v and v' such that:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $h' : \Delta', h'_\tau : \Delta'_\tau \Vdash v : \Delta, v' : \Delta_\tau[\alpha_i/\tau_i]$, and
- $C = (\lambda x.\Lambda h', h'_\tau.x(\!(v)\!)(\!(v')\!))$

Conversions are only used in the theory of qualified types to relate different translations for the same term, and for that reason there is no need to distinguish between the scope of h' and that of h'_τ in the previous definition — observe that *both* h' and h'_τ are abstracted in the conversion.

To fully comprehend the idea of conversion it is useful to consider an example from the Haskell system. We consider again the expression `member 2 [1]` from Example 5.13: in this context, the general function `member` whose principal type is $\forall a.Eq\ a \Rightarrow a \to [a] \to Bool$ has to be instantiated to work on *Int*s — that is, we need an expression of type $Int \to [Int] \to Bool$ obtained from the polymorphic function. The first type is more general than the second one, and the conversion proving that is $\lambda x.x(\!(\texttt{dEqInt})\!)$, which, when applied to `member`, will produce the desired effect.

Several useful properties of conversions can be established. First of all, they are reflexive and transitive (Proposition 5.15), and they respect substitutions (Proposition 5.16). Finally, some rules to modify the predicate assignments constraining the type schemes are given by Proposition 5.17.

PROPOSITION 5.15. *The following assertions hold:*

1. $\lambda x.x : (\Delta \mid \sigma) \geq (\Delta \mid \sigma)$

[1]Remember that this is an abbreviation for $(\Delta \Rightarrow \sigma) \to (\Delta' \Rightarrow \sigma')$.

2. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $C' : (\Delta' \mid \sigma') \geq (\Delta'' \mid \sigma'')$ *then*

$$C'' : (\Delta \mid \sigma) \geq (\Delta'' \mid \sigma'')$$

where for all e', $C''e' = C'(Ce')$

PROPOSITION 5.16. *If* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, *then* $C : S\,(\Delta \mid \sigma) \geq S\,(\Delta' \mid \sigma')$.

PROPOSITION 5.17. *For any qualified type* ρ *and predicate assignments* $h : \Delta$ *and* $h' : \Delta'$, *there are conversions such that:*

1. $\lambda x.x : (\Delta, \Delta' \mid \rho) \geq (\Delta \mid \Delta' \Rightarrow \rho)$
2. $\lambda x.x : (\Delta \mid \Delta' \Rightarrow \rho) \geq (\Delta, \Delta' \mid \rho)$

It is also the case that

3. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $h : \Delta''' \Vdash v : \Delta''$, *then*

$$C' : (\Delta, \Delta'' \mid \sigma) \geq (\Delta', \Delta''' \mid \sigma')$$

where $C' = \lambda x.\Lambda h.C\,(x(\!(v)\!))$

Observe that by taking $\Delta = \emptyset$, we have, by Proposition 5.17-1,2 that

$$\lambda x.x : (\Delta \mid \rho) \geq \Delta \Rightarrow \rho \text{ and } \lambda x.x : \Delta \Rightarrow \rho \geq (\Delta \mid \rho)$$

The algorithm calculating OML principal types can be extended to an algorithm calculating principal translations from OML to OP. Then, it can be proved that any translation of an OML term e to OP can be written in the form $C\,(\Lambda h'.e')(\!(h)\!)$ where e' is the principal translation and C the corresponding conversion. This allows to characterize two translations as equivalent if the corresponding conversions are equivalent. One way to assure that property is to define the notion of ambiguous type scheme as a type scheme that qualifies variables not appearing in the type, and then restrict our language to those terms with unambiguous principal type schemes.

5.6 Fine Tuning of Predicates

With the algorithm mentioned in the previous section, an OML term e of type $\forall \alpha_i.\Delta \Rightarrow \tau$ is implemented by a translation of the form $\Lambda h.e'$, where h is a collection of evidence variables for Δ, and e' is an OP term corresponding to e that uses those variables to obtain appropriate evidence values. It is desirable, then, that the number of predicates appearing in Δ be kept as small as possible, avoiding duplications and redundancy, and also that the predicates in Δ be as accurate as possible (avoiding ambiguities whenever possible, etc.). There are two mechanisms to do this — they have been introduced by Jones [1994b] and applied by Jones [2000].

The first one is to replace Δ by another assignment Δ', equivalent to Δ (in the sense that they entail each other), but with fewer predicates or reduced by some other

measure of complexity depending on the application. The process of taking a predicate assignment Δ and producing the mentioned Δ' is called *simplification*, and it is very important to produce type schemes that are small.

As the theory of qualified types is abstract (in the sense that no particular predicates are used), the exact way in which the simplification process is performed is not given, because it will vary from one system of predicates to another — and the notion of an *optimal* predicate assignment may even not exist for some systems. Simplification can be expressed by asking the implementation to provide a function *simp* that given a predicate set returns another equivalent one in a simpler form, and then using it in the following rule:

$$\frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \qquad \Delta' = \mathit{simp}\,(\Delta)}{\Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau}$$

Observe that it is always possible to take $\mathit{simp}\,(\Delta) = \Delta$.

The second notion is that of improvement of predicates. It is similar to simplification, but it fixes the value of some variables. An *improving substitution* for a predicate set Δ is a substitution S that can be applied to it without changing its satisfiability properties, that is, such that $S\,\Delta$ has the same satisfiable instances as Δ. As with simplification, the specific way in which improving is implemented cannot be given in the abstract setting, but it can be expressed by assuming a function *impr* that, given a predicate set returns an improving substitution, and using it in the rule:

$$\frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \qquad T = \mathit{impr}\,(\Delta)}{T\,\Delta \mid T S\,\Gamma \vdash_{\mathrm{W}} e : T\,\tau}$$

There is always an improving function, that is, the one returning always the identity substitution.

A special case of the notion of improvement — or even it can be said an application of it — is the idea of *functional dependencies*. When considering a predicate P with more than one argument, it may be the case that in every use of the predicate, the value of some of the arguments, y, depend uniquely on the value of some others, x. In that case we say that y *functionally depends* on x, and write $x \rightsquigarrow y$. These dependencies can be used to deduce some improving substitutions, to detect some ambiguities, and to accurately generalize over inferred types.

Calculating improving substitutions If we have a predicate $P(x, y)$ with instances $P(x, y_1)$ and $P(x, y_2)$, then if $x \rightsquigarrow y$, we can unify y_1 and y_2, or fail if they are not unifiable. Note that, in general, this process has to be iterated until no further improvements can be found.

Detecting ambiguity Instead of considering a type ambiguous when some variable appears in the qualifiers but not in the type, we can relax this condition to variables appearing in the qualifiers that are not dependent on variables appearing in the type.

Accurate generalization When generalizing a qualified type, we usually quantify all the variables appearing in the type and not in the environment. Using functional

dependencies, those variables not appearing in the environment but that depend on variables in it, must not be quantified.

These elements allow the language designer a fine control over these aspects of the type inference algorithm.

5.7 Summary

We have briefly presented the main aspects of the theory of qualified types. The main idea in this theory is to restrict the scope of type variables by qualifying them with predicates; also the theory provides a small collection of properties that predicates have to satisfy to allow the existence of principal types. The theory presents a system of rules specifying typing, and another one giving an algorithm to calculate principal type schemes. It also uses the concept of evidence to implement overloaded terms with qualified types, and the concept of conversions to establish the coherence of the language.

In the next chapter we use all these ideas to present a system specifying type specialization in such a way that principal specializations exist. To do that, we introduce the notion of conversion into the language, and extend all the properties accordingly. In the process we change the definition of conversions slightly, in order to avoid the use of β-reduction when proving equality of conversions, and to keep better control of the scope of evidence variables.

Chapter 6

Principal Type Specialization

...

Q: *How do you kill a white elephant?*

A: *You strangle him until he turns purple, and then shoot him with a purple elephant gun.*

Popular joke (conclusion)

The problem of lack of principality in the original formulation of type specialization is important, because it forces an algorithm to wait until all the context is known before making any attempt to specialize a given expression. As we have seen, this implies that the specialization of polyvariant expressions must be deferred until a specialization is required, and that of static functions until an application is performed; in the case of a polyvariant recursive function, the 'context' may be hidden in the body of the function, thus causing problems to determine the specialization needed for a given expression. With these restrictions, the additions of modules or polymorphism is very difficult. Our claim is that by solving the problem of lack of principality, we can improve our understanding of the flow of information, and express the problems in a much clearer way, thus facilitating the search for solutions. So this chapter addresses that problem, and removes the lack of principality, by providing a different formulation that has the property of existence of principal specializations.

Lack of principality is very similar to the problem appearing in simply typed λ-calculus when typing an expression like $\lambda x.x$: the type of x is determined by the context of use, and different typings for this expression have no relation between them expressible in the system. The solution to this problem for the Hindley-Milner type system is to extend the type language to allow polymorphism — by introducing type variables — modifying the typing rules accordingly [Damas and Milner, 1982], and defining a notion of instantiation for types. Then it can be proved that for every typable term there exists a particular type scheme — the *principal* type scheme — such that every typing for the same term is expressible using it.

The contribution of this chapter is to achieve a similar result for specializations: the existence of a *principal type specialization*, that is, a specialization such that every other valid specialization for the same source source term can be obtained as an instance of it. The key idea is that the principal specialization of an expression can be done in isolation, without any context, because it will perform only the *minimum* work required, and it will wait for the static information that the context must provide. A first step in this direction is to use residual type variables — here written using the symbol t — to defer the specialization of expressions depending on the context. Unfortunately, this is not

enough, as subtle dependencies between types (such as the relation between n and n' in the specialization of Example 4.7-1), cannot be expressed. Using type variables, we expect a specialization of the form $\vdash \lambda^D x.x +^S 1^S : Int^S \rightarrow^D Int^S \hookrightarrow \lambda x.\bullet : \forall t, t'.t \rightarrow t'$ but with an extra condition relating t and t'. The *theory of qualified types*, briefly described in Chapter 5, presents a type framework that allows expressing conditions relating universally quantified variables [Jones, 1994a]. In this framework, types are enriched with predicates constraining variables, and type inference, the 'more general' ordering, instantiation, etc. from the Hindley-Milner system, are redefined to take the predicates into consideration.

In the example above, we can introduce a predicate expressing the relation between type variables, and thus produce $\vdash \lambda^D x.x +^S 1^S : Int^S \rightarrow^D Int^S \hookrightarrow \lambda x.\bullet : \forall t, t'.t' := t + \hat{1} \Rightarrow t \rightarrow t'$, in which the predicate $t' := t + \hat{1}$ is *qualifying* the type $t \rightarrow t'$ and thus restricting the quantification.

In this chapter we present a new formulation for type specialization based on the theory of qualified types, explain the changes introduced to that theory, and the main differences that our formulation has with the original formulation of type specialization. We show with several examples that more expressiveness is possible, and also show that this new formulation allows the better understanding of the problems posed in Section 4.4. When presenting propositions or lemmas, their proofs are omitted during this chapter, and collected in Appendix A.

We proceed as follows. In Section 6.1 we present the residual language extended to express the new features of the system. After that, in Section 6.3 we give the rules allowing the derivation of specializations in the new system, and enunciate the main result of the chapter — Theorem 6.26 establishing the existence of principal specializations; its proof is presented in detail in Chapter 7. Finally, in Section 6.4 we illustrate the expressiveness of the new system with examples.

6.1 Residual Language, Revisited

The key idea when extending type specialization to achieve the existence of principal specializations is the extension of the residual type language with predicates following the framework of the theory of qualified types presented in the previous chapter. Predicates will be used to express restrictions imposed by the context on the residual type of an expression; in this way, the specialization of any expression can proceed in isolation, independent of the context, and the result can be instantiated in accordance with its uses in different contexts.

Extending the residual type language with predicates implies that the residual term language must also be extended to manipulate evidence. The extensions have two parts: the "structural" components taken from the theory of qualified types, and the particular constructs needed to express specialization features. In this section we present the residual type and term languages extended in this way, and discuss the main differences that our extension has: we use conversions as part of the language to express polyvariance. When defining conversions, we have chosen to give a slightly different formulation than the one given by Jones [1994a]; the reason is that in this way we avoid the need for conventional β-reduction in the definition of equality, and to keep better control on

the scope of evidence variables. We also think that this presentation makes it easier to distinguish conversions from other terms only by syntactic means.

Following the theory of qualified types, the residual type language is extended with type variables (t), and the syntactic categories of qualified types (ρ) and type schemes (σ); also particular predicates (δ) are defined. The most important innovations with respect to the theory of qualified types are the new type construct **poly** σ, and the use of scheme variables (s). We use the name α to refer either to a type variable t or a type scheme variable s. It is very important to note that the construct **poly** now takes a type scheme as its argument, instead of a type as before, and the reason of why scheme variables are needed. These two changes are the key to obtain principality.

DEFINITION 6.1. Let t denote a *type variable* from an countably infinite set of variables, and s a *type scheme variable* from another countably infinite set of variables, both disjoint with any other set of variables already used. A *residual type*, written using the symbol τ', is an element of the language given by the grammar

$$\begin{array}{rcl}
\tau' & ::= & t \mid \mathit{Int} \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau') \mid \mathbf{poly}\ \sigma \\
\rho & ::= & \delta \Rightarrow \rho \mid \tau' \\
\sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta & ::= & \text{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \text{IsMG}\ \sigma\ \sigma
\end{array}$$

For example, $(\hat{42} \to (t, \mathbf{poly}\ s))$ is a residual type, $(\text{IsInt}\ t, \text{IsMG}\ s'\ s \Rightarrow \hat{42} \to (t, \mathbf{poly}\ s))$ is a qualified residual type, and $(\forall s, t.\text{IsInt}\ t, \text{IsMG}\ s'\ s \Rightarrow \hat{42} \to (t, \mathbf{poly}\ s))$ is a residual type scheme. (Observe that we are using the conventions introduced in Notation 5.3). The intuition for predicate IsInt is that its argument type has to be a one-point type, $\hat{n}$, and that for IsMG is the internalization of the "more general" relation corresponding to this theory.

Given a type scheme $\sigma = \forall\alpha_1. \cdots \forall\alpha_m.\rho$, we define the set of bound variables of σ, written $BV(\sigma)$, as the set of variables $A = \{\alpha_1, \cdots, \alpha_m\}$, and the set of free variables of σ, written $FV(\sigma)$, as the set of all variables appearing in σ (even those in predicates), with the exception of those that are bound. These sets will be naturally divided into two subsets of type variables and scheme variables, but we do not explicitly distinguish this fact unless necessary. We assume implicit rules for α-conversion of universal quantification.

We define a notion of substitution for types and type schemes. A substitution is a pair of functions from type variables to types and from type scheme variables to type schemes, such that they are different from the identity function only in a finite number of variables. We usually do not distinguish between these two, and use the notational convention defined in Notation 5.5.

The residual term language is extended with evidence (v), including evidence variables (h), evidence abstractions ($\Lambda h.e'$), and evidence applications ($e'((v))$). Evidence is very important in this formulation of type specialization because it allows us to abstract differences among different residual terms of a given source term, and is one of the cornerstones for the principality result. Two particular kinds of evidence are used: numbers, as evidence for predicates of the form IsInt and $_ := _ + _$, and conversions, as evidence for predicates of the form IsMG. Observe that conversions, written using the

$$(\beta_v) \quad (\Lambda h.e'_1)((v)) \;\triangleright\; e'_1[h/v]$$

$$(\eta_v) \quad \Lambda h.e'_1((h)) \;\triangleright\; e'_1 \quad (h \notin EV(e'_1))$$

$$(\mathbf{let}_v) \quad \mathbf{let}_v\; x = e'_1 \;\mathbf{in}\; e'_2 \;\triangleright\; e'_2[x/e'_1]$$

$$(\circ_v) \quad (v_1 \circ v_2)[e'] \;\triangleright\; v_1[v_2[e']]$$

Figure 6.1: Reduction for residual terms.

symbol C, are defined separately from other elements in the language, and that they are contexts instead of (families of) terms; the particular forms $v \circ v$ and $\mathbf{let}_v\; x = e'\;\mathbf{in}\; e'$ are used for composition of conversions, necessary for technical reasons — see, for example, Theorem 7.10.

Definition 6.2. A *residual term*, written using the symbol e', is an element of the language defined by the following grammar:

$$\begin{array}{lllllll}
e' & ::= x' & \mid n & \mid e' + e' & \mid \bullet & & \\
 & \mid \lambda x'.e' & \mid e'@e' & \mid \mathbf{let}\; x' = e' \;\mathbf{in}\; e' & & & \\
 & \mid (e'_1, \ldots, e'_n) & & \mid \pi_{n,n}\, e' & & & \\
 & \mid h & \mid v[e'] & \mid \Lambda h.e' & \mid e'((v)) & \mid \mathbf{let}_v\; x = e' \;\mathbf{in}\; e' & \\
v & ::= h & \mid n & \mid C & \mid v \circ v & & \\
C & ::= [] & \mid \Lambda h.C & \mid C((v)) & \mid \mathbf{let}_v\; x = C \;\mathbf{in}\; e' & &
\end{array}$$

The particular uses of each kind of evidence is shown in the examples. The expressions $\Lambda h.e'$, $e'((v))$, and $\mathbf{let}_v\; x = e'_1\;\mathbf{in}\; e'_2$ are the evidence version of the abstraction, application, and local definition, and they are considered separately because they will have reduction rules associated with them — see Definition 6.3 — while their counterparts for regular variables will not. In particular, the $\mathbf{let}_v$ construct is used to differentiate clearly the scope of evidence variables appearing in the composing terms. Given a residual term e, we denote by $EV(e)$ the set of free evidence variables of e.

To identify different equivalent ways of abstracting or providing evidence, we define a notion of equivalence on residual terms.

Definition 6.3. The equivalence $=$ on residual terms is defined as the smallest congruence containing the reduction rules appearing on Figure 6.1. Observe that there is no rule of β-reduction for normal applications — only evidence applications can be reduced.

We assume implicitly rules for α-conversion for both λ and Λ-abstractions.

The equivalence on residual terms can be extended to conversions:

Definition 6.4. Two conversions C and C' are equivalent, written $C = C'$, if for every residual expression e', $C[e'] = C'[e']$, where $=$ is the equivalence given in Definition 6.3.

Properties relating lists of predicates and evidence are captured by an *entailment* relation ($\Vdash$), as described in the previous chapter, which satisfies the structural properties

$$\text{(IsInt)} \quad \Delta \Vdash n : \text{IsInt } \hat{n}$$

$$\text{(IsOp)} \quad h : \Delta \Vdash n : \hat{n} := \hat{n}_1 + \hat{n}_2 \quad (\textit{whenever } n{=}n_1{+}n_2)$$

$$\text{(IsOpIsInt)} \quad \Delta, \tau' := \tau'_1 + \tau'_2, \Delta' \Vdash h : \text{IsInt } \tau'$$

$$\text{(IsMG)} \quad \frac{C : (\Delta \mid \sigma') \geq (\Delta \mid \sigma)}{\Delta \Vdash C : \text{IsMG } \sigma' \; \sigma}$$

$$\text{(Comp)} \quad \frac{\Delta \Vdash v : \text{IsMG } \sigma_1 \; \sigma_2 \qquad \Delta \Vdash v' : \text{IsMG } \sigma_2 \; \sigma_3}{\Delta \Vdash v' \circ v : \text{IsMG } \sigma_1 \; \sigma_3}$$

Figure 6.2: Entailment for evidence construction.

established by the theory of qualified types (see Figure 5.4). The particular meaning of predicates is defined by completing the definition of $\Vdash$ with rules to entail those predicates; these rules are presented in Figure 6.2. The predicate IsInt is provable when the type is a one-point type representing a number (rule (IsInt)), and the evidence is the value of that number. Similarly, the predicate $_ := _ + _$ is provable when the three arguments are one-point types with the corresponding numbers related by addition (rule (IsOp)), and the evidence is the number corresponding to the result of the addition. The predicate IsMG internalizes the ordering $\geq$ (rules (IsMG) and (Comp)), and the evidence is the corresponding conversion; rule (Comp) captures the transitivity of $\geq$, which is important in the proofs.

6.1.1 Ordering between residual types

As described in Chapter 5, the comparison between different types and type schemes can be done by using a "more general" ordering, defined as in Definition 5.14. As we have mentioned, we define conversions as special kinds of contexts, rather than as terms in the residual language. Additionally, as we use conversions as part of the evidence language, we need to be careful in the treatment of free and bound evidence variables in both the term to convert and in the converted one. Our definition of "more general" reflects those changes (compare Definition 5.14 and the following definition).

Definition 6.5. Let $\sigma = \forall \alpha_i.\Delta_\tau \Rightarrow \tau$ and $\sigma' = \forall \beta_i.\Delta'_\tau \Rightarrow \tau'$ be two type schemes, and suppose that none of the β_i appears free in σ, $h : \Delta$, or $h' : \Delta'$. A term C is called a *conversion* from $(\Delta \mid \sigma)$ to $(\Delta' \mid \sigma')$, written $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, if and only if there are types τ_i, evidence variables h'_τ, and evidence expressions v and v' such that:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $h' : \Delta', h'_\tau : \Delta'_\tau \Vdash v : \Delta, v' : \Delta_\tau[\alpha_i/\tau_i]$, and
- $C = (\mathbf{let}_v \; x = \Lambda h.[] \;\mathbf{in}\; \Lambda h'_\tau.x((v))((v')))$

The most important property of conversions is that they can be used to transform an object e' of type σ under a predicate assignment Δ into an element of type σ' under a predicate assignment Δ', changing only the evidence that appears at top level of e' — we prove this fact in Theorem 6.12. Assuming that C is used to convert such e', we know that the only free evidence variables in it are those appearing in h — see Lemma 6.23 — and so the x in $\mathbf{let}_v$ does not contain free evidence variables. The variables in h' and h'_τ may only appear in the evidence values v and v'. This allows us to apply the (η_v) rule on conversions.

EXAMPLE 6.6. Conversions are used to adjust the evidence demanded by different type schemes. For all Δ it holds that

1. $[]((42)) : (\Delta \mid \forall t.\text{IsInt } t \Rightarrow t \rightarrow Int) \geq (\Delta \mid \hat{42} \rightarrow Int)$
2. $C : (\Delta \mid \forall t_1, t_2.\text{IsInt } t_1, \text{IsInt } t_2 \Rightarrow t_1 \rightarrow t_2) \geq (\Delta \mid \forall t.\text{IsInt } t \Rightarrow t \rightarrow t)$ where $C = \Lambda h.[]((h))((h))$
3. $\Lambda h.[] : (\Delta \mid \hat{42} \rightarrow Int) \geq (\Delta \mid \forall t.\text{IsInt } t \Rightarrow \hat{42} \rightarrow Int)$

Observe in Example 6.6-1 that the conversion provides the evidence needed to prove the predicate IsInt $\hat{42}$, resulting from the instantiation of variable t, and thus the resulting type does not depend anymore on the predicate. Example 6.6-2 shows a situation where both abstraction and application are combined: both t_1 and t_2 are instantiated to t, so IsInt t_1 and IsInt t_2 collapse into IsInt t, and that is reflected by the conversion, which abstracts h for the evidence proving IsInt t and instantiates its arguments twice with h to prove the predicates qualifying the original term. Finally, Example 6.6-3 shows how it is possible to qualify a type with needless predicates, as long as the conversion abstracts the evidence for them (although it will not be used at all).

The *composition* of two conversions $C[C'[e']]$ can be expressed in the syntax as $\mathbf{let}_v\ x = C'\ \mathbf{in}\ C[x]$; we use the abbreviation $(C \circ C')$ for this last expression, which is justified because, for every e', $(C \circ C')[e'] = C[C'[e']] = (\mathbf{let}_v\ x = C'[e']\ \mathbf{in}\ C[x]) = (\mathbf{let}_v\ x = C'\ \mathbf{in}\ C[x])[e']$. This is also compatible with the composition of conversions when considered as evidence $(v \circ v)$.

Properties similar to those enunciated in Chapter 5 hold for this definition of conversions, too.

PROPOSITION 6.7. *The following assertions hold when $\sigma, \sigma', \sigma''$ are not scheme variables:*

1. $[] : (\Delta \mid \sigma) \geq (\Delta \mid \sigma)$
2. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $C' : (\Delta' \mid \sigma') \geq (\Delta'' \mid \sigma'')$ *then*

$$C' \circ C : (\Delta \mid \sigma) \geq (\Delta'' \mid \sigma'')$$

PROPOSITION 6.8. *The following assertions hold:*

1. *If* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, *then* $C : S\,(\Delta \mid \sigma) \geq S\,(\Delta' \mid \sigma')$.
2. *If* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \forall\alpha.\sigma')$, *and* $dom(S) = \alpha$ *then* $C : (\Delta \mid \sigma) \geq (\Delta' \mid S\,\sigma')$.

PROPOSITION 6.9. *For any qualified type ρ and predicate assignments $h : \Delta$ and $h' : \Delta'$,*

1. $\Lambda h'.[] : (\Delta, h' : \Delta' \mid \rho) \geq (\Delta \mid \Delta' \Rightarrow \rho)$
2. $[]((h')) : (\Delta \mid \Delta' \Rightarrow \rho) \geq (\Delta, h' : \Delta' \mid \rho)$
3. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $h''' : \Delta''' \Vdash v'' : \Delta''$, *then*
$$C' : (\Delta, \Delta'' \mid \sigma) \geq (\Delta', \Delta''' \mid \sigma')$$
where $C' = (\mathbf{let}_v\ x = \Lambda h'''.C[]\ \mathbf{in}\ x((v'')))$
4. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $\alpha \notin FV(\Delta, \Delta' \Rightarrow \sigma)$, *then*
$$C : (\Delta \mid \sigma) \geq (\Delta' \mid \forall\alpha.\sigma')$$

6.1.2 Typing residual terms

Instead of letting the typing of residual terms be implicitly defined in the specialization process (as it was defined by Hughes [1996b]), we give a separate system defining this typing. In this way we can show that specialization is well behaved with respect to this system (Theorem 6.20). We first define the notion of residual type assignment, and then the judgements and rules used to derive the typing of a residual term. It is very important to note that the system RT is not designed to *infer* residual types given residual terms, but only to *check* a given typing; this is so because the residual language is not intended to be used manually by a programmer, but to be automatically produced by a program generator, so there is no need for type inference: the type is produced at the same time as the term! This is the reason allowing us to use the form of higher-order polymorphism (controlled by annotations) provided by the new construct **poly** σ.

DEFINITION 6.10. A *residual type assignment*, written using the symbol Γ_R, is a (finite) list of *residual type statements* of the form $x' : \tau'$, where no x' appears more than once.

Judgements for residual typing have the form $\Delta \mid \Gamma_R \vdash_{RT} e' : \sigma$, expressing that the residual term e' has type σ under the assumptions stated in Δ and Γ_R. Residual terms are assigned a type scheme according to the RT system, whose rules are given in Figures 6.3 and 6.4.

An example of a residual typing showing the use of the **poly** wrapper is $\vdash_{RT} \Lambda h_t.\lambda x'.x' :$ **poly** $(\forall t.\text{IsInt}\ t \Rightarrow t \rightarrow t)$. Another one, showing the use of a **poly**-type in a function argument, is, for any given n, $h : \text{IsMG}\ s\ (\hat{n} \rightarrow t) \mid \emptyset \vdash_{RT} \lambda f'.h[f']@\bullet :$ **poly** $s \rightarrow t$ — observe the use of the evidence variable h, that allows to introduce a conversion when the scheme variable s is instantiated.

The following properties show that contexts can be weakened in residual judgements, and that conversions indeed relate types σ and σ' in their contexts.

PROPOSITION 6.11. *If $h : \Delta \mid \Gamma_R \vdash_{RT} e' : \sigma$, and $\Delta' \Vdash v : \Delta$, then $\Delta' \mid \Gamma_R \vdash_{RT} e'[h/v] : \sigma$.*

THEOREM 6.12. *If $h : \Delta \mid \Gamma_R \vdash_{RT} e' : \sigma$, and $C : (h : \Delta \mid \sigma) \geq (h' : \Delta' \mid \sigma')$, then $h' : \Delta' \mid \Gamma_R \vdash_{RT} C[e'] : \sigma'$.*

This last theorem is important, because it shows that conversions behave as expected.

$$\text{(RT-VAR)}\quad \frac{x' : \tau' \in \Gamma_R}{\Delta \mid \Gamma_R \vdash_{RT} x' : \tau'}$$

$$\text{(RT-DINT)}\quad \frac{\Delta \Vdash v : \text{IsInt } \tau'}{\Delta \mid \Gamma_R \vdash_{RT} v : \mathit{Int}}$$

$$\text{(RT-D+)}\quad \frac{(\Delta \mid \Gamma_R \vdash_{RT} e'_i : \mathit{Int})_{i=1,2}}{\Delta \mid \Gamma_R \vdash_{RT} e'_1 + e'_2 : \mathit{Int}}$$

$$\text{(RT-SINT)}\quad \frac{\Delta \Vdash v : \text{IsInt } \tau'}{\Delta \mid \Gamma_R \vdash_{RT} \bullet : \tau'}$$

$$\text{(RT-TUPLE)}\quad \frac{(\Delta \mid \Gamma_R \vdash_{RT} e'_i : \tau'_i)_{i=1,..,n}}{\Delta \mid \Gamma_R \vdash_{RT} (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\text{(RT-PRJ)}\quad \frac{\Delta \mid \Gamma_R \vdash_{RT} e' : (\tau'_1, \ldots, \tau'_n)}{\Delta \mid \Gamma_R \vdash_{RT} \pi_{i,n}\, e' : \tau'_i}$$

$$\text{(RT-LAM)}\quad \frac{\Delta \mid \Gamma_R, x' : \tau'_2 \vdash_{RT} e' : \tau'_1}{\Delta \mid \Gamma_R \vdash_{RT} \lambda x'.e' : \tau'_2 \to \tau'_1}$$

$$\text{(RT-APP)}\quad \frac{\Delta \mid \Gamma_R \vdash_{RT} e'_1 : \tau'_2 \to \tau'_1 \qquad \Delta \mid \Gamma_R \vdash_{RT} e'_2 : \tau'_2}{\Delta \mid \Gamma_R \vdash_{RT} e'_1 @ e'_2 : \tau'_1}$$

$$\text{(RT-LET)}\quad \frac{\Delta \mid \Gamma_R \vdash_{RT} e'_2 : \tau'_2 \qquad \Delta \mid \Gamma_R, x' : \tau'_2 \vdash_{RT} e'_1 : \tau'_1}{\Delta \mid \Gamma_R \vdash_{RT} \textbf{let } x' = e'_2 \textbf{ in } e'_1 : \tau'_1}$$

$$\text{(RT-POLY)}\quad \frac{\Delta \mid \Gamma_R \vdash_{RT} e' : \sigma' \qquad \Delta \Vdash v : \text{IsMG } \sigma'\ \sigma}{\Delta \mid \Gamma_R \vdash_{RT} v[e'] : \textbf{poly } \sigma}$$

$$\text{(RT-SPEC)}\quad \frac{\Delta \mid \Gamma_R \vdash_{RT} e' : \textbf{poly } \sigma \qquad \Delta \Vdash v : \text{IsMG } \sigma\ \tau'}{\Delta \mid \Gamma_R \vdash_{RT} v[e'] : \tau'}$$

Figure 6.3: Typing rules for the residual language (first part).

$$\text{(RT-QIN)}\quad \frac{\Delta, h : \delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \rho}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \Lambda h.e' : \delta \Rightarrow \rho}$$

$$\text{(RT-QOUT)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v : \delta}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'((v)) : \rho}$$

$$\text{(RT-GEN)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \forall\alpha.\sigma} \left(\alpha \notin FV(\Delta) \cup FV(\Gamma_{\mathrm{R}})\right)$$

$$\text{(RT-INST)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \forall\alpha.\sigma}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : S\,\sigma} \;(\mathrm{dom}(S) = \alpha)$$

Figure 6.4: Typing rules for the residual language (second part).

6.2 Roadmap Example

Before proceeding with the development of the theory of principal type specialization, we provide an example to be used as a roadmap, to understand some of the motivations under the technicalities. We choose the source program introduced in Example 3.12, which is further developed in Examples 6.17, 7.22 and 8.13. We present again the source term, and each of the specializations here, with a brief explanation of what is intended in each case. However, the details are explained in each of the corresponding examples.

The term we consider is

$$\begin{array}{l}\mathbf{let}^{D}\ f = \mathbf{poly}\ (\lambda^{D} x.\mathbf{lift}\ x +^{D} 1^{D})\\ \ \mathbf{in}\ (\mathbf{spec}\ f\ @^{D}\ 42^{S}, \mathbf{spec}\ f\ @^{D}\ 17^{S})^{D}\ : (Int^{D}, Int^{D})^{D}\end{array}$$

It contains a polyvariant function that is applied to two different static values, thus producing two different specializations.

Recalling Example 3.12, possible residual programs for this term in the original type specialization framework are

1. $\mathbf{let}\ f' = (\lambda x'.42 + 1, \lambda x'.17 + 1)$
   $\mathbf{in}\ (\mathbf{fst}\ f'@\bullet, \mathbf{snd}\ f'@\bullet) : (Int, Int)$

2. $\mathbf{let}\ f' = (\lambda x'.17 + 1, \lambda x'.55 + 1, \lambda x'.42 + 1)$
   $\mathbf{in}\ (\pi_{3,3}\, f'@\bullet, \pi_{1,3}\, f'@\bullet) : (Int, Int)$

where the polyvariant function is specialized to a tuple, and each specialization to the corresponding projection. However, the size and ordering of the tuple is not specified by the translation, thus allowing us to choose any of them.

The idea we develop with principal type specialization is that the specialization will be divided in two phases: in a first syntax directed pass, information about the term is collected by means of predicates, and a second phase will solve those predicates to calculate a corresponding solution. We first present a specification for the system in the

present chapter, then an algorithm to perform the first phase (Chapter 7) and then the second phase (Chapter 8), with two variants.

With the system as specified in this chapter, several different specializations for the same term can be obtained. In Example 6.17 we present some of them. Of particular interest is one specialization for a given term with particular properties: every other specialization for the same term can be obtained from that one by instantiation (see Section 6.3.3, specially Definition 6.25 for details). We call this specialization *principal* because of the similarity of this property with the notion of principal types. For our source term, the principal specialization is

$$\begin{array}{l}\Lambda h^u_s, h^\ell_{s_1}, h^\ell_{s_2}.\mathbf{let}\ f' = h^u_s[\Lambda h_t.\lambda x'.h_t + 1] \\ \qquad\qquad\quad \mathbf{in}\ (h^\ell_{s_1}[f']@\bullet, h^\ell_{s_2}[f']@\bullet) \\ :\ \forall s.\text{IsMG}\ (\forall t.\text{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int})\ s, \\ \qquad \text{IsMG}\ s\ (\hat{42} \rightarrow \mathit{Int}), \text{IsMG}\ s\ (\hat{17} \rightarrow \mathit{Int}) \Rightarrow (\mathit{Int}, \mathit{Int})^D\end{array}$$

Instead of a tuple and projections, we use variables standing for conversions to allow further decisions on the final form for them. The residual program obtained with the algorithm of Chapter 7, given in Example 7.22, is equivalent to this one after simplification of the predicates.

We can choose to produce either the principal residual program, or to perform the constraint solving phase (Chapter 8), giving a different residual (see Example 8.13):

$$\begin{array}{l}\mathbf{let}\ f' = \Lambda h.\lambda x'.h + 1 \\ \ \mathbf{in}\ (f'((42))@\bullet, f'((17))@\bullet) : (\mathit{Int}, \mathit{Int})^D\end{array}$$

where polyvariance is expressed using evidence abstraction, and each instance is expressed using evidence application.

To obtain the residual program produced with the original formulation, we present a slight modification of the constraint solving phase, which we call *evidence elimination* (Section 8.3) — by using this phase, the same residuals as those of Example 3.12 can be obtained.

6.3 Specifying Principal Specialization

The system specifying type specialization is composed by two sets of rules.

The first one relates source types with residual types, expressing which residual type can be obtained by specializing a given source one. This system, called SR, is important because it is needed to restrict the possible choices of residuals for the bound variable when specializing a lambda-abstraction, and the residual types of the specializations of a polyvariant expression (see rules (DLAM) and (SPEC) in Figure 6.6). Section 6.3.1 presents the rules for it, and some useful properties.

The second one presents the specialization rules themselves. The rules correspond with those introduced in Chapter 3, but extended with predicates, and dealing with the extended residual language. They are presented in Section 6.3.2.

$$\text{(SR-DINT)} \quad \Delta \vdash_{\text{SR}} \mathit{Int}^D \hookrightarrow \mathit{Int}$$

$$\text{(SR-SINT)} \quad \frac{\Delta \Vdash \text{IsInt } \tau'}{\Delta \vdash_{\text{SR}} \mathit{Int}^S \hookrightarrow \tau'}$$

$$\text{(SR-DFUN)} \quad \frac{\Delta \vdash_{\text{SR}} \tau_1 \hookrightarrow \tau_1' \qquad \Delta \vdash_{\text{SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta \vdash_{\text{SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau_2' \rightarrow \tau_1'}$$

$$\text{(SR-TUPLE)} \quad \frac{(\Delta \vdash_{\text{SR}} \tau_i \hookrightarrow \tau_i')_{i=1,..,n}}{\Delta \vdash_{\text{SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

$$\text{(SR-POLY)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma' \qquad \Delta \Vdash \text{IsMG } \sigma' \, \sigma}{\Delta \vdash_{\text{SR}} \textbf{poly } \tau \hookrightarrow \textbf{poly } \sigma}$$

$$\text{(SR-QIN)} \quad \frac{\Delta, \delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho}$$

$$\text{(SR-QOUT)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}$$

$$\text{(SR-GEN)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall \alpha.\sigma} \ (\alpha \notin FV(\Delta))$$

$$\text{(SR-INST)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall \alpha.\sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow S\,\sigma} \ (\text{dom}(S){=}\alpha)$$

Figure 6.5: Rules defining the source-residual relationship.

6.3.1 Source-Residual relationship

In the original formulation, the residual type assigned to the residual variable in the rule for lambda abstraction is not constrained in any way. In that case it was not a problem, because only whole programs were the target for specialization, so it was expected that every function be applied at least once. But when looking for principal specializations, this becomes a problem, because certain terms have more specializations than expected (e.g. $\vdash \lambda^D x.x : \mathit{Int}^S \rightarrow^D \mathit{Int}^S \hookrightarrow \lambda x'.x' : \mathit{Bool} \rightarrow \mathit{Bool}$ is a valid specialization), and *every* valid specialization should be expressed by the principal one. So, we have added a source-residual relationship (τ' is a residual of τ) — expressed by a new kind of judgement: $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$. Rules to derive this judgement are more or less straightforward, (SR-SINT) for static integers and (SR-POLY) for polyvariance being the most interesting ones — see Figure 6.5. In this way, we cure a simple omission in the original paper, which is necessary to achieve our contribution.

The following properties of the SR system are useful.

PROPOSITION 6.13. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *then* $S\,\Delta \vdash_{\text{SR}} \tau \hookrightarrow S\,\sigma$.

PROPOSITION 6.14. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *and* $\Delta' \Vdash \Delta$, *then* $\Delta' \vdash_{\text{SR}} \tau \hookrightarrow \sigma$.

THEOREM 6.15. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *and* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *then* $\Delta' \vdash_{\text{SR}} \tau \hookrightarrow \sigma'$.

This last theorem shows that if a residual type can be obtained from a source one, any instance of it can be obtained too.

6.3.2 Specialization rules: the system $\vdash_{\text{P}}$

The specialization judgements of the original system are extended with a predicate assignment and produce residual type schemes, becoming $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$. The original assignments map source variables to residual *expressions*, allowing the unfolding of the variable in static lets and functions; while powerful, that trick is troublesome in proofs, because the expressions may contain evidence variables limiting the use of predicates. For that reason, we have chosen to allow only a restricted form of specialization assignments, where source variables are mapped to residual variables; the unfolding of static lets and functions will be handled in a different way (see Chapter 9).

The rules to specify type specialization are changed accordingly; they appear in Figures 6.6 and 6.7. Observe the use of $\Vdash$ on the premises, allowing residual types to be properly constrained type variables when corresponding static information is missing.

EXAMPLE 6.16. The predicate IsInt constrains a residual type such that it can only be a one-point type, and $_ := _ + _$ constrains three types such that they are one-point types, and the first is the result of adding the other two.

1. $\vdash_{\text{P}} \lambda^D x.\mathbf{lift}\ x : Int^S \to^D Int^D \hookrightarrow \Lambda h_t.\lambda x'.h_t : \forall t.\text{IsInt}\ t \Rightarrow t \to Int$

2. $\vdash_{\text{P}} \lambda^D x.x +^S 1^S : Int^S \to^D Int^S$
   $\hookrightarrow \Lambda h_t, h_{t'}.\lambda x'.\bullet : \forall t, t'.\text{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t'$

3. $\vdash_{\text{P}} \lambda^D x.\mathbf{lift}\ x +^D 1^D : Int^S \to^D Int^D \hookrightarrow \Lambda h_t.\lambda x'.h_t + 1 : \forall t.\text{IsInt}\ t \Rightarrow t \to Int$

4. $\vdash_{\text{P}} \lambda^D x.\mathbf{lift}\ x +^D \mathbf{lift}\ (x +^S 1^S) : Int^S \to^D Int^D$
   $\hookrightarrow \Lambda h_t, h_{t'}.\lambda x'.h_t + h_{t'} : \forall t, t'.\text{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to Int$

The residual term $\Lambda h_t.\lambda x'.h_t$ in Example 6.16-1 can be converted into $\lambda x'.42$ of type $\hat{42} \to Int$ using the conversion in Example 6.6-1, and reducing the resulting redexes.

Observe how every predicate appearing in a residual type has a corresponding evidence abstraction on the term level. This is obtained by rules used to move predicates from the predicate assignment into the type, and vice-versa: (QIN) and (QOUT) in Figure 6.7. Another thing to observe is the predicate IsInt appearing in all the specializations of Example 6.16; it is introduced by the combination of rules (DLAM) and (SR-SINT).

One important thing to take into account is that the predicate $_ := _ + _$ constrains many variables at once and creates some dependencies between them; for example, in $t' := t + \hat{1}$, the variable t' depends on t. If not all variables with dependencies are quantified, then there will be only fewer solutions than one would expect. For example, the type $(\forall t.\text{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t')$ is possible in Example 6.16-2, but as t'

$$\text{(VAR)}\quad \frac{x : \tau \hookrightarrow x' : \tau' \in \Gamma}{\Delta \mid \Gamma \vdash_{\text{P}} x : \tau \ \hookrightarrow\ x' : \tau'}$$

$$\text{(DINT)}\quad \Delta \mid \Gamma \vdash_{\text{P}} n^{D} : Int^{D} \ \hookrightarrow\ n : Int$$

$$\text{(D+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : Int^{D} \ \hookrightarrow\ e'_i : Int)_{i=1,2}}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 +^{D} e_2 : Int^{D} \ \hookrightarrow\ e'_1 + e'_2 : Int}$$

$$\text{(LIFT)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : Int^{S} \ \hookrightarrow\ e' : \tau' \qquad \Delta \Vdash v : \text{IsInt}\ \tau'}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{lift}\ e : Int^{D} \ \hookrightarrow\ v : Int}$$

$$\text{(SINT)}\quad \Delta \mid \Gamma \vdash_{\text{P}} n^{S} : Int^{S} \ \hookrightarrow\ \bullet : \hat{n}$$

$$\text{(S+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : Int^{S} \ \hookrightarrow\ e'_i : \tau'_i)_{i=1,2} \qquad \Delta \Vdash v : \tau' := \tau'_1 + \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 +^{S} e_2 : Int^{S} \ \hookrightarrow\ \bullet : \tau'}$$

$$\text{(DTUPLE)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : \tau_i \ \hookrightarrow\ e'_i : \tau'_i)_{i=1,..,n}}{\Delta \mid \Gamma \vdash_{\text{P}} (e_1, \ldots, e_n)^{D} : (\tau_1, \ldots, \tau_n)^{D} \ \hookrightarrow\ (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\text{(DPRJ)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : (\tau_1, \ldots, \tau_n)^{D} \ \hookrightarrow\ e' : (\tau'_1, \ldots, \tau'_n)}{\Delta \mid \Gamma \vdash_{\text{P}} \pi^{D}_{i,n}\, e : \tau_i \ \hookrightarrow\ \pi_{i,n}\, e' : \tau'_i}$$

$$\text{(DLAM)}\quad \frac{\Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{P}} e : \tau_1 \ \hookrightarrow\ e' : \tau'_1 \qquad \Delta \vdash_{\text{SR}} \tau_2 \ \hookrightarrow\ \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} \lambda^{D} x.e : \tau_2 \rightarrow^{D} \tau_1 \ \hookrightarrow\ \lambda x'.e' : \tau'_2 \rightarrow \tau'_1}\ (x' \textit{ fresh})$$

$$\text{(DAPP)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e_1 : \tau_2 \rightarrow^{D} \tau_1 \ \hookrightarrow\ e'_1 : \tau'_2 \rightarrow \tau'_1 \qquad \Delta \mid \Gamma \vdash_{\text{P}} e_2 : \tau_2 \ \hookrightarrow\ e'_2 : \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 \mathbin{@^{D}} e_2 : \tau_1 \ \hookrightarrow\ e'_1 @ e'_2 : \tau'_1}$$

$$\text{(DLET)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e_2 : \tau_2 \ \hookrightarrow\ e'_2 : \tau'_2 \quad \Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{P}} e_1 : \tau_1 \ \hookrightarrow\ e'_1 : \tau'_1}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{let}^{D}\ x = e_2\ \textbf{in}\ e_1 : \tau_1 \ \hookrightarrow\ \textbf{let}\ x' = e'_2\ \textbf{in}\ e'_1 : \tau'_1}\ (x' \textit{ fresh})$$

$$\text{(POLY)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : \sigma' \qquad \Delta \Vdash v : \text{IsMG}\ \sigma'\ \sigma}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{poly}\ e : \textbf{poly}\ \tau \ \hookrightarrow\ v[e'] : \textbf{poly}\ \sigma}$$

$$\text{(SPEC)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \textbf{poly}\ \tau \ \hookrightarrow\ e' : \textbf{poly}\ \sigma \qquad \Delta \Vdash v : \text{IsMG}\ \sigma\ \tau' \qquad \Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \tau'}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{spec}\ e : \tau \ \hookrightarrow\ v[e'] : \tau'}$$

Figure 6.6: Specialization rules (first part)

$$\text{(QIN)}\quad \frac{\Delta, h_\delta : \delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \rho}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow \Lambda h_\delta . e' : \delta \Rightarrow \rho}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v_\delta : \delta}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'((v_\delta)) : \rho}$$

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \forall \alpha . \sigma}\ (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

$$\text{(INST)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \forall \alpha . \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma}\ (\mathrm{dom}(S) = \alpha)$$

Figure 6.7: Specialization rules (second part).

depends on t, there will be only one solution for t, although it is universally quantified — this does not seem to be the intention when universal quantification is used. To quantify residual types properly, a notion of functional dependency should be used in generalization, exactly as it is done by Jones [2000] — see Definition 7.1.

Type specializing polyvariance in a principal manner is involved; its treatment is the key notion allowing principality. The basic idea is that the specialization of a polyvariant expression e should have a scheme as its residual type (instead of a tuple type), and **spec**'s of e provide adequate instances (instead of projections); this is easier to see with an example.

EXAMPLE 6.17. A specialization of the expression in Example 3.12 using this idea is

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)\\ \quad\ \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D\ : (Int^D, Int^D)^D \hookrightarrow\\ \quad \mathbf{let}\ f' = \Lambda h.\lambda x'.h + 1\\ \quad\ \mathbf{in}\ (f'((42))@\bullet, f'((17))@\bullet) : (Int, Int)^D\end{array}$$

Observe the use of an evidence abstraction corresponding to the use of **poly** and the use of evidence applications corresponding to the use of every **spec** (instead of the previous use of tuples and projections), so no decision about the size and order of the tuple is needed. Unfortunately, this is not enough, because a given source expression may have different residual schemes in different specializations (e.g. $\lambda^D x.\lambda^D y.\mathbf{lift}\ x : Int^S \to^D Int^S \to^D Int^D$ specializes to $\Lambda h_t, h_{t'}.\lambda x'.\lambda y'.h_t : \forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t \to t' \to Int$ and also to $\Lambda h_t.\lambda x'.\lambda y'.h_t : \forall t.\mathrm{IsInt}\ t \Rightarrow t \to t \to Int$) and the principal one should express both of them. To express this, we use the predicate IsMG in the definition of rules (POLY) and (SPEC) in Figure 6.6. The type scheme for **poly** e cannot be derived entirely from the type of e, because the context can place further constraints on it, for example by passing the expression to a function which expects an argument with a more restricted residual type — see Example 6.31; the use of the predicate IsMG in the rule (POLY) permits expressing the principal specialization of a **poly** by allowing to abstract over those constraints placed by the context: instead of calculating the residual type directly, a scheme variable s can be introduced and constrained with an upper bound;

further constraints can be expressed as additional upper bounds to s. Conversely, the use of IsMG in the rule (SPEC) allows the selection of the proper instance for a **spec**; it introduces a lower bound for s, whose conversion establishes how to instantiate the polyvariant expression to the type needed. The rule (SPEC) also uses the source-residual relation, for just the same reason as it is used in the rule (DLAM). The rule for **poly** types — (SR-POLY) in Figure 6.5 — is used when a lambda-bound variable is of a **poly** type — see Example 6.18-2.

The principal specialization for the expression in Example 6.17 then is

$$\begin{array}{ll} & \Lambda h^u_s, h^\ell_{s_1}, h^\ell_{s_2}.\textbf{let } f' = h^u_s[\Lambda h_t.\lambda x'.h_t + 1] \textbf{ in } (h^\ell_{s_1}[f']@\bullet, h^\ell_{s_2}[f']@\bullet) \\ : & \forall s.\text{IsMG } (\forall t.\text{IsInt } t \Rightarrow t \rightarrow \mathit{Int})\ s, \\ & \quad \text{IsMG } s\ (\hat{42} \rightarrow \mathit{Int}), \text{IsMG } s\ (\hat{17} \rightarrow \mathit{Int}) \Rightarrow (\mathit{Int}, \mathit{Int})^D \end{array}$$

The upper bound (IsMG $(\forall t.\text{IsInt } t \Rightarrow t \rightarrow \mathit{Int})\ s$) introduced by (POLY) is responsible for the use of h^u_s in the principal specialization for f, and the lower bounds (IsMG $s\ (\hat{42} \rightarrow \mathit{Int})$) and (IsMG $s\ (\hat{17} \rightarrow \mathit{Int})$) introduced by (SPEC), for the conversion variables $h^\ell_{s_1}$ and $h^\ell_{s_2}$, respectively.

EXAMPLE 6.18. These are the principal specializations for the expressions in Example 4.7. (Remember that the source type — including binding time annotations — is part of the input!)

1. $\vdash_{\text{P}} \textbf{poly } (\lambda^D x.\textbf{lift } x +^D 1^D) : \textbf{poly } (\mathit{Int}^S \rightarrow^D \mathit{Int}^D)$
   $\hookrightarrow \Lambda h^u_s.h^u_s[\Lambda h_x.\lambda^D x'.h_x + 1]:$
   $\forall s.\text{IsMG } (\forall t.\text{IsInt } t \Rightarrow t \rightarrow \mathit{Int})\ s \Rightarrow \textbf{poly } s$

2. $\vdash_{\text{P}} \lambda^D f.\textbf{spec } f\ @^D\ 13^S : \textbf{poly } (\mathit{Int}^S \rightarrow^D \mathit{Int}^D) \rightarrow^D \mathit{Int}^D$
   $\hookrightarrow \Lambda h^u_s, h^\ell_s.\lambda f'.h^\ell_s[f']@\bullet:$
   $\forall s.\text{IsMG } (\forall t.\text{IsInt } t \Rightarrow t \rightarrow \mathit{Int})\ s, \text{IsMG } s\ (\hat{13} \rightarrow \mathit{Int}) \Rightarrow \textbf{poly } s \rightarrow \mathit{Int}$

Observe that scheme variables are used when a **poly** appears in the source type, and that the predicates constraining them are upper and lower bounds: the upper bounds come from **poly**'s (h^u_s in expression 1), and the lower bounds come from **spec**'s (h^ℓ_s in expression 2). Also observe the upper bound in expression 2: this is an additional constraint that every **poly** expression used as argument to the function must satisfy.

The following two theorems show that the specialization system is well behaved with respect to the residual typing and the restrictions imposed by the source-residual relationship.

THEOREM 6.19. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, *and for all* $x : \tau_x \hookrightarrow x' : \tau'_x \in \Gamma$, $\Delta \vdash_{\text{SR}} \tau_x \hookrightarrow \tau'_x$, *then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$.

Given a specialization assignment, $\Gamma = [x_i : \tau_i \hookrightarrow x'_i : \sigma_i \mid i = 1..n]$, we define the projection of Γ to the residual language to be $\Gamma_{(\text{RT})} = [x'_i : \sigma_i \mid i = 1..n]$.

THEOREM 6.20. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, *then* $\Delta \mid \Gamma_{(\text{RT})} \vdash_{\text{RT}} e' : \sigma$.

Additionally, we need the following properties of the system during the proof of principality.

PROPOSITION 6.21. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \tau'$ *and* $h' : \Delta' \Vdash v : \Delta$, *then* $h' : \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[h/v] : \tau'$

PROPOSITION 6.22. *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then* $S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma$.

LEMMA 6.23. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then* $EV(e') \subseteq h$

Proof: By induction on the P derivation.

The next lemma establishes that the residual type of a specialization cannot be a scheme variable (s).

LEMMA 6.24. *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then there exist* β_j, Δ_σ, *and* τ' *such that* $\sigma = \forall \beta_j.\Delta_\sigma \Rightarrow \tau'$.

Proof: By induction on the P derivation.

6.3.3 Existence of principal specializations

Similar to the notion of well-typed terms, we say that a source term is *specializable* under a given specialization assignment if there is a predicate assignment Δ, a residual term e', and a residual type scheme σ such that $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$. The purpose of this section is to characterize the existence of a specific pair of residual term and type corresponding to every specializable source term e of type τ. The notion of *principal type scheme*, originally introduced in the study of combinatory logic [Curry and Feys, 1958; Hindley, 1969], and studied by Mark Jones in the theory of qualified types, is particularly useful, corresponding to the most general derivable typing with respect to the ordering of types under a given assignment. We can state a similar result for specializations.

DEFINITION 6.25. A *principal type specialization* of a source term e of type τ under the specialization assignment Γ is a residual term e'_p of type σ_p such that $\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'_p : \sigma_p$ and it is the case that for every $\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ there exist some conversion C and substitution R satisfying $C : R\,\sigma_p \geq (\Delta' \mid \sigma)$ and $C[e'_p] = e'$.

The main result in this chapter is the existence of principal type specializations. The complete proof is given in Chapter 7.

THEOREM 6.26. *Let us consider a specialization assignment* Γ *and a source term* e *of type* τ *such that* e *is specializable under* Γ. *Then, there exists a principal type specialization of* e *under* Γ.

Proof: deferred.

The proof follows the lines of the proof of principality for the theory of qualified types [Jones, 1994a], the main difference being the rules for polyvariance and the use of conversions inside the language of evidence. The proof proceeds in two steps: first, we define $\vdash_{\mathrm{S}}$, a syntax directed version of $\vdash_{\mathrm{P}}$, and prove that they are equivalent (in an appropriate way), and then, we define an algorithm $\vdash_{\mathrm{W}}$, and prove that the $\vdash_{\mathrm{S}}$ system is equivalent to $\vdash_{\mathrm{W}}$. The reason for this separation is that comparing the algorithm against a syntax directed system is easier, and so the proofs are simpler. The algorithm $\vdash_{\mathrm{W}}$ (based

on the algorithm W [Milner, 1978]) has two interesting cases: in the rule for polyvariant expressions, and in lambda abstractions when the domain type is polyvariant. In both cases the algorithm introduces a new type scheme variable, and constrains it with an appropriate IsMG predicate.

It is important to observe that the proof of principality is constructive, involving an algorithm producing principal specializations (or failing when none exists). In the next chapter we develop the proof in detail, and in Chapter 10 we present a prototype implementation of it.

6.4 Examples

We revisit here some of the problematic examples presented in Section 4.4; the rest of them are considered in Chapter 9, after extending the new formulation of type specialization to handle booleans, datatypes and recursion.

The examples discussed here show that more expressiveness is indeed possible with the new approach. As a convention, from now on we use the following notation for evidence variables: a superscript ^{u} will indicate a variable corresponding to a predicate IsMG where the scheme variable is constrained by an upper bound, and a superscript $^\ell$, the same but for lower bounds.

Example 6.27. The source terms in Example 4.7, which have static information missing (and for that reason many different specializations for them can be obtained), can be specialized with our approach as follows.

