Tagging, Encoding, and Jones Optimality

Olivier Danvy[1] and Pablo E. Martínez López[2]

[1] BRICS**,
Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
(danvy@brics.dk)

[2] LIFIA
Universidad Nacional de La Plata
CC. 11 Correo Central, (1900) La Plata, Bs.As. Argentina
(fidel@info.unlp.edu.ar)

Abstract. A partial evaluator is said to be Jones-optimal if the result of specializing a self-interpreter with respect to a source program is textually identical to the source program, modulo renaming. Jones optimality has already been obtained if the self-interpreter is untyped. If the self-interpreter is typed, however, residual programs are cluttered with type tags. To obtain the original source program, these tags must be removed.

A number of sophisticated solutions have already been proposed. We observe, however, that with a simple representation shift, ordinary partial evaluation is already Jones-optimal, modulo an encoding. The representation shift amounts to reading the type tags as constructors for higher-order abstract syntax. We substantiate our observation by considering a typed self-interpreter whose input syntax is higher-order. Specializing this interpreter with respect to a source program yields a residual program that is textually identical to the source program, modulo renaming.

1 Introduction

Specializing an interpreter with respect to a program has the effect of translating the program from the source language of the interpreter to the implementation language (or to use Reynolds's words, from the defined language to the defining language [36]). For example, if an interpreter for Pascal is written in Scheme, specializing it with respect to a Pascal program yields an equivalent Scheme program. Numerous instances of this specialization are documented in the literature, e.g., for imperative languages [5, 9], for functional languages [2], for logic languages [7], for object-oriented languages [27], for reflective languages [31], and for action notation [3]. Interpreter specialization is also known as the first Futamura projection [15, 16, 26].

** Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

P. Degano (Ed.): ESOP 2003, LNCS 2618, pp. 335–347, 2003.

One automatic technique for carrying out program specialization is partial evaluation [6, 25]. An effective partial evaluator completely removes the interpretive overhead of the interpreter. This complete removal is difficult to characterize in general and therefore it has been characterized for a particular case, self-interpreters, i.e., interpreters whose source language is (a subset of) their implementation language. A partial evaluator is said to be *Jones optimal* if it completely removes the interpretation overhead of a self-interpreter, i.e., if the result of specializing a self-interpreter with respect to a well-formed source program is textually identical to the source program, modulo renaming. Jones optimality was obtained early after the development of offline partial evaluation for untyped interpreters, with lambda-Mix [18, 25].

A typed interpreter, however, requires a universal data type to represent expressible values. Specializing such an interpreter, e.g., with lambda-Mix, yields a residual program with many tag and untag operations. Ordinary, Mix-style, partial evaluation is thus not Jones optimal [24].

Obtaining Jones optimality has proven a source of inspiration for a number of new forays into partial evaluation, e.g., handwritten generators of program generators [19,1], constructor specialization [10,33], type specialization [11,21,20, 22, 23, 29], coercions [8], and more recently tag elimination [30, 37, 38] and staged tagless interpreters [34]. Furthermore, the term "identical modulo renaming" in the definition of Jones optimality has evolved into "at least as efficient" [17, 25].

Here, we identify a simple representation shift of the specialized version of a typed lambda-interpreter and we show that with this representation shift, ordinary partial evaluation is already Jones optimal in the original sense.

Prerequisites and notation: We assume a basic familiarity with partial evaluation in general, as can be gathered in Jones, Gomard, and Sestoft's textbook [25]. We use Standard ML [32] and the notion of higher-order abstract syntax as introduced by Pfenning and Elliot [35] and used by Thiemann [39, 40]: Whereas the first-order abstract syntax of lambda-terms reads as

```
datatype foexp = FOVAR of string
               | FOLAM of string * foexp
               | FOAPP of foexp * foexp
```

the higher-order abstract syntax of lambda-terms reads as

```
datatype hoexp = HOVAR of string
               | HOLAM of hoexp -> hoexp
               | HOAPP of hoexp * hoexp
```

where the constructor `HOVAR` is only used to represent free variables. For example, the first-order abstract syntax of the K combinator $\lambda x.\lambda y.x$ reads as

```
FOLAM ("x", FOLAM ("y", FOVAR "x"))
```

and its higher-order abstract syntax reads as

```
HOLAM (fn x => HOLAM (fn y => x))
```

2 The Problem

The problem of specializing a typed interpreter is usually presented as follows [38]. Given