1. $\vdash_{\mathrm{P}} \lambda^D x.x +^S 1^S : \mathit{Int}^S \to^D \mathit{Int}^S \hookrightarrow \Lambda h_t, h_{t'}.\lambda x'.\bullet : \forall t, t'.\mathrm{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t'$

2. $\vdash_{\mathrm{P}} \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) : \mathbf{poly}\ (\mathit{Int}^S \to^D \mathit{Int}^D)$
   $\hookrightarrow \Lambda h_t.\lambda x'.h_t + 1 : \mathbf{poly}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int}^D)$

3. $\vdash_{\mathrm{P}} \lambda^D f.\mathbf{spec}\ f\ @^D\ 13^S : \mathbf{poly}\ (\mathit{Int}^S \to^D \mathit{Int}^D) \to^D \mathit{Int}^D$
   $\hookrightarrow \Lambda h^u, h^\ell.\lambda f'.h^\ell[f']@\bullet : \forall s.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int})\ s,$
   $\mathrm{IsMG}\ s\ (\hat{13} \to \mathit{Int}) \Rightarrow \mathbf{poly}\ s \to \mathit{Int}$

Observe the use of evidence abstractions to wait for the residual static information. This is one of the keys allowing principal specialization. In the first two cases, the evidence are the numbers corresponding to the static values of x and resulting operations. In the last case, the evidence are conversions, one as upper bound and one as lower bound; the upper bound, introduced by the rule (SR-POLY) premise of rule (DLAM), constrains the type of f' to respect the source type, and the lower bound establishes that f' is used applied to a value of type $\hat{13}$, the corresponding conversion (h^ℓ) converting f' to fit this use.

One interesting thing about type specialization is the role played by the source type. In the following example we can see that by changing the source type only, the residual expression is different, because it expects for different evidence.

EXAMPLE 6.28. Compare this specialization with that in Example 6.27-3.

$$\begin{array}{l}\vdash_{\mathrm{P}} \lambda^{D} f.\mathbf{spec}\ f\ @^{D}\ 13^{S}\ : \mathbf{poly}\ (Int^{S} \rightarrow^{D} Int^{S}) \rightarrow^{D} Int^{S} \\ \quad \hookrightarrow \Lambda h^{u}, h^{\ell}, h_{t'}.\lambda f'.h^{\ell}[f']@\bullet : \forall s, t'.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow t')\ s, \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{IsMG}\ s\ (\hat{13} \rightarrow t'), \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{IsInt}\ t' \Rightarrow \mathbf{poly}\ s \rightarrow t'\end{array}$$

Observe that we have changed the source type for f to return static integers instead of dynamic ones; this change is reflected in the residual type with the replacement of type *Int* by the residual type variable t', constrained by the predicate IsInt t', and in the residual term by the abstraction of $h_{t'}$.

The inability to decide the size of the residual tuple in Example 4.8 is not present here, because we express polyvariance with type schemes and evidence abstractions.

EXAMPLE 6.29. The source term in Example 4.8 can be specialized in our system as following.

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{let}^{D}\ f = \mathbf{poly}\ (\lambda^{D} x.\mathbf{lift}\ x +^{D} 1^{D}) \\ \quad \mathbf{in}\ (\mathbf{spec}\ f\ @^{D}\ 42^{S}, \mathbf{spec}\ f\ @^{D}\ 17^{S}, \mathbf{spec}\ f)^{D} \\ \quad : (Int^{D}, Int^{D}, Int^{S} \rightarrow^{D} Int^{D})^{D} \\ \quad \hookrightarrow \\ \quad \Lambda h_{t}.\mathbf{let}\ f' = \Lambda h.\lambda x'.h + 1 \\ \qquad \mathbf{in}\ (f'((42))@\bullet, f'((17))@\bullet, f'((h_{t}))) \\ \quad : \forall t.\mathrm{IsInt}\ t \Rightarrow (Int, Int, t \rightarrow Int)\end{array}$$

Remember that there is not enough information to determine which is the right specialization for the third component of the tuple! This can be easily expressed by the new formulation, because polyvariance is represented as terms with abstracted evidence; in this case, the evidence h_t represents the number corresponding to the residual type t. To obtain any of the specializations of Example 4.8, a postprocessing phase called *evidence elimination* (modification of the constraint solving phase) is needed — see Chapter 8.

The next example shows how by applying a function we can alter its residual type.

EXAMPLE 6.30. Observe that the source terms in the following specializations differ in the way the functions are used in the dummy code — the first version is presented for comparison purposes.

1. $\vdash_{\mathrm{P}} \mathbf{poly}\ (\lambda^{D} x.\lambda^{D} y.\mathbf{lift}\ x +^{D} \mathbf{lift}\ y)$
   $: \mathbf{poly}\ (Int^{S} \rightarrow^{D} Int^{S} \rightarrow^{D} Int^{D})$
   $\hookrightarrow \Lambda h_x.\Lambda h_y.\lambda x.\lambda y.h_x + h_y$
   $: \mathbf{poly}\ (\forall t_x, t_y.\mathrm{IsInt}\ t_x, \mathrm{IsInt}\ t_y \Rightarrow t_x \rightarrow t_y \rightarrow Int)$

2. $\vdash_{\mathrm{P}} \mathbf{poly}\ (\mathbf{let}^{D}\ g_1 = \lambda^{D} x.\lambda^{D} y.\mathbf{lift}\ x +^{D} \mathbf{lift}\ y$
   $\mathbf{in}\ \mathbf{let}^{D}\ dummy_1 = g_1\ @^{D}\ 5^{S}\ \mathbf{in}\ g_1)$
   $: \mathbf{poly}\ (Int^{S} \rightarrow^{D} Int^{S} \rightarrow^{D} Int^{D})$
   $\hookrightarrow \Lambda h_y.\mathbf{let}\ g_1 = \lambda x.\lambda y.5 + h_y\ \mathbf{in}\ \mathbf{let}\ dummy_1 = g_1 @ \bullet\ \mathbf{in}\ g_1$
   $: \mathbf{poly}\ (\forall t_y.\mathrm{IsInt}\ t_y \Rightarrow \hat{5} \rightarrow t_y \rightarrow Int)$

3. $\vdash_{\mathrm{P}}$ **poly** $(\mathbf{let}^D\ g_2 = \lambda^D x.\lambda^D y.\mathbf{lift}\ x +^D \mathbf{lift}\ y$
$\quad\quad \mathbf{in}\ \mathbf{let}^D\ dummy_2 = \lambda^D z.g_2\ @^D\ z\ @^D\ 6^S\ \mathbf{in}\ g_2)$
$\quad : \mathbf{poly}\ (Int^S \rightarrow^D Int^S \rightarrow^D Int^D)$
$\quad \hookrightarrow \Lambda h_x.\mathbf{let}\ g_2 = \lambda x.\lambda y.h_x + 6\ \mathbf{in}\ \mathbf{let}\ dummy_2 = \lambda z.g_2 @ z @ \bullet\ \mathbf{in}\ g_2$
$\quad\quad : \mathbf{poly}\ (\forall t_x.\mathrm{IsInt}\ t_x \Rightarrow t_x \rightarrow \hat{6} \rightarrow Int)$

Every different use constrains differently the residual type.

The following example shows the need to use conversions in the rule (POLY).

EXAMPLE 6.31. The source term

$$
\begin{array}{l}
\mathbf{let}^D\ id = \lambda^D x.x \\
\mathbf{in}\ \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\lambda^D y.\mathbf{lift}\ x +^D \mathbf{lift}\ y) \\
\mathbf{in}\ \mathbf{let}^D\ f_1 = id\ @^D \mathbf{poly}\ (\mathbf{let}^D\ g_1 = \lambda^D x.\lambda^D y.0^D \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathbf{let}^D\ dummy_1 = g_1\ @^D\ 5^S\ \mathbf{in}\ g_1) \\
\mathbf{in}\ \mathbf{let}^D\ f_2 = id\ @^D \mathbf{poly}\ (\mathbf{let}^D\ g_2 = \lambda^D x.\lambda^D y.1^D \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathbf{let}^D\ dummy_2 = \lambda^D z.g_2\ @^D\ z\ @^D\ 6^S\ \mathbf{in}\ g_2) \\
\mathbf{in}\ \mathbf{spec}\ (id\ @^D f) \\
:: Int^S \rightarrow^D Int^S \rightarrow^D Int^D
\end{array}
$$

can be specialized to the residual term and type

$$
\begin{array}{l}
\Lambda h^u_{s_1}, h^u_{s_2}, h^u_{s_3}, h_t, h_{t'}, h^\ell. \\
\quad \mathbf{let}\ id = \lambda x.x \\
\quad \mathbf{in}\ \mathbf{let}\ f = h^u_{s_1}[\Lambda h_x, h_y.\lambda x.\lambda y.h_x + h_y] \\
\quad \mathbf{in}\ \mathbf{let}\ f_1 = id @ h^u_{s_2}[\Lambda h_y.\mathbf{let}\ g_1 = \lambda x.\lambda y.0 \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathbf{let}\ dummy_1 = g_1 @ \bullet\ \mathbf{in}\ g_1] \\
\quad \mathbf{in}\ \mathbf{let}\ f_2 = id @ h^u_{s_3}[\Lambda h_x.\mathbf{let}\ g_2 = \lambda x.\lambda y.1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathbf{let}\ dummy_2 = \lambda z.g_2 @ z @ \bullet\ \mathbf{in}\ g_2] \\
\quad \mathbf{in}\ h^\ell[id @ f] \\
\quad :: \forall t, t', s.\mathrm{IsMG}\ (\forall t_x, t_y.\mathrm{IsInt}\ t_x, \mathrm{IsInt}\ t_y \Rightarrow t_x \rightarrow t_y \rightarrow Int)\ s, \\
\quad\quad\quad \mathrm{IsMG}\ (\forall t_y.\mathrm{IsInt}\ t_y \Rightarrow \hat{5} \rightarrow t_y \rightarrow Int)\ s, \\
\quad\quad\quad \mathrm{IsMG}\ (\forall t_x.\mathrm{IsInt}\ t_x \Rightarrow t_x \rightarrow \hat{6} \rightarrow Int)\ s, \\
\quad\quad\quad \mathrm{IsInt}\ t, \\
\quad\quad\quad \mathrm{IsInt}\ t', \\
\quad\quad\quad \mathrm{IsMG}\ s\ (t \rightarrow t' \rightarrow Int) \\
\quad\quad\quad\quad \Rightarrow t \rightarrow t' \rightarrow Int
\end{array}
$$

Observe the use of a monovariant identity function to force all the residual types of the different **poly**s (from the previous example) to be unified. The residual term and type of function f cannot be chosen until *all* the restrictions are known; is for that reason that, in the original formulation, no specialization is done to the body of a **poly** until all the context is known — that is, at the end of the specialization process. But with our formulation we can adapt the specialization of f by constraining it with suitable upper bounds; additional restrictions are expressed as new upper bounds. In the example, there are three upper bounds, one for each poly declaration, and all constraining the same scheme variable s; additionally, the lower bound express the specialization of f, which can be seen in the use of the evidence variable h^ℓ. The final result can be calculated when

there is no possibilities that new constraints may arrive — this is done by *constraint solving*, which is presented in Chapter 8. In this case, the result after constraint solving is the following:

$$
\begin{array}{l}
\textbf{let } id = \lambda x.x \\
\textbf{in let } f = \lambda x.\lambda y.5+6 \\
\textbf{in let } f_1 = id@\textbf{let } \; g_1 = \lambda x.\lambda y.0 \\
\qquad\qquad\qquad\quad \textbf{in let } \; dummy_1 = g_1@\bullet \textbf{ in } g_1 \\
\textbf{in let } f_2 = id@\textbf{let } \; g_2 = \lambda x.\lambda y.1 \\
\qquad\qquad\qquad\quad \textbf{in let } \; dummy_2 = \lambda z.g_2@z@\bullet \textbf{ in } g_2 \\
\textbf{in } id@f \\
\; :: \hat{5} \to \hat{6} \to Int
\end{array}
$$

which is the same as the residual term and type obtained with the original formulation. Observe how the constraints imposed by the residual types of f_1 and f_2 force the arguments of f to the fixed numbers.

As a final example, we show how polyvariant functions can be used polyvariantly.

Example 6.32. The function g expects a polyvariant function as argument, and so, it has to be polyvariant to be able to apply it to different functions (with different residual types).

$$
\begin{array}{l}
\vdash_{\text{P}} \textbf{let}^{D} \; g = \textbf{poly } (\lambda^{D} f.\textbf{spec } f \; @^{D} \, 13^{S}) \\
\quad\; \textbf{in } (\textbf{spec } g \; @^{D} \textbf{poly } (\lambda^{D} x.x), \\
\qquad\quad \textbf{spec } g \; @^{D} \textbf{poly } (\lambda^{D} y.0^{S}))^{D} \\
\quad : (Int^{S}, Int^{S})^{D} \\
\quad \hookrightarrow \textbf{let } g = \Lambda h^{u}, h^{\ell}, h_r.\lambda f.h^{\ell}[f]@\bullet \\
\qquad\quad \textbf{in } (g((\Lambda h.[]((h,h)), []((13)), 13))@(\Lambda h_x.\lambda x.x), \\
\qquad\qquad\; g(([], []((13,0)), 0))@(\Lambda h_y.\lambda y.\bullet)) \\
\qquad : (\hat{13}, \hat{0})
\end{array}
$$

Observe that the evidence used in the residual of each **spec** g is different, corresponding to the instantiation from the principal type of g to the type of the parameters; the residual type of g is

$$
\begin{array}{l}
\forall t, s.\text{IsMG } (\forall t_x, t_r.\text{IsInt } t_x, \text{IsInt } t_r \Rightarrow t_x \to t_r) \; s, \\
\qquad \text{IsMG } s \; (\hat{13} \to t), \\
\qquad \text{IsInt } t \Rightarrow \textbf{poly } s \to t
\end{array}
$$

Chapter 7

The Algorithm and The Proof

The words of the spell toll inside his head. Burgess realizes that he couldn't stop now. Not even if he wanted to...

Preludes & Nocturnes – The Sandman
Neil Gaiman, Sam Kieth, and Mike Diringerberg

In this chapter we present the proof for the principality property of P (Theorem 6.26).

The proof follows the lines of the proof of principality for the theory of qualified types [Jones, 1994a], the main difference being the rules for polyvariance and the use of conversions inside the language of evidence. The proof proceeds in two steps: first, in Section 7.1, we define $\vdash_{\mathrm{S}}$, a syntax directed version of $\vdash_{\mathrm{P}}$, and prove that they are, in some sense, equivalent (Theorems 7.9 and 7.10), and then, in Section 7.2, we define an algorithm $\vdash_{\mathrm{W}}$, and prove that $\vdash_{\mathrm{S}}$ system is, in some sense, equivalent to $\vdash_{\mathrm{W}}$ (Theorems 7.17 and 7.18). The main result is then obtained as a corollary (Corollary 7.19), combining the four theorems stating the equivalence between the systems.

The importance of the proof is that it is constructive, because the algorithm $\vdash_{\mathrm{W}}$ establishes how to calculate the principal specialization of a given source term.

We close the chapter with examples of the specializations generated by the algorithm, and a discussion of the need for simplification, improvement, and constraint solving, topics covered in Chapter 8.

The algorithm presented here is the basis of the prototype described in Chapter 10.

7.1 The Syntax Directed System, S

The rules given in the previous chapter to specify type specialization provide clear descriptions of the treatment of each of the syntactic constructs of the term and type languages. But they are not suitable for use in an algorithm, because they do not completely follow the structure of a term (the are not *syntax directed*). The rules (QIN), (QOUT), (GEN), and (INST), which provide a way to transform a type into a qualified type, and this one into a type scheme, are the cause for the system $\vdash_{\mathrm{P}}$ not being syntax directed: there are apparently no restrictions on when these rules can be used because the terms and types appearing on them are not restricted. But there are restrictions. One that is common to all, the Hindley-Milner system, the theory of qualified types, and our framework, is that the argument of a function has to be a monotype, and thus every use of (QIN) or (GEN) before an application should be eliminated by a corresponding (QOUT) or (INST). The essence of a syntax directed version of $\vdash_{\mathrm{P}}$ is to restrict the places where qualification and quantification are introduced to precisely those places where they are really necessary, and doing so in such a way as to keep the generated types as

general as they can be. The name *syntax directed* is due to an important property that such a system has:

> All derivations for a given term e, if there are any, have the same structure, uniquely determined by the syntactic structure of e.

The syntax directed version allows us to explore the specification by establishing a congruence between the two systems, in such a way that results on one of the systems can be translated into results of the other one. The advantage of a syntax directed version is that the rules are better suited for an algorithm, because there is at most one rule applicable to every term, and only type expressions are involved in the process (type schemes and qualified types are handled by a generalization operator). This approach is the same as that taken by Jones [1994a], which in turn has been inspired by Clément *et al.* [1986], who gives a deterministic set of typing rules for ML and outlines the equivalence to the rules given by Damas and Milner [1982].

In both the Hindley-Milner system and the theory of qualified types the changes are introduced in the rules for typing a let construct and a variable, because as those systems provide an approach in which polymorphism is let-bound, the place where qualification and quantification can be introduced or eliminated is there. But our system is monomorphic in nature, and polymorphism is only used to express polyvariance. For that reason, the places where qualification and quantification are introduced are those annotated by the use of **poly**, so the rule that must change is (POLY).

We need to define first a way to generalize as many variables as possible under a certain assignment. This corresponds to several uses of the rule (GEN).

DEFINITION 7.1. Let $A = (FV(\Delta) \cup FV(\tau'))/(FV(\Gamma) \cup FV(\Delta'))$. We define

$$\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') = \forall A.\Delta \Rightarrow \tau'$$

The correspondence of this notion of generalization with several applications of the rule (GEN) can be stated as the following property.

PROPOSITION 7.2. *If* $\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \Delta \Rightarrow \tau'$, *then* $\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau')$, *and both derivations only differ in the application of rule* (GEN).

Proof: By repeated application of the rule (GEN).

Additionally, type schemes obtained by generalization with Gen can be related by the ordering $\geq$, as stated in the following proposition.

PROPOSITION 7.3. *The relation* $\geq$ *satisfies that, for all* Γ *and* τ',

1. *if* $h' : \Delta' \Vdash v : \Delta$ *and* $C = [\,]((v))$
   then $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq (h' : \Delta' \mid \tau')$
2. *if* $h' : \Delta' \Vdash v : \Delta$ *and* $C = \Lambda h'.[\,]((v))$
   then $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{\Gamma,\Delta''}(\Delta' \Rightarrow \tau')$
3. *for all substitutions* R *and all contexts* Δ,
   $[\,] : R\,\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{R\,\Gamma,R\,\Delta'}(R\,\Delta \Rightarrow R\,\tau')$

The actual notion of generalization used considers functional dependencies between the arguments of predicates, as described in Section 5.6; this amounts also to change the rule (GEN) accordingly. To do this, we need to define first the notion of *closure* of a set wrt. a set of functional dependencies, and the notion of the set of dependencies induced by a set of predicates.

DEFINITION 7.4. If F is a set of functional dependencies, and A is a set of variables, then the *closure* of A wrt. to F, written A^+_F, is the smallest set containing A such that if $t_1 \leadsto t_2 \in F$ and $t_1 \in A$, then $t_2 \in A$.

DEFINITION 7.5. Every set of predicates Δ induces a set of functional dependencies F_Δ on the set of variables $FV(\Delta)$, as follows:

$$F_\Delta = \{FV(\delta)_X \leadsto FV(\delta)_Y \text{ s.t. } \delta \in \Delta, X \leadsto Y \in F_\delta\}$$

where F_δ is the set of functional dependencies induced by δ — defined for each kind of predicate separately — and the subindex X (resp. Y) indicates the restriction of the set to the variables in X (resp. Y).

With the predicates defined so far the only functional dependency is the one in predicate $\tau := \tau_1 + \tau_2$, establishing that the value of τ depends on the values of τ_1 and τ_2, that is $\tau_1, \tau_2 \leadsto \tau$. When introducing new predicates in Chapter 9, some additional functional dependencies will be added.

Now, the generalization operation can be extended to include the idea of functional dependencies.

DEFINITION 7.6. Let $A = FV(\Delta) \cup FV(\tau')$, $B = FV(\Gamma) \cup FV(\Delta')$, and $A' = A/B^+_{F_\Delta}$. We define

$$\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') = \forall A'.\Delta \Rightarrow \tau'$$

The judgements for system $\vdash_{\mathrm{S}}$ have the form $\Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$, with the same intended meaning as those of P system — the main difference is that this system does not produce residual type schemes but residual types only. The rules for S are presented in Figure 7.1. Observe that the only differences between P and S are the rule (POLY), and the absence of rules (QIN), (QOUT), (GEN), and (INST). The use of these rules in inner nodes of the derivation is captured by the use of Gen in the rule (POLY), and the uses of this rules at top level are captured in Theorem 7.10, by the use of conversion C'_s.

The system S is well behaved with respect to entailment and substitutions, as stated by the following propositions.

PROPOSITION 7.7. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$ *then* $h : S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : S\,\tau'$

PROPOSITION 7.8. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$ *and* $\Delta' \Vdash v : \Delta$*, then*

$$\Delta' \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'[h/v] : \tau'$$

We are now able to establish the equivalence between S and P.

THEOREM 7.9. *If* $\Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$ *then* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \tau'$.

$$\text{(S-VAR)}\quad \frac{x : \tau \hookrightarrow x' : \tau' \in \Gamma}{\Delta \mid \Gamma \vdash_{\text{S}} x : \tau \ \hookrightarrow\ x' : \tau'}$$

$$\text{(S-DINT)}\quad \Delta \mid \Gamma \vdash_{\text{S}} n^{D} : \mathit{Int}^{D} \ \hookrightarrow\ n : \mathit{Int}$$

$$\text{(S-D+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{S}} e_i : \mathit{Int}^{D} \ \hookrightarrow\ e'_i : \mathit{Int})_{i=1,2}}{\Delta \mid \Gamma \vdash_{\text{S}} e_1 +^{D} e_2 : \mathit{Int}^{D} \ \hookrightarrow\ e'_1 + e'_2 : \mathit{Int}}$$

$$\text{(S-LIFT)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{S}} e : \mathit{Int}^{S} \ \hookrightarrow\ e' : \tau' \qquad \Delta \Vdash v : \text{IsInt}\ \tau'}{\Delta \mid \Gamma \vdash_{\text{S}} \mathbf{lift}\ e : \mathit{Int}^{D} \ \hookrightarrow\ v : \mathit{Int}}$$

$$\text{(S-SINT)}\quad \Delta \mid \Gamma \vdash_{\text{S}} n^{S} : \mathit{Int}^{S} \ \hookrightarrow\ \bullet : \hat{n}$$

$$\text{(S-S+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{S}} e_i : \mathit{Int}^{S} \ \hookrightarrow\ e'_i : \tau'_i)_{i=1,2} \qquad \Delta \Vdash v : \tau' := \tau'_1 + \tau'_2}{\Delta \mid \Gamma \vdash_{\text{S}} e_1 +^{S} e_2 : \mathit{Int}^{S} \ \hookrightarrow\ \bullet : \tau'}$$

$$\text{(S-DTUPLE)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{S}} e_i : \tau_i \ \hookrightarrow\ e'_i : \tau'_i)_{i=1,..,n}}{\Delta \mid \Gamma \vdash_{\text{S}} (e_1, \ldots, e_n)^{D} : (\tau_1, \ldots, \tau_n)^{D} \ \hookrightarrow\ (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\text{(S-DPRJ)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{S}} e : (\tau_1, \ldots, \tau_n)^{D} \ \hookrightarrow\ e' : (\tau'_1, \ldots, \tau'_n)}{\Delta \mid \Gamma \vdash_{\text{S}} \pi^{D}_{i,n}\, e : \tau_i \ \hookrightarrow\ \pi_{i,n}\, e' : \tau'_i}$$

$$\text{(S-DLAM)}\quad \frac{\Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{S}} e : \tau_1 \ \hookrightarrow\ e' : \tau'_1 \qquad \Delta \vdash_{\text{SR}} \tau_2 \ \hookrightarrow\ \tau'_2}{\Delta \mid \Gamma \vdash_{\text{S}} \lambda^{D} x.e : \tau_2 \to^{D} \tau_1 \ \hookrightarrow\ \lambda x'.e' : \tau'_2 \to \tau'_1}\ (x'\ \mathit{fresh})$$

$$\text{(S-DAPP)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{S}} e_1 : \tau_2 \to^{D} \tau_1 \ \hookrightarrow\ e'_1 : \tau'_2 \to \tau'_1 \qquad \Delta \mid \Gamma \vdash_{\text{S}} e_2 : \tau_2 \ \hookrightarrow\ e'_2 : \tau'_2}{\Delta \mid \Gamma \vdash_{\text{S}} e_1 \mathbin{@^{D}} e_2 : \tau_1 \ \hookrightarrow\ e'_1 @ e'_2 : \tau'_1}$$

$$\text{(S-DLET)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{S}} e_2 : \tau_2 \ \hookrightarrow\ e'_2 : \tau'_2 \quad \Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{S}} e_1 : \tau_1 \ \hookrightarrow\ e'_1 : \tau'_1}{\Delta \mid \Gamma \vdash_{\text{S}} \mathbf{let}^{D}\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \ \hookrightarrow\ \mathbf{let}\ x' = e'_2\ \mathbf{in}\ e'_1 : \tau'_1}\ (x'\ \mathit{fresh})$$

$$\text{(S-POLY)}\quad \frac{h' : \Delta' \mid \Gamma \vdash_{\text{S}} e : \tau \ \hookrightarrow\ e' : \tau' \qquad h : \Delta \Vdash v : \text{IsMG}\ \sigma'\ \sigma}{h : \Delta \mid \Gamma \vdash_{\text{S}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \ \hookrightarrow\ v[\Lambda h'.e'] : \mathbf{poly}\ \sigma}\ \left(\sigma' = \text{Gen}_{\Gamma,\emptyset}(\Delta' \Rightarrow \tau')\right)$$

$$\text{(S-SPEC)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{S}} e : \mathbf{poly}\ \tau \ \hookrightarrow\ e' : \mathbf{poly}\ \sigma' \quad \Delta \Vdash v : \text{IsMG}\ \sigma'\ \tau' \qquad \Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \tau'}{\Delta \mid \Gamma \vdash_{\text{S}} \mathbf{spec}\ e : \tau \ \hookrightarrow\ v[e'] : \tau'}$$

Figure 7.1: Syntax Directed Specialization Rules.

THEOREM 7.10. *If* $h : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : \sigma$, *then there exist* $h'_s, \Delta'_s, e'_s, \tau'_s$, *and* C'_s *such that*

a) $h'_s : \Delta'_s \mid \Gamma \vdash_{\text{S}} e : \tau \ \hookrightarrow\ e'_s : \tau'_s$

b) $C'_s : \text{Gen}_{\Gamma,\emptyset}(\Delta'_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma)$

c) $C'_s[\Lambda h'_s.e'_s] = e'$

Observe that Theorem 7.9 establishes that a derivation in S is also a derivation in P. But the converse is not true: not every derivation in P is a derivation in S; however, there is a way to relate derivations in both systems by using generalization, conversions, and the $\geq$ ordering, as stated by Theorem 7.10.

7.2 The Inference Algorithm, W

In this section we present an algorithm to construct a type specialization for a given typed source term, and prove that every specialization expressed by $\vdash_{\text{S}}$ system can be expressed in terms of the output of this algorithm.

The algorithm is a modification of the algorithm of translation given by Mark Jones [1994a] (in turn an extension of Milner's algorithm W, [Milner, 1978]), and it is presented in a similar way: as a set of inference rules that can be interpreted as an attribute grammar (this, in turn, is attributed to Rèmy [1989]). The main advantage of this style of presentation is that it highlights the relationship with the $\vdash_{\text{S}}$ system. The presentation as the rules of an attribute grammar can also be described in the more conventional style as a partial function W from (Γ, e, τ) to (Δ, S, e', τ'), but its presentation as a system of rules makes more clear the relationship with previous systems.

The algorithm uses a number of subsystems corresponding to algorithmic versions of the different systems used in $\vdash_{\text{P}}$ and $\vdash_{\text{S}}$. We use the letter W to identify algorithmic versions of each system. We present each of those in a different subsection.

7.2.1 A unification algorithm

The unification algorithm is based on Robinson's algorithm, with modifications to deal with substitution under quantification (that is, inside polyvariant residual types). We use a kind of "*skolemisation*" of quantified variables to avoid substituting them — to do this, we extend residual type schemes with *skolem* constants, ranging over c, and belonging to a countably infinite set with no intersection with any variables or types.

To specify the unification algorithm, we use a system of rules to derive judgements of the form $\sigma_c \sim^U \sigma_c$, with U ranging over substitutions. These rules can be interpreted as an attribute grammar in which both residual types are inherited attributes, and the substitution is a synthesized one — that is, a partial function taking two residual types and returning the substitution that unifies them, if it exists. The rules are presented in Figure 7.2.

The following properties establish that the result of unification, if it exists, is really a most general unifier.

$$c \sim^{\mathrm{Id}} c$$

$$\hat{n} \sim^{\mathrm{Id}} \hat{n}$$

$$Int \sim^{\mathrm{Id}} Int$$

$$\alpha \sim^{\mathrm{Id}} \alpha$$

$$\frac{\alpha \notin FV(\sigma)}{\alpha \sim^{[\alpha/\sigma]} \sigma}$$

$$\frac{\alpha \notin FV(\sigma)}{\sigma \sim^{[\alpha/\sigma]} \alpha}$$

$$\frac{\tau_1' \sim^{T} \tau_1'' \qquad T\,\tau_2' \sim^{U} T\,\tau_2''}{\tau_1' \rightarrow \tau_2' \sim^{UT} \tau_1'' \rightarrow \tau_2''}$$

$$\frac{\tau_{11}' \sim^{T_1} \tau_{21}' \qquad T_1\,\tau_{12}' \sim^{T_2} T_1\,\tau_{22}' \qquad \ldots \qquad T_{n-1}\ldots T_1\,\tau_{1n}' \sim^{T_n} T_{n-1}\ldots T_1\,\tau_{2n}'}{(\tau_{11}', \ldots, \tau_{1n}') \sim^{T_n \ldots T_1} (\tau_{21}', \ldots, \tau_{2n}')}$$

$$\frac{\sigma \sim^{U} \sigma'}{\textbf{poly}\ \sigma \sim^{U} \textbf{poly}\ \sigma'}$$

$$\frac{\sigma[\alpha/c] \sim^{U} \sigma'[\alpha'/c]}{\forall\alpha.\sigma \sim^{U} \forall\alpha'.\sigma'}\ (c\ \mathit{fresh})$$

$$\frac{\delta \sim^{U} \delta' \qquad \rho \sim^{U} \rho'}{\delta \Rightarrow \rho \sim^{U} \delta' \Rightarrow \rho'}$$

$$\frac{\tau \sim^{U} \tau'}{\mathrm{IsInt}\ \tau \sim^{U} \mathrm{IsInt}\ \tau'}$$

$$\frac{\tau \sim^{T} \tau' \qquad T\,\tau_1 \sim^{U} T\,\tau_1' \qquad UT\,\tau_2 \sim^{V} UT\,\tau_2'}{\tau := \tau_1 + \tau_2 \sim^{VUT} \tau' := \tau_1' + \tau_2'}$$

$$\frac{\sigma_1 \sim^{T} \sigma_2 \qquad T\,\sigma_1' \sim^{U} T\,\sigma_2'}{\mathrm{IsMG}\ \sigma_1\ \sigma_1' \sim^{UT} \mathrm{IsMG}\ \sigma_2\ \sigma_2'}$$

Figure 7.2: Rules for unification.

Proposition 7.11. *If $\sigma \sim^{U} \sigma'$ then $U\,\sigma = U\,\sigma'$.*

Proposition 7.12. *If $S\,\sigma = S\,\sigma'$, then $\sigma \sim^{U} \sigma'$ and there exists a substitution T such that $S = TU$.*

7.2.2 An entailment algorithm

The idea of an algorithm for entailment is to calculate a set of predicates that should be added to the current predicate assignment Δ to be able to entail a given predicate δ. The input is the current predicate assignment and the predicate δ to entail, and the output is the set of predicates to add and the evidence proving δ. The result can be easily achieved by adding δ to Δ with a new variable h. So, one possible algorithm is the one consisting on this only rule:

$$h : \delta \mid \Delta \Vdash_{\mathrm{W}} h : \delta \quad (h \textit{ fresh})$$

that is, generate a new fresh variable h and add $h : \delta$ to the current predicate assignment. Observe that this choice makes the use of Δ unnecessary; however, we keep the complete formulation to allow future improvements.

More refined algorithms can be designed — for example, to handle ground predicates (such as IsInt $\hat{n}$) or predicates already appearing in Δ, but all these cases can be handled by the phase of simplification and constraint solving — see Chapter 8. For that reason, we limit ourselves to the basic choice.

It is very easy to verify that the following property holds.

Proposition 7.13. *If $\Delta' \mid \Delta \Vdash_{\mathrm{W}} \delta$ then $\Delta', \Delta \Vdash \delta$.*

Proof: By definition of $\Delta' \mid \Delta \Vdash_{\mathrm{W}} \delta$ and (Fst).

7.2.3 An algorithm for source-residual relationship

The source-residual relationship between types is calculated by providing the algorithm with the source type as its input, so that it produces the residual type and a predicate assignment expressing the restrictions on variables as output. It can be given by an attribute grammar with judgements of the form $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$, where τ is an inherited attribute (that is, input to the algorithm), and Δ and τ' are synthesized ones (that is, output). The rules are given in Figure 7.3.

The following propositions relate the algorithm $\vdash_{\text{W-SR}}$ with the specification of the relationship, $\vdash_{\text{SR}}$.

Proposition 7.14. *If $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$ then $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$.*

Proposition 7.15. *If $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ then $\Delta'_w \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_w$ with all the residual variables fresh, and there exists C'_w such that $C'_w : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$.*

This last property establishes that the residual type produced by $\vdash_{\text{W-SR}}$ can be generalized to a type that is more general than any other to which the source term can be specialized. This is important when using $\vdash_{\text{W-SR}}$ to constrain the type of a lambda-bound variable in rule (W-DLAM).

$$h : \text{IsInt}\ t \vdash_{\text{W-SR}} \mathit{Int}^S \hookrightarrow t \quad (t \textit{ fresh})$$

$$\emptyset \vdash_{\text{W-SR}} \mathit{Int}^D \hookrightarrow \mathit{Int}$$

$$\frac{\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1' \qquad \Delta_2 \vdash_{\text{W-SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta_1, \Delta_2 \vdash_{\text{W-SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau_2' \rightarrow \tau_1'}$$

$$\frac{(\Delta_i \vdash_{\text{W-SR}} \tau_i \hookrightarrow \tau_i')_{i=1,\ldots,n}}{\Delta_1, \ldots, \Delta_n \vdash_{\text{W-SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

$$\frac{\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'}{\text{IsMG}\ \sigma\ s \vdash_{\text{W-SR}} \mathbf{poly}\ \tau \hookrightarrow \mathbf{poly}\ s} \left(\sigma{=}\text{Gen}_{\emptyset,\emptyset}(\Delta{\Rightarrow}\tau') \textit{ and } s \textit{ fresh}\right)$$

Figure 7.3: Rules calculating principal source-residual relationship.

7.2.4 An algorithm for type specialization

The algorithm for type specialization takes an assignment Γ, a source term e and its type τ and return a residual term e', a residual type τ', a predicate assignment Δ, and a substitution S that has to be applied to Γ to adjust the types appearing in it. It is specified by means of a set of rules used to derive judgements of the form $\Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \hookrightarrow e' : \tau'$, with the same meaning as in the $\vdash_{\text{P}}$ system. The rules can be interpreted as an attribute grammar in which e, τ, and Γ are inherited attributes, and Δ, S, e', and τ' are synthesized ones, and are presented in Figures 7.4 and 7.5.

The results obtained by W are equivalent, in the sense established in Theorems 7.17 and 7.18, to the results obtained by S. To establish the equivalence we use a notion of similarity between substitutions defined in the same way as it was done by Jones [1994a], that is, two substitutions R and S are similar (written $R \approx S$), if they only differ in a finite number of variables. This is useful to compare substitutions produced by the algorithm, given that it introduces several fresh variables that will be substituted.

Lemma 7.16. *If* $h : \Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \hookrightarrow e' : \tau'$ *then* $EV(e') \subseteq h$

Proof: By induction on the W derivation.

Theorem 7.17. *If* $\Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \hookrightarrow e' : \tau'$ *then* $\Delta \mid S\,\Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : \tau'$.

Theorem 7.18. *If* $h : \Delta \mid S\,\Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : \tau'$, *then* $h_w' : \Delta_w' \mid T_w'\,\Gamma \vdash_{\text{W}} e : \tau \hookrightarrow e_w' : \tau_w'$ *and there exist a substitution* R *and evidence* v_w' *such that*

a) $S \approx RT_w'$

b) $\tau' = R\,\tau_w'$

c) $h : \Delta \Vdash v_w : R\,\Delta_w'$

d) $e' = e_w'[h_w'/v_w']$

The meaning of this last theorem is that every residual term and type obtained by the syntax directed system can be expressed as a particular case of the residual term and type produced by the algorithm.

$$\text{(W-VAR)}\quad \frac{x : \tau \hookrightarrow x' : \tau' \in \Gamma}{\emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} x : \tau \;\hookrightarrow\; x' : \tau'}$$

$$\text{(W-DINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^{D} : \mathit{Int}^{D} \;\hookrightarrow\; n : \mathit{Int}$$

$$\text{(W-D+)}\quad \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \mathit{Int}^{D} \;\hookrightarrow\; e_1' : \mathit{Int} \qquad \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e_2' : \mathit{Int}}{S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^{D} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e_1' + e_2' : \mathit{Int}}$$

$$\text{(W-LIFT)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \mathit{Int}^{S} \;\hookrightarrow\; e' : \tau' \qquad \Delta' \mid \Delta \Vdash_{\mathrm{W}} v : \mathrm{IsInt}\;\tau'}{\Delta', \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{lift}\; e : \mathit{Int}^{D} \;\hookrightarrow\; v : \mathit{Int}}$$

$$\text{(W-SINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^{S} : \mathit{Int}^{S} \;\hookrightarrow\; \bullet : \hat{n}$$

$$\text{(W-S+)}\quad \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \mathit{Int}^{S} \;\hookrightarrow\; e_1' : \tau_1' \quad \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \mathit{Int}^{S} \;\hookrightarrow\; e_2' : \tau_2' \quad \Delta \mid S_2\,\Delta_1, \Delta_2 \Vdash_{\mathrm{W}} v : t := S}{\Delta, S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^{S} e_2 : \mathit{Int}^{S} \;\hookrightarrow\; \bullet : t}\;(t\ \mathit{fresh})$$

$$\text{(W-DLAM)}\quad \frac{\Delta \vdash_{\text{W-SR}} \tau_2 \;\hookrightarrow\; \tau_2' \qquad \Delta' \mid S\,(\Gamma, x : \tau_2 \hookrightarrow x' : \tau_2') \vdash_{\mathrm{W}} e : \tau_1 \;\hookrightarrow\; e' : \tau_1'}{\Delta', S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \lambda^{D} x.e : \tau_2 \to^{D} \tau_1 \;\hookrightarrow\; \lambda x'.e' : S\,\tau_2' \to \tau_1'}\;(x'\ \mathit{fresh})$$

$$\text{(W-DAPP)}\quad \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_2 \to^{D} \tau_1 \;\hookrightarrow\; e_1' : \tau_1' \quad \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2' \qquad S_2\,\tau_1' \sim^{U} \tau_2' \to t}{U S_2\,\Delta_1, U\,\Delta_2 \mid U S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 @^{D} e_2 : \tau_1 \;\hookrightarrow\; e_1' @ e_2' : U\,t}\;(t\ \mathit{fresh})$$

$$\text{(W-POLY)}\quad \frac{h : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e' : \tau' \quad \Delta' \mid \emptyset \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\;(\mathrm{Gen}_{S\,\Gamma,\emptyset}(\Delta \Rightarrow \tau'))\;s}{\Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{poly}\; e : \mathbf{poly}\;\tau \;\hookrightarrow\; v[\Lambda h.e'] : \mathbf{poly}\; s}\;(s\ \mathit{fresh})$$

$$\text{(W-SPEC)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \mathbf{poly}\;\tau \;\hookrightarrow\; e' : \tau_\sigma' \qquad \tau_\sigma' \sim^{U} \mathbf{poly}\; s \quad \Delta' \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau' \qquad \Delta'' \mid U\,\Delta, \Delta' \Vdash_{\mathrm{W}} v : \mathrm{IsMG}}{\Delta'', U\,\Delta, \Delta' \mid U S\,\Gamma \vdash_{\mathrm{W}} \mathbf{spec}\; e : \tau \;\hookrightarrow\; v[e'] : \tau'}\;(s\ \mathit{fresh})$$

Figure 7.4: Type Specialization Algorithm (first part).

$$\text{(W-DTUPLE)}\ \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_1 \;\hookrightarrow\; e'_1 : \tau'_1 \quad \ldots \quad \Delta_n \mid S_n\,S_{n-1}\ldots S_1\,\Gamma \vdash_{\mathrm{W}} e_n : \tau_n \;\hookrightarrow\; e'_n : \tau'_n}{\begin{array}{l} S_n \ldots S_2\,\Delta_1, \ldots, \Delta_n \mid S_n \ldots S_1\,\Gamma \\ \quad \vdash_{\mathrm{W}} (e_1, \ldots, e_n)^D : (\tau_1, \ldots, \tau_n)^D \\ \qquad \hookrightarrow\; (e'_1, \ldots, e'_n) : (S_n \ldots S_2\,\tau'_1, S_n \ldots S_3\,\tau'_2, \ldots, \tau'_n) \end{array}}$$

$$\text{(W-DPRJ)}\ \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : (\tau_1, \ldots, \tau_n)^D \;\hookrightarrow\; e' : \tau' \qquad \tau' \sim^U (t_1, \ldots, t_n)}{U\,\Delta \mid US\,\Gamma \vdash_{\mathrm{W}} \pi^D_{i,n}\,e : \tau_i \;\hookrightarrow\; \pi_{i,n}\,e' : U\,t_i}\ (t_1,\ldots,t_n \textit{ fresh})$$

$$\text{(W-DLET)}\ \frac{\Delta_2 \mid S_2\,\Gamma \vdash_{\mathrm{W}} e_2 : \tau_2 \;\hookrightarrow\; e'_2 : \tau'_2 \quad \Delta_1 \mid S_1\,(S_2\,\Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2) \vdash_{\mathrm{W}} e_1 : \tau_1 \;\hookrightarrow\; e'_1 : \tau'_1}{\begin{array}{l} S_1\,\Delta_2, \Delta_1 \mid S_1 S_2\,\Gamma \vdash_{\mathrm{W}} \mathbf{let}^D\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \\ \qquad\qquad \hookrightarrow\; \mathbf{let}\ x' = e'_2\ \mathbf{in}\ e'_1 : \tau'_1 \end{array}}\ (x' \textit{ fresh})$$

Figure 7.5: Type Specialization Algorithm (second part).

7.3 Proof of Principality of P

We are finally in position to prove Theorem 6.26; the result is a corollary of the theorems established in the previous sections.

COROLLARY 7.19. [Proof of Theorem 6.26] *Let us consider specialization assignment* Γ *and a source term* e *of type* τ *such that* e *is specializable under* Γ*. Then, there exists a principal type specialization* $e'_p : \sigma_p$ *of* e*.*

Proof: Let us consider a specialization assignment Γ and a source term e of type τ, specializable under Γ. Recall that a principal type specialization of e under Γ is a residual term e'_p of type σ_p such that $\Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e'_p : \sigma_p$, and for all Δ, e', and σ, such that $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma$ there exists a conversion C and a substitution R satisfying $C : R\,\sigma_p \geq (\Delta \mid \sigma)$ and $C[e'_p] = e'$.

So, consider any derivation $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma$. By Theorem 7.10, there exist $h_s, \Delta_s, e'_s, \tau'_s$, and C_s such that

(1) $h_s : \Delta_s \mid \Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e'_s : \tau'_s$

(2) $C_s : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma)$

(3) $C_s[\Lambda h_s.e'_s] = e'$

By Theorem 7.18 on **(1)**, $h_w : \Delta_w \mid T\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e'_w : \tau'_w$ and there exist a substitution R and evidence v_w such that

(4) $\mathrm{Id} \approx RT$

(5) $\tau'_s = R\,\tau'_w$

(6) $h_s : \Delta_s \Vdash v_w : R\,\Delta_w$

(7) $e'_s = e'_w[h_w/v_w]$

By Theorems 7.17 and 7.9, we have

$$h'_w : \Delta'_w \mid T\,\Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e'_w : \tau'_w.$$

Using rules (QIN) and (GEN), we have

$$T\,\Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; \Lambda h'_w.e'_w : \mathrm{Gen}_{T\,\Gamma,\emptyset}(h'_w : \Delta'_w \Rightarrow \tau'_w),$$

and by Proposition 6.22, and using **(4)**, we have that

$$\Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; \Lambda h'_w.e'_w : R\,\mathrm{Gen}_{T\,\Gamma,\emptyset}(h'_w : \Delta'_w \Rightarrow \tau'_w).$$

We take $e_p = \Lambda h'_w.e'_w$, and $\sigma_p = \mathrm{Gen}_{T\,\Gamma,\emptyset}(h'_w : \Delta'_w \Rightarrow \tau'_w)$.
The needed conversion, C, is obtained by a composition between

- $C_s : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma)$ from **(2)**,
- $\Lambda h'_s.[]((v_w)) : \mathrm{Gen}_{\Gamma,\emptyset}(R\,\Delta'_w \Rightarrow \tau'_s) \geq \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_s \Rightarrow \tau'_s)$, obtained from Proposition 7.3-2 on **(6)**, and
- $[] : R\,\mathrm{Gen}_{T\,\Gamma,\emptyset}(\Delta'_w \Rightarrow \tau'_w) \geq \mathrm{Gen}_{\Gamma,\emptyset}(R\,\Delta'_w \Rightarrow \tau'_s)$, obtained from Proposition 7.3-3, **(4)**, and **(5)**.

The equality $C[e'_p] = e''$ is obtained using β_v, **(7)**, and **(3)**, and the result follows.

7.4 Examples

In this section we present the principal specialization of some of the examples considered in previous chapters, as produced by the algorithm described, without any simplification or improvement. It is notorious that the set of predicates produced is usually larger than expected, a situation that is managed by the notions of simplification and improvement discussed in Section 5.6. Additionally, those examples involving polyvariance specialize to their most general forms, but are of little use without a process calculating particular cases of it. The exact description and implementation of simplification, improvement, and *constraint solving* are described in the next chapter.

Our first example shows that several copies of the same predicate are produced, even ground. These cases are easily handled by simplification.

EXAMPLE 7.20. The first copy of the predicate IsInt is produced by rule (W-DLAM), and the second one by the rule (W-LIFT). In the second case, the type variable t is unified with the residual of 54^S.

1. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D \hookrightarrow \Lambda h_t, h_{t'}.\lambda x.h_{t'} : \forall t.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int}$
2. $\vdash_{\mathrm{P}} (\lambda^D x.\mathbf{lift}\ x)\ @^D\ 54^S : \mathit{Int}^D \hookrightarrow \Lambda h_t, h_{t'}.(\lambda x.h_{t'})@\bullet : \mathrm{IsInt}\ \hat{54}, \mathrm{IsInt}\ \hat{54} \Rightarrow \hat{54} \to \mathit{Int}$

Observe that the predicate set and quantification, and corresponding evidence abstraction, are added to the type by virtue of Corollary 7.19.

When static operations (such as addition) are considered, the resulting predicate set requires the use of improvements to be able to produce the final result, as can be seen in the following example.

EXAMPLE 7.21. The value of residual type variable t, introduced to express the result of the static addition by rule (W-S+), is completely determined by the other arguments in $t := \hat{12} + \hat{1}$. An improving substitution can use this fact to produce a better result.

1. $\vdash_{\mathrm{P}} (\lambda^D x.\mathbf{lift}\ (x +^S 1^S))\ @^D\ 12^S\ : Int^D$
   $\hookrightarrow \Lambda h_{12}, h_t, h'_t.\lambda x.h'_t : \forall t.\text{IsInt}\ \hat{12}, t := \hat{12} + \hat{1}, \text{IsInt}\ t \Rightarrow Int$

The next example shows that the principal specialization does not take decisions involving polyvariant expressions.

EXAMPLE 7.22. The source term is the same as in Examples 3.12 and 6.17.

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)\\
\quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D\\
\quad : (Int^D, Int^D)^D\\
\quad \hookrightarrow \Lambda h^u_s, h_{42}, h^\ell_{s_1}, h_{17}, h^\ell_{s_2}.\mathbf{let}\ f = h^u_s[\Lambda h_x, h'_x.\lambda x.h'_x + 1]\ \mathbf{in}\ (h^\ell_{s_1}[f]@\bullet, h^\ell_{s_2}[f]@\bullet)\\
\qquad : \forall s.\text{IsMG}\ (\forall t.\text{IsInt}\ t, \text{IsInt}\ t \Rightarrow t \to Int)\ s,\\
\qquad\qquad \text{IsInt}\ \hat{42},\\
\qquad\qquad \text{IsMG}\ s\ (\hat{42} \to Int),\\
\qquad\qquad \text{IsInt}\ \hat{17},\\
\qquad\qquad \text{IsMG}\ s\ (\hat{17} \to Int)\\
\qquad\quad \Rightarrow (Int, Int)
\end{array}
$$

Observe the use of evidence variables h^u_s, $h^\ell_{s_1}$, and $h^\ell_{s_2}$ expressing the (still unknown) conversions adapting the actual data to the type expressed by the scheme variable s; they are introduced in rules (W-POLY) and (W-SPEC).

The process of constraint solving described in the next chapter finds a suitable value for s, and calculates the corresponding conversions, giving the same answer as in Example 6.17.

Chapter 8

Constraint Solving and Postprocessing

"The cutting edge isn't the weapon. The cutting edge is skill, the endless effort of the inventor."

Silent Shichirōbei

The Guns of Sakai – Lone Wolf and Cub
Kazuo Koike & Goseki Kojima

The algorithm presented in the previous chapter producing the principal specialization of a term introduces potentially many predicates, several of which are redundant or expressible in simpler forms. Additionally, in cases when there are several possibilities for a certain variable (as is the case for polyvariant expressions), the given algorithm just defers the decision of which value to assign, adding predicates to the context to express that decision.

In this chapter we describe a phase that takes a predicate assignment and solves those variables whose solution can be calculated. As a first step we present a variation of the process of *simplification* and *improvement*, introduced by Mark Jones [1994b], whose goal is to reduce the number of predicates, eliminating the redundant ones, and deciding the values of variables with a unique solution. This simplification is the basis for the constraint solving phase, where the decisions that were deferred during the previous phases are solved, when possible (observe that in general, some decisions depend on contextual information that may still not be present). Finally, a variation of the algorithm of constraint solving, called *evidence elimination*, is described; its purpose is to eliminate some evidence abstraction and applications remaining after constraint solving, in favour of tuples, following the original formulation, and thus producing output that can be accepted by a standard arity raiser.

With this separation, the specialization can be regarded as a static analysis of the program, performed locally and collecting the restrictions that specify the properties of the final residual program; the constraint solving phase can be viewed as the implementation of the actual calculation of the residual.

The work presented in this section has been published by Martínez López and Badenes [2003].

We begin, in Section 8.1, by describing the process of simplification, which will be the basis for the other phases. In Section 8.2 we present the constraint solving, calculating solutions for those variables that have more than one possibility. Then, in Section 8.3 we present the evidence elimination phase. The chapter concludes with some examples.

8.1 Simplification and Improvement

8.1.1 Motivation

The algorithm calculating the principal specialization of an expression introduces several predicates to express the dependencies of subexpressions on static data. But, as this algorithm operates locally, often redundant predicates are introduced. With the goal of reducing the number of predicates, both because of legibility and to lower the computational effort of subsequent phases, we introduce a process of *simplification* of predicates.

For example, with the algorithm W presented in Figures 7.4 and 7.5, the specialization of the term

$$\lambda^{D} x.\mathbf{lift}\ ((x +^{S} 1^{S}) +^{S} (x +^{S} 1^{S})) : Int^{S} \to^{D} Int^{D}$$

is the following residual term and type:

$$\begin{array}{ll}\Lambda h_t\ h_a\ h_b\ h_c.\lambda x.h_c : \forall t\ t'\ t''. & \text{IsInt } t,\\ & t' := t + \hat{1},\\ & t'' := t + \hat{1},\\ & t''' := t'' + t' \Rightarrow t \to Int\end{array}$$

where the redundancy of predicates can be observed.

By the use of a simplification process, this residual can be converted into this other one:

$$\Lambda h, h'.\lambda x.h' : \forall t, t', t'''.t' := t + \hat{1}, t''' := t' + t' \Rightarrow t \to Int$$

which, in some sense, is 'simpler' than the original, but equivalent.

8.1.2 Specification

To establish formally the notion of simplification, we recall the properties we expect of a simplification relation. We also use a special notation.

NOTATION 8.1. In a simplification we use conversions of the form $(\Lambda h.[])((v))$ and compositions of these. To simplify the reading, we use a particular notation for this restricted form of conversions (we call them *replacements*): $h \leftarrow v$ is denoting the previous conversion, and the composition of replacements is written $h \leftarrow v \cdot C$ to denote $(\Lambda h.C)((v))$ (the operator $\cdot$ associates to the right) In this way, $h_1 \leftarrow v_1 \cdot \ldots \cdot h_n \leftarrow v_n$ denotes the conversion $(\Lambda h_1 \ldots h_n.[])((v_1 \ldots v_n))$.

The following property of the operator $(\cdot)$ will be very useful:

LEMMA 8.2. *A conversion $h \leftarrow h$ is neutral for $\cdot$ (observe that $[]$ is a particular case of this.) That is, for every conversion C and evidence variable h, it holds that $h \leftarrow h \cdot C = C = C \cdot h \leftarrow h$.*

Proof: For all e', $(h \leftarrow h \cdot C)(([])e'] = (\Lambda h.C)((h))[e'] = (\Lambda h.C[e'])((h)) =_{\beta_v} C[e'][h/h] = C[e']$ The last step is justified by Proposition 8.3. The case $C = C \cdot h \leftarrow h$ is analogous.

PROPOSITION 8.3. $(\Lambda h.e')((h))\!=\!e'$

Proof: By (β_v) and an easy induction.

Now we are ready to define simplification:

DEFINITION 8.4. A relation $S; C \mid h : \Delta \trianglerighteq h' : \Delta'$ is a *simplification* for Δ if $C = h{\leftarrow}v$ and the following conditions hold:

$$\begin{array}{ll} \text{(i)} & h' : \Delta' \Vdash v : S\Delta \\ \text{(ii)} & S\Delta \Vdash \Delta' \end{array}$$

The conditions establish that the predicate assignments are equivalent with respect to entailment (under S); as we intend to use this process to replace one predicate assignment with another, it is a natural condition to ask. The condition about the form of conversion C expresses that it can be used to transform an expression assuming predicates in Δ into another one assuming predicates in Δ'.

Observe that with this definition, $\mathrm{Id}; [] \mid \Delta \trianglerighteq \Delta$ is a valid simplification for Δ. However, we expect that any interesting simplification will be able to do more work, as the following example shows.

EXAMPLE 8.5. Given

$$\Delta_1 = h_1 : \mathrm{IsInt}\ \hat{9}, h_2 : \mathrm{IsInt}\ t''', h_3 : t := \hat{1} + \hat{2}, h_4 : t' := t + \hat{3}, h_5 : t''' := t'' + t'$$

$$\Delta_2 = h_5 : t''' := t'' + 6$$

we would like our implementation of $\trianglerighteq$ to satisfy

$$S; C \mid \Delta_1 \trianglerighteq \Delta_2$$

where $S = _[t/\hat{3}][t'/\hat{6}]$ and $C = h_1{\leftarrow}9 \cdot h_2{\leftarrow}h_5 \cdot h_3{\leftarrow}3 \cdot h_4{\leftarrow}6$; the reasons for that are:

- $h_1 : \mathrm{IsInt}\ \hat{9}$ can be trivially simplified by (IsInt), and 9 is its evidence.
- $h_2 : \mathrm{IsInt}\ t'''$ is entailed by the fifth predicate.
- $h_3 : t := \hat{1} + \hat{2}$ can be simplified calculating the result of $1 + 2$ and generating the substitution that changes t for $\hat{3}$ in the fourth predicate.
- $h_4 : t' := t + \hat{3}$, can be simplified in a similar way, once the value of t is known (from the previous predicate).

To conclude this subsection, we present a closure property of the simplification relation with respect to substitutions. It states that if two predicate assignments are related, the instances of them will be related (provided the substitutions are 'well behaved').

DEFINITION 8.6. Two substitutions S and T are said to be *compatible* with respect to a type τ, written $S \sim_\tau T$, if $TS\tau = ST\tau$. This notion extends naturally to type schemes σ, predicates δ, and predicate assignments Δ.

LEMMA 8.7. *Let $T; C \mid \Delta \trianglerighteq \Delta'$ be a simplification for Δ. If S and T are compatible under Δ, i.e. $S \sim_\Delta T$, then $T; C \mid S\Delta \trianglerighteq S\Delta'$ is a simplification for $S\Delta$.*

This property is important to ensure that sequential steps of an algorithm give a sound solution. This is presented in Section 8.2, where we combine simplification with constraint solving.

$$\text{(SimEntl)}\quad \frac{h : \Delta \Vdash v_\delta : \delta}{\text{Id}; h_\delta \leftarrow v_\delta \mid h : \Delta, h_\delta : \delta \trianglerighteq h : \Delta}$$

$$\text{(SimTrans)}\quad \frac{S; C \mid h : \Delta \trianglerighteq h' : \Delta' \qquad S'; C' \mid h' : \Delta' \trianglerighteq h'' : \Delta''}{S'S; C' \circ C \mid h : \Delta \trianglerighteq h'' : \Delta''}$$

$$\text{(SimCtx)}\quad \frac{S; C \mid h_1 : \Delta_1 \trianglerighteq h_2 : \Delta_2}{S; C \mid h_1 : \Delta_1, h' : \Delta' \trianglerighteq h_2 : \Delta_2, h' : S\Delta'}$$

$$\text{(SimPerm)}\quad \frac{S; h_1, h_2 \leftarrow v_1, v_2 \mid h_1 : \Delta_1, h_2 : \Delta_2 \trianglerighteq h'_1 : \Delta'_1, h'_2 : \Delta'_2}{S; h_2, h_1 \leftarrow v_2, v_1[h_2/v_2] \mid h_2 : \Delta_2, h_1 : \Delta_1 \trianglerighteq h'_2 : \Delta'_2, h'_1 : \Delta'_1}$$

Figure 8.1: Structural rules for simplification

8.1.3 Implementing a Simplification

Our next step is to define a set of rules implementing a simplification relation.

We start with structural rules, which should be present in any good simplification; they are presented in Figure 8.1. Rule (SimEntl) allows the elimination of redundant predicates; this includes both predicates that are deducible from others, but also those that are true by their form. For example, predicates of the form IsInt $\hat{n}$ for known $\hat{n}$s, or predicates IsMG σ σ' for which it can be shown that $C : \sigma \geq \sigma'$. The second rule, (SimTrans), provides transitivity, giving us a way to compose simplifications. The third rule, (SimCtx), expresses how to simplify only part of an assignment; it is important to note the use of the substitution S on the right hand side to cancel variables that may have been simplified. Finally, the last rule, (SimPerm), establishes that predicate assignments can be treated as if they had no order, closing the relation with respect to permutations.

The last two rules are complementary, and usually used together. In order to express this, we define a derived rule, (SimCHAM), which allows the application of simplification in any context, following the ideas of the *Chemical Abstract Machine* [Berry and Boudol, 1990].

$$\text{(SimCHAM)}\quad \frac{\Delta'_1 \approx \Delta_1 \qquad S; C \mid \Delta_1 \trianglerighteq \Delta_2 \qquad \Delta_2 \approx \Delta'_2}{S; C^{\approx} \mid \Delta'_1, \Delta \trianglerighteq \Delta'_2, S\Delta}$$

In this last rule ((SimCHAM)), the equivalence $\approx$ is defined as the least congruence on predicate assignments containing $\Delta, \delta, \delta', \Delta' \approx \Delta, \delta', \delta, \Delta'$, allowing assignments to be considered as lists without order for the application of the simplification rules. It is important to note that the order of predicates can be changed only when they are still labeled with evidence variables in a predicate assignment $(h : \delta)$; after they are introduced in a type with the (QIN) rule of qualified types theory, the link from the variables to their predicates is only given by the order in which they appear in the expressions ($\Lambda h._$ in terms and $\delta \Rightarrow _$ in types).

We have to prove that the given structural rules (in Figure 8.1) indeed define a simplification relation according to Definition 8.4.

$$\text{(SimOp}_{res}\text{)}\quad \frac{t \sim^S \hat{n}}{S; h_\delta \leftarrow n \mid h_\delta : t := \hat{n_1} \otimes \hat{n_2} \trianglerighteq \emptyset}\ (n = n_1 \otimes n_2)$$

$$\text{(SimMG}_U\text{)}\quad \frac{C : \sigma_2 \geq \sigma_1}{\mathrm{Id}; h_2 \leftarrow h_1 \circ C \mid h_1 : \mathrm{IsMG}\ \sigma_1\ s, h_2 : \mathrm{IsMG}\ \sigma_2\ s \trianglerighteq h_1 : \mathrm{IsMG}\ \sigma_1\ s}$$

Figure 8.2: Language-dependent simplification rules

THEOREM 8.8. *The rules* (SimEntl), (SimTrans), (SimCtx), *and* (SimPerm) *(of Figure 8.1) define a simplification relation, and the derived rule* (SimCHAM) *is consistent with it.*

As a second step in implementing a simplification, we complete the relation defined by the structural rules with those given in Figure 8.2, dealing with some constructs of our language. Rule (SimOp_{res}) internalizes the computation of binary operators, when all the operands are known. A similar rule will exist for unary operators as well. The rule (SimMG_U) eliminates redundant uses of predicate IsMG, when two upper bounds of the same variable are comparable. A similar rule for lower bounds would not make sense in this system: as lower bounds are produced by the rules (SPEC), they have types instead of schemes; in addition, evidence elimination will need to use all the lower bounds (see Section 8.3). Regarding predicates as IsInt $\hat{n}$ or IsMG $\sigma\ \sigma'$ when both σ and σ' are not scheme variables, they can be simplified using rule (SimEntl), as the entailment relation can deal with them.

Again we have to show that these rules define a simplification relation.

THEOREM 8.9. *A system defining a simplification relation, extended with rules* (SimOp_{res}) *and* (SimMG_U) *still defines a simplification relation.*

Although the rules presented here as an implementation may seem restricted, its goal is to simplify exactly the predicates generated by the specialization algorithm (not including a rule for lower bounds is an example of this tailoring). When designing a system as this one, the trade off between generality and specificity has to be taken into account — in one end, a very general but useless simplification, and in the other one, a non-tractable or unsolvable simplification would be obtained.

Extensions to the system presented here are possible, and in the case of extending the language of predicates, necessary. Some of those are presented in the discussion below (see Section 8.1.5).

8.1.4 Simplification during specialization

To use simplification during the specialization phase, we need to add a rule to the system P; this rule can be used in any place, but in practice is only needed before the use of a (POLY), or at the end of the derivation.

$$\text{(SIMP)}\quad \frac{h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ e' : \sigma \qquad S; C \mid h : \Delta \trianglerighteq h' : \Delta'}{h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ C[e'] : S\sigma}$$

We can prove that the new rule is consistent with the rest of the system

THEOREM 8.10. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *and* $S; C \mid h : \Delta \trianglerighteq h' : \Delta'$ *then* $h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow C[e'] : S\sigma$.

Proof: By Proposition 6.22, $h : S\Delta \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\sigma$. By definition of $\trianglerighteq$, we have $h' : \Delta' \Vdash v : S\Delta$ and then, by Proposition 6.21, $h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[h/v] : S\sigma$. Finally $e'[h/v] = (\Lambda h.[])((v))[e'] = C[e']$.

As we have seen, the relation of simplification is not necessarily functional, and then there is the possibility to choose among different assignments to replace the current one. In practice, we use a function *simplify* such that *simplify*(h, Δ) returns a triple $\langle v : \Delta', S, C\rangle$ such that $S; C \mid h : \Delta \trianglerighteq v : \Delta'$. This follows Mark Jones [1994b].

Continuing with this idea, we also extend the specialization algorithm with the following rule:

$$\text{(W-POLY)} \quad \frac{h : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \hookrightarrow e' : \tau'}{h'' : \mathrm{IsMG}\ \sigma\ s \mid T S\,\Gamma \vdash_{\mathrm{W}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow e'' : \mathbf{poly}\ s}$$

where:

$$\begin{array}{l} e'' = h''[\Lambda h'.C[e']] \\ (h' : \Delta', T, C) = \textit{simplify}(h : \Delta) \\ \sigma = \mathrm{Gen}_{TS\Gamma,\emptyset}((T\Delta' \Rightarrow T\tau')) \\ s \text{ and } h'' \text{ fresh} \end{array}$$

With this version of the algorithm, predicate assignments are simplified before their introduction in the type schemes of **poly** s. It is important to note, however, that not all predicates can be completely simplified before being introduced in the type schemes: predicates with free variables will remain unsolved, and will not be simplified until constraint solving (although the free variables can take their final value much earlier).

With these rules we have completed our goal of incorporating the simplification to the specialization process. Additional features and more possible rules are discussed as new constructs are added to the specializer.

8.1.5 Discussion

We can observe that most of the rules presented for simplification do not introduce new substitutions; the only exception is rule (SimOp$_{res}$). The difference between this rule and the rest coincides with the notions of *simplification* and *improving* presented by Mark Jones [1994b]: improving rules establish the value of some variables, if it is uniquely determined. This differentiation will be important when applying simplification to guarded predicates (in Chapter 9, Section 9.2): improving rules cannot be applied unless we are sure that the substitutions produced do not alter variables that can 'escape' a given guard, because if that guard is going to take a false value, the predicate will simply disappear, and the value assigned to the variable will be unsound. We will complete this discussion when introducing booleans in Section 9.2.

The notion of improvement suggests more rules to be added to the system. In particular, we propose two rules related to the nature of binary operators:

$$\text{(SimOP}_{inv}\text{)} \quad \frac{t \sim^S \hat{n}}{S; h \leftarrow n_1 \mid h : \hat{n_1} := t \otimes \hat{n_2} \trianglerighteq \emptyset} \quad (\exists \text{ unique } n : n_1 = n \otimes n_2)$$

$$\text{(SimOp}_{neu}\text{)}\quad \frac{t \sim^S \hat{k}}{S; [] \mid h : t' := t \otimes t' \trianglerighteq h : \text{IsInt } t'} \quad (k \text{ neutral to } \otimes)$$

Other rules may (and do) exist for other operators, and for other constructs of the language, perhaps based on entailment or reduction (as booleans — Section 9.2, case statements for datatypes — Section 9.6, static functions — Section 9.4, etc.).

Another point to observe is that the incorporation of simplification in the algorithm is suboptimal — that is, using the rule presented to perform simplification during specialization does not always produce optimal results in the predicates appearing in type schemes. This is not a big problem, because those predicates will be simplified when the type scheme is simplified or solved. We show this small drawback with an example.

EXAMPLE 8.11. Observe the predicate context in the upper bound of variable s_f.

$$\begin{array}{l}
\vdash_{\text{P}} \textbf{let}^D\ f = \lambda^D y.\textbf{poly}\ (\lambda^D x.(\textbf{lift}\ x, \textbf{lift}\ y)^D)\\
\quad \textbf{in}\ (\textbf{spec}\ (f\ @^D\ 2^S)\ @^D\ 3^S, \textbf{spec}\ (f\ @^D\ 2^S)\ @^D\ 4^S)^D\\
\quad : ((Int^D, Int^D)^D, (Int^D, Int^D)^D)^D\\
\quad \hookrightarrow \Lambda h^u_f.\Lambda h^\ell_{f_3}.\Lambda h^\ell_{f_4}.\textbf{let}\ f = \lambda y.h^u_f[\Lambda h_x.\Lambda h_y.\lambda x.(h_x, h_y)]\\
\qquad\qquad\qquad\qquad \textbf{in}\ (h^\ell_{f_3}[f@\bullet]@\bullet, h^\ell_{f_4}[f@\bullet]@\bullet)\\
\qquad : \forall s_f.\text{IsMG}\ (\forall t_x.h_x : \text{IsInt}\ t_x, h_y : \text{IsInt}\ \hat{2}\\
\qquad\qquad\qquad\qquad \Rightarrow t_x \rightarrow (Int, Int))\ s_f\\
\qquad\qquad \text{IsMG}\ s_f\ (\hat{3} \rightarrow (Int, Int))\\
\qquad\qquad \text{IsMG}\ s_f\ (\hat{4} \rightarrow (Int, Int))\\
\qquad\qquad\ \Rightarrow ((Int, Int), (Int, Int))
\end{array}$$

The predicate IsInt $\hat{2}$ appears in this context because when it was produced, the value of y was not known, and so, the predicate wasn't simplified. As we have said, this is not a problem, because the predicate will be simplified when performing constraint solving.

8.2 Constraint Solving

8.2.1 Motivation

In the presence of **poly** and **spec** annotations, the specializer does not decide the final form of polyvariant expressions, but abstracts it with evidence variables (used in every definition point **poly** and use point **spec**) until all the information can be gathered. These evidence variables are a key component of principality, as they abstract the different instances of a term, and will be replaced by conversions when the values of scheme variables are decided.

This section presents constraint solving, a process for deciding the final values of scheme and type variables that cannot be decided by simplification, and thus cannot be performed arbitrarily during specialization (in the general case, because global information is needed — indeed, a principal specialization for a term is a kind of intermediate form that takes its final value when integrated in a bigger context.) In this way, complete specializations are produced by the combination of two clearly separated parts:

- the *specification* part, where a description of the problem is constructed, corresponding to the specialization described in previous chapters, and
- an *implementation* part, where a solution for the constructed description is found, corresponding to the constraint solving presented in this section.

Following Aiken [1999], we can see that this approach treats type specialization as a static program analysis where each part of the program is analyzed locally, and then allows the application of resolution techniques for the generated constraints. In the field of type specialization, this approach provides a language allowing to express problems and to look for solutions in a uniform way.

An example of the need for constraint solving was given in the previous chapter, in Example 7.22.

As we have done with simplification, we first specify the idea of constraint solving, and then we implement an heuristic for our particular language, followed by a discussion on different aspects of the notion.

8.2.2 Specifying Solutions

To begin with, we define when a substitution mapping scheme variables to type schemes can be called a *solution*, when it can be performed, and what other components are needed.

Definition 8.12. [Solving] A *solving* from a predicate assignment Δ_1 to Δ_2, requiring the predicates of Δ', is a relation

$$S, T; C \mid \Delta_1 + \Delta' \rhd_V \Delta_2$$

where S and T are substitutions, C a conversion and V a set of type variables, such that

$$\begin{array}{rl} \text{(i)} & T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2 \\ \text{(ii)} & \mathrm{dom}(S) \cap (V \cup FTV(\Delta')) = \emptyset \end{array}$$

We say that S is the *solution* of the solving, and that V restricts the application of S.

While solving may appear similar to simplifying at a first glance, its consequences are stronger. The substitution S, the solution, may decide the values of some scheme variables and it is not required that the new (solved) predicate assignment entails the original one (in contrast to simplifying, where both predicate assignments are equivalent in some sense). It is for this reason that several different solutions may exist for a given predicate assignment, although they can be compared using the notion of 'more general' on the type schemes substituted.

Example 8.13. Recalling Example 7.22, the residual term obtained was

$$\textbf{let } f = h^u_s[\Lambda h_x.\lambda x.h_x + 1] \textbf{ in } (h^\ell_{s_1}[f]@\bullet, h^\ell_{s_2}[f]@\bullet)$$

and the predicate assignment calculated after simplification for the type (Int, Int) was

$$\begin{array}{l} h^u_s : \text{IsMG } (\forall t.\text{IsInt } t \Rightarrow t \to Int)\ s, \\ h^\ell_{s_1} : \text{IsMG } s\ (\hat{42} \to Int), \\ h^\ell_{s_2} : \text{IsMG } s\ (\hat{17} \to Int) \end{array}$$

(simplification cannot eliminate these predicates without knowing the value of s.) This assignment indicates that the value of s has to be a type scheme instance of $(\forall t.\text{IsInt } t \Rightarrow t \to Int)$ more general than $(\hat{42} \to Int)$ and $(\hat{17} \to Int)$. In this case, the only value that satisfies the restrictions is $(\forall t.\text{IsInt } t \Rightarrow t \to Int)$ which is proved by the conversions

$$\begin{array}{l} [] : (\forall t.\text{IsInt } t \Rightarrow t \to Int) \geq (\forall t.\text{IsInt } t \Rightarrow t \to Int) \\ [](\!(42)\!) : (\forall t.\text{IsInt } t \Rightarrow t \to Int) \geq (\hat{42} \to Int) \\ [](\!(17)\!) : (\forall t.\text{IsInt } t \Rightarrow t \to Int) \geq (\hat{17} \to Int) \end{array}$$

By using these conversions as evidence for h^u_s, $h^\ell_{s_1}$, and $h^\ell_{s_2}$ on the term, we obtain (modulo reduction):

$$\mathbf{let}\ f = \Lambda h_x.\lambda x.h_x + 1\ \mathbf{in}\ (f(\!(42)\!)@\bullet, f(\!(17)\!)@\bullet) : (Int, Int)$$

This last term can be transformed using the variation of constraint solving called *evidence elimination*, described in Section 8.3, to a term using tuples for the residuals of polyvariant expressions:

$$\mathbf{let}\ f = (\lambda x.42, \lambda x.17)^S\ \mathbf{in}\ (\pi^S_{1,2}\, f, \pi^S_{2,2}\, f) : (Int, Int)$$

(Observe how the function f takes two residuals in this term.)

8.2.3 Solving and Specialization

We now study how solving can be performed during specialization, by incorporating it to system P.

$$\text{(SOLV)}\ \frac{\Delta_1 \mid \Gamma \vdash_{\text{P}} e : \tau\ \hookrightarrow\ e' : \sigma \qquad S, T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2}{\Delta_2 \mid T\Gamma \vdash_{\text{P}} e : \tau\ \hookrightarrow\ C[e'] : T\sigma}$$

In contrast with the case of simplification, some cautions have to be taken to avoid unsound results: if a variable is decided when some of the information affecting its set of possible values is missing — which can happen if a scheme variable occurs anywhere in the residual type or in the type assignment Γ— then it must not be solved. This situation is captured in the rule by the use of the set $FTV(\Gamma, \sigma)$ in the solving premise (and there used for condition (ii) of Definition 8.12), and its effect shown in the following example.

Example 8.14. Specializing the term e defined as:

$$\begin{array}{l} \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\lambda^D y.(\mathbf{lift}\ x, \mathbf{lift}\ y)) \\ \ \mathbf{in}\ (\mathbf{spec}\ f\ @^D\, 1^S\ @^D\, 2^S, f) \\ : ((Int^D, Int^D), \mathbf{poly}\ (Int^S \to Int^S \to (Int^D, Int^D))) \end{array}$$

gives the residual term

$$\begin{array}{l}\Lambda h^u_f, h^\ell_f.\textbf{let}\ f = h^u_f[\Lambda h_x.\Lambda h_y.\lambda x.\lambda y.(h_x, h_y)] \\ \qquad\quad \textbf{in}\ (h^\ell_f[f]@\bullet@\bullet, f) \\ : \forall s.\text{IsMG}\ (\forall t, t'.\text{IsInt}\ t, \text{IsInt}\ t' \Rightarrow t \to t' \to (\mathit{Int}, \mathit{Int}))\ s, \\ \qquad \text{IsMG}\ s\ (\hat{1} \to \hat{2} \to (\mathit{Int}, \mathit{Int})) \\ \qquad \Rightarrow ((\mathit{Int}, \mathit{Int}), \textbf{poly}\ s)\end{array}$$

Had the variable s been solved before moving the predicates from the context into the type, a suitable solution would have been to choose $\sigma = \forall t, t'.\text{IsInt}\ t, \text{IsInt}\ t' \Rightarrow t \to t' \to (\mathit{Int}, \mathit{Int})$, giving

$$\textbf{let}\ f = \Lambda h_x, h_y.\lambda x.\lambda y.(h_x, h_y)\ \textbf{in}\ (f((1))((2))@\bullet@\bullet, f) : ((\mathit{Int}, \mathit{Int}), \textbf{poly}\ \sigma)$$

However, this solution, although correct, is not general enough; if the original term e appears in the following program (observe the use of monovariant identity functions to force two expressions to have the same residual type):

$$\begin{array}{l}\textbf{let}^D\ e = \ldots \\ \ \textbf{in let}^D\ id_1 = \lambda^D x.x \\ \ \textbf{in let}^D\ id_2 = \lambda^D x.x \\ \ \textbf{in let}^D\ g = id_2 @^D \textbf{poly}\ (\lambda^D x.\lambda^D y.(\textbf{lift}\ (id_1 @^D x), \textbf{lift}\ (id_1 @^D y))) \\ \ \textbf{in spec}\ (id_2 @^D (\textbf{snd}\ e))\ @^D 4^S\ @^D 4^S\ : (\mathit{Int}, \mathit{Int})\end{array}$$

and is specialized in a monolithic fashion, the result is

$$\begin{array}{l}\textbf{let}\ e = \ldots \\ \ \textbf{in let}\ id_1 = \lambda x.x \\ \ \textbf{in let}\ id_2 = \lambda x.x \\ \ \textbf{in let}\ g = id_2@h^u_g[(\Lambda h_{xy}.\lambda^D x.\lambda^D y.(h_{xy}, h_{xy}))] \\ \ \textbf{in}\ (h^\ell_e[(id_2@(\textbf{snd}\ e))])@\bullet@\bullet : (\mathit{Int}, \mathit{Int})\end{array}$$

with the following predicate assignment

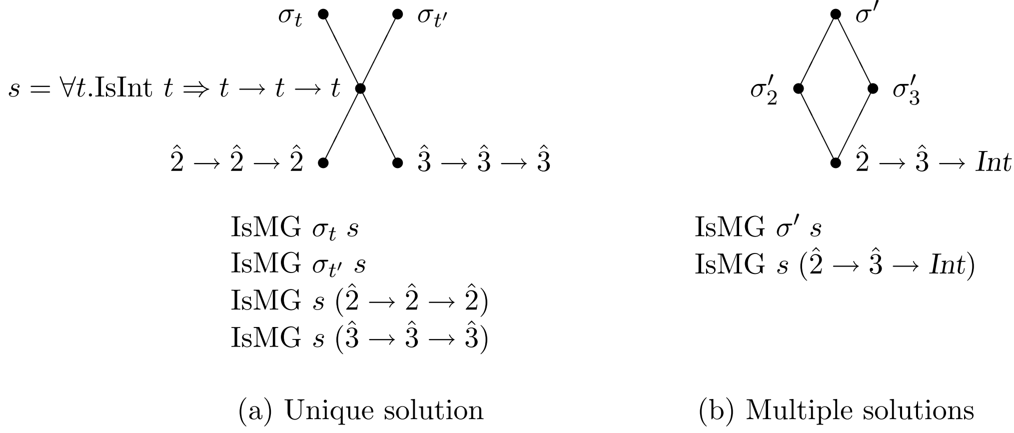
$$\begin{array}{l}h^u_f : \text{IsMG}\ (\forall t', t''.\text{IsInt}\ t', \text{IsInt}\ t'' \Rightarrow t' \to t'' \to (\mathit{Int}, \mathit{Int})) \\ \qquad\qquad (\forall t, t'.\text{IsInt}\ t, \text{IsInt}\ t' \Rightarrow t \to t' \to (\mathit{Int}, \mathit{Int})), \\ h^u_g : \text{IsMG}\ (\text{IsInt}\ t \Rightarrow t \to t \to (\mathit{Int}, \mathit{Int})) \\ \qquad\qquad (\forall t, t'.\text{IsInt}\ t, \text{IsInt}\ t' \Rightarrow t \to t' \to (\mathit{Int}, \mathit{Int})), \\ h^\ell_f : \text{IsMG}\ (\forall t, t'.\text{IsInt}\ t, \text{IsInt}\ t' \Rightarrow t \to t' \to (\mathit{Int}, \mathit{Int})) \\ \qquad\qquad (\hat{1} \to \hat{2} \to (\mathit{Int}, \mathit{Int})), \\ h^\ell_e : \text{IsMG}\ (\forall t, t'.\text{IsInt}\ t, \text{IsInt}\ t' \Rightarrow t \to t' \to (\mathit{Int}, \mathit{Int})) \\ \qquad\qquad (\hat{4} \to \hat{4} \to (\mathit{Int}, \mathit{Int})), \\ h_t : \text{IsInt}\ t\end{array}$$

But this predicate assignment is not solvable! The reason is the second predicate, which has a form forced by the premature decision about the value of s.

As with simplification, we give a proof that the rule (SOLV) is sound.

THEOREM 8.15. *Given* $\Delta_1 \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, *and if* $S, T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2$ *then, it is also the case that*

$$\Delta_2 \mid TS\Gamma \vdash_{\text{P}} e : \tau \hookrightarrow C[e'] : TS\sigma.$$



(a) Unique solution

(b) Multiple solutions

where

$$
\begin{aligned}
\sigma_t &= \forall t, t'.\text{IsInt } t, \text{IsInt } t' \Rightarrow t \rightarrow t' \rightarrow t \\
\sigma_{t'} &= \forall t, t'.\text{IsInt } t, \text{IsInt } t' \Rightarrow t' \rightarrow t \rightarrow t \\
\sigma' &= \forall t, t'.\text{IsInt } t, \text{IsInt } t' \Rightarrow t \rightarrow t' \rightarrow Int \\
\sigma'_2 &= \forall t'.\text{IsInt } t' \Rightarrow \hat{2} \rightarrow t' \rightarrow Int \\
\sigma'_3 &= \forall t.\text{IsInt } t \Rightarrow t \rightarrow \hat{3} \rightarrow Int
\end{aligned}
$$

Figure 8.3: Lattice of solutions

8.2.4 Finding Solutions

In previous sections we have defined the notion of solving, and the way to include it in the specialization process, but we have not said anything about how to actually calculate a solution. The predicates which need the calculation of solutions are those used to express polyvariance, and the scheme variables associated with them. We have seen that IsMG predicates produced by the algorithm (and not simplified away) have two possible forms: they are upper or lower bounds to some scheme variable. So, we may use the lattice of type schemes induced by the $\geq$ order to guide the looking for solutions. We can see this graphically in Figure 8.3. Any value satisfying the conditions is a good solution — if any exists. This is discussed a bit further in Section 8.3.3.

To perform the constraint solving, we proceed incrementally: in the absence of dynamic recursion there is always a scheme variable that does not depend on any other, and thus can be solved. We justify the incremental nature with the following lemma.

LEMMA 8.16. *Composition of solvings is a solving.*

That is, if $S_2 \sim_{S_1\Delta_1,\Delta'} T_1$, $S_1, T_1; C_1 \mid \Delta_1 + \Delta' \rhd_V \Delta_2$, *and* $S_2, T_2; C_2 \mid \Delta_2 + \Delta'' \rhd_V \Delta_3$ *then*

$$S_2S_1, T_2T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \rhd_V \Delta_3$$

In presence of recursion, we need a more powerful method. A discussion about this is deferred to Sections 9.7 and 14.

What about arithmetic predicates whose variables are not functionally determined by the context? This kind of situation may produce ambiguous cases for specialization, as the following example shows.

EXAMPLE 8.17. Consider the following specialization:

$$\begin{array}{l}\vdash \textbf{let}^{D}\ f = \lambda^{D}x.\textbf{lift}\ x\ \textbf{in}\ 2^{D}\ : Int \\ \hookrightarrow \Lambda h_x.\textbf{let}\ f = \lambda x.h_x\ \textbf{in}\ 2 : \forall t_x.h_x : \text{IsInt}\ t_x \Rightarrow Int\end{array}$$

Any value $\hat{n}$ is good for t_x (thus fixing h_x), but none appears to be the right choice — there is no relation between solutions. So, this term has an ambiguous specialization.

However, ambiguity may not arise: if the ambiguous term appears in a polyvariant context that is never specialized, there is no need to solve the predicates.

EXAMPLE 8.18. Consider the following specialization:

$$\begin{array}{l}\vdash \textbf{let}^{D}\ g = \textbf{poly}\ (\lambda^{D}x.\textbf{lift}\ x)\ \textbf{in}\ 2^{D}\ : Int \\ \hookrightarrow \textbf{let}\ g = h^{u}[\Lambda h_x.\lambda x.h_x]\ \textbf{in}\ 2 : Int\end{array}$$

with the predicate assignment

$$h^{u} : \text{IsMG}\ (\forall t_x.h_x : \text{IsInt}\ t_x \Rightarrow Int)\ s$$

which can be solved trivially by choosing s to be the value of the upper bound, with the consequence that the predicate IsInt t_x needs not to be solved.

8.2.5 An Algorithm for Constraint Solving

We have already defined the notion of resolution, and we have discussed some aspects of possible solutions and how to find them. In this section we give an algorithmic implementation of a heuristic to find, in those cases when it is possible, a resolution for all the predicates in a specialization judgement. The goal of the presentation is to establish the form in which solutions can be found, and how to use them, and not to give a detailed executable implementation. For this reason, we present several functions expressed in a pseudo-functional code.

Constraint solving is defined by the function **stepSolve**, that, given a predicate assignment and a (scheme) variable to solve, finds a solution for it and simplifies the resulting predicates.

We use vectorial notation for lists here: for example, $\bar{\sigma}$ represents a list of type schemes, while σ_i will denote each of its elements.

Function **stepSolve** is defined in terms of several functions, described below (from the most primitive to the most complex ones):

- **glb:** it calculates the greatest lower bound of a list of qualified type schemes. For simplicity, we describe the algorithm working on *two* schemes, as its generalization is simple (either by iteration of couples or by a generalization of the method).

  Given $\sigma_1 = \forall\alpha_i.\Delta_1 \Rightarrow \tau_1$ and $\sigma_2 = \forall\beta_j.\Delta_2 \Rightarrow \tau_2$, and assuming $\alpha_i \cap \beta_j = \emptyset$ (in other case, we can perform an α-conversion), the algorithm

  1. calculates the most general unifier U for τ_1 and τ_2, $\tau_1 \sim^{U} \tau_2$.

2. returns the generalization of the type, qualified with both predicate assignments where U has been applied: $glb(\sigma_1, \sigma_2) = \text{Gen}_{\emptyset,\emptyset}(U\Delta_1, U\Delta_2 \Rightarrow U\tau_1)$

- **conversion:** given two type schemes σ and σ', returns, if it exists, a conversion C such that $C : \sigma \geq \sigma'$.

- **makeMoreGeneral:** similar to *conversion*, but 'forcing' the schemes to be comparable (in case they weren't) by adding predicates when necessary. Given two type schemes σ and σ', it returns a conversion C and a predicate assignment Δ such that $C : \sigma \geq (\Delta \mid \sigma')$.

- **findSolution:** implements the searching of a solution, given the upper and lower bounds for a scheme variable.

$$
\begin{array}{lll}
\mathit{findSolution}\,\bar{\sigma}^u\;\bar{\sigma}^\ell\;s & = & \textbf{let} \\
 & & \quad \begin{array}{lcl}
 \sigma & = & \mathit{glb}\,\bar{\sigma}^u \\
 \bar{C}^u & = & \mathit{conversion}\,(\bar{\sigma}^u,\ \sigma) \\
 (\bar{C}^\ell,\ \bar{\mathit{\Delta}}_f) & = & \mathit{makeMoreGeneral}\,(\sigma,\ \bar{\sigma}^l) \\
 \mathit{\Delta}_f & = & \mathit{concat}\ \bar{\mathit{\Delta}}_f
 \end{array} \\
 & & \textbf{in} \\
 & & \quad (\bar{C}^u,\ \bar{C}^\ell,\ \mathit{\Delta}_f,\ \sigma)
\end{array}
$$

- **stepSolve:** it is the main step of our constraint solving heuristic. It takes a predicate assignment of the form

$$\Delta = \{h_{u_i} : \text{IsMG}\ \sigma_{u_i}\ s\},\{h_{l_j} : \text{IsMG}\ s\ \sigma_{l_j}\},\Delta_s$$

and a variable s with $s \notin FTV(\Delta_s, \sigma_{u_i}, \sigma_{l_j})$, it returns a substitution S, a conversion C, two predicate assignments Δ_f and Δ' such that the resolution $S, T; C \mid \Delta + \Delta_f \rhd_V \Delta'$ holds for any V not containing s. It is implemented as follows:

$$
\begin{array}{l}
\mathit{stepSolve}\ \mathit{chooseEv}\ s\ (\{h_{u_i} : \text{IsMG}\ \sigma_{u_i}\ s\},\ \{h_{l_j} : \text{IsMG}\ s\ \sigma_{l_j}\},\ \mathit{\Delta}_s)\ = \\
\quad \textbf{let} \\
\quad\quad \begin{array}{lcl}
(\bar{C}^u, \bar{C}^\ell, \mathit{\Delta}_f,\ \sigma) & = & \mathit{findSolution}\,(\bar{\sigma}^u, \bar{\sigma}^\ell, s) \\
(\bar{C}_2^u, \bar{C}_2^\ell) & = & \mathit{chooseEv}\,(\bar{C}^u, \bar{C}^\ell) \\
(T, C, \mathit{\Delta}') & = & \mathit{simplify}\,(\mathit{\Delta}_s, \mathit{\Delta}_f)
\end{array} \\
\quad \textbf{in} \\
\quad\quad (\,[\sigma/s],\ T,\ C \circ (\,\bar{h}^u \leftarrow \bar{C}_2^u\ \cdot\ \bar{h}^\ell \leftarrow \bar{C}_2^\ell\,),\ \mathit{\Delta}_f,\ \mathit{\Delta}'\,)
\end{array}
$$

The argument *chooseEv* is a function receiving and returning a pair of lists of conversions; it will be used in Section 8.3 to eliminate evidence from the term. If evidence elimination does not have to be performed, *id* can be used.

Function **stepSolve** provides one step of the algorithm. As we have justified in Lemma 8.16, constraint solving can be composed of several individual steps, in such a way that their composition is a solution for the original assignment. The variable to be solved in each individual step can be anyone, provided it satisfies the condition of not being dependent on others — thus any of them can be chosen; an algorithm to select

this variable must check this *dependency condition*, and pick any of those satisfying it. The final algorithm is the composition of the function **stepSolve** with itself, iterating over each of the solvable variables.

This iterative process can be optimized in several ways. For example, the dependence between two type schemes can be "remembered" from one iteration to the next, forcing the resolution order without further calculations.

LEMMA 8.19. *The heuristic presented is correct wrt. the definition of the constraint solving relation. That is:*

1. *If* $\sigma = glb(\sigma_1, \sigma_2)$ *then there exist conversions* C_1, C_2 *such that* $C_i : \sigma_i \geq \sigma$, *and for any* σ' *such that* $\sigma_i \geq \sigma'$ *it will be true that* $\sigma \geq \sigma'$.

2. *findSolution finds a solution for s, respecting the given bounds. That is, the conversions for upper bounds satisfy that* $C_i^u : \sigma_i^u \geq \sigma$ *and similarly for lower bounds,* $C_i^\ell : \sigma_i^\ell \geq (\Delta_f^\ell \mid \sigma)$.

3. *If* $(S, T, C, \Delta_f, \Delta') = stepSolve\ id\ s\ \Delta$ *and* $s \notin V$ *then*

$$S, T; C \mid \Delta + \Delta_f \triangleright_V \Delta'.$$

It is important to remark that the algorithm *solve*, obtained by the repeated composition of **setpSolve** with itself, is not defined for every predicate assignment. In particular, those assignments that cannot be divided in three separate parts for a given scheme variable: (i) its upper bounds, (ii) its lower bounds, and (iii) the predicates not containing it. Such contexts are not generated by specialization for any expression in our simplified language. However, additional constructs in the language may introduce new predicate forms for which determining all the upper and lower bounds depends on the resolution of that variable (and in fact they do: eg. dynamic recursion). This is discussed in detail in Sections 9.7 and 14. For this reason we call this algorithm an *heuristic*.

8.2.6 Examples

We provide a detailed example for the better understanding of the process of constraint solving just defined. It shows, step by step, the application of the constraint solving algorithm, motivating:

1. the iterated resolution of different scheme variables, showing how the solution for one of them provides information to solve the following one.

2. the movement of predicates between the type and the contextual predicate assignment (to and from **poly**s),

3. the flow of evidence information between the term and its type, by means of evidence variables and conversions.

Additionally, it can also be observed that the final result of a constraint solving may be excessively 'complicated', full of evidence expressions (abstractions and applications) that are a byproduct of the internal representation of the process. These constructs are eliminated using the *evidence elimination* procedure, a modification of the constraint solving phase, described in Section 8.3.

Consider the following principal specialization:

$$\begin{array}{l} \Delta \vdash_{\rm P} \textbf{let}^{D}\ f = \textbf{poly}\ (\lambda^{D} x.\textbf{lift}\ (x +^{S} 2^{S})) \\ \quad\quad \textbf{in let}^{D}\ g = \textbf{poly}\ (\lambda^{D} y.\textbf{spec}\ f\ @^{D}\ (y +^{S} 1^{S})) \\ \quad\quad\quad \textbf{in spec}\ g\ @^{D}\ 2^{S} : Int^{D} \\ \quad \hookrightarrow\ \textbf{let}\ f = h_f^u[\Lambda h_x.\Lambda h''_x.\lambda x.h''_x] \\ \quad\quad\quad \textbf{in let}\ g = h_g^u[\Lambda h_y.\Lambda h_f^\ell.\Lambda h'_y.\lambda y.h_f^\ell[f]@\bullet] \\ \quad\quad\quad\quad \textbf{in}\ h_g^\ell[g]@\bullet : Int \end{array}$$

where the context is given by the predicate assignment Δ:

$$\begin{array}{rl} \Delta = & h_g^u : \text{IsMG}\ (\forall t_y, t'_y.\text{IsInt}\ t_y, \text{IsMG}\ s_f\ (t'_y \rightarrow Int), t'_y := t_y + \hat{1} \Rightarrow t_y \rightarrow Int)\ s_g, \\ & h_g^\ell : \text{IsMG}\ s_g\ (\hat{2} \rightarrow Int), \\ & h_f^u : \text{IsMG}\ (\forall t_x, t''_x.\text{IsInt}\ t_x, t''_x := t_x + \hat{2} \Rightarrow t_x \rightarrow Int)\ s_f, \end{array}$$

In the expression there appear two polyvariant functions, each one with a single use. But the application of f is inside the body of g, so the arguments of f will not be known until all the possible forms for g have been determined (note that had g not been applied anywhere, its code would have been the empty tuple, and the same for f.) During specialization, when looking for a variable to solve, both variables s_f and s_g satisfy the dependency condition, but only s_g is solvable at this stage.
The value for s_g is chosen to be the only upper bound, and the resulting predicate assignment is simplified, producing $\hat{2}$ as the value for t_y.

The predicates appearing in the type scheme of the upper bound for s_g are added to the context, but with fresh variables. These predicates are those captured by the assignment named Δ' in the rule (SOLV). The result is

$$\begin{array}{lll} (1) & h_1^* : \text{IsInt}\ \hat{2} & = (\text{IsInt}\ t_y)[\hat{2}/t_y] \\ (2) & h_2^* : \text{IsMG}\ s_f\ (t'_y \rightarrow Int) & \\ (3) & h_3^* : t'_y := \hat{2} + \hat{1} & = (t'_y := t_y + \hat{1})[\hat{2}/t_y] \end{array}$$

The first and third predicates are trivially simplified calculating the addition, and assigning the value $\hat{3}$ to the variable t'_y; the second one is left in the remaining assignment. In the term, the following evidence expressions were used during solving:

$$\begin{array}{ll} [] & \text{for } h_g^u \\ []((2))((h_2^*))((3)) & \text{for } h_g^\ell \end{array}$$

At this step, the specialization is (after applying the two conversions):

$$\begin{array}{l} \textbf{let}\ f = h_f^u[\Lambda h_x, h''_x.\lambda x.h''_x] \\ \ \textbf{in let}\ g = \Lambda h_y, h_f^\ell, h'_y.\lambda y.h_f^\ell[f]@\bullet\ \textbf{in}\ g((2))((h_2^*))((3))@\bullet : Int \end{array}$$

with the following predicate assignment as its context

$$\begin{array}{l} h^u_f : \text{IsMG } (\forall t_x, t''_x.\text{IsInt } t_x, t''_x := t_x + \hat{2} \Rightarrow t_x \to \mathit{Int})\ s_f, \\ h^*_2 : \text{IsMG } s_f\ (\hat{3} \to \mathit{Int}) \end{array}$$

The rest of the work is simple: there is only one single scheme variable to solve, and it occurs in two predicates: an upper bound, and a lower one. Again the upper bound is chosen as the solution for s_f, with $[]$ as the evidence for h^u_f, and then, the last predicate remaining is simplified,

$$h^*_2 : \text{IsMG } (\forall t_x, t''_x.\text{IsInt } t_x, t''_x := t_x + \hat{2} \Rightarrow t_x \to \mathit{Int})\ (\hat{3} \to \mathit{Int})$$

performing the unification between t_x and $\hat{3}$, resulting in the value $\hat{5}$ for t''_x, so using $[]((3))((5))$ as evidence for h^*_2. The solving process finishes, resulting in the term

$$\begin{array}{l} \textbf{let } f = \Lambda h_x, h''_x.\lambda x.h''_x \\ \ \textbf{in let } g = \Lambda h_y, h^\ell_f, h'_y.\lambda y.h^\ell_f[f]@\bullet \textbf{ in } g((2))(([]((3))((5))))((3))@\bullet : \mathit{Int} \end{array}$$

that is a term equivalent to the solution expected.
The equivalence concerns the use of evidence abstraction and application: if all the uses of f and g are known, the applications of evidence can be 'moved' towards the abstractions, and β_v reductions performed (static tuples have to be used), resulting in:

$$\begin{array}{l} \textbf{let } f = \lambda x.5 \\ \ \textbf{in let } g = \lambda y.f@\bullet \\ \quad \textbf{in } g@\bullet : \mathit{Int} \end{array}$$

In the next section, Section 8.3, a process to calculate this kind of transformation is defined, based on the process of context solving.

8.2.7 Discussion

There are some aspects of the constraint solving process that deserve to be discussed a bit further.

The first issue is related with the implementation of the solving process. The rule (SOLV) allows to perform resolution during specialization, and the heuristic defined allows the solving of those variables whose information is completely provided. In the actual implementation, some way to track down dependences among scheme variables would allow us to perform this stage much more efficiently. Additionally, dynamic recursion needs a different treatment, because of the form of constraints it imposes on variables; however, we think that the present formalization clarifies the restrictions and complications of the task, and in doing so goes further into the proper definition of the problem and its solution.

The second issue is the similarities and differences between simplification and solving. As processes, they look similar, instantiating variables and eliminating predicates. However, the fundamental difference is the kind of decisions taken in each case:

- simplifications work only on contexts, independently of the term under specialization, removing redundant predicates, or expressing them in simpler forms. For this reason, all the values decided are unique (by virtue of the 'equivalence' given by the double $\Vdash$), and thus deducible wrt. entailment from the context they appear.

- constraint solving, on the other hand, works both with the predicates and the terms (by transforming them according to the evidence calculated). The process has to take decisions about the values of variables, although those values are not a direct consequence of the context — several possible solutions for some of them may exist. The heuristic presented here chooses the greatest lower bound of all the upper bounds, because it is a simple choice. We conjecture that all the possible solutions are equivalent in some sense, although that needs further treatment to be proved.

The decisions taken during constraint solving represent the answers to the problems found during the specialization phase that cannot be solved locally. So, the predicates to be solved depend essentially on the constructs present in the term under specialization. Our work here has to be regarded mainly as a base for the search for solution algorithms for more complex languages.

8.3 Evidence Elimination

The purpose of evidence elimination is to remove all the evidence constructs (abstractions and applications) introduced during the constraint solving phase, in favour of static tuples, thus obtaining a residual term in the same language used by Hughes [1996b]. Additionally, the repeated abstraction and application of evidence has a computational overhead, so the terms obtained with evidence elimination are more efficient (indeed static tuples also have computational overhead, but the process of arity raising, described by Hughes, removes it.) This process is obtained by a slight modification of the process of constraint solving.

Consider again the principal specialization of Example 6.17.

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)\\ \quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D\ : (Int^D, Int^D)^D\\ \hookrightarrow\ \Lambda h', h_1, h_2.\mathbf{let}\ f' = h'[\Lambda h.\lambda x'.h + 1]\ \mathbf{in}\ (h_1[f']@\bullet, h_2[f']@\bullet)\\ \quad :\ \forall s.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow Int)\ s,\\ \qquad\quad \mathrm{IsMG}\ s\ (\hat{42} \rightarrow Int), \mathrm{IsMG}\ s\ (\hat{17} \rightarrow Int) \Rightarrow (Int, Int)^D\end{array}$$

The constraint solving process generates *one single* version of function f, which is then applied to different evidence:

$$\mathbf{let}\ f = \Lambda h_x.\lambda x.h_x\ \mathbf{in}\ (f((42))@\bullet, f((17))@\bullet) : (Int, Int)$$

One possible alternative residual term, equivalent to the previous one, but with two versions for function f, would have been

$$\mathbf{let}\ f = (\lambda x.42, \lambda x.17)^S\ \mathbf{in}\ (\pi^S_{1,2}\ f@\bullet, \pi^S_{2,2}\ f@\bullet) : (Int, Int) \tag{8.1}$$

We expect evidence elimination to remove all the abstractions and applications of evidence possible, without introducing more elements than needed, and being 'optimal' in some sense (for example, not creating two versions of a function when only one is needed).

In the rest of this section we present an extension to the constraint solving phase performing evidence elimination.

8.3.1 Extensions to the language of evidence

As we have said, evidence constructs will be removed in favour of static tuples. For that reason, we extend the language of evidence with new constructs for static tuples:

$$\begin{array}{rcl} e' & = & \ldots \mid (e', \ldots, e')^{S} \mid \pi^{S}_{n,n}\, e' \\ C & = & \ldots \mid (C, \ldots, C)^{c} \mid \pi^{c}_{n,n}\, [] \end{array}$$

The first construct is used to create static tuples (both in the languages of expressions and conversions), and the second one is used to create projections for those tuples (again in both languages). So, $(e'_1, \ldots, e'_n)^S$ denotes a residual term for every natural number n, and $\pi^S_{i,n}\, e'$ projects the i-th component of a tuple e' with n components, for all natural numbers i and n such that $i \leq n$. The special case of the empty tuple, $()^S$, is also a valid expression, that can be regarded as a term with no information, similar to $\bullet$ (indeed sometimes it is convenient not to distinguish them.)

In the case of conversions, the new forms construct tuples and projections in the following way:

$$\begin{array}{c} \pi^{c}_{i,n}[e'] = \pi^{S}_{i,n}\, e' \\ (C_1, \ldots, C_n)^{c}[e'] = (C_1[e'], \ldots, C_n[e'])^{S} \end{array}$$

The constructs are annotated with ^{c} to distinguish them from their counterparts in the language of expressions — this new label is an annotation representing the fact that we are expressing something in the language of conversions. Observe that the conversions creating static tuples replicate the code of e'!

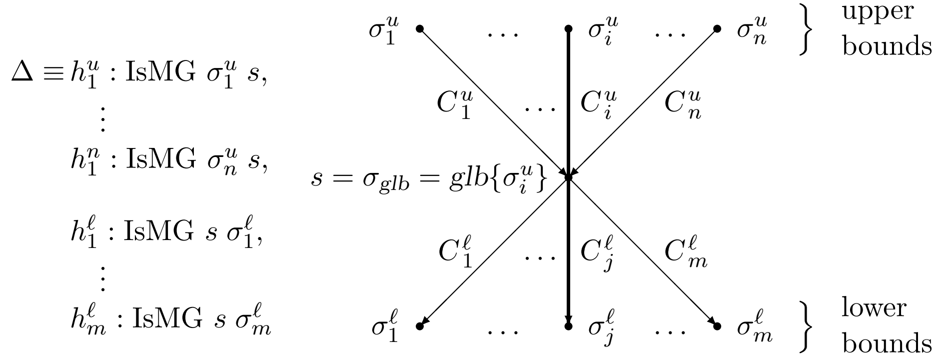
The reason for using static tuples is that this is not the final step of specialization: these tuples can be removed by a later phase of arity raising.

8.3.2 Eliminating Evidence via Constraint Solving

To eliminate evidence expressions, we proceed, during constraint solving, to construct different conversions as evidence for the predicates being solved. Those conversions are constructed based on the set of upper and lower bounds for the scheme variable being solved.

- Let s be the variable selected for solving. We know that it only appears in the bounds, because the dependency condition must be satisfied.
- Let σ^u_i and σ^ℓ_j be the sets of upper and lower bounds for s, respectively.

Two conditions hold: all the upper bounds are more general (wrt. $\geq$) than any lower bound, and there exists the greatest lower bound of all of them, $\sigma_{glb} = glb(\{\sigma^u_i\}_{i=1..n})$;



The conversion between any σ_i^u and σ_j^ℓ is obtained composing $C_j^\ell \circ C_i^u$.

Figure 8.4: Construction of conversions to eliminate evidence

this value, σ_{glb} is more general than any of the lower bounds σ_j^ℓ, and it is the value chosen by the algorithm as the value of s.

The conversions used as evidence for variable h_i^u during the algorithm of Section 8.2 were C_i^u, which converts and expression of type σ_i^u in one of type σ_{glb}. Similarly, C_j^ℓ was used for h_j^ℓ, converting terms of type σ_{glb} into terms of the required type, σ_j^ℓ. The reason for this choice was that, in any point were a polyvariant expression is used (**spec**ed), an expression of type σ_i^u has to be converted in one of type σ_j^ℓ — that is obtained by the composition of the conversions mentioned. This information is depicted graphically in Figure 8.4.

This point is where evidence elimination differs from constraint solving: instead of converting every expression producing upper bounds to the type σ_{glb} in the definition points (**poly**s), and the expressions producing lower bounds to the corresponding type at the use points (**spec**s), to perform evidence elimination the definition points will be converted to *all the possible forms required*, by using the static tuples just introduced, and the use points will only select the component needed by using the projections. This is expressed by changing the evidence constructed to:

$$\begin{array}{rl} (C_1^\ell \circ C_i^u, \ldots, C_m^\ell \circ C_i^u)^c & \text{for IsMG } \sigma_i^u\ s \\ \pi_{j,m}^c\,[] & \text{for IsMG } s\ \sigma_j^\ell \end{array}$$

The justification is that at every use (j-th lower bound) where the i-th definition is used, the composition of $(C_j^\ell \circ C_i^u)^c$ and $\pi_{j,m}^c\,[]$ is

$$\pi_{j,m}^c\,[] \circ (C_1^\ell \circ C_i^u, \ldots, C_m^\ell \circ C_i^u)^c = C_j^\ell \circ C_i^u$$

which is equivalent to the conversion used in the previous algorithm.

For the implementation of this process, let's recall that the function *stepSolve* takes an argument that is a function for transforming two lists of conversions — during constraint solving the identity functions was used. Instead, we define the function *eliminateEv* such that the function *stepSolve eliminateEv* implements the desired process:

$$
\begin{array}{l}
\mathit{eliminateEv}\,(\bar{C}^{u}_{i=1..n},\ \bar{C}^{\ell}_{j=1..m}) \ = \\
\qquad \textbf{let} \\
\qquad\quad \mathit{ubs} = \{\ (C^{\ell}_{1} \circ C^{u}_{i}, \ldots, C^{\ell}_{m} \circ C^{u}_{i})^{c}\ \}_{i=1..n} \\
\qquad\quad \mathit{lbs}\ \ = \{\ \pi^{c}_{j,m}\,[]\ \}_{j=1..m} \\
\qquad \textbf{in} \\
\qquad (\mathit{ubs},\ \mathit{lbs})
\end{array}
$$

Let's see an example.

EXAMPLE 8.20. Consider the following specialization:

$$
\begin{array}{l}
\Delta \vdash_{\mathrm{P}} \textbf{let}^{D}\ f = \textbf{poly}\ (\lambda^{D} y.\textbf{lift}\ y)\ \textbf{in spec}\ f\ @^{D}\,7^{S}\ : \mathit{Int} \\
\qquad \hookrightarrow \\
\qquad \textbf{let}\ f = h^{u}[\Lambda h_{y}.\lambda y.h_{y}]\ \textbf{in}\ h^{\ell}[f]@\bullet : \mathit{Int}
\end{array}
$$

where Δ is

$$
\begin{array}{l}
h^{u} : \mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int})\ s, \\
h^{\ell} : \mathrm{IsMG}\ s\ (\hat{7} \rightarrow \mathit{Int})
\end{array}
$$

There are two possible solutions for s: the upper or the lower bound. In the first case, the evidence will be $[]$ for h^u and $[]((7))$ for h^ℓ, giving, by constraint solving, the term:

$$\textbf{let}\ f = \Lambda h_{y}.\lambda y.h_{y}\ \textbf{in}\ f((7))@\bullet : \mathit{Int}$$

This term contains the evidence abstraction and application that has to be eliminated. In the second case, the evidence should be $[]((7))$ for h^u and $[]$ for h^ℓ, giving the term:

$$\textbf{let}\ f = \lambda y.7\ \textbf{in}\ f@\bullet : \mathit{Int}$$

However, after evidence elimination, both solutions give the same result:

$$\textbf{let}\ f = (\lambda y.7)^{S}\ \textbf{in}\ \pi^{S}_{1,1}\,f@\bullet : \mathit{Int}$$

8.3.3 Discussion

There are some issues that deserve further discussion.

The first one concerns the criteria used to decide how many elements a tuple should have. As it has been said, polyvariance is the ability to produce *different* specializations for a single term. But the processes of proper specialization and constraint solving do not replicate code, so they are not able to give those different versions of the same expression. It is during evidence elimination, by the use of conversions producing tuples, that the code is replicated. It is important, when applying this method, to decide how many copies of an expression will be produced. There are different criteria to decide this.

- One for each *use*: That is, a different conversion for each occurrence of a **spec**— but as there may be different **spec**s with the same arguments, the resulting tuple will usually contain repeated elements, which is not desirable.

- One for each *residual type*: The conversions do not depend on the values of the arguments of a polyvariant function, but on their types. So, another possibility is to produce one element in the tuple for every residual type. However, this choice still produces repeated elements.

- One for each *conversion (syntactically)*: It is usual that two different lower bounds can be satisfied by one conversion, so resulting in the same residual code. This choice will require the syntactic comparison of conversions, something that can be easily implemented, but that can produce few enhancements from the previous choice.

- One for each *conversion (extensionally)*: This variant is similar to the previous case, but testing the conversions by the semantic equivalence. This option, however, is computationally unfeasible.

- One for each *residual code*: A new component will be created only if the result of the composition of conversions produces a new residual code. However, to check this every lower bound has to be compared with every upper bound, and the process is then expensive.

The alternatives presented are ordered from the one identifying fewest elements to the one identifying all of them. Having the implementation cost in mind, the most reasonable one seems to be the second one, because, although it is not optimal, it provides a good trade-off.

The lack of optimality for this choice can be seen in the following example.

EXAMPLE 8.21. Consider the source term

$$\begin{array}{l} \lambda^D y.\mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x) \\ \quad\quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ y) \\ : Int^S \rightarrow^D (Int^D, Int^D, Int^D) \end{array}$$

The following specializations are possible:

1. Specialization as the one given by Hughes [1996b] (considering the term in isolation; with a different context the result may vary):

$$\begin{array}{l} \lambda y.\,\mathbf{let}\ f = (\lambda x.13)^S \\ \quad\quad \mathbf{in}\ (\pi^S_{1,2}\, f@\bullet, \pi^S_{1,2}\, f@\bullet, \pi^S_{1,2}\, f@y) : \hat{13} \rightarrow (Int, Int, Int) \end{array}$$

2. Principal specialization (with simplification):

$$\begin{array}{l} \Lambda h^u, h^\ell_{13}, h_y, h^\ell_y.\lambda y.\mathbf{let}\ f = h^u[\Lambda h_x.\lambda x.h_x] \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ (h^\ell_{13}[f]@\bullet, h^\ell_{13}[f]@\bullet, h_y[f]@\bullet) \\ : \forall t_y, s.\mathrm{IsMG}\ (\forall t_x.\mathrm{IsInt}\ t_x \Rightarrow t_x \rightarrow Int)\ s, \\ \quad \mathrm{IsMG}\ s\ (\hat{13} \rightarrow Int), \\ \quad \mathrm{IsInt}\ t_y, \\ \quad \mathrm{IsMG}\ s\ (t_y \rightarrow Int) \Rightarrow t_y \rightarrow (Int, Int, Int) \end{array}$$

3. Principal specialization and constraint solving:

$$\Lambda h_y.\lambda y.\textbf{let } f = \Lambda h_x.\lambda x.h_x$$
$$\textbf{in } (f((13))@\bullet, f((13))@\bullet, f((h_y))@\bullet)$$
$$: \forall t.\text{IsInt } t \Rightarrow t \rightarrow (Int, Int, Int)$$

4. Constraint solving with evidence elimination:

$$\Lambda h_y.\lambda y.\textbf{let } f = (\lambda x.13, \lambda x.h_y)^S$$
$$\textbf{in } (\pi^S_{1,2}\, f@\bullet, \pi^S_{1,2}\, f@\bullet, \pi^S_{2,2}\, f@\bullet)$$
$$: \forall t.\text{IsInt } t_y \Rightarrow t_y \rightarrow (Int, Int, Int)$$

In the source term, there are two specializations of f for an argument with the same residual type $(\hat{13})$, and another one for an argument with a type still unknown. When specializing, two identical IsMG predicates are generated — and one of them is eliminated by simplification — so the same (abstracted) evidence h^ℓ_{13} is used. However, the third case, **spec** $f\ @^D\, y$, is more problematic: it cannot be known if the value of y will be 13 or not; in the latter case a new specialization is needed, but in the former, there isn't. So, what decision should be taken?

In the specialization given by Hughes [1996b] (8.21-1), the function f is specialized to a tuple with a single element, but keeping internally the possibility of further extension. If during the specialization of the context (remember that in this case, the specialization is monolithic), a different value for y is discovered, some backtracking is done, and a new component is created in the tuple. When the specialization ends, the last task is to 'close' all the tuples, fixing the value of y to 13. This is a decision that is not desirable for modular specialization.

The principal specialization (8.21-2) is the most general form, expressing all the possible solutions. But the term is not in its final form: the value of the variable s has not been decided.

When constraint solving is considered (8.21-3), there is no replication of code. However, some evidence constructs appear that cannot be eliminated, and the term differs for the one desired (using *different* specializations for each **spec**).