- a grammar of source expressions

```
datatype exp = LIT of int
             | VAR of string
             | LAM of string * exp
             | APP of exp * exp
             | ADD of exp * exp
```

- a universal type of expressible values

```
datatype univ = INT of int
              | FUN of univ -> univ
```

- an environment `Env : ENV`

```
signature ENV
= sig
    type 'a env
    val empty : 'a env
    val extend : string * 'a * 'a env -> 'a env
    val lookup : string * 'a env -> 'a
  end
```

- and two untagging functions `app` and `add`

```
exception TYPE_ERROR
(*  app : univ * univ -> univ  *)
fun app (FUN f, v)
    = f v
  | app _
    = raise TYPE_ERROR

(*  add : univ * univ -> univ  *)
fun add (INT i1, INT i2)
    = INT (i1 + i2)
  | add _
    = raise TYPE_ERROR
```

a typed lambda-interpreter is specified as follows:

```
(*  eval : exp -> univ Env.env -> univ  *)
fun eval (LIT i) env
    = INT i
  | eval (VAR x) env
    = Env.lookup (x, env)
  | eval (LAM (x, e)) env
    = FUN (fn v => eval e (Env.extend (x, v, env)))
  | eval (APP (e0, e1)) env
    = app (eval e0 env, eval e1 env)
  | eval (ADD (e1, e2)) env
    = add (eval e1 env, eval e2 env)
```

This evaluator is compositional, i.e., all recursive calls to `eval` on the right-hand side are on proper sub-parts of the term in the left-hand side [41, page 60]. Specializing this evaluator amounts to

1. unfolding all calls to `eval` while keeping the environment partially static, so that variables are looked up at specialization time, and
2. reconstructing all the remaining parts of the evaluator as residual syntax.

Unfolding all calls to `eval` terminates because the evaluator is compositional and its input term is finite.

Specializing the interpreter with respect to the term

```
LAM ("x", APP (VAR "x", VAR "x"))
```

thus yields

```
FUN (fn v => app (v, v))
```

This specialization is not Jones optimal because of the type tag `FUN` and the untagging operation `app`. (N.B. Danvy's type coercions and (depending on the annotations in the interpreter) Hughes's type specialization would actually not produce any result here [8, 21]. Instead, they would yield a type error because the source term is untyped. Raising a type error at specialization time or at run time is inessential with respect to Jones optimality.)

3 But Is It a Problem?

An alternative reading of

```
FUN (fn v => app (v, v))
```

is as higher-order abstract syntax [35]. In this reading, `FUN` is the tag of a lambda-abstraction and `app` is the tag of an application.

In that light, let us define the residual syntax as an ML data type by considering each branch of the evaluator and gathering each result into a data-type constructor:

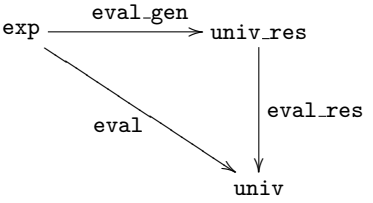
```
datatype univ_res = INT_res of int
                  | FUN_res of univ_res -> univ_res
                  | APP_res of univ_res * univ_res
                  | ADD_res of univ_res * univ_res
```

The corresponding generating extension is a recursive function that traverses the source expression and constructs the result using the constructors of `univ_res`:

```
(*  eval_gen : exp -> univ_res Env.env -> univ_res  *)
fun eval_gen (LIT i) env
    = INT_res i
  | eval_gen (VAR x) env
    = Env.lookup (x, env)
```

```
  | eval_gen (LAM (x, e)) env
    = FUN_res (fn v => eval_gen e (Env.extend (x, v, env)))
  | eval_gen (APP (e0, e1)) env
    = APP_res (eval_gen e0 env, eval_gen e1 env)
  | eval_gen (ADD (e1, e2)) env
    = ADD_res (eval_gen e1 env, eval_gen e2 env)
```

The interpretation of `univ_res` elements is defined with a function `eval_res` that makes the following diagram commute:



First of all, we need a conversion function `u2ur` and its left inverse `ur2u` to write `eval_res`:

```
exception NOT_A_VALUE

(*  u2ur : univ -> univ_res  *)
(*  ur2u : univ_res -> univ  *)
fun u2ur (INT i)
    = INT_res i
  | u2ur (FUN f)
    = FUN_res (fn r => u2ur (f (ur2u r)))
and ur2u (INT_res i)
    = INT i
  | ur2u (FUN_res f)
    = FUN (fn u => ur2u (f (u2ur u)))
  | ur2u (APP_res _)
    = raise NOT_A_VALUE
  | ur2u (ADD_res _)
    = raise NOT_A_VALUE
```