Finally, in (8.21-4) the term after evidence elimination is presented. As there are two predicates as lower bounds, the resulting tuple will have two components. But if later this term were to be applied to an argument with value 13^S, the second component would be identical to the first one, and the process of evidence elimination would not be able to identify this fact.

One possibility to fix this problem is to wait until *all* the lower bounds are closed, but this contradicts our purpose of modularity. Another solution would be to formalize a notion of 'extensible tuples', creating a tuple with a single component but 'open', thus accepting further components in case they are needed.

Another issue to be discussed is the choice of the greatest lower bound as the solution for a given variable. As we have seen, there are several possible solutions that we can choose. But as we have shown in Example 8.20, after evidence elimination all of them may give the same residual term. We conjecture that any choice of the possible values for a given variable s will produce the same term after evidence elimination. There exists

a parallel between this property and the notion of *coherence* as discussed in the theory of qualified types [Jones, 1994a] (and originally defined by Breazu-Tannen *et al.* [1989]); it means, basically, that 'the meaning of a term does not depend on the way that is was type checked' [Jones, 1993].

In the case of Jones' work, the key element for the proof of coherence was the property of uniqueness of evidence. But in the presence of conversions in the language of evidence, this property does no longer hold. However, we are working on a proof that defines 'levels' of evidence, according to the number of nested conversions, and we think that this will allow us to overcome the problems.

This completes our discussion.

Chapter 9

Extending the Source Language

Since practical programming languages are typically large and very complex, programming language design involves careful and separate consideration of various sublanguages. Of course, it is important to keep in mind that a small language with only few constructs may give false impressions. We might conclude that a programming language is much simpler than it really is, or we might rely on properties that are immediately destroyed when important features are added. Therefore, good taste and careful judgment are required. In developing a programming language theory, or applying theoretical analysis to practical situations, we must always keep in mind the nature of our simplifying assumptions and how they may affect the conclusions we reach.

Foundations of Programming Languages
John C. Mitchell

The language considered in the previous chapters is a small subset of a real language. To consider examples of some interest, such as the interpreter for lambda-calculus, we need to extend that language with new constructs, as we have done in Section 3.4, considering what the principal specialization of these constructs should be. Again, adding static characters, strings, etc., and additional operations on numbers is very similar to the treatment we have shown for numbers. Static let, booleans, static functions, recursion, and datatypes will involve the same kinds of ideas, but the details have to be worked out with more care.

In this chapter we describe these extensions to the source language, and the extensions needed in the residual language to express their specialization, and in the constraint solving to produce sensible results. We have implemented some of them in our prototype (described in Chapter 10), in order for it to be able to specialize the interpreter for lambda calculus.

We consider the ability to fail (Section 9.1), dynamic and static booleans (Section 9.2), static lets (Section 9.3), static functions (Section 9.4) and static recursion (Section 9.5), static Haskell-like datatypes — that is, tagged recursive sums-of-products — (Section 9.6), and dynamic recursion (Section 9.7). In almost all cases new kinds of predicates are introduced, and constraint solving is extended to handle them. The most notable exception is dynamic recursion: no new construct or predicate is needed; however, as we have mentioned in Section 4.4.3, this feature involves complex manipulations. Using the framework of principal specialization, we are able to show where the problems are: in particular, constraint solving has to be more involved, because dynamic recursive programs produce non-linear constraints — that is, constraints where the same variable appears on both sides of an inequation. We discuss the problems, and suggest a possible line to extend constraint solving to handle this case.

9.1 Failure

The ability to fail when residual types do not match is very important, as we have seen in Chapter 3, especially when we use specialization to perform type checking. But in the original formulation, failure can only be obtained by forcing two source expressions with different residual types into the same monovariant context (as in $\mathbf{if}^D\ \mathit{True}^D$ **then** e_1 **else** e_2 or $\mathbf{let}^D\ i = \lambda^D x.x$ **in** $(i\ @^D e_1, i\ @^D e_2)$). This will produce strange error messages if we use it when failure is needed — see Example 4.10.

Another characteristic of failure in the original formulation is that failure inside a **poly** has to be delayed until the polyvariant expression is used in some **spec** — see Example 4.9. In the system presented in Chapters 6 and 7, failure inside a **poly** is reported when the polyvariant expression is being specialized. To have the same failure behaviour as in the original formulation, we need to introduce a new predicate expressing failure that will produce an actual failure when constraint solving tries to produce a solution for it. Then we can define

$$\delta ::= \ \ldots\ \mid \text{Fail}\ \mathit{String}$$

with the notion that this predicate cannot be satisfied — there is no evidence for it — and so constraint solving must fail if it tries to solve it. The failure predicate usually results from failed unifications, but if these unifications appear inside a polyvariant expression that is never used, the failure must not be raised.

The delaying effect of this predicate can be observed in the following example.

Example 9.1. Compare this specialization with that of Example 4.9.

$$\begin{array}{l} \vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\mathbf{let}^D\ id = \lambda^D x.x\ \mathbf{in}\ (id\ @^D 1^S, id\ @^D 2^S)^D) \\ \quad \mathbf{in}\ 3^D \\ \quad : \mathit{Int}^D \hookrightarrow \mathbf{let}\ f = \Lambda h.\mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ (id@\bullet, id@\bullet)\ \mathbf{in}\ 3 : \mathit{Int} \end{array}$$

The residual type of expression f is **poly** ($\forall t.$Fail "Cannot unify $\hat{1}$ with $\hat{2}$" $\Rightarrow (\hat{1}, t)$) and the failure will only be produced if the f is specialized (so, for example, replacing 3^D by **spec** f will make the specialization fail with error message "Cannot unify $\hat{1}$ with $\hat{2}$").

Another extension that we can make is to add a primitive for failure in the source language, similar to the **error** primitive in Haskell. The dynamic version of it will be copied in the residual language, but the static version will use the new predicate Fail to express its specialization.

$$e ::= \ \ldots\ \mid \mathbf{error}^S\ \mathit{String}$$

The rule to specialize the new primitive is the following one.

$$\text{(FAIL)}\quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau' \qquad \Delta \Vdash v : \text{Fail s}}{\Delta \mid \Gamma \vdash_{\text{P}} \mathbf{error}^S\ \text{s} : \tau \hookrightarrow v : \tau'}$$

This new construct can be used instead of Wrong^S in any of the examples for the interpreter of lambda-calculus to produce a failure with an appropriate error message (that is, the expression

$$\begin{array}{lcl} e & ::= & \ldots \mid \mathit{True}^{D} \mid \mathit{False}^{D} \mid \mathbf{if}^{D}\, e \;\mathbf{then}\; e \;\mathbf{else}\; e \mid e ==^{D} e \\ \tau & ::= & \ldots \mid \mathit{Bool}^{D} \\ e' & ::= & \ldots \mid \mathit{True} \mid \mathit{False} \mid \mathbf{if}\; e' \;\mathbf{then}\; e' \;\mathbf{else}\; e' \mid e' {==} e' \\ \tau' & ::= & \ldots \mid \mathit{Bool} \end{array}$$

$$\text{(DTRUE)} \quad \Delta \mid \Gamma \vdash_{\mathrm{P}} \mathit{True}^{D} : \mathit{Bool}^{D} \hookrightarrow \mathit{True} : \mathit{Bool}$$

$$\text{(DFALSE)} \quad \Delta \mid \Gamma \vdash_{\mathrm{P}} \mathit{False}^{D} : \mathit{Bool}^{D} \hookrightarrow \mathit{False} : \mathit{Bool}$$

$$\text{(DIF)} \quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \mathit{Bool}^{D} \hookrightarrow e' : \mathit{Bool} \qquad (\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \tau \hookrightarrow e'_i : \tau')_{i=1,2}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{if}^{D}\, e \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 : \tau \hookrightarrow \mathbf{if}\; e' \;\mathbf{then}\; e'_1 \;\mathbf{else}\; e'_2 : \tau'}$$

$$\text{(DEQ)} \quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \tau \hookrightarrow e'_i : \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 ==^{D} e_2 : \mathit{Bool}^{D} \hookrightarrow e'_1 {==} e'_2 : \mathit{Bool}} \quad (\tau \text{ is a dynamic base type})$$

Figure 9.1: Syntax and specialization for dynamic booleans.

$$\begin{array}{l} \mathbf{let}^{S}\; \mathit{meval} = \mathit{preeval} \;@^{S}\; (\lambda^{S} x \rightarrow \mathit{Wrong}^{S}) \\ \;\mathbf{in}\; \ldots \end{array}$$

has to be replaced by

$$\begin{array}{l} \mathbf{let}^{S}\; \mathit{meval} = \mathit{preeval} \;@^{S}\; (\lambda^{S} x \rightarrow \mathbf{error}^{S}\; \text{“Unbound variable”}) \\ \;\mathbf{in}\; \ldots \end{array}$$

and then the expression in Example 4.10 will fail with the message "Unbound variable"). It is important to note that the failure will only be produced if the static function expressing the environment is applied to an unbound variable — see Section 9.4.

9.2 Booleans

Booleans are a particular case of datatypes, and in Hughes' work they are treated precisely like that. However, the techniques needed to work with booleans are simpler than those needed for arbitrary datatypes, and thus we have chosen to treat them separately.

A dynamic version of booleans will generate boolean constants and operators in the residual term, and does not introduce any new interesting problem. The syntax and specialization rules to deal with dynamic booleans are given in Figure 9.1. Only if-then-else and an equality operator are considered; other booleans and relational operators follow the same idea.

The static version of booleans needs a more careful treatment. We begin by extending the source language:

$$\begin{array}{lcl} e & ::= & \ldots \mid \mathit{True}^{S} \mid \mathit{False}^{S} \mid \mathbf{if}^{S}\, e \;\mathbf{then}\; e \;\mathbf{else}\; e \mid e ==^{S} e \\ \tau & ::= & \ldots \mid \mathit{Bool}^{S} \end{array}$$

$$\text{(STRUE)}\quad \Delta \mid \Gamma \vdash_{\text{P}} \mathit{True}^S : \mathit{Bool}^S \hookrightarrow \bullet : \hat{\mathit{True}}$$

$$\text{(SFALSE)}\quad \Delta \mid \Gamma \vdash_{\text{P}} \mathit{False}^S : \mathit{Bool}^S \hookrightarrow \bullet : \hat{\mathit{False}}$$

$$\text{(BLIFT)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \mathit{Bool}^S \hookrightarrow e' : \tau' \qquad \Delta \Vdash v : \text{IsBool } \tau'}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{lift } e : \mathit{Bool}^D \hookrightarrow v : \mathit{Bool}}$$

$$\text{(S==)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : \tau \hookrightarrow e'_i : \tau'_i)_{i=1,2} \qquad \Delta \Vdash v : \tau' := \tau'_1 == \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 ==^S e_2 : \mathit{Bool}^S \hookrightarrow \bullet : \tau'}\ (\tau \text{ is a static base type})$$

$$\text{(SIF)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \mathit{Bool}^S \hookrightarrow e' : \tau'_b \;\; \Delta \Vdash v : \text{IsBool } \tau'_b \qquad (\Delta_i \mid \Gamma \vdash_{\text{P}} e_i : \tau \hookrightarrow e'_i : \tau'_i)_{i=1,2} \;\; \Delta \Vdash \text{IsIf } \tau'_r\ \tau'_b\ \tau}{\Delta, \tau'_b?\Delta_1, !\tau'_b?\Delta_2 \mid \Gamma \vdash_{\text{P}} \textbf{if}^S\, e \textbf{ then } e_1 \textbf{ else } e_2 : \tau \hookrightarrow \textbf{if}_v\ v \textbf{ then } e'_1 \textbf{ else } e'_2 : \tau'_r}$$

Figure 9.2: Rules to specialize static booleans.

The problem is that in the original formulation, only one of the branches of a static conditional is specialized; but to obtain a principal specialization for it, we need to take into account *both* branches, deferring the decisions involved in their specializations until we know which branch to select. In addition to singleton types for boolean constants ($\hat{\mathit{True}}$ and $\hat{\mathit{False}}$) and predicates to express that a type may only represent a boolean constant (IsBool τ'), we will need conditional predicates ($\tau'?\delta$, $!\tau'?\delta$), predicates to express the resulting type of a static if-then-else (IsIf $\tau'_r\ \tau'_b\ \tau'_1\ \tau'_2$, establishing that the resulting type of a static if-then-else, τ'_r, is either τ'_1 or τ'_2 depending on the value of τ'_b), and a new construct in the residual term language to express the result of the static conditional ($\textbf{if}_v\ v \textbf{ then } e' \textbf{ else } e'$).

$$\begin{array}{lcl} e & ::= & \ldots \mid \textbf{if}_v\ v \textbf{ then } e' \textbf{ else } e' \\ v & ::= & \ldots \mid \mathit{True} \mid \mathit{False} \mid \bullet \\ \tau' & ::= & \ldots \mid \hat{\mathit{True}} \mid \hat{\mathit{False}} \\ \delta & ::= & \ldots \mid \text{IsBool } \tau' \mid \text{IsIf } \tau'\ \tau'\ \tau'\ \tau' \mid \tau'?\delta \mid !\tau'?\delta \end{array}$$

Conditional predicates require that $\bullet$ be also considered as evidence, to be the evidence for a predicate guarded by $\hat{\mathit{False}}$ (and some others, as well).

The rules to specialize static booleans are presented in Figure 9.2. The most interesting one is (SIF), specifying how to specialize a static if-then-else: the static boolean condition is specialized and a reference v to its value is obtained using the predicate IsBool τ'_b; specializations of both branches with their own predicate assignments are used to construct a residual *evidence* if-then-else that may be reduced as soon as the value v is known — see Figure 9.3. The resulting type is expressed using the predicate IsIf $\tau'_r\ \tau'_b\ \tau'_1\ \tau'_2$, as described above.

To complete the presentation of static booleans we have to give the rules for entailment of the new predicates, and the reduction rules for evidence if-then-else. They are presented in Figure 9.3

To observe how conditional predicates work, see the following example.

$$\Delta \Vdash b : \text{IsBool } \hat{b}$$

$$\Delta \Vdash \bullet : \text{IsIf } \tau_1' \ \hat{True} \ \tau_1' \ \tau_2'$$

$$\Delta \Vdash \bullet : \text{IsIf } \tau_2' \ \hat{False} \ \tau_1' \ \tau_2'$$

$$\frac{\Delta \Vdash v : \delta}{\Delta \Vdash v : \hat{True}?\delta}$$

$$\Delta \Vdash \bullet : \hat{False}?\delta$$

$$\frac{\Delta \Vdash v : \delta}{\Delta \Vdash v :!\hat{False}?\delta}$$

$$\Delta \Vdash \bullet :!\hat{True}?\delta$$

$$\mathbf{if}_v \ \mathit{True} \ \mathbf{then} \ e_1' \ \mathbf{else} \ e_2' \ \triangleright \ e_1'$$

$$\mathbf{if}_v \ \mathit{False} \ \mathbf{then} \ e_1' \ \mathbf{else} \ e_2' \ \triangleright \ e_2'$$

Figure 9.3: Entailment and reduction rules associated with static booleans.

EXAMPLE 9.2. Consider $f = \lambda^D b.\lambda^D x.\mathbf{if}^S\, b \ \mathbf{then} \ x +^S 1 \ \mathbf{else} \ x +^S 2$. We specialize it alone in the first case, and with a boolean argument in the second case.

1. $\vdash_{\text{P}} f : \mathit{Bool}^S \rightarrow^D \mathit{Int}^S \rightarrow^D \mathit{Int}^S$
   $\hookrightarrow \Lambda h_b, h_x, h_r, h_1, h_2.\lambda b.\lambda x.\mathbf{if}_v \ h_b \ \mathbf{then} \ \bullet \ \mathbf{else} \ \bullet$
   $: \forall t_b, t_x, t_r, t_1, t_2.\text{IsBool } t_b,$
   $\quad \text{IsInt } t_x,$
   $\quad \text{IsIf } t_r \ t_b \ t_1 \ t_2,$
   $\quad t_b?t_1 := t_x + \hat{1},$
   $\quad !t_b?t_2 := t_x + \hat{2}$
   $\quad \Rightarrow t_b \rightarrow t_x \rightarrow t_r$

2. $\vdash_{\text{P}} f \ @^D \ \mathit{True}^S \ : \mathit{Int}^S \rightarrow^D \mathit{Int}^S$
   $\hookrightarrow \Lambda h_x, h_r.(\lambda b.\lambda x.\bullet)@\bullet$
   $: \forall t_x, t_r.\text{IsInt } t_x, t_r := t_x + \hat{1} \Rightarrow t_x \rightarrow t_r$

Observe that in f, the static conditional depends on the argument, which in the first case is unknown, generating the evidence conditional $\mathbf{if}_v$; in the residual type the calculation of the resulting type is deferred using conditional predicates. In the second case, when the boolean is known, the calculation may proceed, and the evidence conditional may be reduced.

Simplification and constraint solving have to be extended for static booleans. The main issue in this respect is how to solve guarded predicates. We have discussed in

Section 8.1.5 the difference between pure simplifications and *improvements* (following the ideas of Mark Jones [1994b]). This difference is important because improving rules — those fixing the value of some variables — cannot be applied unless we are sure that the substitutions produced do not alter variables that can 'escape' a given guard — if that guard is going to take a false value, the predicate will simply disappear, and the value assigned to the variable will be unsound. Let's see this idea in an example.

EXAMPLE 9.3. Observe how sometimes deciding the value of a variable appearing under guards may produce unsound results.

$$
\begin{array}{l}
\Delta \mid \emptyset \vdash_{\text{P}} \lambda^D b.\lambda^D x.\, \mathbf{if}^S\, b \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{if}^D\, \mathit{True} \\
\qquad\qquad\qquad\qquad \mathbf{then}\ x +^S (5^S +^S 7^S) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ 13^S \\
\qquad\qquad\qquad \mathbf{else}\ x \\
\qquad\qquad\quad : \mathit{Bool}^S \to^D \mathit{Int}^S \to^D \mathit{Int}^S \\
\quad \hookrightarrow\ \lambda b.\lambda x.\, \mathbf{if}_v\ h_b \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{if}\ \mathit{True}\ \mathbf{then}\ \bullet\ \mathbf{else}\ \bullet \\
\qquad\qquad\qquad \mathbf{else}\ x \\
\qquad\qquad\quad : t_b \to t_x \to t_r
\end{array}
$$

where

$$
\begin{aligned}
\Delta = {} & h_x : \text{IsInt}\ t_x, h_b : \text{IsBool}\ t_b, h_r : \text{IsInt}\ t_r, \\
& h_1 : t_r := \text{if}\ t_b\ \text{then}\ t_{13}\ \text{else}\ t_x, \\
& h_2 : t_b?t_{12} := \hat{5} + \hat{7}, \\
& h_3 : t_b?\hat{13} := t_x + t_{12},
\end{aligned}
$$

Suppose now that we allow the extension of simplification rules to be applied under guards without restrictions. Then, rules (SimOp$_{res}$) is applied to $t_b?t_{12} := \hat{5} + \hat{7}$, deciding the value of t_{12} to be $\hat{12}$ — resulting in the simplified predicate assignment $\hat{13} := t_x + \hat{12}$.

This last predicate forces t_x to be $\hat{1}$ (using the rule (SimOP$_{inv}$) presented in Section 8.1.5), and thus the function type takes the form $t_b \to \hat{1} \to t_r$ — in particular, in the source code the function can only be applied in its second argument to an expression with value 1. But looking at the initial term, if variable b takes the value of *False*, then the function behaves as the identity function *for any number*, not only for 1. This unsoundness was produced by the (wrong) *decision* about the value of t_x. The simplifications that can produce this effect are exactly those that Jones [1994b] named *improvements*.

We can observe that any decision taken under a guard is restricted: those decisions can only be taken when the variables involved appear only under the given guard. For that reason, improvements can only be applied under guards if they do not alter the context outside the guards (incidentally, pure simplifications produce the identity substitution that never alters the context).

The best simplification that can be obtained in this example — without losing solutions, until more information for b is available — is:

$$
\begin{aligned}
\Delta' = {} & h_x : \text{IsInt}\ t_x, h_b : \text{IsBool}\ t_b, h_r : \text{IsInt}\ t_r, \\
& h_1 : t_r := \text{if}\ t_b\ \text{then}\ \hat{13}\ \text{else}\ t_x, \\
& h_3 : t_b?\hat{13} := t_x + \hat{12},
\end{aligned}
$$

We do not consider, in the rest of this work, the application of improvements under guards, keeping our formulation simple (this extension would require checking the occurrence of type variables on contexts, which may be expensive).

Another example shows that unifications occurring in a branch of a static if-then-else, produce the same kind of effect as improvements, and so they have to be deferred as well, until the particular branch is selected.

EXAMPLE 9.4. The monovariant function *id* is applied to different arguments in each of the branches of the static if-then-else.

$$\begin{array}{l} \mathbf{let}^D\ id = \lambda^D x.x \\ \ \mathbf{in}\ \lambda^D b.\mathbf{if}^S\, b \\ \qquad\quad \mathbf{then}\ id\ @^D\, 2^S \\ \qquad\quad \mathbf{else}\ id\ @^D\, 4^S \\ : Bool^S \rightarrow^D Int^S \end{array}$$

The type variable expressing the type of the argument for *id* has to be unified with either $\hat{2}$ or $\hat{4}$, depending on the value of b.

This deferring is easy to express using predicates: we have to introduce a new predicate reifying the unification of two types, with the entailment rules for it capturing the intended internalization.

$$\delta ::= \ \ldots\ \mid \tau' \sim \tau'$$

$$\frac{\tau'_1 \sim^S \tau'_2}{\Delta \Vdash \bullet : \tau'_1 \sim \tau'_2}$$

Additionally, all the rules in the algorithm that use unification have to replace that use by an entailment of the new predicate. Simplification and improvement have to be extended to resolve the unguarded unification predicates.

Example 9.4 will then specialize to

$$\begin{array}{l} \Lambda h_x, h_b, h_r, h'_r.\mathbf{let}\ id = \lambda x.x \\ \qquad\qquad\quad\ \ \mathbf{in}\ \lambda b.\mathbf{if}_v\ h_b \\ \qquad\qquad\qquad\qquad \mathbf{then}\ id@\bullet \\ \qquad\qquad\qquad\qquad \mathbf{else}\ id@\bullet \\ : \forall t_x, t_b, t_r, t_2, t_4. \\ \quad \text{IsInt}\ t_x, \\ \quad \text{IsBool}\ t_b, \\ \quad \text{IsIf}\ t_r\ t_b\ t_2\ t_4, \\ \quad \text{IsInt}\ t_r, \\ \quad t_b?((t_x \rightarrow t_x) \sim (\hat{2} \rightarrow t_2)), \\ \quad !t_b?((t_x \rightarrow t_x) \sim (\hat{4} \rightarrow t_4)) \\ \quad \Rightarrow t_b \rightarrow t_r \end{array}$$

Observe that the residual type of x, t_x, may be unified with $\hat{2}$ or $\hat{4}$, depending on the residual type of b, t_b. Another interesting point that this example shows is that some simplification can be performed in the body of a guarded predicate, with the restriction that no variable appearing free in some other place is unified (until the guard can be removed); in this case, the predicate $t_b?((t_x \rightarrow t_x) \sim (\hat{2} \rightarrow t_2))$ for example, can be simplified to the set $\{t_b?(t_x \sim \hat{2}), t_b?(t_x \sim t_2)\}$.

9.3 Static Let

In the original formulation of type specialization, the addition of static let was easy: the only difference between the new rule and the one for dynamic let was the degree of unfolding allowed in the context Γ. However, in the principal specialization setting the use of the context to express unfolding is not possible, because of evidence variables that may escape their scopes, as the following example shows.

EXAMPLE 9.5. During this example we assume that we have introduced a rule for static let in the same way as in the original formulation.

$$\begin{array}{l} x : Int^D \hookrightarrow h_z : Int \vdash_{\text{P}} \textbf{poly}\ (\lambda^D y.x +^D \textbf{lift}\ y) : \textbf{poly}\ (Int^S \to^D Int^D) \\ \qquad\qquad \hookrightarrow\ \Lambda h_z, h_y.\lambda y.h_z + h_y : \textbf{poly}\ (\forall t_y.\text{IsInt}\ t_z, \text{IsInt}\ t_y \Rightarrow t_y \to Int) \end{array}$$

Observe that h_z appears in the context, but there is no predicate in the predicate context to bind it — instead, it is bound in the term! The problem is the use of the residual of x in the context.

While this problem is not really manifested with the present formulation for complete programs, because of the way environments are generated, we think that it is better to have a system where all the specializations can be considered correct.

To solve this, we can use the reduction of evidence in the residual language to express static beta reduction, simplifying the treatment of free variables — this is achieved by the new construct $e'@_v e'$ and a corresponding reduction rule.

$$\begin{array}{rcl} e' & ::= & \ldots \mid v@_v e' \\ v & ::= & \ldots \mid e' \end{array}$$

The reduction rule is standard beta reduction.

$$(@_v)\quad (\lambda x'.e'_1)@_v e'_2 \ \triangleright\ e'_1[x'/e'_2]$$

With these elements, the new rule for static let can be defined now as follows.

$$(\text{SLET})\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e_2 : \tau_2 \ \hookrightarrow\ e'_2 : \tau'_2 \quad \Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{P}} e_1 : \tau_1 \ \hookrightarrow\ e'_1 : \tau'_1}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{let}^S\ x = e_2\ \textbf{in}\ e_1 : \tau_1 \ \hookrightarrow\ (\lambda x'.e'_1)@_v e'_2 : \tau'_1}\ \ (x'\ \textit{fresh})$$

Observe how in the result, the fresh variable x' is abstracted from e'_1, and further applied to e'_2 — the construction is thus equivalent to the reduction of this application, by virtue of reduction of evidence.

This new construct is also useful in the next two sections, when defining static functions and static recursion.

9.4 Static Functions

The addition of static functions in the source language is really easy: one simply adds static abstraction and application in the following way.

$$\begin{array}{rcl} e & ::= & \ldots \mid \lambda^S x.e \quad \mid e\ @^S e \\ \tau & ::= & \ldots \mid \tau \to^S \tau \end{array}$$

The typing rules are identical to (ST-DLAM) and (ST-DAPP) from Figure 3.1, the only change being that all annotations are $_^S$ instead of $_^D$. This is possible because, as we have discussed in Chapter 3, type specialization imposes no restrictions on annotations.

In the residual language things are not so easy. We have seen, in the original formulation, that static abstractions are represented as closures containing static source code, because the body of the static function may depend on values given by the context of use. But having principal specializations, we can choose a different representation: the closure will contain the principal specialization of the body, which can be instantiated on each application. Additionally, for each static application it is possible to use the $@_v$-rule of the reduction of evidence to express the static beta reduction, simplifying the treatment of free variables. Finally, a new predicate IsFunS is used to defer the decision of the actual type of a closure until all the contextual information is present — this is almost the same mechanism used with IsMG for polyvariance: upper bounds to the type will be produced from function definitions, and lower bound from function applications — the main difference is that in the case of functions some upper bounds will also contain residual code.

Static closures are added to the residual type language, and also predicates expressing constraints on them. To represent residual code in the static closures, a new syntactic category is added: $\tau'_{e'}$ — it represents a new type composed by (residual) code. The evidence corresponding to predicates involving static closures is the function stored in the closure, which can be reduced when it appears as the body of the application $@_v$.

$$\begin{array}{lcl} \tau'_{e'} & ::= & t \quad \mid e' \\ \tau' & ::= & \ldots \mid \mathbf{clos}(\tau'_{e'} : \sigma') \\ \delta & ::= & \ldots \mid \text{IsFunS } \tau' \, \tau' \end{array}$$

The rules for specialization are extended with (SLAM) and (SAPP). Free variables are treated in a similar way to the original formulation of type specialization (see Chapter 3), but we use the reduction of the special form of application to pass the variables (remember that we cannot allow expressions to appear in contexts). In the rules, that is expressed by using Γ' instead of Γ in the premise of the rule, which replaces every free variable in the static function by a projection on a new variable f' which is further abstracted:

$$\begin{array}{l} \Gamma = x_1 : \tau_1 \hookrightarrow e'_1 : \tau'_1, \ldots, x_n : \tau_n \hookrightarrow e'_n : \tau'_n \\ \Gamma' = x_1 : \tau_1 \hookrightarrow \pi^S_{1,n}\, f' : \tau'_1, \ldots, x_n : \tau_n \hookrightarrow \pi^S_{n,n}\, f' : \tau'_n \\ \tau'_f = (\tau'_1, \ldots, \tau'_n) \end{array}$$

Instead of assigning a closure type directly, we use the predicate IsFunS to defer the decision, for the same reason as in the rule (POLY)— that is, the context may impose further restrictions on the actual type, and so the decision is taken by the constraint solver.

$$\text{(SLAM)} \quad \frac{h' : \Delta' \mid \Gamma' \vdash_{\text{P}} \lambda^D x.e : \tau_x \to^D \tau_e \;\hookrightarrow\; e' : \tau'' \quad \Delta \Vdash \text{IsFunS } \mathbf{clos}(\Lambda h'.\lambda f'.e' : \sigma)\ \tau'}{\Delta \mid \Gamma \vdash_{\text{P}} \lambda^S x.e : \tau_x \to^S \tau_e \;\hookrightarrow\; (e'_1, \ldots, e'_n)^S : \tau'} \;\; \left(f' \textit{ fresh and } \sigma{=}\text{Gen}_{\Gamma',\emptyset}(\Delta' {\Rightarrow} \tau'_f {\to} \tau'')\right)$$

(SAPP)

$$\frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 : \tau_2 \rightarrow^{S} \tau_1 \;\hookrightarrow\; e_1' : \tau' \qquad \Delta \mid \Gamma \vdash_{\mathrm{P}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2' \quad \Delta \Vdash v : \text{IsFunS}\ \tau'\ \mathbf{clos}(\tau_{e'}' : \tau_2' \rightarrow \tau}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 \mathbin{@^{S}} e_2 : \tau_1 \;\hookrightarrow\; v@_v e_1' @_v e_2' : \tau_1'}$$

Observe in rule (SAPP) that $@_v$ is used twice, once to pass the function the residual of its free variables, and once to pass its actual argument.

The specification of specialization for static functions is completed with the rule for entailment of IsFunS and the rule for source-residual relationship.

$$\text{(IsFunS)}\quad \frac{C : (\Delta \mid \sigma') \geq (\Delta \mid \sigma'') \qquad e'' = C[e']}{\Delta \Vdash e'' : \text{IsFunS}\ \mathbf{clos}(e' : \sigma')\ \mathbf{clos}(e'' : \sigma'')}$$

$$\text{(SR-SFUN)}\quad \frac{\Delta' \vdash_{\mathrm{SR}} \tau_x \rightarrow^{D} \tau_e \;\hookrightarrow\; \tau'' \quad \Delta \Vdash \text{IsFunS}\ \mathbf{clos}(\tau_{e'}' : \sigma)\ \tau'}{\Delta \vdash_{\mathrm{SR}} \tau_x \rightarrow^{S} \tau_e \;\hookrightarrow\; \tau'}\quad \left(\sigma = \text{Gen}_{\emptyset,\emptyset}(\Delta' \Rightarrow \tau_f' \rightarrow \tau'')\right)$$

We have said that the predicate IsFunS provides a similar mechanism for static functions as the predicate IsMG does for polyvariance: each predicate provides either an upper bound or a lower bound for a given closure, represented as a type variable t (which the algorithm places instead of τ' in the rules (SLAM), (SAPP), and (SR-SFUN)); additionally, as a function ever applied must have been defined somewhere, we know that one of the upper bounds will provide the function body (in the form of a principal term). These facts are used to define the constraint solving algorithm for it: upper bounds are collected, the greatest lower bound of all of them is calculated (and the function body as well), and the variable t is replaced by it. The evidence for each IsFunS provides the corresponding code.

To compare our approach for static functions with that in the original formulation, we consider again the source term of Example 3.18. Observe that the expression inside the closure corresponding to the function has been processed — in the original formulation, the closure keeps the source code of the function body.

Example 9.6. We consider first the function in isolation, and then the complete term; but instead of offering the final result — which is the same as in the original formulation — we give the principal specialization of it.

1. $\vdash_{\mathrm{P}} \mathbf{let}^{D}\ x = (5^{S}, 6^{D})^{D}$
   $\mathbf{in}\ \lambda^{S}y.\mathbf{lift}\ (\mathbf{fst}^{D}\ x +^{S} y) +^{D} \mathbf{snd}^{D}\ x$
   $: \mathit{Int}^{S} \rightarrow^{S} \mathit{Int}^{D}$
   $\hookrightarrow \Lambda h_f.\mathbf{let}\ x = (\bullet, 6)\ \mathbf{in}\ x$
   $: \forall t.\text{IsFunS}\ (\mathbf{clos}(\Lambda h_y, h_r.\lambda f'.\lambda y.h_r + \mathbf{snd}\ \pi_{1,1}^{S}\ f'$
   $: \forall t_y, t_r.\text{IsInt}\ t_y, t_r := \hat{5} + t_y \Rightarrow (\hat{5}, \mathit{Int})^{S} \rightarrow t_y \rightarrow \mathit{Int}))\ t \Rightarrow t$

2. $\vdash_{\text{P}} \textbf{let}^{D}\ f = \textbf{let}^{D}\ x = (5^{S}, 6^{D})^{D}$
$\qquad\qquad \textbf{in}\ \lambda^{S} y.\textbf{lift}\ (\textbf{fst}^{D}\ x +^{S} y) +^{D} \textbf{snd}^{D}\ x$
$\quad \textbf{in}\ f\ @^{S}\ 2^{S}$
$\quad : \mathit{Int}^{D}$
$\hookrightarrow \Lambda h_f, h_a.\textbf{let}\ f = \textbf{let}\ x = (\bullet, 6)\ \textbf{in}\ x$
$\qquad\qquad \textbf{in}\ h_a @_v f @_v \bullet$
$\quad : \forall t, t'.\text{IsFunS}\ (\textbf{clos}(\Lambda h_y, h_r.\lambda f'.\lambda y.h_r + \textbf{snd}\ f'$
$\qquad\qquad : \forall t_y, t_r.\text{IsInt}\ t_y, t_r := \hat{5} + t_y \Rightarrow t_y \rightarrow \mathit{Int}))\ t,$
$\qquad \text{IsFunS}\ t\ \textbf{clos}(t' : \hat{2} \rightarrow \mathit{Int}) \Rightarrow \mathit{Int}$

When performing constraint solving in the second case, the evidence constructed for the predicate IsFunS t $\textbf{clos}(t' : \hat{2} \rightarrow \mathit{Int})$ will be the term $\lambda f'.\lambda y.7 + \textbf{snd}\ f'$, and when this is assigned to h_a, evidence reduction (in particular, rule $(@_v)$) will produce the right answer.

9.5 Static Recursion

As we have said in Section 3.4.4, static recursion is added to the source language by a construct $\textbf{fix}^{S}$, with its use restricted to functions producing static functions to ensure termination of the specialization process. In the original formulation, the specialization of this new construct was expressed by using a residual type **rec**, and the unfolding was performed at application. In the principal specialization setting, we use a similar approach, but we have to introduce also a new form of predicate, IsFixS, which expresses the relation between the residual type of the argument to $\textbf{fix}^{S}$ and the residual type of the result, bounding a type by a static function; the solution for this predicate is expressed by the residual type $\textbf{rec}(\tau_{e'} : \sigma')$, corresponding to the principal version of **rec**.

The rule to specialize a static **fix** is the following.

$$\text{(SFIX)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \rightarrow^{S} \tau \ \hookrightarrow\ e' : \tau' \qquad \Delta \Vdash \text{IsFixS}\ \tau'\ \tau''}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{fix}^{S}\ e : \tau \ \hookrightarrow\ e' : \tau''}$$

Following the same principles as with polyvariance and static functions, the rule defers the decision of the actual type to use as τ'' until all the information is present. The rules for entailment for the new predicate and type provide the unfolding producing the final residual code.

$$\text{(IsFixS)}\quad \Delta \Vdash \text{IsFixS}\ \textbf{clos}(\tau'_{e'} : \sigma')\ \textbf{rec}(\tau'_{e'} : \sigma')$$

$$\text{(REC-CLOS)}\quad \frac{\Delta \Vdash v : \text{IsFunS}\ \textbf{clos}(\tau'_{e'} : \sigma')\ \textbf{clos}(\tau'''_{e'} : \textbf{rec}(\tau'_{e'} : \sigma') \rightarrow \tau') \quad \Delta \Vdash v' : \text{IsFunS}\ \tau'\ \textbf{clos}(\tau''_{e'} : \sigma'')}{\Delta \Vdash \lambda f.v' @_v (v @_v f @_v f) : \text{IsFunS}\ \textbf{rec}(\tau'_{e'} : \sigma')\ \textbf{clos}(\tau''_{e'} : \sigma'')}\ (f \text{ fresh})$$

$$\frac{\begin{array}{c}\Delta, \text{IsFunS } \mathbf{clos}(\tau'_{e'} : \sigma')\ \mathbf{rec}(\tau''_{e'} : \sigma'') \\ \Vdash \text{IsFunS } \mathbf{clos}(\tau''_{e'} : \sigma'')\ \mathbf{clos}(\tau'''_{e'} : \mathbf{rec}(\tau''_{e'} : \sigma'') \to \tau')\end{array} \qquad \Delta \Vdash \text{IsFunS } \mathbf{clos}(\tau'_{e'} : \sigma')\ \tau'}{\Delta \Vdash \text{IsFunS } \mathbf{clos}(\tau'_{e'} : \sigma')\ \mathbf{rec}(\tau''_{e'} : \sigma'')}\ \text{(CLOS-REC)}$$

When the new type appears as the upper bound in the predicate for a static function, a new unfolding is produced — observe the second argument of predicate IsFunS in the first premise of rule (REC-CLOS). When the new type appears as the lower bound, the predicate that we are trying to entail is used as (inductive) hypothesis — appearing in the predicate assignment of the first premise in rule (CLOS-REC). The case when the argument of a $\mathbf{fix}^S$ is itself a $\mathbf{fix}^S$ can be handled here by adding the following rule.

$$\frac{\Delta \Vdash \text{IsFunS } \mathbf{rec}(\tau'_{e'} : \sigma')\ \tau'' \qquad \Delta \Vdash \text{IsFixS } \tau''\ \tau'}{\Delta \Vdash \text{IsFixS } \mathbf{rec}(\tau'_{e'} : \sigma')\ \tau'}$$

It first calculates a residual type τ'' as the solution for the first **rec**, and then bounds τ' by it.

Constraint solving is performed in a very similar way to that of static functions, relying on simplification to do the actual work.

Let's revisit the example of a recursive static function considered in Section 3.4.4. The source code was

$$\begin{array}{l}\mathbf{let}^D\ n = 35^D \\ \ \mathbf{in\ let}^D\ f = \mathbf{fix}^S\ (\lambda^S g.\lambda^S x.1^D +^D \mathbf{if}^S\ x ==^S 0^S\ \mathbf{then}\ n\ \mathbf{else}\ g\ @^S (x -^S 1^S)) \\ \qquad \mathbf{in}\ f\ @^S 2^S\end{array}$$

Its principal specialization is

$$\begin{array}{l}\Lambda h_1, h_2, h_3.\mathbf{let}\ n = 35\ \mathbf{in\ let}\ f = (n)^S\ \mathbf{in}\ h_3 @_v f @_v \bullet \\ : \forall t, t', t_2.\text{IsFunS closure}_f\ t, \\ \qquad \text{IsFixS } t\ t', \\ \qquad \text{IsFunS } t'\ \mathbf{clos}(t_2 : \hat{2} \to \mathit{Int}) \\ \qquad \Rightarrow \mathit{Int}\end{array}$$

where

$$\begin{array}{l}\text{closure}_f = \mathbf{clos}(\Lambda h_4, h_5.\lambda f_s, g.(g, \pi^S_{1,1}\ f_s)^S \\ \qquad : \forall t_3, t_4, t_5.\text{IsFunS } \mathbf{clos}(t_5 : \forall t_6.\text{IsInt } t_6 \Rightarrow t_6 \to \mathit{Int})\ t_3, \\ \qquad\qquad \text{IsFunS closure}_g(t_3)\ t_4 \\ \qquad\qquad \Rightarrow t_3 \to t_4) \\ \text{closure}_g(t_3) = \mathbf{clos}(\Lambda h_3, h_4, h_5, h_6.\lambda f'_s, x.1 + (\mathbf{if}_v\ h_4 \\ \qquad\qquad \mathbf{then\ snd}^S\ f'_s \\ \qquad\qquad \mathbf{else}\ h_6 @_v \mathbf{fst}^S\ f'_s @_v \bullet) \\ \qquad : \forall t_7, t_8, t_9, t_{10}.\text{IsInt } t_7, \\ \qquad\qquad t_8 := t_7 == \hat{0}, \\ \qquad\qquad !t_8?t_9 := t_7 - \hat{1}, \\ \qquad\qquad !t_8?\text{IsFunS } t_3\ \mathbf{clos}(t_{10} : t_9 \to \mathit{Int}) \\ \qquad\qquad \Rightarrow t_7 \to \mathit{Int})\end{array}$$

The solution uses three predicates expressing the definition and use of the recursive function. The first one is a IsFunS bounding the residual type of the function argument

to $\mathbf{fix}^S$, the second one is a IsFixS expressing the application of $\mathbf{fix}^S$, and the third one, a IsFunS expressing the application of the recursive function f to 2^S. It is the first predicate, the upper bound, which has all the information coming from the body of the recursive function, while the last one, the lower bound, provides the value of the parameter. The unfolding of the static recursive function is performed by the constraint solver, resulting in

$$\begin{array}{l}\mathbf{let}\ n = 35 \\ \ \mathbf{in\ let}\ f = (n)^S \\ \quad\quad \mathbf{in}\ 1 + (1 + (1 + \pi^S_{1,1}\ f)) \\ :\ Int\end{array}$$

Observe that all the subexpressions of the form $(1 + _)$ come from the evidence proving the predicate IsFunS closure$_g(t_3)$ t_4, which is a function adding one to something that depends on the value of the original x.

9.6 Datatypes

As we have said in Section 3.4.5, sum types are a very important addition to the language. Instead of following the original approach of combining anonymous sums and recursive types, we have chosen to use an approach closer to the Haskell language: we extend our language with named sum types, allowing recursion only by using the name of the type as the type of one of the arguments. We allow the use of type arguments in datatype names, but, as the language is still monomorphic, this is only syntactic sugar for declaring several types. We also allow constructors to have multiple arguments, although this can also be obtained as syntactic sugar.

Here we only consider static datatypes. Dynamic datatypes can also be treated, although there require a bit more work to generate the right residual datatype declarations; for that reason we have left them for future work.

We extend the source language with declarations for static datatypes, adding them in front of the expression. The form of datatype declarations resembles Haskell:

$$\mathbf{data}\ D^S\,\alpha_i\ =\ C_1\ \tau_{1,1}\ldots\tau_{1,a_1}\ |\ \ldots\ |\ C_p\ \tau_{1,1}\ldots\tau_{p,a_p}$$

where $0 \leq i \leq arity(D^S)$ (so, α_i represents a vector of $arity(D^S)$ variables), $p \geq 1$, and $a_j \geq 0$, $0 \leq l_j \leq a_j$ for every $j = 1,\ldots,p$. The names D and C_j are distinct identifiers beginning with a capital letter, and the type variables α_i are the only variables that can appear in the τ_{j,l_j} types. Every datatype name D^S followed by $arity(D^S)$ types forms a new type, but every use of a datatype in the program has to be monomorphic (that is, without type variables). Observe that any type can be the argument of a constructor, including datatypes, and that all of them have a unique annotation. In particular, we do not consider *polychronic* datatypes (datatypes with annotation arguments), as it was done by Heldal [2001], although they can be used as syntactic sugar for several declarations. Finally, constructors may be partially applied, because they are η-expanded during source type checking to meet their types — for example, $Two^S\ 1^D$ is expanded to $(\lambda^S x.\lambda^S y.Two^S\ x\ y)\ @^S\,1^D$, because its type is $Two^S\ :\ Int^D \to^S \tau \to^S ZOT^S\ Int^D\ \tau$

for any given ground type τ (see upcoming Example 9.7); in this way, constructor's arguments are always variables, a fact that is used in rule (SCONSTR), below.

The specialization of static datatypes is very similar to the one for booleans. We introduce the new residual construct $\mathbf{case}_v$, corresponding to $\mathbf{if}_v$, and two new predicates, IsConstrOf, that corresponds to IsBool, and IsCase, that corresponds to IsIf; the evidence for these new predicates is a constructor's residual name and void, respectively. The rules to specialize constructors use the fact that every constructor is applied to the right number of variables, because of the η-expansion performed; the specialization of variables is captured by the rule (SLAM). We have left implicit the conditions stating that every residual type introduced is in the source-residual relationship with its corresponding source type — that is, for every τ' used, $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$ — and those stating the type of each C_j — that is $C_j : \tau_{j,1} \rightarrow^S \ldots \rightarrow^S \tau_{j,a_j} \rightarrow^S D^S \tau_i$ — for $j = 1, \ldots, p$.

$$\text{(SCONSTR)} \quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} x_{k,l_k} : \tau_{k,l_k} \hookrightarrow e'_{k,l_k} : \tau'_{k,l_k}}{\Delta \mid \Gamma \vdash_{\text{P}} C_k^S \; x_{k,l_k} : D^S \tau_i \hookrightarrow (e'_{k,l_k})^S : C_k \; \tau'_{k,l_k}}$$

$$\text{(SCASE)} \quad \frac{\begin{array}{c}\Delta \mid \Gamma \vdash_{\text{P}} e : D^S \tau_i \hookrightarrow e' : \tau'_e \\ (\Delta_j \mid \Gamma \vdash_{\text{P}} \lambda^S x_{j,l_j}.e_j : \tau_{j,l_j} \rightarrow^S \tau \hookrightarrow e'_j : \tau'_j)_{j=1,\ldots,p} \\ \Delta \Vdash v : \text{IsConstrOf } (D^S \tau_i) \; \tau'_e \qquad \Delta \Vdash \tau'_r := \mathbf{case} \; \tau'_e \; \mathbf{of} \; \{C_j \rightarrow \tau'_j\}\end{array}}{\begin{array}{r}\Delta, ((v \text{ is a } C_j)?\Delta_j)_{j=1,\ldots,p} \mid \Gamma \vdash_{\text{P}} \mathbf{case}^S e \; \mathbf{of} \; \{C_j \; x_{j,l_j} \rightarrow e_j\} : \tau \\ \hookrightarrow \mathbf{case}_v \; v \; \mathbf{of} \; \{C_j \rightarrow e'_j \pi_{l_j,a_j} e'\} : \tau'_r\end{array}} \; ()$$

Observe how each branch is specialized to a static function that is further applied using the application , resulting in the expansion of variables matched in the case construct with the right projection from the residual of the scrutinized expression.

The entailment rules for the new predicates are similar to their boolean counterparts, except for the argument types. There are also rules for guarded predicates when the guard is a tag-check.

$$\frac{\Delta \vdash_{\text{SR}} \tau_{k,l_k} \hookrightarrow \tau'_{k,l_k}}{\Delta \Vdash C_k : \text{IsConstrOf } (D^S \tau_i) \; (C_k \; \tau'_{k,l_k})}$$

$$\Delta \Vdash \bullet : \tau'_k := \mathbf{case} \; (C_k \; \tau'_{k,l_k}) \; \mathbf{of} \; \{C_j \; \tau'_{j,l_j} \rightarrow \tau'_j\}$$

$$\frac{\Delta \Vdash v : \delta}{\Delta \Vdash v : (C_k \text{ is a } C_k)?\delta}$$

$$\Delta \Vdash \bullet : (C_k \text{ is a } C_j)?\delta \quad (k \neq j)$$

Observe that the argument types have to match in order to select the appropriate branch when entailing an IsCase predicate. Also observe that the evidence proving an IsConstrOf predicate is the name of the tag, a fact used in the construct $\mathbf{case}_v$ in rule (SCASE).

Finally, we also need a reduction rule for $\mathbf{case}_v$, which simply selects the residual expression corresponding to the tag given as evidence.

$$\mathbf{case}_v \; C_k \; \mathbf{of} \; \{C_j \rightarrow e'_j\} \triangleright e'_k$$

The rules for simplification and improvement are essentially the same as for booleans, but also considering the arguments of constructors. Constraint solving uses the rules for static functions to reduce the applications of constructors to their arguments.

An interesting point that arises here is the need for rules with inverse flow of information — we have discussed this idea in Section 8.1.5. For datatypes, the rule involved is

$$\text{(SimCASE}_1\text{)} \quad \text{Id}; h \leftarrow \bullet \;\mid h : t := \textbf{case}\ \tau_v\ \textbf{of}\ \tau_p \ \rightarrow\ \tau_b \trianglerighteq h'' : \tau_v \sim \tau_p, h' : t \sim \tau_b \quad (h',h''\ \textit{fresh})$$

where $\tau_p \ \rightarrow\ \tau_b$ is the only branch in the IsCase predicate. It is less evident why the flow of information works backwards, and a very good example of the power of type specialization. When a static **case** with only one pattern is used, such expression can only be specialized when used with an expression matching the pattern. So, the predicate is simplified into two new ones: one forcing the residual type of the expression to unify to that of the pattern, and another one forcing the residual type of the expression to unify to the body of the **case**-statement.

While this optimization can be regarded as too particular, the examples in Chapter 4 (specially Example 4.4) produce several predicates of this kind — e.g. in every application of the object program, the residual type corresponding to the function expression is matched against the pattern *Fun*. Without this simplification rule those predicates would have remained, and the final result would not be as desired.

It is worth considering some examples.

EXAMPLE 9.7. These are simple examples, showing the specialization of constructors with different number of arguments, and also partially applied. All items assume the following definition:

$$\textbf{data}\ \mathit{ZOT}^S\ t_1\ t_2 \ = \ \mathit{Zero} \mid \mathit{One}\, t_1 \mid \mathit{Two}\, t_1\ t_2.$$

1. $\vdash_P \mathit{Zero}^S : \mathit{ZOT}^S\ \tau_1\ \tau_2 \hookrightarrow ()^S : \mathit{Zero}$

2. $\vdash_P \mathit{One}^S\ 1^D : \mathit{ZOT}^S\ \mathit{Int}^D\ \tau \hookrightarrow (1)^S : \mathit{One\ Int}$

3. $\vdash_P \mathit{Two}^S\ 17^D\ 42^S : \mathit{ZOT}^S\ \mathit{Int}^D\ \mathit{Int}^S \hookrightarrow (17, \bullet)^S : \mathit{Two\ Int}\ \hat{42}$

4. $\vdash_P \mathit{Two}^S\ 17^D : \mathit{Int}^S \rightarrow^S \mathit{ZOT}^S\ \mathit{Int}^D\ \mathit{Int}^S$
   $\hookrightarrow \Lambda h.(17)^S : \forall t.\text{IsFunS closure}_t\ t \Rightarrow t$

   where $\text{closure}_t = \textbf{clos}(\Lambda h'.\lambda f_s.\lambda x_1.(\pi^S_{1,1}\ f_s, x_1)^S : \forall t'.\text{IsInt}\ t' \Rightarrow t' \rightarrow \mathit{Two\ Int}\ t')$

Observe in the last item how static functions are used to represent partially applied constructors; when the constructor is fully applied, constraint solving may eliminate all the predicates. For example, applying the last item to 42^S gives the previous one.

EXAMPLE 9.8. These examples show the use of the case construct, and how the decision of the branch to specialize is deferred using $\textbf{case}_v$.

1. $$\begin{array}{l} \vdash_{\text{P}} (\lambda^{D} d.\mathbf{case}^{S}\, d\ \mathbf{of} \\ \qquad\qquad \mathit{Zero} \ \rightarrow\ 0^{D} \\ \qquad\qquad \mathit{One}\, x \ \rightarrow\ x \\ \qquad\qquad \mathit{Two}\, x\ y \ \rightarrow\ x +^{D} \mathbf{lift}\ y) \\ \quad @^{D} (\mathit{Two}^{S}\ 17^{D}\ 42^{S}) \\ \quad : \mathit{Int}^{D} \\ \quad \hookrightarrow (\lambda d.\mathbf{fst}^{S}\ d + 42)@(17, \bullet)^{S}\ : \mathit{Int} \end{array}$$

2. $$\begin{array}{l} \vdash_{\text{P}} \lambda^{D} d.\mathbf{case}^{S}\, d\ \mathbf{of} \\ \qquad\qquad \mathit{Zero} \ \rightarrow\ 0^{D} \\ \qquad\qquad \mathit{One}\, x \ \rightarrow\ x \\ \qquad\qquad \mathit{Two}\, x\ y \ \rightarrow\ x +^{D} \mathbf{lift}\ y \\ \quad : \mathit{ZOT}^{S}\ \mathit{Int}^{D}\ \mathit{Int}^{S} \rightarrow^{D} \mathit{Int}^{D} \\ \quad \hookrightarrow \Lambda h_d, h_r, h_y.\lambda d.\mathbf{case}_v\ h_d\ \mathbf{of} \\ \qquad\qquad\qquad\qquad \mathit{Zero} \ \rightarrow\ 0 \\ \qquad\qquad\qquad\qquad \mathit{One} \ \rightarrow\ \mathbf{fst}^{S}\ d \\ \qquad\qquad\qquad\qquad \mathit{Two} \ \rightarrow\ \mathbf{fst}^{S}\ d + h_y \\ \quad : \forall t_d, t_y.\text{IsConstrOf}\ (\mathit{ZOT}^{S}\ \mathit{Int}^{D}\ \mathit{Int}^{S})\ t_d, \\ \qquad\qquad \mathit{Int} := \mathbf{case}\ t_d\ \mathbf{of} \\ \qquad\qquad\qquad\qquad \mathit{Zero} \ \rightarrow\ \mathit{Int} \\ \qquad\qquad\qquad\qquad \mathit{One}\ \mathit{Int} \ \rightarrow\ \mathit{Int} \\ \qquad\qquad\qquad\qquad \mathit{Two}\ \mathit{Int}\ t_y \ \rightarrow\ \mathit{Int}, \\ \qquad\qquad (t_d \text{ is a } \mathit{Two})?\text{IsInt}\ t_y \\ \qquad\qquad \Rightarrow t_d \rightarrow \mathit{Int} \end{array}$$

The second item shows how the value of the second argument of constructor *Two* is moved inside the body of the function; the predicate _ := **case** _ **of** _ expresses that information flow, and for this reason it cannot be simplified.

EXAMPLE 9.9. These examples show the specialization of values of a recursive datatype: lists with static spine, defined as $\mathbf{data}\ \mathit{List}^{S}\ t\ =\ \mathit{Nil} \mid \mathit{Cons}\ t\ (\mathit{List}^{S}\ t)$

1. $\vdash_{\text{P}} \mathit{Nil}^{S}\ : \mathit{List}^{S}\ \tau \hookrightarrow ()^{S}\ : \mathit{Nil}$

2. $\vdash_{\text{P}} \mathit{Cons}^{S}\ 17^{D}\ \mathit{Nil}^{S}\ : \mathit{List}^{S}\ \mathit{Int}^{D} \hookrightarrow (17, ()^{S})^{S}\ : \mathit{Cons}\ \mathit{Int}\ \mathit{Nil}$

3. $$\begin{array}{l} \vdash_{\text{P}} \mathbf{fix}^{S}\ (\lambda^{S} f.\lambda^{S} x.\mathit{Cons}^{S}\ 1^{D}\ (f\ @^{S} x)) : \mathit{Int}^{D} \rightarrow^{S} \mathit{List}^{S}\ \mathit{Int}^{D} \\ \quad \hookrightarrow \Lambda h.()^{S}\ : \forall t.\text{IsFixS closure}_{xs}\ t \Rightarrow t \\ \text{where closure}_{xs} = \mathbf{clos}(\Lambda h_f, h_r.\lambda f_s.\lambda f.(f)^{S} \\ \qquad\qquad\qquad\qquad : \forall t_f, t_r, t_e.\text{IsFunS closure}_f(t_e)\ t_f, \\ \qquad\qquad\qquad\qquad\qquad \text{IsFunS closure}_r(t_f)\ t_r \\ \qquad\qquad\qquad\qquad\qquad \Rightarrow t_f \rightarrow t_r) \\ \qquad \text{closure}_f(t_e) = \mathbf{clos}(t_e : \forall t_{xs}.\text{IsConstrOf}\ (\mathit{List}^{S}\ \mathit{Int}^{D})\ t_{xs} \Rightarrow \mathit{Int} \rightarrow t_{xs}) \\ \qquad \text{closure}_r(t_f) = \mathbf{clos}(\Lambda h_{ys}, h_L.\lambda^{S} f'_s.\lambda^{S} x.(1, h_L @_v \pi_{1,1}\ f'_s @_v x)^{S} \\ \qquad\qquad\qquad\qquad : \forall t_{ys}, t'_e.\text{IsConstrOf}\ (\mathit{List}^{S}\ \mathit{Int}^{D})\ t_{ys} \\ \qquad\qquad\qquad\qquad\qquad \text{IsFunS}\ t_f\ (\mathbf{clos}(t'_e : \mathit{Int} \rightarrow t_{ys})) \\ \qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{Int} \rightarrow \mathit{Cons}\ \mathit{Int}\ t_{ys}) \end{array}$$

The type Int^D is an arbitrary choice; any ground type can be used instead. The first two items are similar to previous examples. It is the third example that is interesting; despite its complexity, it shows that we can represent 'infinite' static structures using residual types with predicates. In the original framework, this list can also be represented, but with a residual type containing the source code; the real difference can be appreciated in Example 9.11.

EXAMPLE 9.10. This example shows a (non-recursive) function over the recursive datatype of lists defined in the previous example.

$$
\begin{array}{l}
\vdash_P \mathbf{let}^D\ head = \lambda^D xs.\mathbf{case}^S\, xs\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad Nil \ \to\ \mathbf{error}^S\ \text{“Empty list!”} \\
\qquad\qquad\qquad\qquad\quad Cons\, y\ ys \ \to\ y \\
\quad \mathbf{in}\ head \\
\quad : List^S\ Int^D \to^S Int^D \\
\quad \hookrightarrow \Lambda h_{xs}, h_r, h_f, h_{ys}.\mathbf{let}\ head = \lambda xs.\mathbf{case}_v\ h_{xs}\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil \ \to\ h_f \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Cons \ \to\ \mathbf{fst}^S\ xs \\
\qquad\qquad\qquad\quad \mathbf{in}\ head \\
\quad : \forall t_{xs}, t_{ys}.\text{IsConstrOf}\ (List^S\ Int^D)\ t_{xs}, \\
\qquad\qquad Int := \mathbf{case}\ t_{xs}\ \mathbf{of} \\
\qquad\qquad\qquad\quad Nil \ \to\ Int \\
\qquad\qquad\qquad\quad Cons\ Int\ t_{ys} \ \to\ Int, \\
\qquad\qquad (t_{xs}\ \text{is a}\ Nil)?\text{Fail “Empty list!”}, \\
\qquad\qquad (t_{xs}\ \text{is a}\ Cons)?\text{IsConstrOf}\ (List^S\ Int^D)\ t_{ys} \\
\qquad\qquad \Rightarrow t_{xs} \to Int
\end{array}
$$

Observe how the partiality of *head* has been captured by a guarded predicate Fail; this failure will be raised during specialization if the function is ever applied to the static empty list, causing the whole specialization to fail. In the other item, observe how *head* has been translated to the right projection ($\mathbf{fst}^S$ in this case).

EXAMPLE 9.11. This example shows the specialization of a term accessing a finite part of an infinite static list. The specialization, shown in Fig. 9.4, is obtainable with our approach, but there is no specialization in the original formulation for it. Both the prototype made by Hughes [1997] and our present implementation will loop if the complete specialization of this term is ever attempted; however, with our formulation, we have the possibility to make a constraint solver taking care of static lazy evaluation: only one unfolding of the recursive structure needs to be performed, because arity raising will eliminate the rest. We return to this point in Chapter 14.

Our final example shows the specialization of the source term presented in Example 4.6 with our approach.

EXAMPLE 9.12. The term of Example 4.6 has the following specialization.

$$
\begin{array}{l}
\vdash_P meval\ @^S (Let^S\ \texttt{'i'}^S\ (Lam^S\ \texttt{'x'}^S\ (Var^S\ \texttt{'x'}^S)) \\
\qquad\qquad\quad (App^S (App^S\ (Var^S\ \texttt{'i'}^S)\ (Var^S\ \texttt{'i'}^S)) \\
\qquad\qquad\qquad\quad (Const^S\ 0^S)))
\end{array}
$$

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{let}^{D}\ f = \mathbf{fix}^{S}\ (\lambda^{S} f.\lambda^{S} x.Cons^{S}\ 1^{D}\ (f\ @^{S} x)) \\
\quad \mathbf{in\ case}^{S} f\ @^{S}\ ()^{S}\ \mathbf{of} \\
\qquad Cons\ x\ xs\ \rightarrow\ x \\
\quad : Int^{D} \\
\quad \hookrightarrow \Lambda h_U, h_L, h_{y_s}.\mathbf{let}\ f = \bullet \\
\qquad\qquad\qquad\qquad \mathbf{in\ fst}^{S}\ (h_L @_v f @_v ()^{S}) \\
\qquad : \forall t, t_{x_s}, t_{y_s}, t_e.\mathrm{IsFixS}\ \mathrm{cl}_{x_s}\ t, \\
\qquad\qquad\qquad\qquad \mathrm{IsFunS}\ t\ \mathbf{clos}(t_e : ()^{S} \rightarrow Cons\ Int\ t_{y_s}), \\
\qquad\qquad\qquad\qquad \mathrm{IsConstrOf}\ (List^{S}\ Int^{D})\ t_{y_s} \\
\qquad\qquad\qquad\qquad \Rightarrow Int
\end{array}
$$

where

$$
\begin{array}{l}
\mathrm{cl}_{x_s} = \mathbf{clos}(\Lambda h_f, h_r.\lambda f_s.\lambda f.(f)^{S} \\
\qquad\qquad : \forall t_f, t_r, t_e.\mathrm{IsFunS}\ \mathrm{cl}_f(t_e)\ t_f, \\
\qquad\qquad\qquad\qquad \mathrm{IsFunS}\ \mathrm{cl}_r(t_f)\ t_r \\
\qquad\qquad\qquad\qquad \Rightarrow t_f \rightarrow t_r) \\
\mathrm{cl}_f(t_e) = \mathbf{clos}(t_e : \forall t_{x_s}.\mathrm{IsConstrOf}\ (List^{S}\ Int^{D})\ t_{x_s} \\
\qquad\qquad\qquad\qquad \Rightarrow ()^{S} \rightarrow t_{x_s}) \\
\mathrm{cl}_r(t_f) = \mathbf{clos}(\Lambda h'_{y_s}, h'_L.\lambda^{S} f'_s.\lambda^{S} x.(1, h'_L @_v \pi_{1,1}\ f'_s @_v x)^{S} \\
\qquad\qquad : \forall t'_{y_s}, t'_e.\mathrm{IsConstrOf}\ (List^{S}\ Int^{D})\ t'_{y_s}, \\
\qquad\qquad\qquad\qquad \mathrm{IsFunS}\ t_f\ \mathbf{clos}(t'_e : ()^{S} \rightarrow t'_{y_s}) \\
\qquad\qquad\qquad\qquad \Rightarrow ()^{S} \rightarrow Cons\ Int\ t'_{y_s})
\end{array}
$$

Figure 9.4: A term using an infinite static list.

$$
\begin{array}{l}
: Value^{S} \\
\hookrightarrow \mathbf{let}\ v = \Lambda h.(\lambda v'.v')^{S}\ \mathbf{in}\ \pi_{1,1}\ ((\pi_{1,1}\ (v((Fun)))@(v((Num)))))@(0)^{S} \\
: Num\ Int
\end{array}
$$

Observe that all residuals of $Value^{S}$ are wrapped by a static tuple constructor, and projected before their uses; these constructs are removed by arity raising.

The residual type of the variable v is

$$\mathbf{poly}\ (\forall t.\mathrm{IsConstrOf}\ Value^{S}\ t \Rightarrow Fun\ (t \rightarrow t))$$

showing that we can obtain a (qualified) polymorphic residual function from monomorphic code; in Chapter 11 we show how to make use of this to generate truly polymorphic residuals.

9.7 Dynamic Recursion

Dynamic recursion is added in the same way as in the original formulation, with the rule extended to consider predicates.

$$
\text{(DFIX)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \rightarrow^{D} \tau\ \hookrightarrow\ e' : \tau' \rightarrow \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{fix}^{D}\ e : \tau\ \hookrightarrow\ \mathbf{fix}\ e' : \tau'}
$$

The big difference is that in presence of dynamic recursion, the algorithm produces non-linear IsMG constraints, that is, constraints with the same scheme variable appearing on both sides of the inequation — this is produced by the unification performed by the algorithm between the argument and result types of the residual type of e'. Let's revisit the example of the *power* function.

EXAMPLE 9.13. The source expression for the *power* function is the following one.

$$\begin{array}{l}
\mathbf{let}^{D}\ power = \mathbf{fix}^{D}\ (\lambda^{D} p.\mathbf{poly}\ (\lambda^{D} n.\lambda^{D} x.\mathbf{if}^{S}\ n ==^{S} 1^{S} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ x *^{D} \mathbf{spec}\ p\ @^{D} (n -^{S} 1)\ @^{D} x)) \\
\ \mathbf{in\ spec}\ power\ @^{D} 3^{S}
\end{array}$$

Its principal specialization, as produced by the algorithm, is the following one.

$$\begin{array}{l}
\Lambda h^{u}_{s}, h^{\ell}_{s}.\mathbf{let}\ \ power = \mathbf{fix}\ (\lambda p.h^{u}_{s}[\Lambda h_{n}, h_{b}, h^{u}_{s_2}, h_{n1}. \\
\qquad\qquad\qquad\qquad\qquad \lambda n.\lambda x.\mathbf{if}_{v}\ h_{b} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ x * (h^{u}_{s_2}[p]@\bullet @x)]) \\
\qquad\quad \mathbf{in}\ h^{\ell}_{s}[power]@\bullet \\
: \forall s.\mathrm{IsMG}\ \ (\forall t_{n}, t_{b}, t_{n1}.\mathrm{IsInt}\ \ t_{n}, \\
\qquad\qquad\qquad\qquad t_{b} := t_{n} == \hat{1}, \\
\qquad\qquad\qquad\qquad !t_{b}?\mathrm{IsMG}\ \ s\ (t_{n1} \rightarrow Int \rightarrow Int), \\
\qquad\qquad\qquad\qquad !t_{b}?(t_{n1} := t_{n} - \hat{1}) \\
\qquad\qquad\qquad\qquad\ \ \Rightarrow t_{n} \rightarrow Int \rightarrow Int) \\
\qquad\qquad\quad s, \\
\qquad \mathrm{IsMG}\ \ s\ (\hat{3} \rightarrow Int \rightarrow Int) \\
\qquad\ \Rightarrow Int \rightarrow Int
\end{array}$$

Observe that in the upper bound for the residual scheme variable s, the variable appears in a lower bound for itself! It is not clear what the solution for this kind of inequations should be.

In particular, the constraint solving algorithm we have presented in Chapter 8 is not adequate to handle this kind of 'circular' predicate. Let's see an example.

EXAMPLE 9.14. When an upper bound for a given scheme variable appears in the context of another upper bound for it, our heuristic for constraint solving cannot calculate the greatest lower bound of all the upper bounds as before. The reason is that this heuristic relies on collecting all the upper bounds for a given scheme variable to calculate its actual value.

Observe the following term,

$$\begin{array}{l}
\mathbf{let}^{D}\ f = \mathbf{fix}^{D}\ \ (\lambda^{D} f. \\
\qquad\quad \mathbf{poly}\ (\lambda^{D} n.\lambda^{D} x.\lambda^{D} y. \\
\qquad\qquad\quad \mathbf{let}^{D}\ id = \lambda^{D} z.z \\
\qquad\qquad\quad\ \mathbf{in\ let}^{D}\ g = id\ @^{D} \mathbf{poly}\ (\lambda^{D} n.\lambda^{D} x.\lambda^{D} y.y) \\
\qquad\qquad\qquad \mathbf{in\ if}^{D}\ n ==^{D} 0^{D} \\
\qquad\qquad\qquad\quad \mathbf{then}\ x \\
\qquad\qquad\qquad\quad \mathbf{else\ spec}\ (id\ @^{D} f)\ @^{D} (n -^{D} 1^{D})\ @^{D} x\ @^{D} y)) \\
\ \mathbf{in\ spec}\ f\ @^{D} 3^{D}\ \ @^{D} 4^{S}\ \ @^{D} 4^{S}
\end{array}$$

its principal residual expression,

$$
\begin{array}{l}
\Lambda h^u_{s_1}, h_{xy}, h^\ell_{s_1}.\mathbf{let}\ f = \mathbf{fix}\ (\lambda f.\\
\qquad h^u_{s_1}[\Lambda h^u_{s_2}, h_x, h_y, h^\ell_{s_2}.\\
\qquad\quad \lambda n.\lambda x.\lambda y.\\
\qquad\quad \mathbf{let}\ id = \lambda z.z\\
\qquad\quad \mathbf{in\ let}\ g = id@h^u_{s_2}[\Lambda h'_x, h'_y.\lambda n.\lambda x.\lambda y.y]\\
\qquad\qquad \mathbf{in\ if}\ \ n == 0\\
\qquad\qquad\quad \mathbf{then}\ x\\
\qquad\qquad\quad \mathbf{else}\ h^\ell_{s_2}[(id@f)]@(n-1)@x@y])\\
\quad \mathbf{in}\ h^\ell_{s_1}[f]@3@\bullet@\bullet
\end{array}
$$

and its principal residual type:

$$
\begin{array}{l}
\forall t_x, s.\mathrm{IsMG}\ (\forall t', t_2.\mathrm{IsMG}\ (\forall t_3, t_4.\mathrm{IsInt}\ t_3, \mathrm{IsInt}\ t_4\\
\qquad\qquad\qquad\qquad \Rightarrow Int \rightarrow t_3 \rightarrow t_4 \rightarrow t_4)\\
\qquad\qquad\qquad s,\\
\qquad\qquad \mathrm{IsInt}\ t_2,\\
\qquad\qquad \mathrm{IsInt}\ t',\\
\qquad\qquad \mathrm{IsMG}\ s\ (Int \rightarrow t' \rightarrow t_2 \rightarrow t')\\
\qquad\qquad \Rightarrow Int \rightarrow t' \rightarrow t_2 \rightarrow t')\\
\qquad\quad s,\\
\quad \mathrm{IsInt}\ t_x,\\
\quad \mathrm{IsMG}\ s\ (Int \rightarrow \hat{4} \rightarrow \hat{4} \rightarrow t_x)\\
\quad \Rightarrow t_x
\end{array}
$$

Observe that both upper bounds are necessary to calculate the final solution, because both provide information, restricting the type in the scheme for s to be $Int \rightarrow t \rightarrow t \rightarrow t$. However, the constraint solving heuristic presented in Chapter 8 cannot calculate it.

We consider again the examples of Section 4.4, that are those showing the problems that the interaction between polyvariance and recursion brings. Their principal specializations use predicates, showing clearly where the problems appear — this is one of the important contributions of our formulation.

EXAMPLE 9.15. The source program from Example 4.11

$$
\begin{array}{l}
\mathbf{let}^S\ not = \lambda^S b'.\mathbf{if}^S\ b'\ \mathbf{then}\ False^S\ \mathbf{else}\ True^S\\
\ \mathbf{in}\\
\ \mathbf{let}^D\ f = \mathbf{fix}^D\ (\lambda^D f.\mathbf{poly}\ (\lambda^D b.\mathbf{spec}\ f\ @^D\ (not\ @^S\ b)))\\
\ \ \mathbf{in\ spec}\ f\ @^D\ True^S
\end{array}
$$

making two calls to the function f — one for the value *True* and the other one for the value *False* — has as principal specialization

$$
\begin{array}{l}
\Lambda h, h^u_f, h^\ell_{f_1}.\mathbf{let}\ f = \mathbf{fix}\ (\lambda f.h^u_f[\Lambda h_b, h_r, h^\ell_{f_2}, h_{not}.\lambda b.h^\ell_{f_2}[f]@(h_{not}@_v \bullet @_v b)])\\
\qquad\quad \mathbf{in}\ h^\ell_{f_1}[f]@\bullet
\end{array}
$$

with principal residual type

$$\begin{array}{l}
\forall t_{not}, s.h : \text{IsFunS closure}_{not}\ t_{not} \\
\quad h^u_f : \text{IsMG } (\forall t_b, t_r, t_e.\ \text{IsBool } t_b, \text{IsBool } t_r, \\
\qquad\qquad \text{IsMG } s\ (t_r \to \mathit{Int}), \\
\qquad\qquad \text{IsFunS } t_{not}\ \mathbf{clos}(t_e : t_b \to t_r) \\
\qquad\qquad \Rightarrow t_b \to \mathit{Int}) \\
\qquad\qquad s, \\
\quad h^\ell_{f_1} : \text{IsMG } s\ (\mathit{True} \to \mathit{Int}) \\
\quad \Rightarrow \mathit{Int}
\end{array}$$

where

$$\begin{array}{rcl}
\text{closure}_{not} & = & \mathbf{clos}(\Lambda h_{b'}, h_r, h_{r'}.\ \lambda f_s.\lambda b'.\ \mathbf{if}_v\ h_{b'}\ \mathbf{then}\ \bullet\ \mathbf{else}\ \bullet \\
& & : \forall t_{b'}, t_r.\text{IsBool } t_{b'},\ \text{IsBool } t_r \\
& & \quad t_r := \text{if } t_{b'} \text{ then } \hat{\mathit{False}} \text{ else } \hat{\mathit{True}}, \\
& & \quad \Rightarrow t_{b'} \to t_r)
\end{array}$$

It can be observed that the lower bounds for s are limited to those generated by the values of type variable t_r, which, in turn, is limited by the number of results of the static function *not*. So, to be able to compute the correct residual program, constraint solving needs to detect this bounded static variation of variable t_r, or at least, some kind of memoization of values already computed is needed.

EXAMPLE 9.16. The source program from Example 4.12

$$\begin{array}{l}
\mathbf{let}^D\ f = \mathbf{fix}^D\ (\lambda^D f.\mathbf{poly}\ (\lambda^D x.\lambda^D y. \\
\qquad\qquad \mathbf{if}^D\ \mathbf{lift}\ x ==^D 0^D \\
\qquad\qquad \mathbf{then}\ x +^S 1^S \\
\qquad\qquad \mathbf{else\ spec}\ f\ @^D\ x\ @^D\ y +^S 0^S\ )) \\
\ \mathbf{in}\ \lambda^D z.\mathbf{spec}\ f\ @^D\ 3^S\ @^D\ z \\
\ \ : \mathit{Int}^S \to^D \mathit{Int}^S
\end{array}$$

has as principal specialization

$$\begin{array}{l}
\Lambda h^u_f, h_z, h_r, h^\ell_{f_1}.\mathbf{let}\ f = \mathbf{fix}\ (\lambda f.h^u_f[\Lambda h_x, h_y, h'_x, h'_y, h^\ell_{f_2}. \\
\qquad\qquad \lambda x.\lambda y.\ \mathbf{if}\ h_x == 0 \\
\qquad\qquad\qquad \mathbf{then}\ \bullet \\
\qquad\qquad\qquad \mathbf{else}\ h^\ell_{f_2}[f]@x@\bullet]) \\
\qquad \mathbf{in}\ \lambda z.h^\ell_{f_1}[f]@\bullet@z \\
: \forall t_z, t_r, s. \\
\quad \text{IsMG}(\forall t_x, t_y, t'_x, t'_y. \\
\qquad \text{IsInt } t_x, \\
\qquad \text{IsInt } t_y, \\
\qquad t'_x := t_x + \hat{1}, \\
\qquad t'_y := t_y + \hat{0}, \\
\qquad \text{IsMG } s\ (t_x \to t'_y \to t'_x) \\
\qquad \Rightarrow t_x \to t_y \to t'_x) \\
\qquad s, \\
\quad \text{IsInt } t_z, \\
\quad \text{IsInt } t_r, \\
\quad \text{IsMG } s\ (\hat{3} \to t_z \to t_r) \\
\quad \Rightarrow t_z \to t_r
\end{array}$$

In this example, the values of t_x and t'_x are not related with those of t_y and t'_y; moreover, the values of these last two variables will be the same, because of the predicate $t'_y := t_y + \hat{0}$, implying that there will be only one lower bound for s. However, these facts have to be detected by the algorithm in order for the right residual to be produced.

EXAMPLE 9.17. The source program from Example 4.14

$$
\begin{array}{l}
\mathbf{let}^D\ id = \lambda^D z.z \\
\ \mathbf{in} \\
\ \mathbf{let}^D\ f = \mathbf{fix}^D\ (\lambda^D f. \\
\qquad\qquad\qquad \mathbf{poly}\ (\lambda^D b.\lambda^D x.\lambda^D y. \\
\qquad\qquad\qquad\qquad \mathbf{if}^D\ \mathbf{lift}\ b \\
\qquad\qquad\qquad\qquad\ \mathbf{then\ lift}\ (id\ @^D\ x +^S id\ @^D\ y) \\
\qquad\qquad\qquad\qquad\ \mathbf{else\ spec}\ f\ @^D\ b\ @^D\ x\ @^D\ y)) \\
\ \ \mathbf{in}\ \lambda^D b'.(\mathbf{spec}\ f\ @^D\ b'\ @^D\ 2^S\ @^D\ 2^S, \mathbf{spec}\ f\ @^D\ b'\ @^D\ 3^S\ @^D\ 3^S)
\end{array}
$$

has as principal specialization

$$
\begin{array}{l}
\Lambda h_{b'}, h_x, h^u_f, h^\ell_{f_1}, h^\ell_{f_2}. \\
\ \ \mathbf{let}\ id = \lambda z.z \\
\ \ \ \mathbf{in} \\
\ \ \ \mathbf{let}\ f = \mathbf{fix}^D\ (\lambda f.h^u_f[\Lambda h_b, h'_z, h^\ell_f. \\
\qquad\qquad\qquad \lambda b.\lambda x.\lambda y.\ \mathbf{if}\ h_b \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ h'_z + h'_z \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ h^\ell_f[f]\ @^D\ b\ @^D\ x\ @^D\ y]) \\
\ \ \ \ \mathbf{in}\ \lambda b'.(h^\ell_{f_1}[f]@b'@\bullet@\bullet, h^\ell_{f_2}[f]@b'@\bullet@\bullet) \\
: \forall t_{b'}, t_z, s. \\
\quad h_{b'} : \mathrm{IsBool}\ t_{b'}, \\
\quad h_x : \mathrm{IsInt}\ t_z, \\
\quad h^u_f : \mathrm{IsMG}(\forall t_b.h_b : \mathrm{IsBool}\ t_b, \\
\qquad\qquad\qquad h'_z : \mathrm{IsInt}\ t_z, \\
\qquad\qquad\qquad h^\ell_f : \mathrm{IsMG}\ s\ (t_b \to t_z \to t_z \to Int) \\
\qquad\qquad\qquad \Rightarrow t_b \to t_z \to t_z \to Int) \\
\qquad\qquad\quad s, \\
\quad h^\ell_{f_1} : \mathrm{IsMG}\ s\ (t_{b'} \to \hat{2} \to \hat{2} \to Int), \\
\quad h^\ell_{f_2} : \mathrm{IsMG}\ s\ (t_{b'} \to \hat{3} \to \hat{3} \to Int) \\
\quad \Rightarrow t_{b'} \to (Int, Int)
\end{array}
$$

which has no solution, because any solution requires that t_z be unified with $\hat{2}$ and $\hat{3}$ at the same time — observe that the residual type t_z appears free in the upper bound for s. That is caused because the monovariant function *id* is used with two different static values!

The last example showed how an algorithm using guessing and backtracking can be made to fall into a loop (this is the case for the original prototype of Hughes [1997]); the constraint solving we have presented performs no backtracking, but it may be the case that residual variables can be used as backtracking points: in this case, the fact that t_z

is not local to the upper bound of s can be used to identify the fact that the error is not coming from the identification of lower bounds. This particular point requires further research — see Chapter 14.

9.8 Other Features

There are other features that can be added to the source language to increase the expressiveness; dynamic datatypes, polyvariant sums, and imperative (monadic) features are some of them.

Dynamic datatypes can be defined using named (inductive) sums, and can be treated with similar techniques as those used for polyvariance and static functions; surprisingly, defined in this way, they are independent of the use of recursive types — remember that, as we have mentioned in Section 3.4.5, the original formulation needs recursive types in order to have inductive types. We can express constraints to the residual named sums by new predicates, and let constraint solving produce the required residual types.

It is less clear how to add polyvariant sums and imperative features. The details of the addition of dynamic datatypes and the addition of the other features are left for future work — see Chapter 14.

Chapter 10

The Prototype

Es una linda ración,
con un defecto (con uno o dos).
Y es un cóctel que no se mezcla solo.[1]

Un Poco de Amor Francés
La Mosca y La Sopa
Patricio Rey y sus Redonditos de Ricota

To put the ideas into practice, we have developed a small prototype implementation of a type specializer following our development of previous chapters — that is, the algorithm expressed by the system $\vdash_{\mathrm{W}}$. The intention was to test the ideas with small programs, obtaining feedback for the detection of errors in the development and the finding of new problems and ideas. The prototype accepts a program written in the source language, performs source type inference for it, and then proceed with its type specialization, producing a residual term and residual type. There are flags controlling whether the output should be the principal specialization, the solution found by the constraint solver, or an evidence eliminated residual term. All the examples presented in this thesis were produced by the prototype. We have used the functional language Haskell [Peyton Jones and Hughes (editors), 1999] as our implementation language.

In this chapter we describe the prototype, its features, and the lessons we have learned by implementing it. We start in Section 10.1 with a brief tutorial of how to use the specializer. Then we describe, in Section 10.2, the main elements of its architecture, and some of its relevant features. In Section 10.3 we discuss the insights and lessons we have learned by implementing this prototype, and conclude the chapter in Section 10.4 discussing the things we think are needed to produce a specializer handling more realistic programs.

10.1 A Brief Tutorial

To start with, we present a small tutorial on how to use our specializer. Its interface is intended to have a similar look as the Hugs interpreter. For that reason, the specializer presents a prompt with the name of the last file loaded. At the beginning, when no file has been loaded, the prompt says so.

[1] *It's a fine portion,*
with one defect (one or two).
And it's a cocktail that does not blend itself.

The behaviour of the specializer consists in accepting a command, performing the requested action, and if the command was not the one to quit, returning to the waiting state. The commands accepted can be divided in the following categories: general commands, file management, and expression management.

The general commands are two:

- `:h`, `:?` — present a small help describing the basic commands.
- `:q` — quit the specializer.

File management is the process of loading expressions from different files into the specializer. Each file should contain exactly one expression. The last file loaded is the default one, unless changed by a command. The default file is taken into account by the commands for expression management. The files containing expressions for the type specializer are assumed to have the extension `.pts`, but the names used in the commands must not contain it. In the case of using expressions, the names of the files can be used as variables representing the corresponding expressions — see the example below. The commands for file management are:

- `:a` < directory > — (`a`)dd a new directory to the default path; all the commands in this group look for files in the current directory and in those of the default path.
- `:o` < directory > — rem(`o`)ve the given directory from the default path.
- `:l` < file name > — (`l`)oad a file, parsing the expression in it.
- `:r` [file name] — (`r`)eload the file; if issued without arguments, reload the default file.
- `:d` [file name] — (`d`)elete the file from the list of loaded files; if issued without arguments, the default file is deleted.
- `:c` — (`c`)lear all loaded files.
- `:f` — show the names of all loaded (`f`)iles.
- `:s` < file name > — (`s`)witch the default file to that specified.
- `:u` < script name > — r(`u`)n a script; a script is a special file containing commands for the specializer, and can be used to automate common tasks, such as setting a path.

The commands in the expression management group are the ones implementing the work proper to the specialization process. Each one of them accepts either a file name, or an expression; in the case of receiving a file name, the set of loaded files is searched, and the corresponding expression is used. They can also be invoked without arguments, taking the default file as the one to process. The commands in this group are the following ones:

- `:e` [expression] — show the (`e`)xpression

- `:t` [expression] — infer the (`t`)ype of the expression
- `:ts` [expression] — perform the principal (`t`)ype (`s`)pecialization of the expression, without constraint solving.
- `:cs` [expression] — perform the specialization of the expression, with (`c`)onstraint (`s`)olving, but without evidence elimination.
- `:ee` [expression] — perform the specialization of the expression, with (`c`)onstraint (`s`)olving and (`e`)vidence (`e`)limination.
- `:ar` [expression] — perform the specialization of the expression, with all the post-processing phases; it has not been implemented, because we have not treated arity raising in this thesis.

Supposing that the examples to specialize are contained in the directory `examples`, and that they are called `ex01.pts` (with expression `\x -> x`), `ex02.pts` (with expression `\x -> lift x`), and `ex03.pts` (with expression `(\x -> lift x) @ 2`), the example given in Figures 10.1 and 10.2 is a typical run.

10.2 Overall Architecture and Relevant Features

The prototype is organized into two clearly differentiated parts. The first part includes all the modules providing operations on the source language, including parsing, type inference, and pretty printing. The second part contains those modules involving the residual language and the specialization process, including constraint solving. They are integrated by a module providing interaction with the user.

Parsing is implemented using an variation of monadic parsing combinators in the style of Hutton and Meijer [1998]. Pretty printing is implemented using the library designed by Hughes [1995]. The grammar for the language separates the precedence of different operators, which allows the use of a minimal number of parenthesis when writing expressions; both parsing and printing combinators take advantage of that and thus the user needs only a minimal number of parenthesis.

Source type inference is implemented using a state monad, representing substitutions by a table that associates variables to types, in a very similar way as it is done in the first implementation presented by Sheard [2001]. The state captures the generation of fresh variables and the substitution of type and annotation variables. An interesting feature of our source type inference is that it performs annotation inference: all constructs without explicit annotations are assigned annotation variables, and those variables are unified in accordance with the type inference rules. In this way, if a construct (e.g. a function) is declared static, all its uses (i.e. applications) are automatically inferred as static too — and vice-versa: if any use is static, the construct is inferred static. At the very end, all annotation variables that remain uninstantiated are set dynamic — this is an arbitrary, but conservative, choice. Another arbitrary choice is that, as the source language is monomorphic, all type metavariables that remain uninstantiated at the end are set to *Int*.

```
PPPP  TTTTT  SSSS          1     5555
P   P   T   S             11     5
PPPP    T    SSS     v   v  1     5555
P       T       S     v v   1  ..     5
P       T   SSSS       v   111 .. 5555

 by Pablo E. Martínez López (Fidel)
 E-mail: fidel@info.unlp.edu.ar
 March 2005

No file loaded> :a examples

No file loaded> :l ex01
Parsing ex01...

ex01> :e
\x -> x

ex01> :t
Source program.
---------------
\x -> x
::
Int -> Int

ex01> :l ex02
Parsing ex02...

ex02> :l ex03
Parsing ex03...

ex03> :f
Loaded files:
        ex03
        ex02
        ex01

ex03> :ts
Source program.
---------------
( \x -> lift x ) @ 2^S
::
Int
```

Figure 10.1: Example of run from the type specializer (I).

```
PTS...

Its principal type specialization.
----------------------------------
( \x -> 2 ) @ *
::
Int

ex03> :ts ex02
Source program.
---------------
\x -> lift x
::
Int^S -> Int

PTS...

Its principal type specialization.
----------------------------------
/\h2. \x -> h2
::
\/gt. h2 :: IsInt gt => gt -> Int

ex03> :ts ex02 @ 4
let^S ex02 = \x -> lift x in ex02 @ 4

Source program.
---------------
let^S ex02 = \x -> lift x in ex02 @ 4^S
::
Int

PTS...

Its principal type specialization.
----------------------------------
( \x -> 4 ) @ *
::
Int

ex03> :q
It was good to specialize with you. Bye!
```

Figure 10.2: Example of run from the type specializer (II).

The specialization is also implemented using a state monad, following the same ideas as those used in the source type monad. The main differences are the use of different constructors for universally quantified variables in type schemes, as done by Jones [1999], but using de Bruijn indices to avoid alpha conversions [de Bruijn, 1972; de Bruijn, 1978]. Predicate assignments are implemented using lists of predicates, and simplification and constraint solving just traverses those lists multiple times.

10.3 What Have We Learned?

By implementing the prototype we have learned several things, both about type specialization and about Haskell and typing disciplines.

The most important lesson is related to constraint solving. At the beginning of this thesis, we naively believed that collecting the constraints was enough to express type specialization. This changed substantially after looking at the predicates generated from the most simple examples. Gradually we began to understand that the real challenge in our formulation of type specialization lies in the way constraints are solved. In this thesis we have presented a very primitive constraint solver, not able to manage some complex examples; but we also have detected several different problems posed by type specialization, that have to be tackled before our approach can be used for real programs. While some of those problems were addressed in the prototype of the original formulation, they were very difficult to express with that approach, forcing us to think in terms of their implementation, and thus limiting our ability to understand them; and the solutions proposed (e.g. backtracking) also introduce their own problems (i.e. non-termination of some failing examples). We are convinced that our ability to generate constraints for any source program, even erroneous ones, is important because it allows us to understand the problems, and to help in the searching of solutions. We also explain, in the next chapter, how by modifying the constraint solving phase, different heuristics can be tested, thus showing the significance of constraint solving.

Another lesson was to confirm in practice once again the importance of a disciplined use of types. Not only the Haskell type system has been extremely helpful to avoid hundreds of common mistakes, but also the judicious use of evidence as typed objects (although we have not designed an explicit type system, there is a correspondence between each evidence and the predicates it proves) has been invaluable in avoiding bugs. In particular, every time we have to apply a conversion to an expression e', we checked that e' has the right type scheme (see Theorem 6.12), and use the result as having the resulting scheme. The correspondence between evidence and predicates has been also useful, on occasions, to find errors during the formulation of some rules: our first attempt for the evidence used in rule (REC-CLOS) was wrong, and that was discovered by a type mismatch in the implementation (but not caught by the Haskell type system! It may have been better to design some type system for conversions, after all...).

The third lesson we have learned is the absolute need for annotation inference. Our first version of the prototype only performed annotation checking, and to write a program of certain size that passes the source typecheck was a difficult task, involving several iterations. After we have implemented annotation inference, it was much easier to produce the desired annotations. However, it is still difficult to detect all the points

where decisions have to be taken. For that purpose, an interactive tool performing annotation completion would be essential. This is not separate from the issue of semi-automatic binding time analysis, but it is much easier to achieve, and for that reason we consider that it is an important help to have in a tool.

One important issue that we find by using the prototype is that the size of constraints generated grew quickly with the size of the program. This can be alleviated by simplifying the constraints, as described in Chapter 8, and also by using local declarations, as those we have used in some examples (which also improves readability). The number and size of the constraints have an important impact in the efficiency of constraint solving, which constitutes the most expensive operation in the prototype. However, it remains to be determined if this complexity is inherent to type specialization, or if it is something introduced by our approach.

Finally, we want to remark that the simplicity of the prototype is based almost completely in the right level of detail and the clarity provided by the use of Mark Jones' framework of qualified types. The neat separation of type information from evidence information allowed the separation of those issues in (almost) independent modules, facilitating both coding and maintenance.

10.4 Towards a Proper Implementation

The prototype just described is a small toy to test ideas, and specialize small examples. To be able to perform specialization of bigger and more realistic programs, a full-fledged implementation would have to be constructed. We finish the discussion about the prototype discussing those points we consider essential to produce that implementation of our approach for type specialization.

The use of heap profiling confirmed that the most expensive part of the specializer is the constraint solver. For that reason, it is very important to have an efficient implementation of that phase; our current implementation just performs several iterations over lists of constraints, without any concern for efficiency. There are several possibilities to achieve efficiency. On the one hand, using a representation of predicate assignments that distinguishes them will help in determining which rule can be applied; this is possible because of the separation of constraint solving rules according to the different kinds of predicates. On the other hand, having more compact representation of predicates will help as well; the current implementation uses an algebraic datatype that follows the grammar, with several operations traversing the trees (for example, equality comparisons or unifications). Finally, static recursion gives rise to several predicates that have to be checked but that do not contribute with evidence. With our present solver, those checks can sometimes be repeated; more care in the treatment of those cases will improve the efficiency of programs using static recursion.

Another issue that we detect as a possible source of inefficiency is the use of nested static lambda abstractions. Every static lambda generates a residual closure, and then, nested lambdas generate nested closures. When those nested closures are processed by the constraint solver, they will have to be traversed to obtain the final evidence. For that reason, finding a better representation for nested closures can be helpful to improve efficiency. The main problem consists in the expression of partial applications of those

nested closures.

It is one of our goals to produce a realistic implementation following the ideas described here, but we have left the task for future work (Chapter 14).

Part III

Type Specializing Polymorphism

Chapter 11

Inherited Limit: Polymorphism

You'll reap the harvest you have sown.

Dogs – Animals
Pink Floyd

In this chapter we present an extension of principal type specialization that can produce polymorphic residual programs from monomorphic source ones. This is done to show the possibilities of our approach. However, the result has not been thoroughly tested, and we think that it needs some improvements before being able to state that the inherited limit of polymorphism has been completely removed.

This result is obtained by a new annotation, **polym**, whose behaviour is similar to polyvariance, but with a different rule for evidence elimination: instead of introducing a tuple with copies for each specialization, **polym** forces all the copies to be the same, and thus generates one single element with a more general, *polymorphic*, type. To check whether all the different specializations of a given expression will be the same, some form of usage analysis on evidence variables is needed; but, as we are generating code, instead of performing a separate analysis, we can add some information to the predicates in such a way that the usage information is collected during the construction of the residual term. This usage information can also be used to improve the treatment of polyvariance.

The chapter is organized as follows. In Section 11.1 we revisit the example of the monomorphizer for lambda-calculus given in Chapter 3, showing what is the principal specialization of it, and analyzing it to motivate the need for the new annotation. In Section 11.2 we introduce the new annotation, **polym**, and the rules specifying its behaviour during specialization; the most important change is in the rules for evidence elimination. In Section 11.3 we show how the new annotation can be used to annotate the interpreter for lambda-calculus of the previous example in such a way that by specializing it, we can produce polymorphic residual code. Finally, in Section 11.4 we discuss how the ideas that we used to introduce polymorphism can be used to improve the treatment of polyvariance.

11.1 Monomorphizing Lambda-calculus, Revisited

The interpreter of lambda-calculus we have considered in Chapter 3 can be used to produce a monomorphizer for a let-bound polymorphic lambda calculus; the annotated program to do this was presented in Figure 4.3 (function *meval*), and an example of its specialization using the original formulation of type specialization was presented in

Example 4.6. In that example, the polyvariant term expressing the polymorphic use of a let-bound expression gave rise to a tuple of monomorphic copies. By using our formulation, the same example can be specialized to a different residual code, using evidence abstraction and application.

EXAMPLE 11.1. The result of specializing the expression

$$\begin{array}{l} \mathit{meval} \ @^S (\mathit{Let}^S \ 'i' \ (\mathit{Lam}^S \ 'x' \ (\mathit{Var}^S \ 'x')) \\ \qquad\qquad (\mathit{App}^S \ (\mathit{App}^S \ (\mathit{Var}^S \ 'i') \ (\mathit{Var}^S \ 'i')) \\ \qquad\qquad\qquad (\mathit{Con}^S \ 0^S))) \end{array}$$

is the expression

$$\begin{array}{l} \mathbf{let} \ v = \Lambda h.\lambda v'.v' \\ \ \mathbf{in} \ (v((\mathit{Fun}))@(v((\mathit{Fun}))))@0 \\ \ :: \ \mathit{Num\ Int} \end{array}$$

Observe how the residual of the let-bound identity function is an evidence abstraction, instead of a tuple; its type is

$$\mathbf{poly} \ (\forall t.\text{IsResidualOf} \ t \ \mathit{Value}^S \Rightarrow \mathit{Fun} \ (t \to t))$$

— the evidence variable h is waiting for evidence that type t is the residual type of an expression of source type Value^S. So, in order to use the function, it must first be provided with suitable evidence that the corresponding instance of t satisfies the predicate, which is done by evidence application; the first *Fun* is the evidence that

$$\mathit{Fun} \ ((\mathit{Fun} \ (\mathit{Num\ Int} \to \mathit{Num\ Int})) \to (\mathit{Fun} \ (\mathit{Num\ Int} \to \mathit{Num\ Int}))$$

is the residual of Value^S, and that of the second one is the evidence that

$$\mathit{Fun} \ (\mathit{Num\ Int} \to \mathit{Num\ Int})$$

is also a residual of Value^S.

As can be seen, the evidence h does not play any role in the construction of the residual term; it is needed to distinguish the different uses of the overloaded expression. When performing evidence elimination, every evidence application with a different type will produce a new element in the resulting tuple, but as can be seen, all these elements are equal. This is a key observation that allows us to introduce, in the next section, a new annotation that produces polymorphic code.

11.2 An Annotation to Generate Polymorphism

We begin by extending the source language with a new annotation, **polym**, and its corresponding annotation to eliminate it, **inst**. These are similar to **poly** and **spec**, and thus they will also be reflected in source types in a similar way.

$$\begin{array}{l} e ::= \ \ldots \mid \mathbf{polym} \ e \mid \mathbf{inst} \ e \\ \tau ::= \ \ldots \mid \mathbf{polym} \ \tau \end{array}$$

The rules of source typing for the new annotations are identical to the the ones corresponding to **poly** — see Section 6.3.1.

$$\text{(ST-POLYM)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e : \tau}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \textbf{polym}\ e : \textbf{polym}\ \tau}$$

$$\text{(ST-INSTC)}\quad \frac{\Gamma_{\text{ST}} \vdash_{\text{ST}} e : \textbf{polym}\ \tau}{\Gamma_{\text{ST}} \vdash_{\text{ST}} \textbf{inst}\ e : \tau}$$

For the specification of the specialization of **polym** expressions, we need to introduce a new residual type constructor, **polym**, and a new predicate similar to IsMG, but with a different behaviour during evidence elimination.

$$\begin{array}{lcl} \tau' & ::= & \ldots \mid \textbf{polym}\ \sigma \\ \delta & ::= & \ldots \mid \text{IsMGmorph}\ \sigma\ \sigma \end{array}$$

Using the new constructs, the rules to specialize **polym** are almost identical to those for **poly** — see Section 6.3.2.

$$\text{(POLYM)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma' \qquad \Delta \Vdash v : \text{IsMGmorph}\ \sigma'\ \sigma}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{polym}\ e : \textbf{poly}\ \tau \hookrightarrow v[e'] : \textbf{polym}\ \sigma}$$

$$\text{(INSTC)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \textbf{polym}\ \tau \hookrightarrow e' : \textbf{polym}\ \sigma \quad \Delta \Vdash v : \text{IsMGmorph}\ \sigma\ \tau' \qquad \Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{inst}\ e : \tau \hookrightarrow v[e'] : \tau'}$$

The source-residual relationship must also be extended with a rule for **polym**.

$$\text{(SR-POLYM)}\quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma' \qquad \Delta \Vdash \text{IsMGmorph}\ \sigma'\ \sigma}{\Delta \vdash_{\text{SR}} \textbf{polym}\ \tau \hookrightarrow \textbf{polym}\ \sigma}$$

And finally, rules for the entailment of the new predicate are needed. They are identical to the rules for IsMG.

$$\text{(IsMGmorph)}\quad \frac{C : (\Delta \mid \sigma') \geq (\Delta \mid \sigma)}{\Delta \Vdash C : \text{IsMGmorph}\ \sigma'\ \sigma}$$

$$\text{(MorphComp)}\quad \frac{\Delta \Vdash v : \text{IsMGmorph}\ \sigma_1\ \sigma_2 \qquad \Delta \Vdash v' : \text{IsMGmorph}\ \sigma_2\ \sigma_3}{\Delta \Vdash v' \circ v : \text{IsMGmorph}\ \sigma_1\ \sigma_3}$$

The key difference between **poly** and **polym** is the way evidence elimination is performed. For polyvariant expressions, each different lower bound will generate a tuple element, specialized to the evidence proving that the predicates in the upper bound are satisfied — this can be seen in Examples 11.1 and 4.6. But to generate polymorphism, all the uses of the expression have to share the *same* residual term; this means that the polymorphic residual term have to be equal for all the different uses of evidence — that is, if some evidence has more than one different value, then that evidence must not be used in the construction of the term. This restriction has to be enforced during evidence elimination for **polym**, and thus we need information about the usage of the evidence appearing in evidence applications corresponding to lower bounds. Consider the following example.

Example 11.2. The expression

$$\begin{array}{l}\mathbf{let}^{D}\ f = \mathbf{polym}\ (\lambda^{D}x.\mathbf{lift}\ x +^{D} 1^{D})\\ \ \mathbf{in}\ (\mathbf{inst}\ f\ @^{D}\ 42^{S}, \mathbf{inst}\ f\ @^{D}\ 17^{S})^{D}\\ : (Int^{D}, Int^{D})^{D}\end{array}$$

cannot be specialized. Observe the use of **polym**, and compare this term with that in Example 3.12-1, which uses **poly**. The problem here is that **polym** enforces that all the possible specializations of its argument expression must have the same form, that is, they must not depend on the evidence abstracted. But the expression $\Lambda h_t.\lambda x'.h_t + 1$ that is the residual of f uses the evidence h_t, and thus it must raise an error.

On the other hand, the following specialization is possible

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{let}^{D}\ f = \mathbf{polym}\ (\lambda^{D}x.x +^{S} 1^{S})\\ \quad \mathbf{in}\ (\mathbf{inst}\ f\ @^{D}\ 42^{S}, \mathbf{inst}\ f\ @^{D}\ 17^{S})^{D}\\ \quad : (Int^{S}, Int^{S})^{D}\\ \quad \hookrightarrow\\ \quad \mathbf{let}\ f' = \Lambda h, h'.\lambda x'.\bullet\\ \quad\ \mathbf{in}\ (f'((42, 43))@\bullet, f'((17, 18))@\bullet)\\ \quad : (\hat{43}, \hat{18})\end{array}$$

and when evidence is eliminated, it will produce the following residual term:

$$\mathbf{let}\ f' = \lambda x'.\bullet\ \ \mathbf{in}\ (f'@\bullet, f'@\bullet) : (\hat{43}, \hat{18})$$

Observe that evidence elimination simply removes the evidence abstraction $\Lambda h, h'. \ldots$, and the corresponding evidence applications. The residual term of function f' is then **polym** $(\forall t, t'.t \to t')$ — the predicates relating t with t' were removed by the evidence elimination, because they have no effect in the residual term.

It is important to note that after introducing this change, the residual type produced is no longer in the typing relation for the residual language (the system $\vdash_{\mathrm{RT}}$). It is necessary to modify such typing relation to have polymorphic types into account. We left this as future work.

The way to enforce this restriction is by classifying the predicates in two groups: those whose evidence is actually used (as the one in the first case of the previous example), and those whose evidence is not used. The algorithm calculating the principal specialization for a **polym** has to add to the upper bound only those predicates whose evidence will not be used, leaving the rest in the predicate assignment — the effect of this is that those predicates will not be quantified and thus will be forced to have only one form.

To keep track of the usage of evidence, we use an abstract domain with two elements, N for not used and U for used, and let every predicate be annotated with one of these. The order in the domain is $N \leq U$, meaning that it is always safe to consider as used something that it is not used. In the inference process, usage variables a will be used to propagate unknown usage information.

Specialization rules will propagate the usage information. Each time some evidence is actually used, the information will be set to U — this occurs in the rules (LIFT), (POLY),

$$\text{(LIFT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : Int^{S} \;\hookrightarrow\; e' : \tau' \qquad \Delta \Vdash v : \mathrm{IsInt}^{U}\ \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{lift}\ e : Int^{D} \;\hookrightarrow\; v : Int}$$

$$\text{(POLY)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma' \qquad \Delta \Vdash v : \mathrm{IsMG}^{U}\ \sigma'\ \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[e'] : \mathbf{poly}\ \sigma}$$

$$\text{(SPEC)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \mathbf{poly}\ \tau \;\hookrightarrow\; e' : \mathbf{poly}\ \sigma \qquad \Delta \Vdash v : \mathrm{IsMG}^{U}\ \sigma\ \tau' \qquad \Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{spec}\ e : \tau \;\hookrightarrow\; v[e'] : \tau'}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \delta^{a} \Rightarrow \rho \qquad \Delta \Vdash v : \delta^{a}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e'(\!(v)\!) : \rho}$$

Figure 11.1: Specialization rules extended with usage information.

$$\text{(IsInt)}\quad \Delta \Vdash n : \mathrm{IsInt}^{a}\ \hat{n}$$

$$\text{(IsOp)}\quad h : \Delta \Vdash n : \hat{n} :=^{a} \hat{n}_1 + \hat{n}_2 \quad (\textit{whenever}\ n{=}n_1{+}n_2)$$

$$\text{(IsOpIsInt)}\quad \Delta, h : \tau' :=^{a} \tau'_1 + \tau'_2, \Delta' \Vdash h : \mathrm{IsInt}^{a}\ \tau'$$

$$\text{(IsMG)}\quad \frac{C : (\Delta \mid \sigma') \geq (\Delta \mid \sigma)}{\Delta \Vdash C : \mathrm{IsMG}^{a}\ \sigma'\ \sigma}$$

$$\text{(Comp)}\quad \frac{\Delta \Vdash v : \mathrm{IsMG}^{a'}\ \sigma_1\ \sigma_2 \qquad \Delta \Vdash v' : \mathrm{IsMG}^{a''}\ \sigma_2\ \sigma_3 \qquad a \leq a' \qquad a \leq a''}{\Delta \Vdash v' \circ v : \mathrm{IsMG}^{a}\ \sigma_1\ \sigma_3}$$

Figure 11.2: Entailment for evidence construction with usage information.

(SPEC), (POLYM), and (INSTC), presented in Figure 11.1. Additionally, the rule (QOUT) for discharging a predicate from the assignment will propagate usage information.

Rules for entailment must also propagate usage information, as presented in Figure 11.2

The usage information will be taken into account by the rule (W-POLYM), in order to decide which predicates will be included in the upper bound, and which ones won't.

$$\frac{h : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e' : \tau' \quad \Delta^{U}, \Delta' \mid \emptyset \Vdash_{\mathrm{W}} v : \mathrm{IsMGmorph}\ (\mathrm{Gen}_{S\,\Gamma,\emptyset}(\Delta^{N} \Rightarrow \tau'))\ s}{\Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{polym}\ e : \mathbf{polym}\ \tau \;\hookrightarrow\; v[\Lambda h.e'] : \mathbf{polym}\ s} \;\text{(W-POLYM)}\quad (s\ \textit{fresh})$$

where the notation Δ^{U} (Δ^{N}) means those predicates of Δ marked as used (not used).

The next example shows that, to have all the usage information available when specializing a **polym**, a naive algorithm is not enough.

EXAMPLE 11.3. Observe that function f has to pass its evidence to g, and it is the latter which decides if the evidence will be used or not. In the first example it does not

use it, and in the second one, it does; moreover, the two argument functions presented use different evidence.

1. $\vdash_{\mathrm{P}}$ $\mathbf{let}^{D}\ f = \lambda^{D} g.\mathbf{poly}\ (\lambda^{D} x.\mathbf{spec}\ g\ @^{D}\ x\ @^{D}\ x)$
   $\mathbf{in}\ (\ \mathbf{spec}\ (f\ @^{D}\ \mathbf{poly}\ (\lambda^{D} y_1.\lambda^{D} y_2.0^{D}))$
   $\quad ,\ \mathbf{spec}\ (f\ @^{D}\ \mathbf{poly}\ (\lambda^{D} y_1.\lambda^{D} y_2.1^{D}))$
   $\quad)^{D}$
   $: (\mathit{Int}^{S} \to^{D} \mathit{Int}^{D}, \mathit{Int}^{S} \to^{D} \mathit{Int}^{D})^{D}$
   $\hookrightarrow \Lambda h_1, h_2.$
   $\quad \mathbf{let}\ f = \lambda g.\Lambda h_x, h_g^{\ell}.\lambda x.h_g^{\ell}[g]@x@x$
   $\quad \mathbf{in}\ ((f@(\Lambda h_{y_1}, h_{y_2}.\lambda y_1.\lambda y_2.0))((h_1))(([]((h_1))((h_1))))$
   $\quad\quad ,(f@(\Lambda h_{y_1}, h_{y_2}.\lambda y_1.\lambda y_2.1))((h_2))(([]((h_2))((h_2)))))$
   $\quad : \forall t_1, t_2.\mathrm{IsInt}\ t_1,$
   $\quad\quad \mathrm{IsInt}\ t_2,$
   $\quad\quad \Rightarrow (t_1 \to \mathit{Int}, t_2 \to \mathit{Int})$

2. $\vdash_{\mathrm{P}}$ $\mathbf{let}^{D}\ f = \lambda^{D} g.\mathbf{poly}\ (\lambda^{D} x.\mathbf{spec}\ g\ @^{D}\ x\ @^{D}\ x)$
   $\mathbf{in}\ (\ \mathbf{spec}\ (f\ @^{D}\ \mathbf{poly}\ (\lambda^{D} y_1.\lambda^{D} y_2.\mathbf{lift}\ y_1))$
   $\quad ,\ \mathbf{spec}\ (f\ @^{D}\ \mathbf{poly}\ (\lambda^{D} y_1.\lambda^{D} y_2.\mathbf{lift}\ (y_1 +^{D} y_2)))$
   $\quad)^{D}$
   $: (\mathit{Int}^{S} \to^{D} \mathit{Int}^{D}, \mathit{Int}^{S} \to^{D} \mathit{Int}^{D})^{D}$
   $\hookrightarrow \Lambda h_1, h_2, h'_1, h'_2.$
   $\quad \mathbf{let}\ f = \lambda g.\Lambda h_x, h_g^{\ell}.\lambda x.h_g^{\ell}[g]@x@x$
   $\quad \mathbf{in}\ ((f@(\Lambda h_{y_1}, h_{y_2}, h_{y_{12}}.\lambda y_1.\lambda y_2.h_{y_1}))((h_1))(([]((h_1))((h_1))((h'_1))))$
   $\quad\quad ,(f@(\Lambda h_{y_1}, h_{y_2}, h_{y_{12}}.\lambda y_1.\lambda y_2.h_{y_{12}}))((h_2))(([]((h_2))((h_2))((h'_2)))))$
   $\quad : \forall t_1, t_2, t'_1, t'_2.\mathrm{IsInt}\ t_1,$
   $\quad\quad \mathrm{IsInt}\ t_2,$
   $\quad\quad t'_1 := t_1 + t_1,$
   $\quad\quad t'_2 := t_2 + t_2$
   $\quad\quad \Rightarrow (t_1 \to \mathit{Int}, t_2 \to \mathit{Int})$

Had we used **polym** instead of **poly** in the previous examples, the specialization of the body of function f would not have all the usage information available, until the arguments of f will be known.

This shows that this approach needs further development; however, as our goal is only to show the potential of principal type specialization we will not proceed further, leaving this as a future work — see Chap. 14.

In Section 8.3.3 we have discussed the different possibilities to perform evidence elimination. The new annotation **polym** will be eliminated by considering the last option mentioned — that is, the production of different residual codes — but forcing the process to guarantee that only a one element tuple will be produced. The usage information can be used to avoid repeated computations.

```
data LExp^S = Var Char^S                  | Const Int^S
            | Lam Char^S LExp^S           | App LExp^S LExp^S
            | Let Char^S LExp^S LExp^S
data Value^S = Num Int^D | Fun (Value^S →^D Value^S) | Wrong
data MP^S    = M Value^S | P (polym Value^S)
let^S bind = λ^S x → λ^S v → λ^S env →
               λ^S y → if^S x == y then v else env @^S y
 in
 let^S preeval =
        fix^S (λ^S eval → λ^S env → λ^S expr →
               case^S expr of
                 Var x       → case^S (env @^S x) of
                                  M v →  v
                                  P v →  inst v
                 Const n     → Num^S (lift n)
                 Lam x e     → Fun^S (λ^D v →
                                        let^S env' = bind @^S x @^S v @^S env
                                        in eval @^S env' @^S e)
                 App e1 e2   → case^S (eval @^S env @^S e1) of
                                  Fun f →  f @^D (eval @^S env @^S e2)
                 Let x e1 e2 → let^D v = P^S (polym eval @^S env @^S e1)
                                in let^S env' = bind @^S x @^S v @^S env
                                   in eval @^S env' @^S e2
               )
  in
  let^S meval = preeval @^S (λ^S x → M^S Wrong^S)
   in ⟨...⟩
```

Figure 11.3: An evaluator for lambda-calculus with let-bound polymorphism.

11.3 Generating Polymorphism

We can use the new annotation in the let-bound polymorphic version of the interpreter for lambda-calculus to produce polymorphic code. The resulting annotated program is given in Figure 11.3. Observe that every **poly** was replaced by a **polym**, and that every **spec** was replaced by an **inst**; in this way, the let-bound term will produce a polymorphic residual.

We can observe the effect of the new annotation in the term of Example 11.1.

EXAMPLE 11.4. After evidence elimination, the result of specializing the expression

```
meval @^S ( Let^S 'i' (Lam^S 'x' (Var^S 'x'))
               ( App^S (App^S (Var^S 'i') (Var^S 'i'))
                       (Const^S 0)
               )
          )
```

is the expression

$$\begin{array}{l}\textbf{let } v = \lambda v' \to v' \\ \ \textbf{in } (v\ v)\ 0 \\ \ ::\ \mathit{Num\ Int}\end{array}$$

where the function bound to v has type

$$\textbf{polym } (\forall t.\mathit{Fun}\ (t \to t))$$

We can see that the resulting residual code uses let-bound polymorphism, although the source code of *meval* was completely monomorphic. This shows that the inherited limit of polymorphism was removed for this example. However, the usage analysis may fall short when considering more complex terms. For that reason, this approach needs a deeper consideration to be able to state that the inherited limit of polymorphism has been removed. We have presented this basic treatment here to show the possibilities of our approach.

11.4 Improving Polyvariance

The information about the usage of evidence collected to permit the generation of polymorphism can be used also to have a better treatment of polyvariance. When evidence eliminating polyvariance, *every* different lower bound will generate a different element in the resulting tuple. But some of those elements may have the same residual term, and thus they not need to be different. Consider the following example.

EXAMPLE 11.5. Observe how the evidence corresponding to the first parameter is used, but that of the second is not.

$$\begin{array}{l}\vdash_{\mathrm{P}} \textbf{let}^{D}\ f = \textbf{poly}\ (\lambda^{D}x.\lambda^{D}y.(\textbf{lift}\ x, y)^{D}) \\ \quad \textbf{in}\ (\textbf{spec}\ f\ @^{D}\, 2^{S}\ @^{D}\, 4^{S}, \textbf{spec}\ f\ @^{D}\, 2^{S}\ @^{D}\, 8^{S}, \\ \qquad \textbf{spec}\ f\ @^{D}\, 3^{S}\ @^{D}\, 9^{S}, \textbf{spec}\ f\ @^{D}\, 3^{S}\ @^{D}\, 27^{S})^{D} \\ \quad : ((\mathit{Int}^{D}, \mathit{Int}^{S})^{D}, (\mathit{Int}^{D}, \mathit{Int}^{S})^{D}, (\mathit{Int}^{D}, \mathit{Int}^{S})^{D}, (\mathit{Int}^{D}, \mathit{Int}^{S})^{D})^{D} \\ \quad \hookrightarrow \textbf{let}\ f = \Lambda h_x, h_y.\lambda x.\lambda y.(h_x, y) \\ \qquad \textbf{in}\ (f\,((2,4))@\bullet @\bullet, f\,((2,8))@\bullet @\bullet, \\ \qquad\quad f\,((3,9))@\bullet @\bullet, f\,((3,27))@\bullet @\bullet) \\ \quad : ((\mathit{Int}, \hat{4}), (\mathit{Int}, \hat{8}), (\mathit{Int}, \hat{9}), (\mathit{Int}, \hat{27}))\end{array}$$

The type of the residual function f is

$$\tau_f = \textbf{poly}\ (\forall t_x, t_y.\mathrm{IsInt}\ t_x, \mathrm{IsInt}\ t_y \Rightarrow t_x \to t_y \to (\mathit{Int}, t_y))$$

and the types of the lower bounds are

$$\begin{array}{l}\mathrm{IsMG}\ \tau_f\ (\hat{2} \to \hat{4} \to (\mathit{Int}, \hat{4})) \\ \mathrm{IsMG}\ \tau_f\ (\hat{2} \to \hat{8} \to (\mathit{Int}, \hat{8})) \\ \mathrm{IsMG}\ \tau_f\ (\hat{3} \to \hat{9} \to (\mathit{Int}, \hat{9})) \\ \mathrm{IsMG}\ \tau_f\ (\hat{3} \to \hat{27} \to (\mathit{Int}, \hat{27}))\end{array}$$

Observe that the four are different.

When evidence elimination is performed, four different lower bounds will generate a four-tuple, that is

$$\begin{array}{l}\textbf{let } f = (\lambda x.\lambda y.(2,y), \lambda x.\lambda y.(2,y), \lambda x.\lambda y.(3,y), \lambda x.\lambda y.(3,y)) \\ \;\textbf{in } (\pi_{1,4}\, f@\bullet @\bullet, \pi_{2,4}\, f@\bullet @\bullet, \\ \qquad \pi_{3,4}\, f@\bullet @\bullet, \pi_{4,4}\, f@\bullet @\bullet)\end{array}$$

But if we take the usage information into account, the type of the function f is

$$\tau_f = \textbf{poly } (\forall t_x, t_y.\text{IsInt}^U\ t_x, \text{IsInt}^N\ t_y \Rightarrow t_x \to t_y \to (Int, t_y))$$

which means that the evidence for t_y is not used in the body of the function. Thus, instead of having one element for each different lower bound, we may allow one element for each different evidence marked as used. This will result in the following term

$$\begin{array}{l}\textbf{let } f = (\lambda x.\lambda y.(2,y), \lambda x.\lambda y.(3,y)) \\ \;\textbf{in } (\textbf{fst } f@\bullet @\bullet, \textbf{fst } f@\bullet @\bullet, \\ \qquad \textbf{snd } f@\bullet @\bullet, \textbf{snd } f@\bullet @\bullet)\end{array}$$

The formulation of type specialization using predicates and evidence has been shown to be very useful. We believe that it can be used also to improve the treatment of unresolved issues in type specialization, such as the interaction between polyvariance and dynamic recursion.

Part IV

Other Issues

Chapter 12

About Jones' Optimality

...it has the words DON'T PANIC inscribed in large friendly letters on its cover.

The Hitch Hicker's Guide to the Galaxy
Douglas Adams

In Chapter 2 we have described the notion of compilation by specialization, and we have presented the inherited limit of types based on the compilation using typed interpreters. We also have said that several solutions have been proposed to this problem, including Type Specialization.

However, as we were working in this thesis, we have discovered that indeed the problem of the inherited limit of types is not really there: it takes only a simple representation shift to show that ordinary partial evaluation is Jones-optimal. The representation shift amounts to read the type tags as constructors for higher-order abstract syntax. We substantiate our observation by considering a typed self-interpreter whose input abstract syntax is higher-order — specializing it with respect to a source program yields a residual program that is textually identical to the source program, modulo renaming.

This new way to look at the problem of the inherited limit of types was discovered in joint work with Olivier Danvy, and published by Danvy and Martínez López [2003]. For that reason this chapter departs slightly from the rest of the work — e.g. the examples are self-contained, and developed from scratch — in presenting the ideas that allows us to say that for more than ten years the partial evaluation community has produced a number of really interesting developments while trying to solve a problem that was not really there!

12.1 The Problem

Partial evaluation can be used to compile a program having only an interpreter of a language: by specializing the interpreter to an object program, we can obtain a program written in another language that performs the same task — this scenario was described in Section 2.3. This approach can be used to measure, in some way, the power of the specialization method: if there exists a self-interpreter that, when compiling a program by specialization, is able to obtain essentially the same program (or even a better one), then we can be sure that there is no feature of the interpreter that imposes a limit on the form of residual programs. This phenomenon is referred in the literature by the term *Jones-optimality*, and can be expressed with the following formula:

$[\![\texttt{mix}]\!]_L$ `self-interpreter program` $=_\alpha$ `program`

A partial evaluator such as lambda-Mix, for example, is Jones optimal [Gomard and Jones, 1991; Jones *et al.*, 1993].

A typed interpreter, however, requires a universal data type to represent expressible values, and specializing it with an ordinary partial evaluator yields a residual program with many tag and untag operations. Ordinary, Mix-style, partial evaluation is thus not Jones optimal [Jones, 1988b].

Obtaining Jones optimality for typed interpreters has proved a source of inspiration for a number of new forays into partial evaluation [Makholm, 2000], e.g., constructor specialization [Dussart *et al.*, 1995; ?], type specialization [Dussart *et al.*, 1997b; Hughes, 1996b; Hughes, 1996a; Hughes, 1998a; Hughes, 2000; Martínez López and Hughes, 2002], coercions [Danvy, 1998a], and more recently tag elimination [Taha *et al.*, 2001] and staged tagless interpreters [Pasalic *et al.*, 2002].

To state the problem in full form, we present a self-interpreter for a lambda-calculus with sum-types that is able to represent the universal data type needed to express values, and a generating extension for it. This interpreter is the minimal one able to perform self-interpretation — we present it completely to show what it takes to have sum-types and self-interpretation, and to provide enough information to assess that it is indeed a self-interpreter. The example has been programmed in Haskell, including a type `FOLam` to represent object programs, an evaluation function `eval`, and a generating extension `geval`.

The definition of the language is as follows:

```
data Lam = Var String | Lam (String, Lam) | App (Lam, Lam)
         | Let ((String, Lam), Lam)| Fix Lam
         | Pair (Lam,Lam) | Fst Lam | Snd Lam
         | Con (String, Lam) | Case (Lam, [((String,String),Lam)])
         | Unit ()
         | Num Int
         | Bool Bool | If (Lam,(Lam,Lam))
         | Str String
         | Prim (String, Lam)

data Val = F (Val -> Val)
         | P (Val, Val) | C (String, Val) | U ()
         | N Int        | B Bool          | S String
```

where `Val` is the type of computed values, and `Lam`, the type of expressions in the language. The language has primitive functions, where we consider basic arithmetic operations (here only `(+)`, for simplicity), equality, and a function `error` to provide failure.

The evaluation of closed expressions of the language represented by elements of `Lam` is realized by the function `eval` from `Lam` into `Val`. To define `eval`, we have to use another function, `preeval`, that evaluates any expression, even those with free variables — the usual technique of environments is applied.

```
type Env = (String -> Val)
```

```
bind :: String -> Val -> Env -> Env
bind x v env = \y -> if x==y then v else env y

env0 x = error (x ++ " not bound!")
eval te = preeval te env0

-- This function is used to perform the search of the right branch
lookForC :: String -> [((String,String),Lam)]
             -> ((String,Lam) -> d) -> d
lookForC c [] _ = error ("Non-complete case: " ++ c ++ " not found.")
lookForC c (cxe:bs) k = if c==fst (fst cxe)
                         then k (snd (fst cxe),snd cxe)
                         else lookForC c bs k

preeval :: Lam -> Env -> Val
preeval te env =
  case te of
    Var x    -> env x
    Lam xe   -> F (\v -> let env2 = bind (fst xe) v env
                          in preeval (snd xe) env2)
    App e12  -> appVal (preeval (fst e12) env) (preeval (snd e12) env)
    Let xe21 -> let v = preeval (snd (fst xe21)) env
                 in let env2 = bind (fst (fst xe21)) v env
                     in preeval (snd xe21) env2
    Fix e    -> fixVal (preeval e env)

    Pair e12 -> P (preeval (fst e12) env, preeval (snd e12) env)
    Fst  e   -> fstVal (preeval e env)
    Snd  e   -> sndVal (preeval e env)

    Con ce   -> C (fst ce, preeval (snd ce) env)
    Case ebs -> case preeval (fst ebs) env of
                 C cv -> lookForC (fst cv) (snd ebs)
                          (\xei ->
                            let env2 = bind (fst xei) (snd cv) env
                             in preeval (snd xei) env2)
    Unit u   -> U u
    Num n    -> N n
    Bool b   -> B b
    If  be12 -> ifVal (preeval (fst be12) env)
                      (preeval (fst (snd be12)) env)
                      (preeval (snd (snd be12)) env)
    Str s    -> S s
    Prim ope -> runPrimitive (fst ope) (preeval (snd ope) env)
    _        -> error "Unexpected Lam constructor!"
```

This function uses several auxiliary functions that can easily be defined. For example, the function `appVal` that performs the application of a function to its parameter is defined as

```
appVal :: Val -> Val -> Val
appVal v x = case v of
               F f -> f x
               _   -> error "appVal"
```

the function `fixVal` that computes fixpoints (that uses function `appVal`), as

```
fixVal :: Val -> Val
fixVal vf =
 case vf of
   F f -> F (\v -> appVal (f (F (\v2 -> appVal (fixVal vf) v2)))) v)
   _   -> error "fixVal"
```

the function `runPrimitive`, that executes a primitive function, as

```
runPrimitive :: String -> Val -> Val
runPrimitive p v =
  if p == "+"
   then case v of
         P v12 -> opNumVal (+) (fst v12) (snd v12)
         _     -> error ("Bad arguments to primitive (+)!")
  else if p == "=="
   then case v of
         P v12 -> eqVal (fst v12) (snd v12)
         _     -> error ("Bad arguments to primitive (==)!")
  else if p == "error"
   then case v of
         S msg -> error msg
         _     -> error ("Bad arguments to primitive error!")
   else error ("Unknown primitive: " ++ p)
```

(and uses two other auxiliaries, `opNumVal` and `eqVal`)

```
opNumVal :: (Int -> Int -> Int) -> Val -> Val -> Val
opNumVal op (Val v1) (Val v2) =
  case v1 of
   N n -> case v2 of
           N m -> N (op n m)
           _   -> error "opNumVal"
   _   -> error "opNumVal"
opNumVal _ _ _ = error "opNumVal"

eqVal :: Val -> Val -> Val
eqVal (Val v1) (Val v2) =
```

```
    case (v1,v2) of
     (P v12 , P v12')   -> case eqVal (fst v12) (fst v12') of
                             B b -> if b
                                     then eqVal (snd v12) (snd v12')
                                     else B False
                             _   -> error "eqVal"
     (C cv  , C cv')    -> if fst cv == fst cv'
                             then eqVal (snd cv) (snd cv')
                             else B False
     (U u       , U u') -> B (u == u')
     (N n       , N m)  -> B (n == m)
     (B b       , B b') -> B (b == b')
     (S s       , S s') -> B (s == s')
     _   -> error "eqVal"
  eqVal _ _ = error "eqVal"
```

and the rest, as

```
  ifVal :: Val -> Val -> Val -> Val
  ifVal vb ve1 ve2 = case vb of
                       B b -> if b then ve1 else ve2
                       _   -> error "ifVal"

  fstVal, sndVal :: Val -> Val
  fstVal v = case v of
               P v12 -> case fst v12 of
                          Val v -> v
                          _     -> error "fstVal"
               _     -> error "fstVal"

  sndVal v = case v of
               P v12 -> case snd v12 of
                          Val v -> v
                          _     -> error "sndVal"
               _     -> error "sndVal"
```

The interpreter has been programmed in such a way that it is simple to see that it is indeed a self-interpreter. For example, the object program corresponding to the function `eval` can be defined as follows (assuming the names of object programs for the other functions):

```
  oeval =
   Let (("opNumVal", oopNumVal),
   Let (("eqVal", oeqVal),
   Let (("runPrimitive", orunPrimitive),
   Let (("appVal", oappVal),
   Let (("fixVal", ofixVal),
```

```
Let (("ifVal", oifVal),
Let (("fstVal", ofstVal)    , Let (("sndVal", osndVal),
Let (("lookForC", olookForC),
Let (("bind", obind)        , Let (("env0", oenv0),


Let (("eval", Fix (Lam ("preeval", Lam ("te", Lam ("env"
 Case (Var "te",
  [ (("Var", "x"), App (Var "env", Var "x"))
  , (("Lam", "xe")
     , Con ("F", Lam ("v"
               , Let (("env2", App (App (App
                         (Var "bind", Fst(Var "xe"))
                                    , Var "v")
                                    , Var "env"))
                     , App (App (Var "eval", Snd(Var "xe"))
                                            , Var "env2")))))
  , (("App", "e12")
     , App (App (Var "appVal"
          , App (App (Var "eval", Fst(Var "e12")), Var "env"))
          , App (App (Var "eval", Snd(Var "e12")), Var "env")))
  , (("Let", "xe21")
     , Let (("v", App (App
              (Var "eval", Snd (Fst (Var "xe21"))), Var "env"))
           , Let (("env2", App (App (App
                     (Var "bind", Fst(Fst(Var "xe21")))
                     , Var "v"), Var "env"))
                 , App (App (Var "eval", Snd(Var "xe21"))
                                        , Var "env2"))))
  , (("Fix", "e")
     , App (Var "fixVal"
           , App (App (Var "eval", Var "e"), Var "env")))

  , (("Pair", "e12")
     , Con ("P"
           , Pair ( App (App (Var "eval", Fst(Var "e12"))
                                         , Var "env")
                  , App (App (Var "eval", Snd(Var "e12"))
                                         , Var "env"))))
  , (("Fst",  "e")
     , App (Var "fstVal"
           , App (App (Var "eval", Var "e"), Var "env")))
  , (("Snd",  "e")
     , App (Var "sndVal"
           , App (App (Var "eval", Var "e"), Var "env")))
```

```
, (("Con", "ce")
   , Con ("C", Pair ( Fst(Var "ce")
                    , App (App (Var "eval", Snd(Var"ce"))
                                          , Var "env"))))

, (("Case", "ebs")
   , Case (App (App (Var "eval", Fst(Var "ebs")), Var "env")
          , [ (("C", "cv")
              , App (App (App
                    (Var "lookForC"
                      , Fst(Var "cv"))
                      , Snd(Var "ebs"))
                      , Lam ("xei",
                          Let (("env2", App (App (App (Var "bind"
                                             , Fst(Var "xei"))
                                             , Snd(Var "cv"))
                                             , Var "env"))
                              , App (App (Var "eval"
                                  , Snd(Var "xei"))
                                  , Var "env2")))))
            ]))

, (("Unit", "u"), Con ("U", Var "u"))
, (("Num", "n"), Con ("N", Var "n"))
, (("Bool", "b"), Con ("B", Var "b"))
, (("If", "be12")
   , App (App (App (
       Var "ifVal",
        App (App (Var "eval", Fst(Var "be12")), Var "env")),
        App (App (Var "eval", Fst(Snd(Var "be12"))), Var "env")),
        App (App (Var "eval", Snd(Snd(Var "be12"))), Var "env")))
, (("Str", "s"), Con ("S", Var "s"))
, (("Prim", "ope")
   , App (App (Var "runPrimitive", Fst(Var "ope")),
               App (App (Var "eval", Snd(Var "ope")), Var "env")))
])))))),
Lam ("te"
    , App (App (Var "eval", Var "te"), Var "env0")))))))))))))
```

Observe how constructors of type `Lam` are used to represent program expressions, and how strings are used to represent variables and program constructors. All the auxiliary functions can be defined in the same way.

This shows that our language `Lam` is indeed a self-interpreter. The next step is to define a generating extension for the interpreter — that is, a program that given a representation of an object program, returns the residual of specializing the interpreter to it. To construct it, we need a representation for residual programs, and a function

mapping object programs into residual ones. The representation for residual programs and values is given by the types `GLam` and `GVal` respectively:

```
data GLam = RES_Val GVal
          | RES_App (GLam, GLam)
          | RES_Let (GLam, GLam -> GLam)
          | RES_Fix GLam
          | RES_Case (GLam, [(String,GLam -> GLam)])
          | RES_Fst GLam
          | RES_Snd GLam
          | RES_Eq (GLam,GLam)
          | RES_If (GLam,(GLam,GLam))
          | RES_Prim (String, GLam)
data GVal = RES_F (GLam -> GLam)
          | RES_C (String, GLam)
          | RES_P (GLam,GLam)
          | RES_U ()
          | RES_N Int
          | RES_B Bool
          | RES_S String
```

The function mapping object programs to residual code is called `geval`, and is defined as follows.

```
geval e = gpreeval e genv0

gpreeval :: Lam -> GEnv -> GLam
gpreeval (Var x)          env = env x
gpreeval (Lam (x,e))      env = res_valF
                                 (\v -> let env2 = gbind x v env
                                         in gpreeval e env2)
gpreeval (App (e1,e2))    env = RES_App (gpreeval e1 env
                                        ,gpreeval e2 env)
gpreeval (Let ((x,e2),e1)) env = RES_Let
                                 (gpreeval e2 env
                                 ,\v -> let env2 = gbind x v env
                                         in gpreeval e1 env2)
gpreeval (Fix e)          env = RES_Fix (gpreeval e env)
gpreeval (Unit u)         _   = res_valU u
gpreeval (Con (c,e))      env = res_valC (c, gpreeval e env)
gpreeval (Case (e,bs))    env = RES_Case
                                 (gpreeval e env
                                 ,map
                                   (\((ci,xi),ei) ->
                                     (ci
                                     ,\v -> let env2 = gbind xi v env
                                             in gpreeval ei env2
```

```
                                         )) bs
                                     )
gpreeval (Pair (e1,e2))    env = res_valP (gpreeval e1 env
                                          ,gpreeval e2 env)
gpreeval (Fst  e)          env = RES_Fst (gpreeval e env)
gpreeval (Snd  e)          env = RES_Snd (gpreeval e env)
gpreeval (Num n)           _   = res_valN n
gpreeval (Bool b)          _   = res_valB b
gpreeval (Eq (e1,e2))      env = RES_Eq (gpreeval e1 env
                                        ,gpreeval e2 env)
gpreeval (If (b,(e1,e2)))  env = RES_If (gpreeval b env
                                        ,(gpreeval e1 env
                                         ,gpreeval e2 env))
gpreeval (Str s)           _   = res_valS s
gpreeval (Prim (op,e))     env = RES_Prim (op, gpreeval e env)
```

where environments are defined as

```
type GEnv = (String -> GLam)

gbind :: String -> GLam -> GEnv -> GEnv
gbind x v env = \y -> if x==y then v else env y

genv0 :: GEnv
genv0 _ = error "Variable not found"
```

and the functions `res_val`X are defined as

```
res_valF = RES_Val . RES_F
res_valC = RES_Val . RES_C
res_valP = RES_Val . RES_P
res_valU = RES_Val . RES_U
res_valN = RES_Val . RES_N
res_valB = RES_Val . RES_B
res_valS = RES_Val . RES_S
```

With all this elements, now we are ready to state the problem: the specialization of the interpreter `eval` to a term is not optimal.

For example, if we consider the specialization of the interpreter `eval` to the term given by `"\x.x x"`, we obtain

```
RES_Val (RES_F (\v -> RES_App (v, v)))
```

and this term is not α-equivalent to the original one.

For a bit bigger example we may consider the specialization of the interpreter to the code of the function `appVal`, that is:

```
"\v -> \x ->
  case v of
    F f -> f x"
```

whose result is

```
RES_F (\v -> RES_F (\x ->
       RES_Case (v,
                 [ ("F",\f -> RES_App (f,x))]
                )
      ))
```

Again we can see that the specialization is not α-equivalent to the source code.

One more example, involving primitive functions, is given by the specialization of the function `"runPrimitive"` (we assume defined the operations `"opNumVal"` and `"eqVal"`):

```
"\p -> \v ->
   if p == "+"
    then case v of
           P v12 -> opNumVal (+) (fst v12) (snd v12)
    else if p == "=="
    then case v of
           P v12 -> eqVal (fst v12) (snd v12)
    else if p == "error"
    then case v of
           S msg -> error msg
    else error ("Unknown primitive: " ++ p)"
```

whose result is

```
RES_F (\p -> RES_F (\v ->
   RES_If (RES_Eq (p,RES_S "+"),
      (RES_Case (v,
          [ ("P",\v12 ->
               RES_App (RES_App (RES_App (
                        opNumVal
                        ,RES_F (\x -> RES_F (\y ->
                                  RES_Prim ("+",RES_P (x, y)))))
                        ,RES_Fst v12)
                        ,RES_Snd v12)
           )]),
   RES_If (RES_Eq (p,RES_S "=="),
      (RES_Case (v,
          [ ("P",\v12 ->
               RES_App (RES_App (
                        eqVal
                       ,RES_Fst v12)
                       ,RES_Snd v12)
           )]),
   RES_If (RES_Eq (p,RES_S "error"),
      (RES_Case (v,
```

```
          [ ("S",\msg ->
               RES_Prim ("error",msg)
            )]),
    RES_Prim ("error",RES_S "Unknown primitive!"))
  )))))))))))
```

One more time, the specialization is not producing an α-equivalent version.

Thus, we cannot say that our self-interpreter is Jones-optimal.

12.2 Regaining Jones Optimality

To show that we can have Jones optimality with partial evaluation, we rewrite the self-interpreter, using higher-order abstract syntax as introduced by Pfenning and Elliott [1988] and used by Thiemann, Thiemann [1999a, 1999b]. We also provide a generating extension for this coding of the language. The example has been programmed in Haskell; the definition is as follows:

```
data HOVal = F (HOLam -> HOLam)             -- Functions
           | P (HOLam,HOLam)                -- Pairs
           | C (String, HOLam)              -- Tagged expressions
           | U ()                           -- Unit value
           | N Int                          -- Numbers
           | B Bool                         -- Booleans
           | S String                       -- Strings
data HOLam = Val HOVal                      -- Value injection
           | App (HOLam, HOLam)             -- Functions
           | Let (HOLam, HOLam -> HOLam)    -- Let expression
           | Fix HOLam                      -- Recursion
           | Case (HOLam,
               [(String, HOLam -> HOLam)])  -- Tagged selection
           | Fst HOLam                      -- Pair operations
           | Snd HOLam
           | If (HOLam, (HOLam, HOLam))     -- Boolean operations
           | Prim (String, HOLam)           -- Primitive functions

valF = Val . F    -- Functions
valP = Val . P    -- Pairs
valC = Val . C    -- Tagged expressions
valU = Val . U    -- Unit value
valN = Val . N    -- Numbers
valB = Val . B    -- Booleans
valS = Val . S    -- Strings
```

where `HOVal` is the type of values of the language, and `HOLam`, the type of other expressions. The functions `valF`, `valU`, etc. are used to cast values into expressions (the intended meaning of each of the tags is added as a comment in the code.) The language

has primitive functions, where we consider basic arithmetic operations (here only `(+)`, for simplicity), equality, and a function `error` to provide failure.

The evaluation of the language represented by `HOLam` expressions is performed by a function from `HOLam` into `HOVal`,

```
lookForC :: String -> [(String, HOLam -> HOLam)]
            -> ((HOLam -> HOLam) -> d) -> d
lookForC c [] _ = error ("Non-complete case: " ++ c ++ " not found.")
lookForC c (ce:bs) k = if c==fst ce
                        then k (snd ce)
                        else lookForC c bs k
eval :: HOLam -> HOVal
eval te = case te of
  Val x    -> case x of
               P e12 -> P (Val (eval (fst e12)), Val (eval (snd e12)))
               C ce  -> C (fst ce, Val (eval (snd ce)))
               _     -> x

  App e12  -> eval $ appVal (eval (fst e12)) (Val (eval (snd e12)))
  Let e21  -> let v = Val (eval (fst e21))
               in eval ((snd e21) v)
  Fix e    -> fixVal eval (eval e)

  Case ebs -> case eval (fst ebs) of
               C cv -> lookForC (fst cv) (snd ebs)
                        (\ei -> eval (ei (snd cv)))

  Fst  e   -> fstVal (eval e)
  Snd  e   -> sndVal (eval e)

  If  be12 -> ifVal (eval (fst be12))
                    (eval (fst (snd be12)))
                    (eval (snd (snd be12)))

  Prim ope -> runPrimitive (fst ope) (eval (snd ope))
  _        -> error "Unexpected HOLam constructor!"
```

This function uses auxiliary functions that are similar to those used by the first order version; functions `appVal`, `runPrimitive`, `opNumVal`, `eqVal`, `ifVal`, `fstVal` and `sndVal` are defined in exactly the same way, but with different types. Function `fixVal` that computes fixpoints using function `appVal` has to be redefined slightly, as

```
fixVal :: (HOLam -> HOVal) -> HOVal -> HOVal
fixVal ev vf =
 case vf of
   F f -> F (\v -> appVal
                    (ev (f (valF (\v2 -> appVal (fixVal ev vf) v2)))
```

```
                      v)
    _    -> error "fixVal"
```

The object version of the interpreter, showing that it is indeed a self-interpreter, is defined as follows:

```
oeval =
 Let (oopNumVal,                    \opNumVal     ->
 Let (oeqVal,                       \eqVal        ->
 Let (orunPrimitive eqVal opNumVal, \runPrimitive ->
 Let (oappVal,                      \appVal       ->
 Let (ofixVal appVal,               \fixVal       ->
 Let (oifVal,                       \ifVal        ->
 Let (ofstVal,                      \fstVal       ->
 Let (osndVal,                      \sndVal       ->
 Let (olookForC,                    \lookForC     ->
 Fix (valF (\eval -> valF (\te ->
  Case (te,
   [ ("Val", \x ->
       Case (x,
        [ ("P",\e12 ->
            valC ("P", valP( valC ("Val", App (eval, Fst e12))
                           , valC ("Val", App (eval, Snd e12)))))
        , ("C",\ce ->
            valC ("C", valP ( Fst ce
                            , valC ("Val", App (eval, Snd ce)))))
        , ("U",\_ -> x)
        , ("N",\_ -> x)
        , ("B",\_ -> x)
        , ("S",\_ -> x)
        , ("F",\_ -> x)
        ]))
   , ("App", \e12 ->
       App (eval, App (App (appVal
                            , App (eval, Fst e12))
                            , valC ("Val", App (eval, Snd e12)))))
   , ("Let", \e21 ->
       Let (valC ("Val", App (eval, Fst e21)), \v ->
             App (eval, App (Snd e21, v))))
   , ("Fix", \e -> App (App (fixVal, eval), App (eval, e)))
   , ("Case", \ebs ->
        Case (App (eval, Fst ebs)
             , [ ("C", \cv ->
                  App (App (App
                      (lookForC
                        , Fst cv)
```

```
                        , Snd ebs)
                        , valF (\xei ->
                                App (eval, App (xei, Snd cv)))))
               ]))
    , ("Fst", \e -> App (fstVal, App (eval, e)))
    , ("Snd", \e -> App (sndVal, App (eval, e)))
    , ("If", \be12 -> App (App (App (
                         ifVal, App (eval, Fst be12))
                           , App (eval, Fst(Snd be12)))
                           , App (eval, Snd(Snd be12))))
    , ("Prim", \ope ->
        App (App (runPrimitive, Fst ope), App (eval, Snd ope)))
    ]))))))))))))
```

Observe how constructors of type `HOLam` are used (as well as the functions `valF`, etc.) to represent program expressions, and how strings are used to represent program constructors. Notice as well how higher order syntax allows the use of ordinary variables to represent let-bounded expressions, and how `runPrimitive` and `fixVal` take parameters to represent the use of other functions (appearing in the original code as free variables). All the auxiliary functions can be defined in the same way.

The generating extension for this language is extremely easy to define (we reuse the types `GLam` and `GVal`, and the functions `res_val`X):

```
geval :: HOLam -> GLam
geval e = holam2glam e

holam2glam :: HOLam -> GLam
holam2glam (Val x)           = RES_Val (hoval2gval x)
holam2glam (App (e1,e2))     = RES_App (holam2glam e1, holam2glam e2)
holam2glam (Let (e2,e1))     = RES_Let (holam2glam e2
                                        ,\ge -> holam2glam
                                                (e1 (glam2holam ge)))
holam2glam (Fix e)           = RES_Fix holam2glam (holam2glam e)
holam2glam (Case (e,bs))     = RES_Case
                                  (holam2glam e
                                  ,map (\(ci,fei) ->
                                          (ci,\gei ->
                                                holam2glam
                                                  (fei (glam2holam gei))
                                          )) bs)
holam2glam (Fst  e)          = RES_Fst (holam2glam e)
holam2glam (Snd  e)          = RES_Snd (holam2glam e)
holam2glam (Eq (e1,e2))      = RES_Eq (holam2glam e1, holam2glam e2)
holam2glam (If (b,(e1,e2)))  = RES_If (holam2glam b, (holam2glam e1
                                                      ,holam2glam e2))
holam2glam (Prim (op,e))     = RES_Prim (op, holam2glam e)
```

This function uses another three auxiliary ones, representing the 'inverse' function `glam2holam`, from `GLam` to `HOLam`, and the corresponding conversion functions for values, `hoval2gval` and `gval2hoval`.

```
glam2holam :: GLam -> HOLam
glam2holam (RES_Val gv)            = Val (gval2hoval gv)
glam2holam (RES_App (ge1,ge2))     = App (glam2holam ge1
                                         ,glam2holam ge2)
glam2holam (RES_Let (ge,gfe))      = Let (glam2holam ge
                                         ,\v -> glam2holam
                                                 (gfe (holam2glam v))
                                         )
glam2holam (RES_Fix _ ge)          = Fix (glam2holam ge)
glam2holam (RES_Case (ge,gbs))     = Case
                                      (glam2holam ge
                                      ,map (\(ci,fgei) ->
                                             (ci,\ei ->
                                               glam2holam
                                                (fgei (holam2glam ei))
                                             )
                                           ) gbs
                                      )
glam2holam (RES_Fst ge)            = Fst (glam2holam ge)
glam2holam (RES_Snd ge)            = Snd (glam2holam ge)
glam2holam (RES_Eq (ge1,ge2))      = Eq (glam2holam ge1
                                        ,glam2holam ge2)
glam2holam (RES_If (ge,(ge1,ge2))) = If (glam2holam ge
                                        , (glam2holam ge1
                                          ,glam2holam ge2))
glam2holam (RES_Prim (op,ge))      = Prim (op, glam2holam ge)

-- gval2hoval
gval2hoval :: GVal -> HOVal
gval2hoval (RES_F fg)        = F (\e -> glam2holam
                                         (fg (holam2glam e)))
gval2hoval (RES_C (c, ge))   = C (c, glam2holam ge)
gval2hoval (RES_P (ge1,ge2)) = P (glam2holam ge1, glam2holam ge2)
gval2hoval (RES_U u)         = U u
gval2hoval (RES_N n)         = N n
gval2hoval (RES_B b)         = B b
gval2hoval (RES_S s)         = S s

-- hoval2gval
hoval2gval :: HOVal -> GVal
hoval2gval (F f)       = RES_F (\gv -> holam2glam (f (glam2holam gv)))
hoval2gval (C (c,e))   = RES_C (c, holam2glam e)
```

```
hoval2gval (P (e1,e2)) = RES_P (holam2glam e1, holam2glam e2)
hoval2gval (U u)       = RES_U u
hoval2gval (N n)       = RES_N n
hoval2gval (B b)       = RES_B b
hoval2gval (S s)       = RES_S s
```

This finishes the presentation of the generating extension for this language.

Considering again the specialization of the interpreter `eval` for the codification using higher order syntax (i.e. the function `geval`) to the term given by `"\x.x x"` (that is, `geval (Val (F (\x -> App (x, x)))))`, we obtain

```
RES_Val (RES_F (\v -> RES_App (v, v)))
```

being very easy to verify that it is indeed α-equivalent to the coded version of `"\x.x x"`.

Before ending this chapter, let's consider the higher order versions of the examples ending Section 12.1, to see the specialization of programs using sum-types and primitives.

First we consider the case of the function `appVal`, `\v.\x.case v of F f -> f x"`. The higher order encoding for this function is

```
Val (F (\v -> Val (F (\x ->
   Case (v,
         [("F",\f -> App (f,x))]
        )))))
```

When the generating extension `geval` is applied to it, the result is

```
RES_Val (RES_F (\x_0 -> RES_Val (RES_F (\x_1 ->
   RES_Case (x_0,
             [ ("F",\x_2 -> RES_App (x_2,x_1))]
            )))))
```

which is α-equivalent to the encoding.

The last example is the function `runPrimitive`. Its encoding, assuming `opNumVal` and `eqVal` already defined, is

```
valF (\p -> valF (\v ->
      If (Eq (p, valS "+")
         ,( Case (v,
              [ ("P", \v12 ->
                  App (App (App
                   (opNumVal, valF (\x -> valF (\y ->
                                  Prim ("+", valP (x, y))
                                 ))),
                    Fst v12), Snd v12)
                )
              ])
         ,
      If (Eq (p, valS "==")
```

```
          ,( Case (v,
              [ ("P", \v12 ->
                  App (App (eqVal, Fst v12), Snd v12)
                )
              ])
          ,
        If (Eq (p, valS "error")
          ,( Case (v,
              [ ("S", \msg ->
                  Prim ("error", msg)
                )
              ])
           , Prim ("error", valS "Unknown primitive!")
           ))
          ))))))))
     ))
```

and the resulting specialization is

```
RES_Val (RES_F (\x_0 -> RES_Val (RES_F (\x_1 ->
  RES_If (RES_Eq (x_0,RES_Val (RES_S "+")),
      (RES_Case (x_1,
          [("P",\x_2 ->
              RES_App (RES_App (RES_App (
                 eqVal
                ,RES_Val (RES_F (\x_3 -> RES_Val (RES_F (\x_4 ->
                    RES_Prim ("+",RES_Val (RES_P (x_3, x_4))))))))
                ,RES_Fst x_2)
                ,RES_Snd x_2)
          )]),
  RES_If (RES_Eq (x_0,RES_Val (RES_S "==")),
      (RES_Case (x_1,
          [("P",\x_11 ->
              RES_App (RES_App (
                 opNumVal
                ,RES_Fst x_11)
                ,RES_Snd x_11)
          )]),
  RES_If (RES_Eq (x_0,RES_Val (RES_S "error")),
      (RES_Case (x_1,
          [("S",\x_12 -> RES_Prim ("error",x_12))])
      ,RES_Prim ("error",RES_Val (RES_S "Unknown primitive!"))
      ))))))))))
```

One more time, both versions are α-equivalent.

Thus, if we consider this new way of reading terms, we are allowed to say that our self-interpreter is Jones-optimal.

12.3 Conclusions

Our simple observation complements Danvy [1998a]'s take on coercions for Jones optimality. In hindsight, a similar reading is implicit for Taha *et al.* [2001]'s E function. More than that we cannot say, e.g., about the various other solutions to Jones optimality, or about the frameworks it has inspired.

The generating extension of a lambda-interpreter provides an encoding of a lambda-term into the term model of the meta language of this interpreter. For an untyped self-interpreter, the translation is the identity transformation. For an untyped interpreter in continuation-passing style (CPS), the translation is the untyped CPS transformation. For an untyped interpreter in state-passing style (SPS), the translation is the untyped SPS transformation. And for an untyped interpreter in monadic style, the translation is the untyped monadic-style transformation.

In that light, what we have done here is to identify a similar reading for a typed self-interpreter, identifying its domain of universal values as a representation of higher-order abstract syntax. With this reading, type tags are not a bug but a feature and ordinary partial evaluation is Jones optimal. In particular, for a typed interpreter in CPS, the translation is the typed CPS transformation into higher-order abstract syntax, and similarly for state-passing style, etc., without extraneous type tags but with higher-order abstract syntax.

Chapter 13

Related Work

> *Will considered what to do. When you choose one way out of many, all the ways you don't take are snuffed out like candles, as if they'd never existed. At the moment, all Will's choices existed at once. But to keep them all in existence meant doing nothing. He had to choose, after all.*
>
> The Amber Spyglass
> Philip Pullman

There are many different approaches to program specialization, and each one has different expressive power, and different features. Some of them also solve the optimal specialization of typed interpreters, by alternative ways to the specialization of types considered in this thesis. In this chapter we describe alternative approaches to program specialization, and compare them to our approach, to place type specialization in context with them.

We consider in first place, the approach of partial evaluation, because it is the most popular and well-known (Section 13.1); secondly, we consider other methods that use type information to guide the specialization (Section 13.2); and at the end (Section 13.3) we conclude the section with methods that do not fit in any of the previous categories.

13.1 Partial Evaluation

Partial evaluation [Jones *et al.*, 1993; Consel and Danvy, 1993; Mogensen, 1998b] is a technique that produces the residual programs by using a generalized form of reduction: subexpressions with known arguments are replaced by the result of their evaluation, and combined with those computations that cannot be performed. Different techniques to combine the residual code and the calculated values for static code lead to different kinds of partial evaluation, with different features; polyvariance [Bulyonkov, 1984; Bulyonkov, 1988], constructor specialization [?], partially static data structures [Mogensen, 1988], etc. are just some of those.

The main difference between partial evaluation and type specialization is in the treatment of types. Assuming that the source language has the subject reduction property (i.e. reduction preserves types), the overall type of the residual program obtained by a partial evaluation technique will be exactly the same as the type of the source one. Thus, although the types of subexpressions can change when performing the specialization, arbitrary types cannot be created by partial evaluation, and then it constitutes an inherited limit [Mogensen, 1996], which prevents the optimal specialization of typed interpreters. This problem was stated by Neil Jones in 1987 as one of the open problems in the partial evaluation field [Jones, 1988b]. On the other hand, type specialization is

designed so as to produce both a residual term and a residual type, allowing the production of arbitrary types from a given source program, thus the inherited limit of types have been removed. This allows type specialization to obtain optimal specialization for typed interpreters.

Another, more technical, difference between both approaches is that of annotations. Because of the nature of partial evaluation (specialization by evaluation), a dynamic function cannot have static arguments: the only way in which the actual parameter may be known in the body of the function is by reduction of the function itself, which cannot happen because it is dynamic! This means that the annotation in this example

$$\lambda^D x.x +^S 1$$

is not valid for partial evaluation because there is no way to determine the value of x to perform the static addition. Continuation passing style (CPS for short) is used to improve the binding time annotations, removing some of the restrictions imposed on them. By using continuations, some static constructions can be specialized in a dynamic context. For example, when specializing a dynamic `if-then-else` its continuation can be moved on each branch, thus allowing more constructions to be declared static. Consider the function

$$\lambda^D b.1 +^S \mathbf{if}^D\, b \;\mathbf{then}\; 2 \;\mathbf{else}\; 3$$

Using a direct style partial evaluator makes the annotation incorrect, because the second operand of the sum depends on dynamic computations, and so it cannot be known — no improvement is possible. But a CPS evaluator can move the $1 +^S []$ context on each branch, thus obtaining

$$\lambda^D b.\mathbf{if}^D\, b \;\mathbf{then}\; 3 \;\mathbf{else}\; 4$$

However, there are improvements that cannot be achieved by a CPS partial evaluator: the assumption is that the contexts to be moved are themselves static, but that is not always the case. For example, in the following case, the annotation is invalid even for a CPS partial evaluator.

EXAMPLE 13.1. Observe that the result of the recursive function f contains a static part, but it appears under a dynamic recursion, and thus, under a potentially infinite number of distinct dynamic contexts.

$$\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{let}^S\, f = \mathbf{fix}^D\, (\lambda^D f.\lambda^D n. \\
\qquad\qquad\qquad \mathbf{if}^D\, n ==^D 0^D \\
\qquad\qquad\qquad \mathbf{then}\; (1^S, 2^D)^D \\
\qquad\qquad\qquad \mathbf{else}\; \mathbf{let}^D\; p = f \mathbin{@^D} (n -^D 1^D) \\
\qquad\qquad\qquad\qquad \mathbf{in}\; (\mathbf{fst}^D\, p, \mathbf{snd}^D\, p *^D \mathbf{snd}^D\, p)^D) \\
\quad \mathbf{in}\; \lambda^D n.\mathbf{lift}\; (\mathbf{fst}^D\, (f \mathbin{@^D} n)) \\
\quad : Int^D \to^D Int^D \\
\quad \hookrightarrow \lambda n.1 : Int \to Int
\end{array}$$

Restrictions in annotations imply the existence of a "best" way to annotate a program that can be computed automatically by means of an analysis called Binding Time Analysis (or BTA) [Jones *et al.*, 1993; Glenstrup and Jones, 1996]. In type specialization, though, we have shown that annotations are not restricted (except by a kind of

'correctness' formation), and thus, there does not exist anything such as a best annotation; for that reason, we have made annotations part of the input. This issue has been misunderstood in the past, with several people asking "why worry about this contrived annotation, while this other one will do?": because by deciding the annotations, we decide which specializations are allowed and which not; indeed, the only difference between an interpreter for lambda-calculus in untyped and simply typed versions is the way it is annotated — see Chapter 4. As we have shown in this work, type specialization has features that cannot be obtained by any partial evaluator; for example, type checking of an object program by specialization.

Inspired by type specialization, Peter Thiemann has proposed a partial evaluator with first-class polyvariance and co-arity raising [Thiemann, 2000a]. He shows that these two features are enough to have optimal specialization of typed interpreters. He has also shown that by writing a typed interpreter using dependent types, optimality can also be achieved [Thiemann, 1999c].

13.1.1 Similix

Similix [Bondorf and Danvy, 1991; Bondorf, 1993] is a self-applicable partial evaluator for a large higher-order subset of the strict functional language Scheme [Abelson *et al.*, 1998], that uses CPS style, and can handle partially static data structures [Mogensen, 1988]. It also handles source programs that use a limited class of side-effects, for instance input/output operations. The order in which effects are performed is preserved, and thus also the termination behaviour of the program; this is achieved by a process called let-insertion, which also avoid unnecessary duplication of computations. In type specialization, we could take a different approach to preservation of termination: we can have non-terminating programs specialized to terminating ones. Let's see an example.

Example 13.2. Consider the following source term:

$$\begin{array}{l} 1^S \;+^S \mathbf{if}^D \; \mathit{True}^D \\ \qquad\quad \mathbf{then}\ \mathbf{fix}^D \;(\lambda^D x.x) \\ \qquad\quad \mathbf{else}\ 2^S \\ : \mathit{Int}^S \end{array}$$

Clearly, the term is non-terminating, as the fixpoint of the identity function is not defined. However, as we ask both branches of a dynamic conditional to have the same residual type, the specialization of this term is $\bullet : \hat{3}$, which is saying that the result of the program is the number 3.

As we are considering a *pure* language — i.e. one with the only effect of non-termination — we have more freedom regarding termination: we can either try to preserve the termination behaviour of the source program in the residual one, or we allow the two to have different termination behaviours. Preservation of termination can be achieved by introducing extra computations that may loop in the residual, but at the cost of efficiency of the generated program — these extra computations take time to run. For example, in the case of the previous example, the residual program with the extra computations may look like $\mathbf{fix}\ (\lambda x.x); \bullet : \hat{3}$ indicating that, if the residual

computation terminates, its result must be 3 — and in this case, the residual computation does not terminate. Although some of these extra computations can be removed afterwards when it is clear they don't loop, it is clearly impossible to remove all of them. So we have taken the approach of not introducing them in the first place, thus altering the termination behaviour of some programs.

13.1.2 Polymorphic and modular partial evaluation

Regarding polymorphism and modules, we can mention the works of Heldal [Heldal, 2001; Heldal and Hughes, 2000; Dussart *et al.*, 1997a; Heldal and Hughes, 1997], and of Helsen and Thiemann [**?**; **?**].

Heldal shows

- how to generate a residual program with different modules from a single source program, by staging the static data — thus removing the inherited limit of modules,
- how to partially evaluate polymorphic programs, with a stress on the BTA required, making essential use of coercions as arguments to polymorphic functions, and
- how to specialize a multi-module program, although in an orthogonal way to the first result.

However, this approach cannot produce polymorphic programs from monomorphic ones.

13.1.3 Other works on partial evaluation

Peter Thiemann [1999c, 2000a] has been inspired by Hughes' original work on type specialization, and produced some works in the field of partial evaluation closely related to it.

Thiemann [1999c] presents a formalization of partial evaluation for a two-level lambda calculus in a Martin-Löf-style type theory [Nordström *et al.*, 1990] with some non-standard extensions. A typed source program generates a residual expression from its dynamic part, a residual (dependent) type from its static part, and a residual kind from its type. A big difference with type specialization is that residual types do not form a free algebra, but a quotient algebra based on a computation relation. Another important difference is that annotated source programs have to be *well-annotated* in order for the specialization to proceed; this restriction basically establishes that all subexpressions from a dynamic expression have to be dynamic — it is related to the existence of a translation function from fully dynamic types to monotypes. These two features make this approach fundamentally different wrt. to our work, because the free nature of residual types and the unrestricted nature of source annotations are fundamental ingredients of it, as we have shown throughout this thesis.

Thiemann [2000a] identifies the minimum number of features from type specialization needed to solve the type specialization problem as stated by Neil Jones [1988b]. He states that these features amount to first class polyvariance and what he calls co-arity raising.

Co-arity raising is the dual of arity raising: while the latter is the splitting of a function's argument into many arguments (and thus *raising* the arity of the function), the former is the splitting of a function's result into many results (which have to be further reinserted in the proper places in the code). Hughes [1996b] presents this feature as part of the whole process of arity raising. An important consequence of co-arity raising is that the usual restrictions on annotations are no longer needed, thus resulting in similar annotation requirements as in our work. Thiemann [2000b] showed type soundness of his system, and implemented it on his partial evaluation system for Scheme. This approach is the closest to type specialization, but it contains only a minimum number of features — it is not clear if all the possibilities of type specialization can be achieved by extending it or not. However, this contribution is very important because it clearly explains why type specialization is able to perform the optimal specialization of typed interpreters.

13.2 Type Directed Methods

To overcome the limitation imposed by partial evaluation on the type of the residual program, several methods for program specialization have been developed that make use of type information to guide the process. We review three of them: Olivier Danvy's type directed partial evaluation, Walid Taha's tag elimination, and Atshushi Ohori's approach to the compilation of polymorphic primitives.

13.2.1 Type Directed Partial Evaluation

Type-directed partial evaluation [Danvy, 1996; Sheard, 1997; Danvy, 1998b] is a simple method for implementing powerful partial evaluators that uses reification in a key way. *Reification* is a translation from a semantic domain back to an equivalent expression in the syntactic domain; it can be viewed as the reverse process of evaluation. In this way, arbitrary static expressions can be used in dynamic contexts. One important property of reification is that the returned expressions are always in normal form, and then, by composing an evaluator with a reification function, static computations can be carried out without any notion of symbolic computation.

TDPE has been implemented in Scheme [Danvy, 1996], Haskell [Rose, 1998], and ML [Sheard, 1997], the latter allowing the specialization of polymorphic functions. Similarly to our work, different residual terms can be obtained from the same source one, and this is achieved by varying the type guiding the reification. The key difference with our work, then, is that the residual type is an *input* to the specialization process, and thus types cannot be produced by specialization: the residual term is *adapted* to meet the desired type. So, similarly to partial evaluation, no annotations are needed in TDPE — but the residual type is!

The benefit of TDPE over type specialization is that it is a simpler approach, where the symbolic reduction mechanism *is* the operational semantics of the language. In our approach, constraint solving is used for symbolic reduction and it remains to be proved that it coincides with the semantics.

The fact that the generated programs returned by a type directed partial evaluator are always in (long $\beta\eta$-)normal form eliminates the need for polyvariance: if a function is applied, then it will be unfolded, and if it is not applied, it will be reified accordingly with the desired type.

A difficulty of TDPE is that to handle sum types, an abstraction of control [Felleisen, 1988; Danvy and Filinski, 1990; Danvy and Filinski, 1992] is needed, because, as described in the case of CPS partial evaluation, the context of a case over the sum is moved on every branch.

Although the ML implementation of TDPE can handle polymorphism in the source terms, there is no attempt to take polymorphic types to guide the reification, so the residual terms have no more polymorphism than the source program.

13.2.2 Tag-elimination

Tag elimination [Taha and Makholm, 2000; Taha *et al.*, 2001] is a transformation that removes type tags as a post-processing phase to traditional partial evaluation. Similarly to TDPE, it uses the desired residual type as input, and performs a type checking of the subject program after the interpretation, removing those tags that are superfluous.

TE is described as "specialization of types" (quoted in the original paper [Taha and Makholm, 2000]), but it is presented as a transformation that needs the residual type as input. We would like to see a presentation of TE showing that there is no need for the residual type to be provided as input.

The main contribution of TE is that theoretical results about the process can be easily proved: for example, it can be shown that Jones-optimality is obtained for a typed language, and that performing tag elimination is exactly the same as type-checking the term being interpreted; the only theoretical results about the power of TS are its correctness [Hughes, 2000], and the principality established in this thesis, although there is an example showing that optimal interpretation for the typed lambda calculus can be achieved [Hughes, 1996b]. Jones-optimality requires self-interpretation of the interpreter, and the application of the second and third Futamura projections [Jones *et al.*, 1993] require self-specialization; we are far from self-specialization, yet, but that is one of the goals that guide our efforts to make TS more expressive.

13.2.3 Ohori's specialization

Ohori [1999] has developed a framework that can be used to implement a language with polymorphic primitives efficiently. His work resembles that of Mark Jones [1994a] (his *kinds* corresponds to Mark Jones' predicates), and it is very similar in some technical aspects to the work presented here. His transformation of a polymorphic primitive into a pair of a low-level generic operation and a type attribute required for executing it on a given type resembles closely our treatment of polyvariant functions. The key difference is that his work is intended to be used as a compilation mechanism (it cannot be used to generate polymorphic programs; only to compile them), while ours is intended to generate arbitrary typed programs.

13.3 Other Approaches

There are several other methods for program specialization. We consider three of them here: supercompilation, generalized partial computation, and data specialization.

13.3.1 Supercompilation

Supercompilation [Turchin, 1986; Turchin, 1985; Sørensen and Glück, 1998; Sørensen *et al.*, 1996; Secher and Sørensen, 2000] is an approach to program specialization more general than partial evaluation. The idea is that a supercompiler supervises the evaluation of a program, and compiles a residual program for it, even when no input data is present. Instead of proceeding by a step-by-step transformation, a supercompiler builds a model of the program under treatment, and uses it to produce an equivalent version of the program, but more efficient — the authors state that this is the most important feature of the method. These results are obtained by using two techniques called *driving* and *generalization*. Driving consists in the construction of a possibly infinite *process tree* from the text of a program, and generalization consists in a criterion of when to stop driving and what to do with the nodes in the process tree, so warranting that it becomes finite.

An important difference of supercompilation with respect to partial evaluation is that in driving across case-expressions it uses the information that the pattern succeeded when specializing a given branch. This is achieved by propagating information by unification — this is called *positive supercompilation* [Sørensen *et al.*, 1996] — in a similar way as in type specialization. But the type of information propagation is different; consider the function $\lambda x.\textbf{if}\ x == 3\ \textbf{then}\ x + 1\ \textbf{else}\ x$: it can be transformed into $\lambda x.\textbf{if}\ x == 3\ \textbf{then}\ 4\ \textbf{else}\ x$ by supercompilation, because x is unified with 3. Although this example is not obtainable with type specialization, a similar propagation of information under the branches of a case expression is possible with our method, as we have shown in Example 3.22.

Another possibility is that also the **else** branch uses some information from the test. This is achieved by constraint-based propagation of information, and then the technique is called *perfect supercompilation* [Turchin, 1986; Secher and Sørensen, 2000]. This is similar to what we achieve with predicates in the case of static conditionals — the main difference is that in perfect supercompilation this is done with arbitrary conditionals.

An interesting feature of supercompilation is related to the order in which nested function calls are treated. In contrast to partial evaluation, it uses a call-by-name-style strategy: inner calls are treated first. This is called outside-in evaluation by Turchin [1986], and it is a feature that enables the method to eliminate intermediate structures [Sørensen and Glück, 1998], in a very similar way as deforestation.

13.3.2 Generalized Partial Computation

Generalized Partial Computation (GPC) was first proposed by Futamura and Nogi [1988a], and its power further demonstrated through examples [Futamura and Nogi, 1988b; Futamura *et al.*, 1991]. The original approach has been designed for a restricted form of

a language, but it has been extended also to first-order functional languages, with both strict and lazy semantics [Takano, 1991].

GPC is a program transformation technique based on partial evaluation and theorem proving [Takano, 1991; Futamura *et al.*, 2002]: instead of relying on the information coming from the values of static data, GPC also uses the logical structure of programs, axioms for abstract data types, and algebraic properties of primitive functions, obtained by means of the theorem prover.

It is a technique very similar to supercompilation, but the use of a theorem prover to propagate information makes it more powerful — while supercompilation can only propagate structural predicates (assertions and restrictions about atoms and constructors), GPC can propagate arbitrary predicates. Like supercompilation, when using a call-by-name-style strategy it can eliminate intermediate structures.

Regarding the relation of GPC with our work, the same comments as in the case of supercompilation apply.

13.3.3 Data specialization

Data specialization is an approach to program specialization where the results of static computations are stored in an intermediate data structure, instead of being coded as a residual program. It was introduced by Barzdins and Bulyonkov [1988], and further explored by Malmkjær [1989]; later, the technique was developed for a subset of C and applied to graphics applications by Knoblock and Ruf [1996], and afterwards combined with classical program specialization by Chirokoff *et al.* [1999].

The idea of data specialization is that the program to be specialized will produce two programs:

- one, called the *loader*, will calculate all the static values, and store those needed by dynamic computations into a data structure, called the *cache*;
- the other, called the *reader*, will perform the dynamic computations, using the cache when static data is needed.

Several considerations apply when deciding what data should be stored in the cache, because it is important that the size of the cache does not grow too big, and accessing it must not be more expensive than recalculating the value accessed.

In practice, data specialization is best suited for programs working with huge amounts of data and with big data dependencies, because it does not modify the control flow of the source program in any way. One advantage of this technique is that it solves the problem of code explosion arising with the usual program specialization when the amount of data is big. Good results can also be obtained by combining it with program specialization.

Regarding our work, a program with unknown static parts is specialized to a residual program using predicates, which can be further used in the calculation of the right version when the static data is available. This residual program can be, in some way, compared with the reader from the data specialization method. Despite the fact that we do not store static values in intermediate data structures, the ideas coming from this approach may be used to improve type specialization in some way, by exploiting the similarity we have mentioned.

We conclude this chapter with the observation that no specialization method is a panacea: each one of them has its strengths and weaknesses, and our method is no exception. Some methods, such as partial evaluation, have the great advantage that are very well engineered, and thus have wider applicability. However, the rich cross-fertilization of ideas produced by the consideration of variations is worth enough to justify the exploration of the possibilities.

Chapter 14

Future Work

Then he waited, marshaling his thoughts and brooding over his still untested powers. For (. . .) he was not quite sure what to do next.
But he would think of something.

2001. A Space Odyssey
Arthur C. Clarke

As we are approaching the end of this thesis, we found that there are several possibilities, weaknesses, extensions, and improvements that we have not considered because of time limitations. In this chapter we describe some of those things that are worth considering.

We begin, in Section 14.1, by describing the improvements needed in the constraint solver. As we have said, there are several programs that cannot be specialized to their final forms with our current solver, dynamic recursive programs being the most notorious. Our idea is to change the constraint solver to allow incremental solutions for the upper bound of a polyvariant residual type. Then, in Section 14.2, we describe another important need: a better implementation. The prototype we have presented is just an inefficient implementation designed to test the ideas, and is not suitable for the specialization of large programs. These two features are extremely important to turn type specialization into a usable tool. In other line of work, more extensions to the source language have to be considered, such as dynamic sum types, polyvariant sums, completing the specialization of polymorphism, and considering the specialization of lazy languages. We are already working on dynamic sum types, and we expect to have some results soon. All the extensions we plan are discussed in Section 14.3. Finally, in Section 14.4, we discuss the possibilities to generate a binding time assistant. As we have said, no automatic binding time analysis is possible for type specialization, but we may think of a tool helping the annotation of a program, perhaps by highlighting the sensible points in a program, or suggesting different possible choices.

We have found that our work has opened several different lines of research, and by following them we can get closer to a type specializer for Haskell.

14.1 Improving Constraint Solving

Constraint solving proved to be the most important part in our approach to type specialization — it is here where the actual calculation of static data, and the movement of information between different parts of a program is performed.

Our present heuristic is able to solve only simple cases, when a strong side condition is satisfied: all upper and lower bounds for some scheme variable should be present

to be able to calculate a solution for it. However, that condition is not met when one considers programs with dynamic recursion. For that reason, one important future work is the search for better algorithms for constraint solving, including dynamically recursive programs.

Another point where constraint solving shows its importance is in the production of polymorphic programs. Our attempts to perform this extension are based on a modified version of the evidence elimination phase (that is, in turn, an extension of constraint solving). However, we found that a more involved development is needed to tackle this problem in its full length. This means that more work has to be done in the development of this variation of constraint solving as well.

Finally, when adding static functions and static recursion we have done it with straightforward rules. The result for this is that when curried static functions are used in the source program, the residual type contains a lot of nested closures. We think that a more compact representation will be much more efficient, but of course, constraint solving has to be taken into account when this improved representation is designed. In the same spirit, static recursion is also not optimal, because it generates a lot of predicates that have to be checked for consistency, but which actually do not contribute with evidence or with new information to the solving. We think that some property relaxing the need for checking those predicates can be established, thus reducing the amount of work needed during constraint solving.

14.2 Better Implementation

Working with a complex theory like the one we have presented cannot be complete without an appropriate tool to test the ideas; additionally, as our final goal is to produce programs automatically, a proper implementation is a must.

The prototype we have designed with this work is very naive, and we have made no attempts to make it efficient. As a result, only small examples can be tested, because when the examples grew bigger, the time needed to produce the residual program increases to unfeasible limits.

There are many opportunities for improvement in this tool. The most time consuming part is that of constraint solving, so working on better algorithms as have been pointed out in the previous section, will surely improve the performance of the whole process (for example, by avoiding checking redundant predicates as in static recursion). However, there are also implementation enhancements that can be done — for example, storing the predicates in different structures depending on their nature, which will minimize the effort needed to look them up to detect solvable variables. Another possible enhancement is related with the way terms, types, and conversions are represented, so that comparison for equality between those, performing substitutions to them, etc. is much more efficient.

From another perspective, the compilation of the Haskell code has been done plainly, with no hints to the compiler on how to optimize the final code, and no profiling of the memory has been performed. It is usual that Haskell programs can be improved by performing minor changes related with memory consumption and lazy evaluation — examples have been given by Röjemo and Runciman [1996], Röjemo [1995], and Martínez

López [1998].

Introducing all these observations into the tool, to turn the prototype into a full-fledged implementation, is one work remaining to be done.

14.3 Extensions to the Source Language

The language for which we have developed the theory in its full dimension is a very small one. Even considering the extensions presented in Chapter 9, there are a number of important features that can be added to the source language to improve the level of expressiveness.

Among the features that can be added, we can mention the specialization of dynamic sum types, and the specialization of dynamic recursion. These two features are important to have a complete language to work with. As we have mentioned in Section 14.1, to be able to specialize dynamic recursion we only need to design a better algorithm for constraint solving. Dynamic sum types, on the other hand, need a more involved work, including the design of new predicates to express their specialization, and the corresponding constraint solving for them. We are already working on this topic, in collaboration with Alejandro Russo, and we expect to publish some results soon.

Two important features that were considered in John Hughes' work that we have not included here are type specialization for imperative features (monads), and polyvariant sums. The specialization of monadic operations has been presented by Dussart *et al.* [1997b], and we think that those ideas can be easily introduced in our framework. Polyvariant sums, on the other hand, provide a form of constructor specialization, and it is not so obvious how to provide principal specialization for programs using them. However, the work on dynamic sum types can enlighten this task, providing a way to tackle the problems presented.

On another dimension, we have features that have not been considered before, either in type specialization or in any other known framework of program specialization. We are referring to advanced features of modern languages, such as parametric polymorphism (in the source language), specialization of ad-hoc polymorphism, or overloading (type classes, for example), and specialization of programs with lazy behaviour.

Regarding source polymorphism, it seems like an easy addition, once residual polymorphism has been issued — see Chapter 11 and Section 14.1 for discussion on how to obtain residual polymorphism.

Regarding type classes, both source and residual, our framework has already predicates expressing other constructs, so it looks like an easy extension to have type classes — consider Section 5.3 that shows how Haskell type classes can be expressed with predicates and qualified types.

Finally, the specialization of languages with lazy evaluation semantics was never considered before. The main problem is that if a fragment of a program containing an infinite computation is marked static, it seems necessary that the specializer has to compute *all* the meaning of that expression before proceeding. However, as we have shown in Example 9.11, it is possible to combine constraint solving with arity raising to compute only those parts that are really needed, thus obtaining a framework with the ability to specialize lazy programs.

All these extensions can be addressed in isolation, showing that our formulation has indeed brought new potentials to the field of program specialization.

14.4 Binding Time Assistant

One important difference between type specialization and partial evaluation is the role of binding time annotations. As we have illustrated in Chapters 4 and 11, in type specialization we can choose the semantics of the object language just by varying the annotations.

It is important to note that we consider the constructs for lifting and polyvariance as annotations, even though they are not directly binding time annotations, as considered by the partial evaluation community. Polyvariance is first class in this framework, and thus it needs to be explicitly indicated by the programmer, which affects the way in which a program is specialized — observe the monomorphizer of Section 4.3, where polyvariance plays a central role.

For that reason, it is impossible to calculate the 'best' annotation automatically, as is done in partial evaluation. However, we have observed that usually there are some rules of thumb on how to annotate a given program, and perhaps it is possible to construct a tool for assisting the annotation process.

The assistant we are thinking of will suggest program points where it may be a good idea to make an expression polyvariant, or where some variable is better considered as dynamic (for example, because it is the control variable of a recursive function), and when the programmer fixes one particular annotation, it will calculate other annotations that are consequences of the given one. Even though this process is similar to the annotation inference we perform in the prototype, we also think of suggesting where to place **spec**s to specialize a polyvariant expression, which cannot be inferred.

A binding time assistant will be an excellent complement in an environment for automatic program production.

Chapter 15

Conclusions

One never reaches a horizon. It is not a line; it has no place; it enclosed no field; its location is always relative to the view. To move toward a horizon is simply to have a new horizon.

Finite and Infinite Games. A Vision of Life as Play and Possibility
James P. Carse

Despite the fact that it was introduced in 1996, Type Specialization is still not well-known and well-understood. There are several subtleties and dark alleys in the process of moving static information into the type. Those problems produced the impression that type specialization is just a weird way to do partial evaluation. However, as the examples have shown, it is a powerful method for specialization, including the possibility to type check an object program or to perform closure conversion on it, just by specializing it.

In this thesis we have contributed to the field of program specialization by designing a way to perform type specialization that clarifies some of the difficult aspects of the process, and that allows the possibility of trying different heuristics in the finding of solutions when calculating the final form of the residual program. We have shown that by varying some parts of the solving process we can produce polymorphic residual programs from monomorphic ones, thus introducing the possibility to break the inherited limit of the degree of polymorphism. In the process we have used the theory of qualified types, adapting it when necessary. The clear separation of concepts provided by that theory was a great help during the development of our framework.

The main property of the system we have introduced — principality — is of central importance in this process, because it allows to start the process of specialization of program fragments without the need for the whole program, opening the door for modular specialization. Principality is obtained by separating the specialization process in two phases: first traverse the syntactic tree of the source term collecting restrictions, and then solve those restrictions. Additionally, some criteria of when it is safe to solve certain restrictions can be established; for example, when a scheme variable is no longer free in the type, all its upper and lower bounds are present, then its value can be decided.

There were times during our search for a formulation allowing principality when it looked like there was a property that cannot be established. Several different attempts on how to formulate the rules were made, and discarded, before the right path was found. This is something that deserves to be mentioned, because when one looks at a finished type system, it usually seems that the rules are obvious. But when designing it, the properties one wants the system to have impose several restrictions on the way to formulate them. This is a warning to people thinking on entering the fascinating world of designing type systems, and one that we have learned in the hard way.

Our formulation, on the other hand, has shown enormous potential for obtaining new developments in the field of program specialization: specialization of polymorphism, lazy evaluation, and type classes are the most important ones. There are a lot of open paths to follow, and we are sure that they will end in a type specializer for the Haskell language.

Appendix A

Proofs

– *'How do you know I'm mad?' said Alice.*
– *'You must be,' said the Cat, – 'or you wouldn't have come here.'*

Alice's Adventures in Wonderland
Lewis Carroll

In this appendix, we present the proofs of propositions, lemmas, and theorems used to prove the property of principality for the specialization framework introduced in this thesis.

A.1 Proof of proposition 6.7 from section 6.1

PROPOSITION 6.7. *The following assertions hold when $\sigma, \sigma', \sigma''$ are not scheme variables:*

1. $[] : (\Delta \mid \sigma) \geq (\Delta \mid \sigma)$
2. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $C' : (\Delta' \mid \sigma') \geq (\Delta'' \mid \sigma'')$ *then*

$$C' \circ C : (\Delta \mid \sigma) \geq (\Delta'' \mid \sigma'')$$

Proof:

1. We can assume that $\sigma = \forall\alpha.\Delta_\sigma \Rightarrow \tau_\sigma$. To prove this item, we only need to prove that

$$\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h_\sigma.x((h))((h_\sigma)) = []$$

(which can be done using $(\mathbf{let}_v)$, Proposition 8.3, and (η_v)) and then use Definition 6.5.

2. Suppose that $\sigma = \forall\alpha.\Delta_\sigma \Rightarrow \tau_\sigma$, $\sigma' = \forall\beta.\Delta'_\sigma \Rightarrow \tau'_\sigma$, $\sigma'' = \forall\gamma.\Delta''_\sigma \Rightarrow \tau''_\sigma$. Then, applying Definition 6.5 to the hypothesis, we know that there exist substutions S and S', and evidence $v, v_\sigma, v', v'_\sigma$ such that

$$\tau'_\sigma = S\,\tau_\sigma \tag{A.1}$$
$$h' : \Delta', h'_\sigma : \Delta'_\sigma \Vdash v : \Delta, v_\sigma : S\,\Delta_\sigma \tag{A.2}$$
$$C = (\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h'_\sigma.x((v))((v_\sigma))) \tag{A.3}$$

and

$$\tau''_\sigma = S'\,\tau'_\sigma \tag{A.4}$$
$$h'' : \Delta'', h''_\sigma : \Delta''_\sigma \Vdash v' : \Delta', v'_\sigma : S'\,\Delta'_\sigma \tag{A.5}$$
$$C' = (\mathbf{let}_v\ x' = \Lambda h'.[]\ \mathbf{in}\ \Lambda h''_\sigma.x'((v'))((v'_\sigma))) \tag{A.6}$$

Let's call $v'' = v[h', h'_\sigma/v', v'_\sigma]$, and $v''_\sigma = v_\sigma[h', h'_\sigma/v', v'_\sigma]$. As dom$(S) = \alpha$, by (Close) on A.2,

$$h' : \Delta', h'_\sigma : S' \Delta'_\sigma \Vdash v : \Delta, v_\sigma : S' S\, \Delta_\sigma \tag{A.7}$$

and by (Trans) on A.5 and A.7,

$$h'' : \Delta'', h''_\sigma : \Delta''_\sigma \Vdash v'' : \Delta, v''_\sigma : S' S\, \Delta_\sigma \tag{A.8}$$

The result follows from Definition 6.5, using A.1 and A.4, A.8, and

$$C' \circ C = (\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h''_\sigma . x((v''))((v''_\sigma))) \tag{A.9}$$

obtained using ($\mathbf{let}_v$) and the fact that h', h'_σ only appear free on v, v_σ.

■

A.2 Proof of proposition 6.8 from section 6.1

Proposition 6.8. *The following assertions hold:*

1. *If* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, *then* $C : S\,(\Delta \mid \sigma) \geq S\,(\Delta' \mid \sigma')$.
2. *If* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \forall\alpha.\sigma')$, *and dom*$(S) = \alpha$ *then* $C : (\Delta \mid \sigma) \geq (\Delta' \mid S\,\sigma')$.

Proof:

1. Suppose $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, with $\sigma = \forall\alpha_i.\Delta_\sigma \Rightarrow \tau_\sigma$, and $\sigma' = \forall\beta_j.\Delta'_\sigma \Rightarrow \tau'_\sigma$, and such that none of the variables α_i, β_j are involved in S.

   By Definition 6.5, we know that there exist types τ_i, and evidence v, v_σ such that

$$\tau'_\sigma = \tau_\sigma[\alpha_i/\tau_i] \tag{A.10}$$
$$h' : \Delta', h'_\sigma : \Delta'_\sigma \Vdash v : \Delta, v_\sigma : \Delta_\sigma[\alpha_i/\tau_i] \tag{A.11}$$
$$C = (\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h'_\sigma . x((v))((v_\sigma))) \tag{A.12}$$

   We can apply S to A.10, and the fact that none of α_i are involved in S to obtain

$$S\,\tau'_\sigma = S\,(\tau_\sigma[\alpha_i/\tau_i]) = (S\,\tau_\sigma)[\alpha_i/S\,\tau_i] \tag{A.13}$$

   By the same reasoning applied to A.11 (using (Close)), we know that

$$h' : S\,\Delta', h'_\sigma : S\,\Delta'_\sigma \Vdash v : S\,\Delta, v_\sigma : (S\,\Delta_\sigma)[\alpha_i/S\,\tau_i] \tag{A.14}$$

   The result follows from Definition 6.5 using A.13, A.14, and A.12.

2. It is easy to see that $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ and then, using the previous item, C : $(S\,\Delta \mid S\,\sigma) \geq (S\,\Delta' \mid S\,\sigma')$ The result follows from the fact that $\alpha \notin \Delta, \Delta' \Rightarrow \sigma$.

■

A.3 Proof of proposition 6.9 from section 6.1

PROPOSITION 6.9. *For any qualified type ρ and predicate assignments $h : \Delta$ and $h' : \Delta'$,*

1. $\Lambda h'.[] : (\Delta, h' : \Delta' \mid \rho) \geq (\Delta \mid \Delta' \Rightarrow \rho)$
2. $[]((h')) : (\Delta \mid \Delta' \Rightarrow \rho) \geq (\Delta, h' : \Delta' \mid \rho)$
3. *if $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ and $h''' : \Delta''' \Vdash v'' : \Delta''$, then*
$$C' : (\Delta, \Delta'' \mid \sigma) \geq (\Delta', \Delta''' \mid \sigma')$$
where $C' = (\mathbf{let}_v\ x = \Lambda h'''.C[]\ \mathbf{in}\ x((v'')))$
4. *if $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ and $\alpha \notin FV(\Delta, \Delta' \Rightarrow \sigma)$, then*
$$C : (\Delta \mid \sigma) \geq (\Delta' \mid \forall\alpha.\sigma')$$

Proof:

1. The result is immediate using Definition 6.5, and the equality
$$(\mathbf{let}_v\ x = \Lambda h, h'.[]\ \mathbf{in}\ \Lambda h'.x((h))((h'))) = \Lambda h'.[]$$

2. $[]((h')) : (\Delta \mid \Delta' \Rightarrow \rho) \geq (\Delta, h' : \Delta' \mid \rho)$
The result is immediate using Definition 6.5, and the equality
$$(\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ x((h))((h'))) = []((h'))$$

3. Suppose that $\sigma = \forall\alpha.\Delta_\sigma \Rightarrow \tau_\sigma$, and $\sigma' = \forall\beta.\Delta'_\sigma \Rightarrow \tau'_\sigma$. Then, applying Definition 6.5 to the hypothesis, we know that there exist a substution S and evidence expressions v, v_σ such that
$$\tau'_\sigma = S\,\tau_\sigma \tag{A.15}$$
$$h' : \Delta', h'_\sigma : \Delta'_\sigma \Vdash v : \Delta, v_\sigma : S\,\Delta_\sigma \tag{A.16}$$
$$C = (\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h'_\sigma.x((v))((v_\sigma))) \tag{A.17}$$
It is easy to obtain, from A.16 and the hypothesis that
$$h' : \Delta', h''' : \Delta''', h'_\sigma : \Delta'_\sigma \Vdash v : \Delta, v'' : \Delta'', v_\sigma : S\,\Delta_\sigma \tag{A.18}$$
We also need the equality
$$(\mathbf{let}_v\ x = \Lambda h.C[]\ \mathbf{in}\ x((v))) = (\mathbf{let}_v\ x = \Lambda h, h'''.[]\ \mathbf{in}\ \Lambda h'_\sigma.x((v, v''))((v_\sigma))) \tag{A.19}$$
which follows from $(\mathbf{let}_v)$, (β_v), and the fact that h''' is not free in v, v_σ.
The result follows from Definition 6.5 using A.15, A.18, and A.19.

4. The result follows immediately from Definition 6.5.

■

A.4 Proof of proposition 6.11 from section 6.1

PROPOSITION 6.11. *If* $h : \Delta \mid \Gamma_{\!R} \vdash_{RT} e' : \sigma$*, and* $\Delta' \Vdash v : \Delta$*, then* $\Delta' \mid \Gamma_{\!R} \vdash_{RT} e'[h/v] : \sigma$*.*

Proof: By induction on the RT derivation. The only interesting case is that of (RT-QIN); the rest follows from IH, and (Trans) when necessary.

Case (RT-QIN): We know that

$$\text{(RT-QIN)}\quad \frac{h : \Delta, h_\delta : \delta \mid \Gamma_{\!R} \vdash_{RT} e' : \rho}{h : \Delta \mid \Gamma_{\!R} \vdash_{RT} \Lambda h_\delta . e' : \delta \Rightarrow \rho}$$

By (Fst), (Snd), and (Univ) on the hypothesis, we know that

$$\Delta', h_\delta : \delta \Vdash v : \Delta, h_\delta : \delta$$

By IH on the premise, we know that,

$$\Delta', h_\delta : \delta \mid \Gamma_{\!R} \vdash_{RT} e'[h/v] : \rho \tag{A.20}$$

The result follows from (RT-QIN) on A.20 and the facts that $h \neq h_\delta$ and $h_\delta \notin EV(v)$.

■

A.5 Proof of theorem 6.12 from section 6.1

THEOREM 6.12. *If* $h : \Delta \mid \Gamma_{\!R} \vdash_{RT} e' : \sigma$*, and* $C : (h : \Delta \mid \sigma) \geq (h' : \Delta' \mid \sigma')$*, then* $h' : \Delta' \mid \Gamma_{\!R} \vdash_{RT} C[e'] : \sigma'$*.*

Proof: By Definition 6.5 on the second hypothesis, we know that $\sigma = \forall \alpha_i . \Delta_\tau \Rightarrow \tau$ and $\sigma' = \forall \beta_j . \Delta'_\tau \Rightarrow \tau'$, with α_i, β_j only appearing in the bodies of σ, σ' respectively, and there are a substutition S, evidence variables h_τ and h'_τ, and evidence expressions v and v' such that:

$$\tau' = S\,\tau \tag{A.21}$$

$$h' : \Delta', h'_\tau : \Delta'_\tau \Vdash v : \Delta, v' : S\,\Delta_\tau \tag{A.22}$$

$$C = (\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h'_\tau . x(\!(v)\!)(\!(v')\!)) \tag{A.23}$$

By (RT-INST) on the first hypotesis, we know that

$$h : \Delta \mid \Gamma_{\!R} \vdash_{RT} e' : S\,(\Delta_\tau \Rightarrow \tau) \tag{A.24}$$

Then, by (RT-QIN) on A.24

$$\emptyset \mid \Gamma_{\!R} \vdash_{RT} \Lambda h.e' : \Delta, S\,\Delta_\tau \Rightarrow S\,\tau \tag{A.25}$$

By Proposition 6.11 on A.25 and then (RT-QOUT) on the result and A.22,

$$h' : \Delta', h'_\tau : \Delta'_\tau \mid \Gamma_{\!R} \vdash_{RT} (\Lambda h.e')(\!(v)\!)(\!(v')\!) : S\,\tau \tag{A.26}$$

Then, by (RT-QIN) on A.26 (also using A.21),

$$h' : \Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \Lambda h'_{\tau}.(\Lambda h.e')((v))((v')) : \Delta'_{\tau} \Rightarrow \tau' \tag{A.27}$$

Finally, by (RT-GEN) on A.27

$$h' : \Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \Lambda h'_{\tau}.(\Lambda h.e')((v))((v')) : \forall \beta_j.\Delta'_{\tau} \Rightarrow \tau' \tag{A.28}$$

The result follows from A.28, using A.23 and ($\mathbf{let}_v$) to show that

$$C[e'] = \Lambda h'_{\tau}.(\Lambda h.e')((v))((v'))$$

■

A.6 Proof of proposition 6.13 from section 6.3

PROPOSITION 6.13. *If* $\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma$ *then* $S\,\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow S\,\sigma$.

Proof: By induction on the SR derivation using IH and (Close). In the cases (SR-GEN) and (SR-INST), we can assume that $\alpha \notin \mathrm{dom}(S)$ by α-conversion, and then apply IH.

■

A.7 Proof of proposition 6.14 from section 6.3

PROPOSITION 6.14. *If* $\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma$ *and* $\Delta' \Vdash \Delta$*, then* $\Delta' \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma$.

Proof: By induction on the SR derivation, using (Trans) when necessary.

■

A.8 Proof of theorem 6.15 from section 6.3

THEOREM 6.15. *If* $\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma$ *and* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *then* $\Delta' \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma'$.

Proof: By Definition 6.5 on the second hypothesis, we know that $\sigma = \forall \alpha_i.\Delta_{\tau} \Rightarrow \tau$ and $\sigma' = \forall \beta_j.\Delta'_{\tau} \Rightarrow \tau'$, with α_i, β_j only appearing in the bodies of σ, σ' respectively, and there is a substutition S such that:

$$\tau' = S\,\tau \tag{A.29}$$

$$\Delta', \Delta'_{\tau} \Vdash \Delta, S\,\Delta_{\tau} \tag{A.30}$$

By (SR-INST) on the first hypotesis, we know that

$$h : \Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow S\,(\Delta_{\tau} \Rightarrow \tau) \tag{A.31}$$

Then, by (SR-QIN) on A.31

$$\emptyset \vdash_{\mathrm{SR}} \tau \hookrightarrow \Delta, S\,\Delta_{\tau} \Rightarrow S\,\tau \tag{A.32}$$

By Proposition 6.14 on A.32 and then (SR-QOUT) on the result and A.30,

$$\Delta', \Delta'_\tau \vdash_{\text{SR}} \tau \hookrightarrow S\,\tau \tag{A.33}$$

Then, by (SR-QIN) on A.33 (also using A.29),

$$\Delta' \vdash_{\text{SR}} \tau \hookrightarrow \Delta'_\tau \Rightarrow \tau' \tag{A.34}$$

Finally, by (SR-GEN) on A.34

$$\Delta' \vdash_{\text{SR}} \tau \hookrightarrow \forall \beta_j.\Delta'_\tau \Rightarrow \tau' \tag{A.35}$$

finishing the proof.

■

A.9 Proof of theorem 6.19 from section 6.3

THEOREM 6.19. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, *and for all* $x : \tau_x \hookrightarrow x' : \tau'_x \in \Gamma$, $\Delta \vdash_{\text{SR}} \tau_x \hookrightarrow \tau'_x$, *then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$.

Proof: By induction on the P derivation.

■

A.10 Proof of theorem 6.20 from section 6.3

THEOREM 6.20. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, *then* $\Delta \mid \Gamma_{(\text{RT})} \vdash_{\text{RT}} e' : \sigma$.

Proof: By induction on the P derivation.

■

A.11 Proof of proposition 6.21 from section 6.3

PROPOSITION 6.21. *If* $h : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \tau'$ *and* $h' : \Delta' \Vdash v : \Delta$, *then* $h' : \Delta' \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e'[h/v] : \tau'$

Proof: By induction on the P derivation. The only interesting case is (QIN); the rest follows from IH, and (Trans) when necessary.

Case (QIN): We know that

$$\text{(QIN)}\quad \frac{h : \Delta, h_\delta : \delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \rho}{h : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow \Lambda h_\delta.e' : \delta \Rightarrow \rho}$$

By (Fst), (Snd), and (Univ) on the hypothesis, we know that

$$\Delta', h_\delta : \delta \Vdash v : \Delta, h_\delta : \delta$$

By IH on the premise, we know that,

$$\Delta', h_\delta : \delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e'[h/v] : \rho \tag{A.36}$$

The result follows from (QIN) on A.36 and the facts that $h \neq h_\delta$ and $h_\delta \notin EV(v)$.

■

A.12 Proof of proposition 6.22 from section 6.3

PROPOSITION 6.22. *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then* $S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma$.

Proof: By induction on the P derivation using IH, Proposition 6.13, and (Close). In the cases (GEN) and (INST), we can assume that $\alpha \notin \mathrm{dom}(S)$ by α-conversion, and then apply IH.

■

A.13 Proof of proposition 7.3 from section 7.1

PROPOSITION 7.3. *The relation* $\geq$ *satisfies that, for all* Γ *and* τ',

1. *if* $h' : \Delta' \Vdash v : \Delta$ *and* $C = [\,]((v))$
   then $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq (h' : \Delta' \mid \tau')$
2. *if* $h' : \Delta' \Vdash v : \Delta$ *and* $C = \Lambda h'.[\,]((v))$
   then $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{\Gamma,\Delta''}(\Delta' \Rightarrow \tau')$
3. *for all substitutions* R *and all contexts* Δ,
   $[\,] : R\,\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{R\,\Gamma,R\,\Delta'}(R\,\Delta \Rightarrow R\,\tau')$

Proof:

1. Let's assume that $h' : \Delta' \Vdash v : \Delta$ and take $C = [\,]((v))$. The result $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq (h' : \Delta' \mid \tau')$ follows from Definition 6.5, taking Id as the substitution, using the hypothesis as the entailment needed, and using ($\mathbf{let}_v$) to show that $[\,]((v)) = \mathbf{let}_v\ x = [\,]\ \mathbf{in}\ x((v))$.

2. Let's assume that $h' : \Delta' \Vdash v : \Delta$. By the previous item, we know that

$$C_v : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq (h' : \Delta' \mid \tau') \tag{A.37}$$

   where $C_v = [\,]((v))$.

   By Proposition 6.9-1,

$$\Lambda h'.[\,] : (h' : \Delta' \mid \tau') \geq \Delta' \Rightarrow \tau' \tag{A.38}$$

   By Proposition 6.7-2 on A.37 and A.38,

$$C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq \Delta' \Rightarrow \tau' \tag{A.39}$$

   where $C = \Lambda h'.C_v[\,] = \Lambda h'.[\,]((v))$

   The result follows from A.39 by Proposition 6.9-4.

3. Let $\{\alpha_i\} = (FV(\Delta) \cup FV(\tau'))/(FV(\Gamma) \cup FV(\Delta'))$, and choose new variables γ_i not involved in R, such that

$$R\,\text{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') = \forall\gamma_i.R\,(\Delta \Rightarrow \tau')[\alpha_i/\gamma_i] = \forall\gamma_i.T\,\Delta \Rightarrow T\,\tau'$$

where $S = _[\alpha_i/\gamma_i]$, and $T = RS$.

Let $\{\beta_j\} = (FV(R\,\Delta) \cup FV()\,R\tau')/(FV()\,R\Gamma \cup FV()\,R\Delta')$, so we have

$$\text{Gen}_{R\,\Gamma,R\,\Delta'}(R\,\Delta \Rightarrow R\,\tau') = \forall\beta_j.R\,\Delta \Rightarrow R\,\tau'$$

We can observe that none of the variables β_j appear free in $R\,\text{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau')$ (suppose that $\beta \in FV(R\,\text{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau'))$; then, $\beta \in FV(R\,\alpha)$ for some $\alpha \in FV(\Delta \Rightarrow \tau')/\{\alpha_i\}$, which in turn implies that $\alpha \in (FV(\Gamma) \cup FV(\Delta'))$, and hence that $\beta \in FV(R\,\alpha) \subseteq (FV(R\,\Gamma) \cup FV(R\,\Delta'))$; it follows that $\beta \notin \{\beta_j\} = (FV(R\,\Delta) \cup FV()\,R\tau')/(FV()\,R\Gamma \cup FV()\,R\Delta'))$.

Note that

$$R\,\tau' = (T\,\tau')[R\,\alpha_i/\gamma_i]$$

and

$$h : R\,\Delta \Vdash h : R\,\Delta = (T\,\Delta)[R\,\alpha_i/\gamma_i]$$

The result follows from Definition 6.5:

$$[] : \forall\gamma_i.T\,\Delta \Rightarrow T\,\tau' \geq \forall\beta_j.R\,\Delta \Rightarrow R\,\tau'$$

■

A.14 Proof of proposition 7.7 from section 7.1

PROPOSITION 7.7. *If* $h : \Delta \mid \Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : \tau'$ *then* $h : S\,\Delta \mid S\,\Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : S\,\tau'$

Proof: By induction on the S derivation. The only interesting case is that of (S-POLY); all the rest follow from IH and the same rule, using, when needed Proposition 6.13 or (Close).

For the case (S-POLY), we have the following proof.

Case (S-POLY): We know that

$$\text{(S-POLY)}\quad \frac{h' : \Delta' \mid \Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : \tau' \qquad h : \Delta \Vdash v : \text{IsMG}\;\sigma'\;\sigma}{h : \Delta \mid \Gamma \vdash_{\text{S}} \textbf{poly}\;e : \textbf{poly}\;\tau \hookrightarrow v[\Lambda h'.e'] : \textbf{poly}\;\sigma}\;{\scriptstyle (\sigma'=\text{Gen}_{\Gamma,\emptyset}(\Delta'\Rightarrow\tau'))}$$

By IH on the first premise, we have that

$$h' : S\,\Delta' \mid S\,\Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : S\,\tau' \tag{A.40}$$

By (Close) on the second premise,

$$h : S\,\Delta \Vdash v : \text{IsMG}\;(S\,\sigma')\;(S\,\sigma) \tag{A.41}$$

By Proposition 7.3-3,

$$[] : S\,\mathrm{Gen}_{\Gamma,\emptyset}(\Delta' \Rightarrow \tau') \geq \mathrm{Gen}_{S\,\Gamma,\emptyset}(S\,\Delta' \Rightarrow S\,\tau') \tag{A.42}$$

and using (Comp) on A.41 and A.42,

$$h : S\,\Delta \Vdash v : \mathrm{IsMG}\ \mathrm{Gen}_{S\,\Gamma,\emptyset}(S\,\Delta' \Rightarrow S\,\tau')\ (S\,\sigma) \tag{A.43}$$

The result follows from (S-POLY) on A.40 and A.43.

■

A.15 Proof of proposition 7.8 from section 7.1

Proposition 7.8. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$ *and* $\Delta' \Vdash v : \Delta$, *then*

$$\Delta' \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'[h/v] : \tau'$$

Proof: By induction on the S derivation. The only interesting case is that of (S-POLY); all the rest follow from IH and the same rule.

For the case (S-POLY), we have the following proof.

Case (S-POLY): We know that

$$\text{(S-POLY)}\quad \frac{h'' : \Delta'' \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau' \quad h : \Delta \Vdash v_\sigma : \mathrm{IsMG}\ \sigma'\ \sigma}{h : \Delta \mid \Gamma \vdash_{\mathrm{S}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow v_\sigma[\Lambda h''.e'] : \mathbf{poly}\ \sigma}\ \left(\sigma' = \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'' \Rightarrow \tau')\right)$$

By (Trans) on the second hypotesis and the second premise, we have that

$$h' : \Delta' \Vdash v_\sigma[h/v] : \mathrm{IsMG}\ \sigma'\ \sigma \tag{A.44}$$

On the other hand, by Lemma 6.23 on the first premise we know that $h \notin EV(e')$, and thus

$$(v_\sigma[h/v])[\Lambda h''.e'] = (v_\sigma[\Lambda h''.e'])[h/v] \tag{A.45}$$

The result follows from (S-POLY) on the first premise and A.44, and using A.45.

■

A.16 Proof of theorem 7.9 from section 7.1

Theorem 7.9. *If* $\Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$ *then* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \tau'$.

Proof: By induction on the S derivation. The only interesting case is that of (S-POLY); all the rest follow from IH and the corresponding rule in system P.

For the case (S-POLY), we have the following proof.

Case (S-POLY): We know that

$$\text{(S-POLY)}\quad \frac{h' : \Delta' \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau' \qquad h : \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{h : \Delta \mid \Gamma \vdash_{\mathrm{S}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow v[\Lambda h'.e'] : \mathbf{poly}\ \sigma}\ \left(\sigma' = \mathrm{Gen}_{\Gamma,\emptyset}(\Delta' \Rightarrow \tau')\right)$$

By IH on the first premise, we have that

$$h' : \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \tau' \tag{A.46}$$

Using rule (QIN) on A.46 several times, we get

$$\emptyset \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow \Lambda h'.e' : \Delta' \Rightarrow \tau' \tag{A.47}$$

and then, using rule (GEN) several times on this equation, we get

$$\emptyset \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow \Lambda h'.e' : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta' \Rightarrow \tau') \tag{A.48}$$

Using Proposition 6.21 on this last equation, we obtain

$$h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow \Lambda h'.e' : \sigma' \tag{A.49}$$

The result follows from (POLY) on A.49 and the second premise.

■

A.17 Proof of theorem 7.10 from section 7.1

THEOREM 7.10. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$, *then there exist* $h'_s, \Delta'_s, e'_s, \tau'_s$, *and* C'_s *such that*

a) $h'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'_s : \tau'_s$

b) $C'_s : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma)$

c) $C'_s[\Lambda h'_s.e'_s] = e'$

Proof: By induction on the P derivation. Several cases are similar to others, so we will present only the proof of some of them, representative of the rest.

Case (VAR): We know that

$$\text{(VAR)}\quad \frac{x : \tau \hookrightarrow e' : \tau' \in \Gamma}{h : \Delta \mid \Gamma \vdash_{\mathrm{P}} x : \tau \hookrightarrow e' : \tau'}$$

Let's take $h'_s = h$, $\Delta'_s = \Delta$, $e'_s = e'$, $\tau'_s = \tau'$, and $C'_s = [](\!(h)\!)$.

Item **a)** holds by (S-VAR) on the premise.

$$h : \Delta \mid \Gamma \vdash_{\mathrm{S}} x : \tau \hookrightarrow e' : \tau'$$

Item **b)** holds by Proposition 7.3-1.

$$[]((h)) : \text{Gen}_{\Gamma,\emptyset}(\Delta \Rightarrow \tau') \geq (h : \Delta \mid \tau')$$

Item **c)** holds by Proposition 8.3.

$$(\Lambda h.e')((h)) = e'$$

Case (D+)**:** We know that

$$\text{(D+)}\quad \frac{(h : \Delta \mid \Gamma \vdash_{\text{P}} e_i : Int^D \hookrightarrow e'_i : Int)_{i=1,2}}{h : \Delta \mid \Gamma \vdash_{\text{P}} e_1 +^D e_2 : Int^D \hookrightarrow e'_1 + e'_2 : Int}$$

By IH on the premise, we have that for $i = 1, 2$ there exist h'_{Ii}, Δ'_{Ii}, e'_{Ii}, τ'_{Ii}, and C'_{Ii} such that

$$h'_{Ii} : \Delta'_{Ii} \mid \Gamma \vdash_{\text{S}} e_i : Int^D \hookrightarrow e'_{Ii} : \tau'_{Ii} \quad \text{(A.50)}$$
$$C'_{Ii} : \text{Gen}_{\Gamma,\emptyset}(\Delta'_{Ii} \Rightarrow \tau'_{Ii}) \geq (h : \Delta \mid Int) \quad \text{(A.51)}$$
$$C'_{Ii}[\Lambda h'_{Ii}.e'_{Ii}] = e'_i \quad \text{(A.52)}$$

By Definition 6.5 on A.51, there exist, for $i = 1, 2$, S'_{Ii} and v'_{Ii} such that

$$Int = S'_{Ii}\, \tau'_{Ii} \quad \text{(A.53)}$$
$$h : \Delta \Vdash v'_{Ii} : S'_{Ii}\, \Delta'_{Ii} \quad \text{(A.54)}$$
$$C'_{Ii} = []((v'_{Ii})) \quad \text{(A.55)}$$

By Propositions 7.7 and 7.8 on A.50 and A.54, and using A.53, we have for $i = 1, 2$

$$h : \Delta \mid \Gamma \vdash_{\text{S}} e_i : Int^D \hookrightarrow e'_{Ii}[h'_{Ii}/v'_{Ii}] : Int \quad \text{(A.56)}$$

Finally, by replacing A.55 on A.52, and then using (β_v), we obtain

$$e'_{Ii}[h'_{Ii}/v'_{Ii}] = e'_i \quad \text{(A.57)}$$

Let's take $h'_s = h$, $\Delta'_s = \Delta$, $e'_s = e'_{I1}[h'_{I1}/v'_{I1}] + e'_{I2}[h'_{I2}/v'_{I2}]$, $\tau'_s = Int$, and $C'_s = []((h))$.

Item **a)** holds by (S-D+) on A.56.

$$h : \Delta \mid \Gamma \vdash_{\text{S}} e_1 +^D e_2 : Int^D \hookrightarrow e'_{I1}[h'_{I1}/v'_{I1}] + e'_{I2}[h'_{I2}/v'_{I2}] : Int$$

Item **b)** holds by Proposition 7.3-1.

$$[]((h)) : \text{Gen}_{\Gamma,\emptyset}(\Delta \Rightarrow Int) \geq (h : \Delta \mid Int)$$

Item **c)** holds by Proposition 8.3 and A.57.

$$(\Lambda h.e'_{I1}[h'_{I1}/v'_{I1}] + e'_{I2}[h'_{I2}/v'_{I2}])((h)) = e'_1 + e'_2$$

Case (POLY): We know that

$$\text{(POLY)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma' \qquad \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[e'] : \mathbf{poly}\ \sigma}$$

By IH on the first premise, there exist h'_I, Δ'_I, e'_I, τ'_I, and C'_I such that

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e'_I : \tau'_I \tag{A.58}$$

$$C'_I : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \sigma') \tag{A.59}$$

$$C'_I[\Lambda h'_I.e'_I] = e' \tag{A.60}$$

By Lemma 6.24, there exist β_j, Δ_σ, and τ' such that β_j does not appear free in Δ or Γ, and

$$\sigma' = \forall \beta_j.\Delta_\sigma \Rightarrow \tau' \tag{A.61}$$

Then, by Definition 6.5 on A.59 and A.61, there exist S'_I and v'_I such that

$$\tau' = S'_I\, \tau'_I \tag{A.62}$$

$$h : \Delta, h_\sigma : \Delta_\sigma \Vdash v'_I : S'_I\, \Delta'_I \tag{A.63}$$

$$C'_I = \Lambda h_\sigma.[]((v'_I)) \tag{A.64}$$

By Proposition 7.7 on A.58,

$$h'_I : S'_I\, \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e'_I : S'_I\, \tau'_I \tag{A.65}$$

and by Proposition 7.8 on A.65 and A.63, and using A.62,

$$h : \Delta, h_\sigma : \Delta_\sigma \mid \Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e'_I[h'_I/v'_I] : \tau' \tag{A.66}$$

Let's define

$$\sigma'' = \mathrm{Gen}_{\Gamma,\emptyset}(\Delta, \Delta_\sigma \Rightarrow \tau') \tag{A.67}$$

As β_j does not appear free in Δ, all the variables of Δ appear free on σ', and thus, by Definition 6.5,

$$C'' : (\Delta \mid \sigma'') \geq (\Delta \mid \sigma') \tag{A.68}$$

where $C'' = \Lambda h_\sigma.[]((h, h_\sigma))$, and using (IsMG) on A.68, we have that

$$\Delta \Vdash C'' : \mathrm{IsMG}\ \sigma''\ \sigma' \tag{A.69}$$

Now, using (Comp) on A.69 and second premise,

$$\Delta \Vdash v \circ C'' : \mathrm{IsMG}\ \sigma''\ \sigma \tag{A.70}$$

On the other hand, by replacing A.64 on A.60, and then using (β_v), we obtain

$$\Lambda h_\sigma.e'_I[h'_I/v'_I] = e' \tag{A.71}$$

and by Proposition 8.3, and A.71,

$$C''[\Lambda h, h_\sigma.e'_I[h'_I/v'_I]] = e' \tag{A.72}$$

Let's take $h'_s = h$, $\Delta'_s = \Delta$, $e'_s = v[C''[\Lambda h, h_\sigma.e'_I[h'_I/v'_I]]]$, $\tau'_s = \mathbf{poly}\ \sigma$, and $C'_s = []((h))$.

Item **a)** holds by (S-POLY) on A.66 and A.70, using A.67.

$$h : \Delta \mid \Gamma \vdash_{\mathrm{S}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow v[C''[\Lambda h, h_\sigma.e'_I[h'_I/v'_I]]] : \mathbf{poly}\ \sigma$$

Item **b)** holds by Proposition 7.3-1.

$$[]((h)) : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta \Rightarrow \mathbf{poly}\ \sigma) \geq (h : \Delta \mid \mathbf{poly}\ \sigma)$$

Item **c)** holds by Proposition 8.3 and A.72.

$$(\Lambda h.e'_s)((h)) = v[e']$$

Case (QIN): We know that

$$\text{(QIN)}\ \frac{\Delta, h_\delta : \delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \rho}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow \Lambda h_\delta.e' : \delta \Rightarrow \rho}$$

By IH on the first premise, there exist h'_I, Δ'_I, e'_I, τ'_I, and C'_I such that

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'_I : \tau'_I \tag{A.73}$$
$$C'_I : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta, h_\delta : \delta \mid \rho) \tag{A.74}$$
$$C'_I[\Lambda h'_I.e'_I] = e' \tag{A.75}$$

By Lemma 6.24, there exist Δ_ρ and τ'_ρ such that

$$\rho = \Delta_\rho \Rightarrow \tau'_\rho \tag{A.76}$$

Then, by Definition 6.5 on A.74 and A.76, there exist S'_I and v'_I such that

$$\tau'_\rho = S'_I\,\tau'_I \tag{A.77}$$
$$h : \Delta, h_\delta : \delta, h_\rho : \Delta_\rho \Vdash v'_I : S'_I\,\Delta'_I \tag{A.78}$$
$$C'_I = \Lambda h_\rho.[]((v'_I)) \tag{A.79}$$

Let's take $h'_s = h'_I$, $\Delta'_s = \Delta'_I$, $e'_s = e'_I$, $\tau'_s = \tau'_I$, and $C'_s = \Lambda h_\delta, h_\rho.[]((v'_I))$.

Item **a)** holds by A.73.

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'_I : \tau'_I$$

Item **b)** holds by Definition 6.5 on A.77 and A.78, using S'_I as substitution.

$$\Lambda h_\delta, h_\rho.[]((v'_I)) : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \delta \Rightarrow \rho)$$

Item **c)** holds by expanding A.79 on A.75.

$$\Lambda h_\delta, h_\rho.(\Lambda h'_I.e'_I)((v'_I)) = \Lambda h_\delta.e'$$

Case (QOUT): We know that

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v_\delta : \delta}{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e'(\!(v_\delta)\!) : \rho}$$

By IH on the first premise, there exist h'_I, Δ'_I, e'_I, τ'_I, and C'_I such that

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\text{S}} e : \tau \ \hookrightarrow\ e'_I : \tau'_I \tag{A.80}$$
$$C'_I : \text{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \delta \Rightarrow \rho) \tag{A.81}$$
$$C'_I[\Lambda h'_I.e'_I] = e' \tag{A.82}$$

By Lemma 6.24, there exist Δ_ρ and τ'_ρ such that

$$\rho = \Delta_\rho \Rightarrow \tau'_\rho \tag{A.83}$$

Then, by Definition 6.5 on A.74 and A.83, there exist S'_I and v'_I such that

$$\tau'_\rho = S'_I\, \tau'_I \tag{A.84}$$
$$h : \Delta, h_\delta : \delta, h_\rho : \Delta_\rho \Vdash v'_I : S'_I\, \Delta'_I \tag{A.85}$$
$$C'_I = \Lambda h_\delta, h_\rho.[](\!(v'_I)\!) \tag{A.86}$$

Using (Trans) on A.85 and second premise, we get

$$h : \Delta, h_\rho : \Delta_\rho \Vdash v'_I[h_\delta/v_\delta] : S'_I\, \Delta'_I \tag{A.87}$$

Let's take $h'_s = h'_I$, $\Delta'_s = \Delta'_I$, $e'_s = e'_I$, $\tau'_s = \tau'_I$, and $C'_s = \Lambda h_\rho.[](\!(v'_I[h_\delta/v_\delta])\!)$.

Item **a)** holds by A.80.

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\text{S}} e : \tau \ \hookrightarrow\ e'_I : \tau'_I$$

Item **b)** holds by Definition 6.5 on A.84 and A.87, using S'_I as substitution.

$$\Lambda h_\rho.[](\!(v'_I[h_\delta/v_\delta])\!) : \text{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \rho)$$

Item **c)** holds by expanding A.86 on A.82, using (β_v), and the fact that $h_\delta \notin EV(e'_I)$ (because h_δ is fresh).

$$\Lambda h_\rho.(\Lambda h'_I.e'_I)(\!(v'_I[h_\delta/v_\delta])\!) = e'(\!(v_\delta)\!)$$

Case (GEN): We know that

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : \sigma}{\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : \forall\alpha.\sigma}\ (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

By IH on the first premise, there exist h'_I, Δ'_I, e'_I, τ'_I, and C'_I such that

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\text{S}} e : \tau \ \hookrightarrow\ e'_I : \tau'_I \tag{A.88}$$
$$C'_I : \text{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \sigma) \tag{A.89}$$
$$C'_I[\Lambda h'_I.e'_I] = e' \tag{A.90}$$

Let's take $h'_s = h'_I$, $\Delta'_s = \Delta'_I$, $e'_s = e'_I$, $\tau'_s = \tau'_I$, and $C'_s = C'_I$.

Item **a)** holds by A.88.

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'_I : \tau'_I$$

Item **b)** holds by A.89 and the fact that α does not appear free on $\mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I)$ because of the side condition $\alpha \notin FV(\Delta) \cup FV(\Gamma)$.

$$C'_I : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \forall\alpha.\sigma)$$

Item **c)** holds by A.90.

$$C'_I[\Lambda h'_I.e'_I] = e'$$

Case (INST): We know that

$$(\text{INST})\ \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \forall\alpha.\sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma}\ (\mathrm{dom}(S){=}\alpha)$$

By IH on the first premise, there exist h'_I, Δ'_I, e'_I, τ'_I, and C'_I such that

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'_I : \tau'_I \quad (A.91)$$
$$C'_I : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid \forall\alpha.\sigma) \quad (A.92)$$
$$C'_I[\Lambda h'_I.e'_I] = e' \quad (A.93)$$

Let's take $h'_s = h'_I$, $\Delta'_s = \Delta'_I$, $e'_s = e'_I$, $\tau'_s = \tau'_I$, and $C'_s = C'_I$.

Item **a)** holds by A.91.

$$h'_I : \Delta'_I \mid \Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e'_I : \tau'_I$$

Item **b)** holds by transitivity of $\geq$ with A.92 and Proposition 6.8-2.

$$C'_I : \mathrm{Gen}_{\Gamma,\emptyset}(\Delta'_I \Rightarrow \tau'_I) \geq (h : \Delta \mid S\,\sigma)$$

Item **c)** holds by A.93.

$$C'_I[\Lambda h'_I.e'_I] = e'$$

■

A.18 Proof of proposition 7.11 from section 7.2

PROPOSITION 7.11. *If* $\sigma \sim^U \sigma'$ *then* $U\,\sigma = U\,\sigma'$.

Proof: By induction on the derivation of $\sigma \sim^U \sigma'$.

Cases $c \sim^{\mathrm{Id}} c$, $\hat{n} \sim^{\mathrm{Id}} \hat{n}$, $Int \sim^{\mathrm{Id}} Int$, and $\alpha \sim^{\mathrm{Id}} \alpha$ are completely trivial. Cases $\alpha \sim^{[\alpha/\sigma]} \sigma$, $\tau'_1 \rightarrow \tau'_2 \sim^{UT} \tau''_1 \rightarrow \tau''_2$, and $\forall\alpha.\sigma \sim^U \forall\alpha'.\sigma'$ are presented below. For $\sigma \sim^{[\alpha/\sigma]} \alpha$, the proof is identical to its symmetrical case. For all the rest, the result follows from IH and the definition of substitution.

Case $\alpha \sim^{[\alpha/\sigma]} \sigma$: We know that

$$\frac{\alpha \notin FV(\sigma)}{\alpha \sim^{[\alpha/\sigma]} \sigma}$$

It is clear that, as $\alpha \notin FV(\sigma)$ by the premise, then $\alpha[\alpha/\sigma] = \sigma[\alpha/\sigma]$.

Case $\tau_1' \rightarrow \tau_2' \sim^{UT} \tau_1'' \rightarrow \tau_2''$ **:** We know that

$$\frac{\tau_1' \sim^{T} \tau_2' \qquad T\,\tau_1'' \sim^{U} T\,\tau_2''}{\tau_1' \rightarrow \tau_2' \sim^{UT} \tau_1'' \rightarrow \tau_2''}$$

By IH on the first premise, we have that

$$T\,\tau_1' = T\,\tau_1'' \tag{A.94}$$

and then, it is clear that

$$UT\,\tau_1' = UT\,\tau_1'' \tag{A.95}$$

Again by IH, but this time on the second premise, we have that

$$U\,T\,\tau_2' = U\,T\,\tau_2'' \tag{A.96}$$

The result follows from A.95 and A.96 by definition of substitution.

Case $\forall\alpha.\sigma \sim^{U} \forall\alpha'.\sigma'$**:** We know that

$$\frac{\sigma[\alpha/c] \sim^{U} \sigma'[\alpha'/c]}{\forall\alpha.\sigma \sim^{U} \forall\alpha'.\sigma'}\ (c\ \textit{fresh})$$

By α-conversion on type schemes, we can assume that both α and α' do not appear free on U. By IH on the premise,

$$U\,(\sigma[\alpha/c]) = U\,(\sigma'[\alpha'/c]) \tag{A.97}$$

and then

$$(U\,\sigma)[\alpha/c] = (U\,\sigma')[\alpha'/c] \tag{A.98}$$

The result follows from the definition of substitution, and the fact that α and α' do not appear free on U.

∎

A.19 Proof of proposition 7.12 from section 7.2

PROPOSITION 7.12. *If* $S\,\sigma = S\,\sigma'$*, then* $\sigma \sim^U \sigma'$ *and there exists a substitution* T *such that* $S = TU$*.*

Proof:

We first have to show that every derivation of $\sigma \sim^U \sigma'$ is finite. To do that, we define a complexity measure consisting on the pair (n, m) where n is the number of free variables and m is the number of constructors in σ and σ'; the lexicographic ordering on this measure is well-founded. The premises of every rule have a smaller complexity than the conclusion (we need the property that the free variables in the range of U are included in the set of free variables of σ, σ'), and so every derivation tree must be finite.

By well-founded induction on the derivation of $\sigma \sim^U \sigma'$. There are four cases to consider: either

1. σ and σ' are identical, and no unification is necessary,
2. σ and σ' are of the form $\sigma = \forall\alpha.\sigma_I$ and $\sigma' = \forall\alpha'.\sigma'_I$, with α, α' not involved in S and $S\,\sigma_I = S\,\sigma'_I$,
3. σ is a variable α not in σ', or vice versa, or
4. both types began with the same constructor, and their corresponding parts are unified by S.

In the first case, it can be shown by induction on the derivation $\sigma \sim^U \sigma'$ that $U = \mathrm{Id}$, and the result trivilly follows.

In the second case, as α, α' are not involved in S, $S\,(\sigma_I[\alpha/c]) = (S\,\sigma_I)[\alpha/c] = (S\,\sigma'_I)[\alpha'/c] = S\,(\sigma'_I[\alpha'/c])$ for a fresh constant c. The result follows from IH and the rule for type schemes.

In the third case, the algorithm produces the substitution $U = _[\alpha/\sigma']$; but, as we know that $S\,\alpha = S\,\sigma'$, then $S\,U\,\alpha = S\,\sigma' = S\,\alpha$, and $S\,U\,\beta = S\,\beta$, for $\beta \neq \alpha$. Hence, $S = SU$, finishing the case.

In the fourth case, we have to apply IH to the premises, and construct the result. We present the case for function types as example; the rest are analogous. We know that

$$\frac{\tau'_1 \sim^{U'} \tau'_2 \qquad U'\,\tau''_1 \sim^{U''} U'\,\tau''_2}{\tau'_1 \to \tau'_2 \sim^{U''U'} \tau''_1 \to \tau''_2}$$

where $U = U''U'$, and we want to find a substitution T such that $S = TU = TU''U'$. By IH on the first premise, we have that there exists a substitution T' such that $S = T'U'$. Then we can rewrite $S\,\tau'_i$ as $T'\,U'\,\tau'_i$, and so, applying the IH to the second premise, we obtain that there exists T'' such that $T' = T''U''$. We observe that composing the two equalities $S = T'U'$ and $T' = T''U''$, we obtain $S = T''U''U'$, and so the result follows by taking $T = T''$.

■

A.20 Proof of proposition 7.14 from section 7.2

PROPOSITION 7.14. *If* $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$ *then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$.

Proof: By induction on the derivation of $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$.

Case *Int*^{S}: The result follows from (SR-SINT) on the trivial judgment IsInt $t \Vdash$ IsInt t.

Case *Int*^{D}: The result follows from (SR-DINT).

Case $\tau_2 \rightarrow^D \tau_1$: We know that

$$\frac{\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1' \qquad \Delta_2 \vdash_{\text{W-SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta_1, \Delta_2 \vdash_{\text{W-SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau_2' \rightarrow \tau_1'}$$

By IH on the premises, we know that $\Delta_1 \vdash_{\text{SR}} \tau_1 \hookrightarrow \tau_1'$ and $\Delta_2 \vdash_{\text{SR}} \tau_2 \hookrightarrow \tau_2'$

Then, by Proposition 6.14 on those, we know that

$$\Delta_1, \Delta_2 \vdash_{\text{SR}} \tau_1 \hookrightarrow \tau_1' \tag{A.99}$$

and

$$\Delta_1, \Delta_2 \vdash_{\text{SR}} \tau_2 \hookrightarrow \tau_2' \tag{A.100}$$

The result follows from (SR-DFUN) on A.99 and A.100.

Case $(\tau_1, \ldots, \tau_n)^D$: We know that

$$\frac{(\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1')_{i=1,\ldots,n}}{\Delta_1, \ldots, \Delta_n \vdash_{\text{W-SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

The result is obtained applying IH to the premises, then Proposition 6.14 to the resulting judgments to obtain the same predicate assignment on all of them, and finally using (SR-TUPLE) on those.

Case poly τ: We know that

$$\frac{\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'}{\text{IsMG}\ \sigma\ s \vdash_{\text{W-SR}} \textbf{poly}\ \tau \hookrightarrow \textbf{poly}\ s} \left(\sigma = \text{Gen}_{\emptyset,\emptyset}(\Delta \Rightarrow \tau')\ \textit{and}\ s\ \textit{fresh}\right)$$

By IH on the premise, we know that $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$

Then, by (SR-QIN), $\vdash_{\text{SR}} \tau \hookrightarrow \Delta \Rightarrow \tau'$ and by (SR-GEN), $\vdash_{\text{SR}} \tau \hookrightarrow \text{Gen}_{\emptyset,\emptyset}(\Delta \Rightarrow \tau')$

Finally, by Proposition 6.14

$$\text{IsMG}\ \sigma\ s \vdash_{\text{SR}} \tau \hookrightarrow \text{Gen}_{\emptyset,\emptyset}(\Delta \Rightarrow \tau') \tag{A.101}$$

The result follows from (SR-POLY) on A.101 and IsMG σ $s \Vdash$ IsMG σ s.

■

A.21 Proof of proposition 7.15 from section 7.2

PROPOSITION 7.15. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *then* $\Delta'_w \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_w$ *with all the residual variables fresh, and there exists* C'_w *such that* $C'_w : \text{Gen}_{\emptyset,\emptyset}(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$.

Proof: By induction on the SR derivation.

Case (SR-DINT): We know that

$$\text{(SR-DINT)} \quad \Delta \vdash_{\text{SR}} \mathit{Int}^D \hookrightarrow \mathit{Int}$$

The result follows from $\emptyset \vdash_{\text{W-SR}} \mathit{Int}^D \hookrightarrow \mathit{Int}$.

Case (SR-SINT): We know that

$$\text{(SR-SINT)} \quad \frac{\Delta \Vdash \text{IsInt } \tau'}{\Delta \vdash_{\text{SR}} \mathit{Int}^S \hookrightarrow \tau'}$$

Applying the algorithm, we have that IsInt $t \vdash_{\text{W-SR}} \mathit{Int}^S \hookrightarrow t$, with t fresh. The fact that there exists C'_w such that

$$C'_w : (\forall t.\text{IsInt } t \Rightarrow t) \geq (\Delta \mid \tau')$$

follows from Definition 6.5 using the premise for the required entailment.

Case (SR-DFUN): We know that

$$\text{(SR-DFUN)} \quad \frac{\Delta \vdash_{\text{SR}} \tau_1 \hookrightarrow \tau'_1 \qquad \Delta \vdash_{\text{SR}} \tau_2 \hookrightarrow \tau'_2}{\Delta \vdash_{\text{SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau'_2 \rightarrow \tau'_1}$$

By IH on the first premise,

$$\Delta'_{w_1} \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau'_{w_1} \tag{A.102}$$

with all the variables fresh, and there exists C'_{w_1} such that

$$C'_{w_1} : \text{Gen}_{\emptyset,\emptyset}(\Delta'_{w_1} \Rightarrow \tau'_{w_1}) \geq (\Delta \mid \tau'_1) \tag{A.103}$$

that is, by Definition 6.5, there exist S'_{w_1} and v'_{w_1} such that

$$\tau'_1 = S'_{w_1}\, \tau'_{w_1} \tag{A.104}$$
$$h : \Delta \Vdash v'_{w_1} : S'_{w_1}\, \Delta'_{w_1} \tag{A.105}$$
$$C'_{w_1} = [\,]((v'_{w_1})) \tag{A.106}$$

By IH on the second premise,

$$\Delta'_{w_2} \vdash_{\text{W-SR}} \tau_2 \hookrightarrow \tau'_{w_2} \tag{A.107}$$

with all the variables fresh, and there exists C_{w_2} such that

$$C'_{w_2} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_2} \Rightarrow \tau'_{w_2}) \geq (\Delta \mid \tau'_2) \tag{A.108}$$

that is, by Definition 6.5, there exist S'_{w_2} and v'_{w_2} such that

$$\tau'_2 = S'_{w_2}\, \tau'_{w_2} \tag{A.109}$$

$$h : \Delta \Vdash v'_{w_2} : S'_{w_2}\, \Delta'_{w_2} \tag{A.110}$$

$$C'_{w_2} = [\,](\!(v'_{w_2})\!) \tag{A.111}$$

We define S'_w such that $\mathrm{dom}(S'_w) = \mathrm{dom}(S'_{w_1}) \cup \mathrm{dom}(S'_{w_2})$ and $S'_w\, t = S'_{w_i}\, t$, if $t \in \mathrm{dom}(S'_{w_i})$ (it is well defined because all the variables are fresh). Then, it is easy to check (using A.104, A.105, A.109, and A.110) that

$$\tau'_2 \rightarrow \tau'_1 = S'_w\, (\tau'_{w_2} \rightarrow \tau'_{w_1}) \tag{A.112}$$

$$h : \Delta \Vdash v'_{w_1} : S'_{w_1}\, \Delta'_{w_1}, v'_{w_2} : S'_{w_2}\, \Delta'_{w_2} \tag{A.113}$$

and we can define $C'_w = [\,](\!(v'_{w_1}, v'_{w_2})\!)$.

By application of the corresponding rule in the algorithm to A.102 and A.107, we can infer that

$$\Delta'_{w_1}, \Delta'_{w_2} \vdash_{\text{W-SR}} \tau_2 \rightarrow^{D} \tau_1 \;\hookrightarrow\; \tau'_{w_2} \rightarrow \tau'_{w_1} \tag{A.114}$$

and the variables are fresh, as obtained by IH. Finally,

$$C'_w : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_1}, \Delta'_{w_2} \Rightarrow \tau'_{w_2} \rightarrow \tau'_{w_1}) \geq (\Delta \mid \tau'_2 \rightarrow \tau'_1) \tag{A.115}$$

by Definition 6.5 using A.112 and A.113.

Case (SR-TUPLE): Analogous to (SR-DFUN).

Case (SR-POLY): We know that

$$\text{(SR-POLY)}\quad \frac{\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma' \qquad \Delta \Vdash \mathrm{IsMG}\; \sigma'\; \sigma}{\Delta \vdash_{\text{SR}} \mathbf{poly}\; \tau \;\hookrightarrow\; \mathbf{poly}\; \sigma}$$

By IH on the first premise,

$$\Delta'_{w_I} \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau'_{w_I} \tag{A.116}$$

with all the variables fresh, and there exist C'_{w_I} such that

$$C'_{w_I} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I}) \geq (\Delta \mid \sigma') \tag{A.117}$$

Let's call σ_{w_I} to $\mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I})$.

Using (IsMG) on A.117, we have that

$$\Delta \Vdash \mathrm{IsMG}\; \sigma_{w_I}\; \sigma'$$

and then by (Comp) on this and the second premise,

$$\Delta \Vdash \mathrm{IsMG}\; \sigma_{w_I}\; \sigma \tag{A.118}$$

The first part of the result follows from the application of the corresponding rule of the algorithm to A.116, and the second part by Definition 6.5 using A.118 for the entailment needed.

Case (SR-QIN): We know that

$$\text{(SR-QIN)}\quad \frac{\Delta, \delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho}$$

By IH on the premise,

$$\Delta'_{w_I} \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_{w_I} \tag{A.119}$$

with all the variables fresh, and there exist C'_{w_I} such that

$$C'_{w_I} : \text{Gen}_{\emptyset,\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I}) \geq (\Delta, \delta \mid \rho) \tag{A.120}$$

that is, by Definition 6.5, there exist S'_{w_I} and v'_{w_I} such that

$$\tau_\rho = S'_{w_I}\, \tau'_{w_I} \tag{A.121}$$

$$h : \Delta, h_\delta : \delta, h_\rho : \Delta_\rho \Vdash v'_{w_I} : S'_{w_I}\, \Delta'_{w_I} \tag{A.122}$$

$$C'_{w_I} = \Lambda h_\rho.[]((v'_{w_I})) \tag{A.123}$$

(assuming $\rho = \Delta_\rho \Rightarrow \tau_\rho$).

The first part of the result is exactly A.119. The second part follows from Definition 6.5 using A.121, A.122, and the conversion $C'_w = \Lambda h_\delta, h_\rho.[]((v'_{w_I}))$.

Case (SR-QOUT): We know that

$$\text{(SR-QOUT)}\quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}$$

By IH on the first premise,

$$\Delta'_{w_I} \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_{w_I} \tag{A.124}$$

with all the variables fresh, and there exist C'_{w_I} such that

$$C'_{w_I} : \text{Gen}_{\emptyset,\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I}) \geq (\Delta \mid \delta \Rightarrow \rho) \tag{A.125}$$

that is, by Definition 6.5, there exist S'_{w_I} and v'_{w_I} such that

$$\tau_\rho = S'_{w_I}\, \tau'_{w_I} \tag{A.126}$$

$$h : \Delta, h_\delta : \delta, h_\rho : \Delta_\rho \Vdash v'_{w_I} : S'_{w_I}\, \Delta'_{w_I} \tag{A.127}$$

$$C'_{w_I} = \Lambda h_\delta, h_\rho.[]((v'_{w_I})) \tag{A.128}$$

(assuming $\rho = \Delta_\rho \Rightarrow \tau_\rho$).

By (Trans) on the second premise and A.127

$$h : \Delta, h_\rho : \Delta_\rho \Vdash v'_{w_I}[h_\delta/v_\delta] : S'_{w_I}\, \Delta'_{w_I} \tag{A.129}$$

The first part of the result is exactly A.124. The second part follows from Definition 6.5 using A.126, A.129, and the conversion $C'_w = \Lambda h_\rho.[]((v'_{w_I}[h_\delta/v_\delta]))$.

Case (SR-GEN): We know that

$$\text{(SR-GEN)}\quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall\alpha.\sigma}\ (\alpha \notin FV(\Delta))$$

By IH on the first premise,

$$\Delta'_{w_I} \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_{w_I} \tag{A.130}$$

with all the variables fresh, and there exist C'_{w_I} such that

$$C'_{w_I} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I}) \geq (\Delta \mid \sigma) \tag{A.131}$$

The first part of the result is exactly A.130. The second part follows immediately from A.131, because $\alpha \notin FV(\Delta)$.

Case (SR-INST): We know that

$$\text{(SR-INST)}\quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall\alpha.\sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow S\,\sigma}\ (\mathrm{dom}(S){=}\alpha)$$

By IH on the first premise,

$$\Delta'_{w_I} \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_{w_I} \tag{A.132}$$

with all the variables fresh, and there exist C'_{w_I} such that

$$C'_{w_I} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I}) \geq (\Delta \mid \forall\alpha.\sigma) \tag{A.133}$$

that is, by Definition 6.5, there exist S'_{w_I} and v'_{w_I} such that

$$\tau_\sigma = S'_{w_I}\,\tau'_{w_I} \tag{A.134}$$

$$h : \Delta, h_\sigma : \Delta_\sigma \Vdash v'_{w_I} : S'_{w_I}\,\Delta'_{w_I} \tag{A.135}$$

$$C'_{w_I} = \Lambda h_\sigma.[]((v'_{w_I})) \tag{A.136}$$

(assuming $\sigma = \forall\alpha_i.\Delta_\sigma \Rightarrow \tau_\sigma$).

By (Close) on A.135 (and using the fact that α does not appear free on Δ or $S'_{w_I}\,\Delta'_{w_I}$), we have that

$$h : \Delta, h_\sigma : S\,\Delta_\sigma \Vdash v'_{w_I} : S'_{w_I}\,\Delta'_{w_I} \tag{A.137}$$

The first part of the result is exactly A.132. The second part follows from Definition 6.5 on A.134, A.137, and A.136.

■

A.22 Proof of theorem 7.17 from section 7.2

THEOREM 7.17. *If* $\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e' : \tau'$ *then* $\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e' : \tau'$.

Proof: By induction on the W derivation. We have to use Propositions 7.7 and 7.8, Proposition 7.13, Proposition 7.14, Propositions 6.13 and 6.14, and Proposition 7.11 — in addition to the IH — depending on the premises on each of the rules.

■

A.23 Proof of theorem 7.18 from section 7.2

THEOREM 7.18. *If* $h : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e' : \tau'$, *then* $h'_w : \Delta'_w \mid T'_w\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e'_w : \tau'_w$ *and there exist a substitution* R *and evidence* v'_w *such that*

a) $S \approx RT'_w$

b) $\tau' = R\,\tau'_w$

c) $h : \Delta \Vdash v_w : R\,\Delta'_w$

d) $e' = e'_w[h'_w/v'_w]$

Proof: By induction on the S derivation. We show the interesting cases; the rest are shown analogously.

Case (S-VAR)**:** We know that

$$\text{(S-VAR)}\quad \frac{x : \tau \hookrightarrow e' : \tau' \in S\,\Gamma}{\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} x : \tau \;\hookrightarrow\; e' : \tau'}$$

From the premise, we can conclude, by definition of substitution on assignments, that there exists τ'_s such that

$$x : \tau \hookrightarrow e' : \tau'_s \in \Gamma \tag{A.138}$$

and

$$S\,\tau'_s = \tau' \tag{A.139}$$

By (W-VAR) on A.138,

$$\emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} x : \tau \;\hookrightarrow\; e' : \tau'_s \tag{A.140}$$

Let's take $R = S$.

Item **a)** holds trivially from the definition of Id.

$$S \approx S\mathrm{Id}$$

Item **b)** holds by A.139.

$$\tau' = S\,\tau'_s$$

Item **c)** holds trivially from the definition of entailment.

$$\Delta \Vdash \emptyset$$

Item **d)** holds trivially (because there are no evidence variables to replace).

$$e' = e'$$

Case (S-D+)**:** We know that

$$\text{(S-D+)} \quad \frac{(\Delta \mid S\,\Gamma \vdash_{\text{S}} e_i : \mathit{Int}^{D} \;\hookrightarrow\; e'_i : \mathit{Int})_{i=1,2}}{\Delta \mid S\,\Gamma \vdash_{\text{S}} e_1 +^{D} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e'_1 + e'_2 : \mathit{Int}}$$

By IH on the premise, with $i = 1$,

$$h'_{w_1} : \Delta'_{w_1} \mid T'_{w_1}\,\Gamma \vdash_{\text{W}} e_1 : \mathit{Int}^{D} \;\hookrightarrow\; e'_{w_1} : \mathit{Int} \tag{A.141}$$

and there exist R_1 and v'_{w_1} such that

$$S \approx R_1 T'_{w_1} \tag{A.142}$$
$$h : \Delta \Vdash v'_{w_1} : R_1\,\Delta'_{w_1} \tag{A.143}$$
$$e'_1 = e'_{w_1}[h'_{w_1}/v'_{w_1}] \tag{A.144}$$

Writing $S\,\Gamma$ as $R_1\,(T'_{w_1}\,\Gamma)$ by A.142, we can use HI on the premise with $i = 2$ to get

$$h'_{w_2} : \Delta'_{w_2} \mid T'_{w_2}\,(T'_{w_1}\,\Gamma) \vdash_{\text{W}} e_1 : \mathit{Int}^{D} \;\hookrightarrow\; e'_{w_2} : \mathit{Int} \tag{A.145}$$

and there exist R_2 and v'_{w_2} such that

$$R_1 \approx R_2 T'_{w_2} \tag{A.146}$$
$$h : \Delta \Vdash v'_{w_2} : R_2\,\Delta'_{w_2} \tag{A.147}$$
$$e'_2 = e'_{w_2}[h'_{w_2}/v'_{w_2}] \tag{A.148}$$

Using (W-D+) on A.141 and A.145,

$$h'_{w_1} : T'_{w_2}\,\Delta'_{w_1}, h'_{w_2} : \Delta'_{w_2} \mid T'_{w_2} T'_{w_1}\,\Gamma \vdash_{\text{W}} e_1 +^{D} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e'_{w_1} + e'_{w_2} : \mathit{Int} \tag{A.149}$$

Let's take $R = R_2$ and $v_w = (v'_{w_1}, v'_{w_2})$.

Item **a)** holds by A.142 and A.146.

$$S \approx R_2 T'_{w_2} T'_{w_1}$$

Item **b)** holds trivially.

$$\mathit{Int} = R_2\,\mathit{Int}$$

Item **c)** holds by A.143 (using A.146) and A.147.

$$h : \Delta \Vdash v'_{w_1} : R_2\,(T'_{w_2}\,\Delta'_{w_1}), v'_{w_2} : R_2\,\Delta'_{w_2}$$

Item **d)** holds by A.144 and A.148, using Lemma 7.16 on A.141 and A.145.

$$e'_1 + e'_2 = (e'_{w_1} + e'_{w_2})[h'_{w_1}, h'_{w_2} / v'_{w_1}, v'_{w_2}]$$

Case (S-DLAM): We know that

$$\text{(S-DLAM)} \quad \frac{\Delta \mid S\,\Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\mathrm{S}} e : \tau_1 \;\hookrightarrow\; e' : \tau'_1 \qquad \Delta \vdash_{\mathrm{SR}} \tau_2 \;\hookrightarrow\; \tau'_2}{\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} \lambda^{D} x.e : \tau_2 \to^{D} \tau_1 \;\hookrightarrow\; \lambda x'.e' : \tau'_2 \to \tau'_1} \quad (x' \textit{ fresh})$$

Let t_2 be a fresh variable and define S' such that $S'\,t_2 = \tau'_2$ and $S'\,t' = S\,t$ for all $t \neq t_2$. So, the derivation for e in the first premise can be rewritten

$$\Delta \mid S'\,(\Gamma, x : \tau_2 \hookrightarrow x' : t_2) \vdash_{\mathrm{S}} e : \tau_1 \;\hookrightarrow\; e' : \tau'_1 \tag{A.150}$$

By IH on A.150, we have that

$$h'_{w_1} : \Delta'_{w_1} \mid T'_{w_1}\,\Gamma \vdash_{\mathrm{W}} e : \tau_1 \;\hookrightarrow\; e'_{w_1} : \tau'_{w_1} \tag{A.151}$$

and there exist R_1 and v'_{w_1} such that

$$S' \approx R_1 T'_{w_1} \tag{A.152}$$
$$\tau'_1 = R_1\,\tau'_{w_1} \tag{A.153}$$
$$h : \Delta \Vdash v'_{w_1} : R_1\,\Delta'_{w_1} \tag{A.154}$$
$$e' = e'_{w_1}[h'_{w_1} / v'_{w_1}] \tag{A.155}$$

By Proposition 7.15 on the second premise,

$$\Delta'_{w_2} \vdash_{\mathrm{W\text{-}SR}} \tau_2 \;\hookrightarrow\; \tau'_{w_2} \tag{A.156}$$

with all variables fresh, and there exists $C_{w_2} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w_2} \Rightarrow \tau'_{w_2}) \geq (\Delta \mid \tau'_2)$, that is (by Definition 6.5), there exist R_2 and v'_{w_2} such that

$$\tau'_2 = R_2\,\tau'_{w_2} \tag{A.157}$$
$$h : \Delta \Vdash v'_{w_2} : R_2\,\Delta'_{w_2} \tag{A.158}$$
$$C'_{w_2} = [\,]((v'_{w_2})) \tag{A.159}$$

By (W-DLAM) on A.151 and A.156,

$$\begin{aligned} h'_{w_1} : \Delta'_{w_1}, h'_{w_2} : T'_{w_1}\,\Delta'_{w_2} \mid T'_{w_1}\,\Gamma \vdash_{\mathrm{W}} \lambda^{D} x.e &: \tau_2 \to^{D} \tau_1 \\ \hookrightarrow\; \lambda x'.e'_{w_1} &: (T'_{w_1}\,\tau'_{w_2}) \to \tau'_{w_1} \end{aligned} \tag{A.160}$$

Additionally, as the variables in $\Delta'_{w_2} \Rightarrow \tau'_{w_2}$ are all fresh, we know that $T'_{w_1}\,\Delta'_{w_2} = \Delta'_{w_2}$ and $T'_{w_1}\,\tau'_{w_2} = \tau'_{w_2}$.

Let's take R such that $R\,t = R_2\,t$ if $t \in \mathrm{dom}(R_2)$, and $R\,t = R_1\,t$ if $t \notin \mathrm{dom}(R_2)$, and $v'_w = (v'_{w_1}, v'_{w_2})$.

Item **a)** holds by A.152, and the fact that $S' \approx S$ and $R \approx R_1$ by definition.

$$S \approx R T'_{w_1}$$

Item **b)** holds by A.157, A.153, and the definition of R.

$$\tau'_2 \to \tau'_1 = R\,(\tau'_{w_2} \to \tau'_{w_1})$$

Item **c)** holds by A.154, A.158, and the definition of R.

$$h : \Delta \Vdash v'_{w_1} : R\,\Delta'_{w_1}, v'_{w_2} : R\,\Delta'_{w_2}$$

Item **d)** holds by A.155.

$$\lambda x'.e' = \lambda x'.(e'_{w_I}[h'_{w_1}, h'_{w_2}/v'_{w_1}, v'_{w_2}])$$

Case (S-DAPP)**:** We know that

$$\text{(S-DAPP)}\ \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e_1 : \tau_2 \to^{D} \tau_1 \ \hookrightarrow\ e'_1 : \tau'_2 \to \tau'_1 \quad \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e_2 : \tau_2 \ \hookrightarrow\ e'_2 : \tau'_2}{\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e_1 \,@^{D}\, e_2 : \tau_1 \ \hookrightarrow\ e'_1 @ e'_2 : \tau'_1}$$

By IH we have that

$$h'_{w_1} : \Delta'_{w_1} \mid T'_{w_1}\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_2 \to^{D} \tau_1 \ \hookrightarrow\ e'_{w_1} : \tau'_{w_1} \tag{A.161}$$

and there exist R_1 and v'_{w_1} such that

$$S \approx R_1 T'_{w_1} \tag{A.162}$$
$$\tau'_2 \to \tau'_1 = R_1\,\tau'_{w_1} \tag{A.163}$$
$$h : \Delta \Vdash v'_{w_1} : R_1\,\Delta'_{w_1} \tag{A.164}$$
$$e'_1 = e'_{w_1}[h'_{w_1}/v'_{w_1}] \tag{A.165}$$

Writing $S\,\Gamma$ as $R_1\,(T'_{w_1}\,\Gamma)$ by A.162, we can use HI on the second premise to get

$$h'_{w_2} : \Delta'_{w_2} \mid T'_{w_2}\,(T'_{w_1}\,\Gamma) \vdash_{\mathrm{W}} e_2 : \tau_2 \ \hookrightarrow\ e'_{w_2} : \tau'_{w_2} \tag{A.166}$$

and there exist R_2 and v'_{w_2} such that

$$R_1 \approx R_2 T'_{w_2} \tag{A.167}$$
$$\tau'_2 = R_2\,\tau'_{w_2} \tag{A.168}$$
$$h : \Delta \Vdash v'_{w_2} : R_2\,\Delta'_{w_2} \tag{A.169}$$
$$e'_2 = e'_{w_2}[h'_{w_2}/v'_{w_2}] \tag{A.170}$$

Let's define R' such that $R'\,t = \tau'_1$ for a fresh variable t, and $R'\,t' = R_2\,t'$ for every $t' \neq t$. By A.168 and A.163 (using A.167)

$$R'\,T'_{w_2}\,\tau'_{w_1} = R'\,(\tau'_{w_2} \to t) \tag{A.171}$$

By Proposition 7.12 on A.171 (and the fact that t is fresh), there exist substitutions U and R such that

$$T'_{w_2}\,\tau'_{w_1} \sim^{U} \tau'_{w_2} \to t \tag{A.172}$$

and

$$R' = RU \tag{A.173}$$

Using (W-DAPP) on A.161, A.166, and A.172,

$$\begin{array}{c} h'_{w_1} : UT'_{w_2}\,\Delta'_{w_1}, h'_{w_2} : U\,\Delta'_{w_2} \mid UT'_{w_2}T'_{w_1}\,\Gamma \\ \vdash_{\mathrm{W}} e_1 \mathbin{@^{D}} e_2 : \tau_1 \;\hookrightarrow\; e'_{w_1} @ e'_{w_2} : U\,t \end{array} \tag{A.174}$$

Let's take $v_w = (v'_{w_1}, v'_{w_2})$.

Item **a)** holds by A.162, A.167, A.173, and the fact that $R' \approx R_2$.

$$S \approx RUT'_{w_2}T'_{w_1}$$

Item **b)** holds by A.173 and definition of R'.

$$\tau'_1 = R\,(U\,t)$$

Item **c)** holds by A.173, A.164 (using A.167), A.169, and the fact that $R' \approx R_2$.

$$h : \Delta \Vdash v'_{w_1} : R\,(UT'_{w_2}\,\Delta'_{w_1}), v'_{w_2} : RU\,\Delta'_{w_2}$$

Item **d)** holds by A.165 and A.170, using Lemma 7.16 on A.161 and A.166.

$$e'_1 @ e'_2 = (e'_{w_1} @ e'_{w_2})[h'_{w_1}, h'_{w_2} / v'_{w_1}, v'_{w_2}]$$

Case (S-POLY): We know that

$$\text{(S-POLY)}\quad \frac{h' : \Delta' \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e' : \tau' \quad h : \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{h : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[\Lambda h'.e'] : \mathbf{poly}\ \sigma}\ \scriptstyle{\left(\sigma' = \mathrm{Gen}_{S\,\Gamma,\emptyset}(\Delta' \Rightarrow \tau')\right)}$$

By IH on the first premise, we have that

$$h'_{w_I} : \Delta'_{w_I} \mid T'_{w_I}\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e'_{w_I} : \tau'_{w_I} \tag{A.175}$$

and there exist R_I and v'_{w_I} such that

$$S \approx R_I T'_{w_I} \tag{A.176}$$

$$\tau' = R_I\,\tau'_{w_I} \tag{A.177}$$

$$h' : \Delta' \Vdash v'_{w_I} : R_I\,\Delta'_{w_I} \tag{A.178}$$

$$e' = e'_{w_I}[h'_{w_I}/v'_{w_I}] \tag{A.179}$$

Let's define $\sigma'_{w_I} = \text{Gen}_{(T'_{w_I}\,\Gamma),\emptyset}(\Delta'_{w_I} \Rightarrow \tau'_{w_I})$. Then, for s and h'_w fresh, we have that

$$h'_w : \text{IsMG } \sigma'_{w_I}\ s \mid \emptyset \Vdash_{\text{W}} h'_w : \text{IsMG } \sigma'_{w_I}\ s \qquad \text{(A.180)}$$

On the other hand, by Definition 6.5 on A.177 and A.178.

$$\Lambda h'.[]((v'_{w_I})) : \text{Gen}_{(R_I T'_{w_I}\,\Gamma),\emptyset}(R_I\,\Delta'_{w_I} \Rightarrow R_I\,\tau'_{w_I}) \geq \text{Gen}_{(S\,\Gamma),\emptyset}(\Delta' \Rightarrow \tau') \qquad \text{(A.181)}$$

Finally, by (W-POLY) on A.175 and A.180,

$$h'_w : \text{IsMG } \sigma'_{w_I}\ s \mid T'_{w_I}\,\Gamma \vdash_{\text{W}} \textbf{poly}\ e : \textbf{poly}\ \tau \ \hookrightarrow\ h'_w[\Lambda h'_{w_I}.e'_{w_I}] : \textbf{poly}\ s \qquad \text{(A.182)}$$

Let's take R such that $R\,s = \sigma$ and $R\,t = R_I\,t$ for all $t \neq s$, and $v'_w = v \circ (\Lambda h'.[]((v'_{w_I})))$.

Item **a)** holds by A.176 and because $R \approx R_I$ by definition.

$$S \approx RT'_{w_I}$$

Item **b)** holds trivially, by definition of R.

$$\textbf{poly}\ \sigma = \textbf{poly}\ (R\,s)$$

Item **c)** holds by using (Comp) on A.181 (combining it with Proposition 7.3-3) and the second premise.

$$h : \Delta \Vdash v'_w : \text{IsMG } (R\,\sigma'_{w_I})\ \sigma$$

Item **d)** holds by definition of v'_w, (β_v), and A.179.

$$v[\Lambda h'.e'] = (h'_w[\Lambda h'_{w_I}.e'_{w_I}])[h'_w/v'_w]$$

■

A.24 Proof of lemma 8.7 from section 8.1

LEMMA 8.7. *Let $T; C \mid \Delta \trianglerighteq \Delta'$ be a simplification for Δ. If S and T are compatible under Δ, i.e. $S \sim_\Delta T$, then $T; C \mid S\Delta \trianglerighteq S\Delta'$ is a simplification for $S\Delta$.*

Proof: If $T; C \mid \Delta \trianglerighteq \Delta'$, then $h' : \Delta' \Vdash v : T\Delta$ y $T\Delta \Vdash \Delta$. Applying $\Vdash_{Clos}$ we obtain $h' : S\Delta' \Vdash v : ST\Delta$ and $ST\Delta \Vdash S\Delta$. As $S \sim_\Delta T$, this is equivalent to $h' : S\Delta' \Vdash v : TS\Delta$ and $TS\Delta \Vdash S\Delta$, which is the same as $T; C \mid S\Delta \trianglerighteq S\Delta'$.

■

A.25 Proof of theorem 8.8 from section 8.1

THEOREM 8.8. *The rules* (SimEntl), (SimTrans), (SimCtx), *and* (SimPerm) *(of Figure 8.1) define a simplification relation, and the derived rule* (SimCHAM) *is consistent with it.*

Proof: By induction in the derivation of the simplification:

- (SimEntl): Using the hypotesis $h : \Delta \Vdash v_\delta : \delta$, we obtain, by (Univ) that (i) $h : \Delta \Vdash h : \Delta, v_\delta : \delta$. Similarly, it is also true that (ii) $h : \Delta, h_\delta : \delta \Vdash v_\delta : \delta$. Moreover, the conversion statisfies the conditions in the definition, because (iii) $h_\delta{\leftarrow}\delta = h{\leftarrow}h \cdot h_\delta{\leftarrow}\delta$, as it has been proved in Lemma 8.2.

- (SimTrans): The Inductive Hypothesis are those corresponding to the definition of simplification on the premises of the rule: (i) $h' : \Delta' \Vdash v : S\Delta$ y $h'' : \Delta'' \Vdash v' : S'\Delta'$; (ii) $S\Delta \Vdash \Delta'$ y $S'\Delta' \Vdash \Delta''$; (iii) $C = h{\leftarrow}v$ y $C' = h'{\leftarrow}v'$. We obtain the evidence $v^* = v[v'/h']$.
  To obtain (i) the two IH from the first condition is used. Applying (Close) (from Figure 5.4) with substitution S' on the first hypothesis, it holds that $h' : S'\Delta' \Vdash v : S'S\Delta$. Then, by (Trans) on this last sentence and the second hypothesis, we get $h'' : \Delta'' \Vdash v[v'/h'] : S'S\Delta$, as needed (observe that the evidence thus obtained is $v[v'/h'] = v^*$).
  To prove (ii) the two corresponding IH is considered. Applying the same rules as in the previous case, it holds that $S'S\Delta \Vdash \Delta''$.
  Finally, (iii) holds by observing the composition of conversions from the hypothesis: $(C' \circ C)[e'] = e'[{}^{v[v'/h']}/_{h}] = (h{\leftarrow}v^*)[e']$, then $C' \circ C = h{\leftarrow}v^*$ as needed.

- (SimCtx): The Inductive Hypotesis establish that $h_2 : \Delta_2 \Vdash v : S\Delta_1$ and $S\Delta_1 \Vdash \Delta_2$. adding $h' : S\Delta'$ to the first one, and $S\Delta'$ to the second one, the conditions for (i) and (ii) are obtained. It should be noted that the original conversion, $h_1{\leftarrow}v_1$, is equal (under the congruence used) to the conversion needed for the rule: $h_1, h'{\leftarrow}v, h'$

- (SimPerm): This rule is an extension of the closure property for permutations in the predicate assignments in the entailment relationship. The only difficulties in the proof are that the conversion C^* may not be the same (because the order of abstraction and application of evidence matters), and the manipulation of predicates.
  The condition (i) is obtained immediately from the corresponding IH.
  Given the following IH

  $$h_1' : \Delta_1', h_2' : \Delta_2' \Vdash v_1 : S\Delta_1, v_2 : S\Delta_2,$$

  condition (ii) is obtained in the following way: the left hand side is permuted, and the right hand side is enlarged with $v_2 : \Delta_2$), it holds that

  $$h_2' : \Delta_2', h_1' : \Delta_1' \Vdash h_1' : \Delta_1', h_2' : \Delta_2', v_2 : S\Delta_2.$$

Using once more the original hypothesis, this time for $v_1 : S\Delta_1$, and adding $h_2 : \Delta_2$ to its left hand side, it holds that

$$h'_1 : \Delta'_1, h'_2 : \Delta'_2, h_2 : S\Delta_2 \Vdash v_1 : S\Delta_1.$$

The last two judgments can be composed using (Trans), to obtain

$$h'_2 : \Delta'_2, h'_1 : \Delta'_1 \Vdash v_1[h'_1/h'_1][h'_2/h'_2][v_2/h_2] : S\Delta_1$$

wich is equivalent to $h'_2 : \Delta'_2, h'_1 : \Delta'_1 \Vdash v_1[v_2/h_2] : S\Delta_1$. Additionally, the IH implies that $h'_2 : \Delta'_2, h'_1 : \Delta'_1 \Vdash v_2 : S\Delta_2$, and by (Univ), the judgment

$$h'_2 : \Delta'_2, h'_1 : \Delta'_1 \Vdash v_2 : S\Delta_2, v_1[h_2/v_2] : S\Delta_1$$

is obtained, and thus the condition (ii) .

So, we have shown the first part (structural rules define a simplification relation).

To conclude the proof, we show how to obtain rule (SimCHAM): If $\Delta_1 \approx \Delta'_1$ and $\Delta_2 \approx \Delta'_2$, then Δ_1 can be obtained in a finite number of exchanges between sublists of Δ'_1, in the way established by (SimPerm) — and similarly for Δ_2 y Δ'_2. In this way, if $S; C \mid \Delta_1 \trianglerighteq \Delta_2$ then $S; C^{\approx} \mid \Delta'_1 \trianglerighteq \Delta'_2$ to the right conversion (as it is explained in the definition of the rule). A final application of (SimCtx) allows obtaining $S; C^{\approx} \mid \Delta'_1, \Delta \trianglerighteq \Delta'_2, S\Delta$, as needed.

■

A.26 Proof of theorem 8.9 from section 8.1

THEOREM 8.9. *A system defining a simplification relation, extended with rules* (SimOp$_{res}$) *and* (SimMG$_U$) *still defines a simplification relation.*

Proof: The proof is, as in the case of basic rules, by induction on the derivation, taking one case for each rule:

- (SimOp$_{res}$): (i) It holds that $\emptyset \Vdash n : \hat{n} := \hat{n_1} + \hat{n_2}$, and also (ii) $\hat{n} := \hat{n_1} + \hat{n_2} \Vdash \emptyset$. The conversion is exactly as required.
- (SimMG$_U$): (i) As $C : \sigma_2 \geq \sigma_1$ by (IsMG), it holds that IsMG $\sigma_2\ \sigma_1$. Additionally, h_1 : IsMG $\sigma_1\ s$, so applying (Trans), $h_1 \circ C$: IsMG $\sigma_2\ s$ is obtained. (ii) holds trivially, with the required conversion $C = h_2 \leftarrow (h_1 \circ C) = h_1 \leftarrow h_2 \cdot h_2 \leftarrow (h_1 \circ C)$.

■

A.27 Proof of theorem 8.15 from section 8.2

THEOREM 8.15. *Given* $\Delta_1 \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, *and if* $S, T; C \mid \Delta_1 + \Delta' \vartriangleright_{FTV(\Gamma,\sigma)} \Delta_2$ *then, it is also the case that*

$$\Delta_2 \mid TS\Gamma \vdash_{\text{P}} e : \tau \hookrightarrow C[e'] : TS\sigma.$$

Proof: The premise $S, T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2$ implies, by definition of resoulution, that the following simplification holds

$$T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2$$

where $\mathrm{dom}(S) \cap (FTV(\Gamma, \sigma) \cup FTV(\Delta')) = \emptyset$.
Enriching the context of the hypotesis $\Delta_1 \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ with Δ' (by Th. A.12 of [Martínez López and Hughes, 2001]) the following judgment is obtained:

$$\Delta_1, \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma.$$

Then, applying the substitution S to the judgment (Proposition 6.22) it holds that

$$S\Delta_1, S\Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\sigma.$$

Because of the restrictions in the domain of S, we know that $S\Delta' = \Delta'$, $S\Gamma = \Gamma$ y $S\sigma = \sigma$, and then, the previous judgment is equivalent to

$$S\Delta_1, \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma.$$

Finally, using the premise $T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2$, the rule (SIMP) can be applied, obtaining

$$\Delta_2 \mid T\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow C[e'] : T\sigma$$

as needed.

■

A.28 Proof of lemma 8.16 from section 8.2

LEMMA 8.16. *Composition of solvings is a solving.*
That is, if $S_2 \sim_{S_1\Delta_1,\Delta'} T_1$, $S_1, T_1; C_1 \mid \Delta_1 + \Delta' \rhd_V \Delta_2$, *and* $S_2, T_2; C_2 \mid \Delta_2 + \Delta'' \rhd_V \Delta_3$ *then*

$$S_2S_1, T_2T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \rhd_V \Delta_3$$

Proof: By hypothesis in the first resolution, it holds that

$$T_1; C_1 \mid S_1\Delta_1, \Delta' \trianglerighteq \Delta_2.$$

By comptibility between S_2 and T_1 (Lemma 8.7), substitution S_2 can be applied to this simplification, obtaining

$$T_1; C_1 \mid S_2S_1\Delta_1, S_2\Delta' \trianglerighteq S_2\Delta_2.$$

Enlarging both lists of predicates, using (SimCHAM), we obtain

$$T_1; C_1 \mid S_2S_1\Delta_1, S_2\Delta', \Delta'' \trianglerighteq S_2\Delta_2, \Delta'',$$

Moreover, the first resolution establish that

$$T_2; C_2 \mid S_2\Delta_2, \Delta'' \trianglerighteq \Delta_3,$$

and, by composing the last two simplifications (rule (SimTrans)), given that the required restrictions on the substitutions hold by hypothesis, it is true that

$$S_2S_1, T_2T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \rhd_V \Delta_3$$

■

Bibliography

Amongst them in the Tower's high room Kurremkarmerruk sat on a high seat, writing down lists of names that must be learned before the ink faded at midnight leaving the parchment blank again.

A Wizard of Earthsea
Úrsula K. Le Guin

[Abelson *et al.*, 1998] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo J. Rozas, Norman I. Adams IV, Daniel P. Friedman, Eugene Kohlbecker, Guy Lewis Steele Jr., David H. Bartley, Robert Halstead, Don Oxley, Gerald Jay Sussman, Gary Brooks, Chris Hanson, Kent M. Pitman, and Mitchell Wand. Revised^5 report on the algorithmic language Scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7–105, August 1998.

[Aiken, 1999] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

[Andersen, 1992] Lars Ole Andersen. Self-applicable C program specialisation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, (PEPM '92)*, pages 54–61, San Francisco, California, USA, June 1992. Yale University. Technical Report YALEU/DCS/RR-909.

[Andersen, 1993] Lars Ole Andersen. Binding-time analysis and the taming of C pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, (PEPM '93)*, pages 47–58, Copenhagen, Denmark, June 1993. ACM.

[Au *et al.*, 1991] Wing-Yee Au, Daniel Weise, and Scott Seligman. Automatic generation of compiled simulations through program specialization. In *Proceedings of the 28th Design Automation Conference*, pages 205–210, San Francisco, California, USA, June 1991. IEEE Computer Society Press.

[Augustsson, 1997] Lennart Augustsson. Partial evaluation in aircraft crew planning. In Gallagher [1997], pages 127–136.

[Augustsson, 1998] Lennart Augustsson. Partial evaluation in aircraft crew planning. In Hatcliff et al. [1998], pages 231–245. Verbatim reprint of [Augustsson, 1997].

[Barzdins and Bulyonkov, 1988] G. J. Barzdins and M. A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers, Preprint 791 from Computing Centre of Siberian division of USSR Academy of Sciences, 1988.

[Berlin and Surati, 1994] Andrew A. Berlin and Rajeev J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pages 133–141, Orlando, Florida, USA, June 1994.

[Berlin and Weise, 1990] Andrew A. Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[Berlin, 1990] Andrew A. Berlin. Partial evaluation applied to numerical computation. In Wand [1990], pages 139–150.

[Berry and Boudol, 1990] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, 1990.

[Beshers and Feiner, 1997] Clifford Beshers and Steven Feiner. Generating efficient virtual worlds for visualization using partial evaluation and dynamic compilation. In Gallagher [1997], pages 107–115.

[Bjørner *et al.*, 1988] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*, North Holland, 1988. IFIP World Congress Proceedings, Elsevier Science Publishers B.V.

[Bjørner *et al.*, 1996] Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors. *Perspectives of System Informatics: Proceedings of the 2nd International Andrei Ershov Memorial Conference*, volume 1181 of *Lecture Notes in Computer Science (LNCS)*, Akademgorodok, Novosibirsk, Russia, June 1996. Springer-Verlag.

[Bjørner *et al.*, 2001] Dines Bjørner, Manfred Broy, and Alexander V. Zamulin, editors. *Perspectives of System Informatics: Proceedings of 4th International Andrei Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science (LNCS)*, Akademgorodok, Novosibirsk, Russia, July 2001. Springer-Verlag.

[Bondorf and Danvy, 1991] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, October 1991.

[Bondorf, 1991] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991.

[Bondorf, 1993] Anders Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, Denmark, May 1993. Available at
URL: `ftp://ftp.diku.dk/pub/diku/semantics/similix/`,
file: `similix-manual-5.0.ps.gZ`.

[Breazu-Tannen *et al.*, 1989] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and coercion. In *Proceedings of the fourth annual symposium on logic in computer science*, 1989.

[Breazu-Tannen *et al.*, 1991] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[Bulyonkov, 1984] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[Bulyonkov, 1988] Mikhail A. Bulyonkov. A theoretical approach to polyvariant mixed computation. In Bjørner et al. [1988], pages 51–64.

[Burstall and Darlington, 1977] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Chirokoff *et al.*, 1999] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. *Higher Order Symbol. Computation*, 12(4):309–335, 1999.

[Clément *et al.*, 1986] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 13–27, Cambridge, Massachussets, USA, August 1986. ACM Press.

[Consel and Danvy, 1991] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [1991], pages 496–519.

[Consel and Danvy, 1993] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of 20th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL '93)*, pages 493–501, Charleston, South Carolina, USA, January 1993. ACM Press.

[Consel, 1990a] Charles Consel. Binding time analysis for higher order untyped functional languages. In Wand [1990], pages 264–272.

[Consel, 1990b] Charles Consel. The Schism manual, version 1.0, December 1990.

[Curry and Feys, 1958] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North Holland, Amsterdam, 1958.

[Damas and Milner, 1982] Luis Damas and Robin Milner. Principal type-schemes for functional languages. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, January 1982.

[Danvy and Filinski, 1990] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [1990], pages 151–160.

[Danvy and Filinski, 1992] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[Danvy and Filinski, 2001] Olivier Danvy and Andrzej Filinski, editors. *Programs as Data Objects II*, volume 2053 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, May 2001.

[Danvy and Martínez López, 2003] Olivier Danvy and Pablo E. Martínez López. Tagging, encoding, and Jones optimality. In Pierpaolo Degano, editor, *Proceedings of the European Symposium on Programming (ESOP 2003), part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 2618 of *Lecture Notes in Computer Science (LNCS)*, pages 335–347, Warsaw, Poland, April 2003. Springer-Verlag.

[Danvy *et al.*, 1996] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Selected papers of the International Seminar "Partial Evaluation"*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, February 1996. Springer-Verlag, Heidelberg, Germany.

[Danvy, 1996] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of 23rd ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL '96)*, pages 242–257, St. Petersburg Beach, Florida, USA, January 1996. ACM Press.

[Danvy, 1998a] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science (LNCS)*, pages 908–917. Springer-Verlag, 1998.

[Danvy, 1998b] Olivier Danvy. Type-directed partial evaluation. In Hatcliff et al. [1998], pages 367–411.

[de Bruijn, 1972] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the church-rosser theorem. *Indag. Mat.*, 5(35), 1972.

[de Bruijn, 1978] Nicolaas G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report 78-WSK-03, Eindhoven University of Technology, 1978.

[Dussart *et al.*, 1995] Dirk Dussart, Eddy Bevers, and Karel de Vlaminck. Polyvariant constructor specialisation. In William L. Scherlis, editor, *Proceedings of the ACM SIGPLAN*

Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '95), pages 54–65, La Jolla, California, USA, June 1995. ACM Press.

[Dussart *et al.*, 1997a] Dirk Dussart, Rogardt Heldal, and John Hughes. Module-sensitive program specialisation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)*, Las Vegas, Nevada, USA, June 1997. ACM SIGPLAN.

[Dussart *et al.*, 1997b] Dirk Dussart, John Hughes, and Peter Thiemann. Type specialisation for imperative languages. In *International Conference on Functional Programming (ICFP'97)*, pages 204–216. ACM Press, June 1997.

[Ershov, 1982] Andrei P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[Ershov, 1988] Andrei P. Ershov. Opening key-note speech. *New Generation Computing*, 6(2 and 3), 1988. Also in [Bjørner *et al.*, 1988], pages 133-151.

[Felleisen, 1988] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of 15th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL '88)*, pages 180–190, San Diego, California, USA, January 1988. ACM Press.

[Fuller and Abramsky, 1988] David A. Fuller and Samson Abramsky. Mixed computation of Prolog programs. *New Generation Computing*, 6(2-3):119–141, 1988.

[Futamura and Nogi, 1988a] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In D. Bjørner, A.P.Ershov, and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[Futamura and Nogi, 1988b] Yoshihiko Futamura and Kenroku Nogi. Program evaluation and generalized partial computation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 685–692, 1988.

[Futamura *et al.*, 1991] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.

[Futamura *et al.*, 2002] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 20(1):75–99, January 2002.

[Futamura, 1971] Yoshihiko Futamura. Partial evaluation of computation process - An approach to a compiler-compiler. *Computer, Systems, Controls*, 2(5):45–50, 1971.

[Gallagher, 1997] John Gallagher, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM '97)*, Amsterdam, The Netherlands, June 1997. ACM.

[Ganzinger and Jones, 1985] Harald Ganzinger and Neil D. Jones, editors. *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science (LNCS)*, Copnehagen, Denmark, October 1985. Springer-Verlag.

[Glenstrup and Jones, 1996] Arne John Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In Bjørner et al. [1996], pages 273–284.

[Gomard and Jones, 1991] Carster K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. In *Journal of Functional Programming*, volume 1 of *1*, pages 21–70, January 1991.

[Hannan and Hicks, 1998] John J. Hannan and Patrick Hicks. Higher-order arity raising. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 27–38. ACM Press, September 1998.

[Hannan and Miller, 1990] John Hannan and Dale Miller. From operational semantics to abstract machines. In *Proceedings of 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, June 1990. New York ACM.

[Hannan and Miller, 1992] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[Hatcliff *et al.*, 1998] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation - Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science (LNCS)*, Copenhagen, Denmark, June 1998. Springer-Verlag.

[Heldal and Hughes, 1997] Rogardt Heldal and John Hughes. Partial evaluation and separate compilation. In Gallagher [1997], pages 1–11.

[Heldal and Hughes, 2000] Rogardt Heldal and John Hughes. Extending a partial evaluator which supports separate compilation. *Theoretical Computer Science*, 248(1-2):99–145, December 2000.

[Heldal and Hughes, 2001] Rogardt Heldal and John Hughes. Binding-time analysis for polymorphic types. In Bjørner et al. [2001], pages 191–204.

[Heldal, 2001] Rogardt Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator*. PhD thesis, Chalmers University of Technology and Göteborg University, 2001.

[Helsen and Thiemann, 2000] Simon Helsen and Peter Thiemann. Fragmental specialization. [**?**], pages 51–71.

[Helsen and Thiemann, 2004] Simon Helsen and Peter Thiemann. Polymorphic specialization for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(4):652–701, July 2004.

[Henglein, 1991] Fritz Henglein. Efficient type inference for higher-order binding time analysis. In Hughes [1991], pages 448–472.

[Hindley, 1969] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[Hogg, 1996] Jonathan A. H. Hogg. Dynamic hardware generation mechanism based on partial evaluation. In *Proceedings of 3rd Workshop on Designing Correct Circuits (DCC '96)*, Electronic Workshops in Computing, Båstad, Sweden, September 1996. Springer-Verlag.

[Holst, 1988] Carsten Kehler Holst. Language triplets: The AMIX approach. In Bjørner et al. [1988], pages 167–186.

[Hughes, 1991] John Hughes, editor. *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science (LNCS)*, Cambridge, Massachussets, August 1991. Springer-Verlag.

[Hughes, 1995] John Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science (LNCS)*, pages 53–96. Springer-Verlag, May 1995.

[Hughes, 1996a] John Hughes. An introduction to program specialisation by type inference. In *Functional Programming*. Glasgow University, July 1996. Published electronically.

[Hughes, 1996b] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [1996], pages 183–215.

[Hughes, 1998a] John Hughes. A type specialisation tutorial. In Hatcliff et al. [1998], pages 293–325.

[Hughes, 1998b] John Hughes. Type specialiser prototype, 1998.
URL: `http://www.cs.chalmers.se/~rjmh/TypeSpec2/`.

[Hughes, 1998c] John Hughes. Type specialization. In *ACM Computing Surveys*, volume 30. ACM Press, September 1998. Article 14. Special issue: electronic supplement to the September 1998 issue.

[Hughes, 2000] John Hughes. The correctness of type specialisation. In Smolka [2000], pages 215–229.

[Hutton and Meijer, 1998] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[Jones *et al.*, 1985] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science (LNCS)*, pages 124–140, Dijon, France, May 1985. Springer-Verlag.

[Jones *et al.*, 1989] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, February 1989.

[Jones *et al.*, 1993] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science, 1993.
Available online at URL: `http://www.dina.dk/~sestoft/pebook/pebook.html`.

[Jones, 1988a] Neil D. Jones. Automatic program specialization: A re-examination from first principles. In Bjørner et al. [1988], pages 225–282.

[Jones, 1988b] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Bjørner et al. [1988], pages 1–14.

[Jones, 1993] Mark P. Jones. Coherence for qualified types, September 1993. Research Report YALEU/DCS/RR-989, Yale University.

[Jones, 1994a] Mark P. Jones. *Qualified Types: Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Jones, 1994b] Mark P. Jones. Simplifying and improving qualified types, June 1994. Research Report YALEU/DCS/RR-1040, Yale University.

[Jones, 1996] Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [1996], pages 216–237.

[Jones, 1999] Mark P. Jones. Typing Haskell in Haskell. In *Proceedings of 1999 Haskell Workshop*, Paris, France, October 1999. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[Jones, 2000] Mark P. Jones. Type classes with functional dependencies. In Smolka [2000], pages 230–244.

[Khoo and Sundaresh, 1991] Siau-Cheng Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 211–222, New Haven, Conneticut, USA, June 1991. ACM. SIGPLAN Notices Vol.26, Nro.9, September 1991.

[Kleene, 1952] Stephen C. Kleene. *Introduction to Metamathematics.* Princeton NJ: D. van Nostrand, 1952.

[Knoblock and Ruf, 1996] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, pages 215–225. ACM Press, 1996.

[Lawall and Danvy, 1994] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM.

[Lawall, 1998] Julia Lawall. Faster Fourier Transforms via automatic program specialization. In Hatcliff et al. [1998], pages 338–355.

[Lloyd and Shepherdson, 1991] John W. Lloyd and John C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.

[Makholm, 2000] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Taha [?], pages 129–148.

[Malmkjær, 1989] K. Malmkjær. Program and data specialisation: Principles, applications, and self-application, Master's Thesis, DIKU University of Copenhagen, August 1989.

[Marlet *et al.*, 1997] Renaud Marlet, Scott Thibault, and Charles Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engeneering*, pages 183–192. IEEE Computer Society, November 1997.

[Marlet *et al.*, 1999] Renaud Marlet, Scott Thibault, and Charles Consel. Efficient implementations of software architectures via partial evaluation. *Automated Software Engineering: An International Journal*, 6(4):411–440, October 1999.

[Marquard and Steensgaard, 1992] Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language, Master Thesis, DIKU, University of Copenhagen, Denmark, 1992.

[Martínez López and Badenes, 2003] Pablo E. Martínez López and Hernán Badenes. Simplifying and solving qualified types for principal type specialisation. In *Proceedings of 7th Workshop Argentino de Informática Teórica (WAIT 2003)*, September 2003.

[Martínez López and Hughes, 2001] Pablo E. Martínez López and John Hughes. Towards principal type specialisation. Technical report, University of Buenos Aires, 2001. URL: `http://www-lifia.info.unlp.edu.ar/~fidel/Works/PTS/towardsPTS.dvi.tgz`.

[Martínez López and Hughes, 2002] Pablo E. Martínez López and John Hughes. Principal type specialisation. In Wei-Ngan Chin, editor, *Proceedings of the 2002 ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105. ACM Press, September 2002.

[Martínez López, 1998] Pablo E. Martínez López. *Application of Functional Languages to Massive and Complex Computational Problems*, Master Thesis, Universidad de la República and Pedeciba (Programa de Desarrollo de Ciencias Básicas), Montevideo, Uruguay, August 1998.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, volume 17 of *3*, 1978.

[Mogensen, 1988] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Bjørner et al. [1988], pages 325–347.

[Mogensen, 1989] Torben Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In Josep Diaz and Fernando Orejas, editors, *TAPSOFT '89. Proceedings of the International Conference on Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science (LNCS)*, pages 298–312, Barcelona, Spain, March 1989. Springer-Verlag.

[Mogensen, 1993] Torben Æ. Mogensen. Constructor specialisation. In David Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, pages 22–32. ACM Press, June 1993.

[Mogensen, 1996] Torben Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In Danvy et al. [1996], pages 303–321.

[Mogensen, 1998a] Torben Æ. Mogensen. Inherited limits. *ACM Computing Surveys*, 30(3), September 1998.

[Mogensen, 1998b] Torben Æ. Mogensen. Partial evaluation: Concepts and applications. In Hatcliff et al. [1998], pages 1–19.

[Muller *et al.*, 1998] Gilles Muller, Renaud Marlet, Eugen N. Volanschi, Charles Consel, Calton Pu, and Ashvin Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems (ICDCS '98)*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[Nielson and Nielson, 1992] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Nordström *et al.*, 1990] Bengt Nordström, Ken Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[Ohori, 1999] Atsushi Ohori. Type-directed specialization of polymorphism. *Information and Computation*, 1999.

[Pasalic *et al.*, 2002] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[Penello, 1986] Thomas J. Penello. Very fast LR parsing. In *Proceedings of SIGPLAN '86 Conference on Compiler Construction*, pages 145–151, Palo Alto, California, USA, 1986. ACM.

[Peyton Jones and Hughes (editors), 1999] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999.
URL: `http://www.haskell.org/onlinereport/`.

[Pfenning and Elliott, 1988] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation (PLDI)*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press. Also in SIGPLAN Notices, 23(7).

[Pierce, 2002] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts 02142, February 2002.

[Pottier, 2000] François Pottier. A 3-part type inference engine. In Smolka [2000], pages 320–335.

[Rèmy, 1989] Didier Rèmy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–88, January 1989. Austin, Texas.

[Röjemo and Runciman, 1996] Niklas Röjemo and Colin Runciman. Lag, drag, void and use: heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN International Conference on Functional Programminga (ICFP '96)*, pages 34–41, Philadelphia, Pennsylvania, United States, 1996. ACM Press. Also published in ACM SIGPLAN Notices 31(6), June 1996.

[Röjemo, 1995] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, 1995.

[Romanenko, 1990] Sergei Romanenko. Arity raising and its use in program specialisation. In Neil D. Jones, editor, *Proceedings of 3rd European Symposium on Programming (ESOP'90)*, volume 432 of *Lecture Notes in Computer Science (LNCS)*, pages 341–360, Copenhagen, Denmark, May 1990. Springer-Verlag.

[Rose, 1998] Kristoffer H. Rose. Type-directed partial evaluation in Haskell. In *Preliminary Proceedings of the 1988 APPSEM Workshop on Normalization by Evaluation (NbE '98)*, Chalmers Universty, Sweden, May 1998. BRICS Note Series number NS-98-1, Department of Computer Science, University of Aarhus.

[Secher and Sørensen, 2000] Jens Peter Secher and Morten Heine B. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *Lecture Notes in Computer Science (LNCS)*, pages 113–127. Springer-Verlag, 2000.

[Sestoft, 1985] Peter Sestoft. The structure of a self-applicable partial evaluator. In Ganzinger and Jones [1985], pages 236–256.

[Sheard, 1997] Tim Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In Gallagher [1997], pages 22–35.

[Sheard, 2001] Tim Sheard. Generic unification via two-level types and parameterized modules. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, pages 86–97, Florence, Italy, September 2001.

[Shields and Meijer, 2001] Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, pages 261–275, London, United Kingdom, January 2001.

[Singh and McKay, 1998] Satnam Singh and Nicholas McKay. Partial evaluation of hardware. In Hatcliff et al. [1998], pages 221–230.

[Smolka, 2000] Gert Smolka, editor. *Proceedings of 9th European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, March/April 2000.

[Sørensen *et al.*, 1996] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[Sørensen and Glück, 1998] Morten Heine Sørensen and Robert Glück. Introduction to supercompilation. In Hatcliff et al. [1998], pages 246–270.

[Taha and Makholm, 2000] Walid Taha and Henning Makholm. Tag elimination - or - type specialisation is a type-indexed effect. In *APPSEM Workshop on Subtyping & Dependent Types in Programming*, Ponte de Lima, Portugal, July 2000.

[Taha *et al.*, 2001] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Danvy and Filinski [2001], pages 257–275.

[Taha, 2000] Walid Taha, editor. *Proceedings of the First Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, volume 1924 of *Lecture Notes in Computer Science (LNCS)*, Montréal, Canada, September 2000. Springer-Verlag.

[Takano, 1991] Akihiko Takano. Generalized partial computation for a lazy functional language. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '91)*, pages 1–11. ACM Press, 1991. Also published in ACM SIGPLAN Notices, Volume 26, Issue 9 (September 1991).

[Thatte, 1992] Satish R. Thatte. Typechecking with ad-hoc polymorphism (preliminary report). Technical report, Department of Mathematics and Computer Science, Clarkson University, Potsdan, NY, USA, May 1992. Manuscript.

[Thibault *et al.*, 1998] Scott Thibault, Charles Consel, and Gilles Muller. Safe and efficient active network programming. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, USA, October 1998.

[Thibault *et al.*, 1999] Scott Thibault, Renaud Marlet, and Charles Consel. Domain-specific languages: from design to implementation - Application to video device drivers generation. *IEEE Transactions on Software Engineering (TSE)*, 25(3):363–377, May-June 1999.

[Thiemann and Sperber, 1996] Peter Thiemann and Michael Sperber. Polyvariant expansion and compiler generators. In Bjørner et al. [1996], pages 285–296.

[Thiemann, 1999a] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[Thiemann, 1999b] Peter Thiemann. Higher-order code splicing. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science (LNCS)*, pages 243–257. Springer-Verlag, March 1999.

[Thiemann, 1999c] Peter Thiemann. Interpreting specialization in type theory. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 30–43, San Antonio, Texas, USA, January 1999.

[Thiemann, 2000a] Peter Thiemann. First-class polyvariant functions and co-arity raising, November 2000. Unpublished manuscript. Available from URL: `http://www.informatik.uni-freiburg.de/~thiemann/papers/fcpcr.ps.gz`.

[Thiemann, 2000b] Peter Thiemann. The PGG system-user manual, March 2000. Available from URL: `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.

[Turchin, 1985] Valentin F. Turchin. Program transformation by supercompilation. In Ganzinger and Jones [1985], pages 257–281.

[Turchin, 1986] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wand, 1982] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

[Wand, 1990] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, Nice, France, 1990. ACM Press.

[Wand, 1991] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.