The corresponding evaluator reads as follows (the auxiliary functions `app` and `add` are that of Section 2):

```
(*  eval_res : univ_res -> univ  *)
fun eval_res (INT_res i)
    = INT i
  | eval_res (FUN_res f)
    = FUN (fn u => eval_res (f (u2ur u)))
  | eval_res (APP_res (r0, r1))
    = app (eval_res r0, eval_res r1)
  | eval_res (ADD_res (r1, r2))
    = add (eval_res r1, eval_res r2)
```

The generating extension, `eval_gen`, is an encoding function from first-order abstract syntax to higher-order abstract syntax. (In fact, it is the standard such encoding [35].) It maps a term such as

```
LAM ("x", APP (VAR "x", VAR "x"))
```

into the value of

```
FUN_res (fn v => APP_res (v, v))
```

With this reading of residual syntax as higher-order abstract syntax, ordinary partial evaluation (i.e., the generating extension) maps the first-order abstract-syntax representation of $\lambda x.x\,x$ into the higher-order abstract-syntax representation of $\lambda x.x\,x$, and it does so optimally.

(Incidentally, partial evaluation (of an interpreter in a typed setting) connects to Ershov's mixed computation, since the specialized version of an evaluator is both (1) a residual program and (2) the data type used to represent this residual program together with the interpretation of the constructors of the data type [12, 13, 14].)

4 An Interpreter for Higher-Order Abstract Syntax

Let us now verify that Jones optimality is obtained for a typed interpreter whose input syntax is higher order. It is a simple matter to adapt the representation of the input of the typed interpreter from Section 3 to be higher order instead of first order. In the fashion of Section 2, the grammar of source expressions and the universal type of expressible values read as follows:

```
datatype exp = LIT of int
             | LAM of exp -> exp
             | APP of exp * exp
             | ADD of exp * exp

datatype univ = INT of int
              | FUN of univ -> univ
```

The auxiliary functions `app` and `add` read just as in Section 2. As in Section 3, we need a conversion function `u2e` and its left inverse `e2u`:

```
exception NOT_A_VALUE

(*  u2e : univ -> exp  *)
(*  e2u : exp -> univ  *)
fun u2e (INT i)
    = LIT i
  | u2e (FUN f)
    = LAM (fn e => u2e (f (e2u e)))
```

```
and e2u (LIT i)
    = INT i
  | e2u (LAM f)
    = FUN (fn u => e2u (f (u2e u)))
  | e2u (APP _)
    = raise NOT_A_VALUE
  | e2u (ADD _)
    = raise NOT_A_VALUE
```

The corresponding evaluator reads as follows:

```
(*  eval : exp -> univ  *)
fun eval (LIT i)
    = INT i
  | eval (LAM f)
    = FUN (fn u => eval (f (u2e u)))
  | eval (APP (e0, e1))
    = app (eval e0, eval e1)
  | eval (ADD (e1, e2))
    = add (eval e1, eval e2)
```

As in Section 3, we define the residual syntax as an ML data type by enumerating each branch of the evaluator and gathering each result into a data-type constructor. The result and its interpretation (i.e., `eval_res` and its two auxiliary conversion functions) are the same as in Section 3:

```
datatype univ_res = INT_res of int
                  | FUN_res of univ_res -> univ_res
                  | APP_res of univ_res * univ_res
                  | ADD_res of univ_res * univ_res
```

As in Section 3, we need two conversion functions `ur2e` and `e2ur` to write the generating extension:

```
(*  ur2e : univ_res -> exp  *)
(*  e2ur : exp -> univ_res  *)
fun ur2e (INT_res i)
    = LIT i
  | ur2e (FUN_res f)
    = LAM (fn e => ur2e (f (e2ur e)))
  | ur2e (APP_res (e0, e1))
    = APP (ur2e e0, ur2e e1)
  | ur2e (ADD_res (e1, e2))
    = ADD (ur2e e1, ur2e e2)
and e2ur (LIT i)
    = INT_res i
  | e2ur (LAM f)
    = FUN_res (fn r => e2ur (f (ur2e r)))
  | e2ur (APP (e0, e1))
    = APP_res (e2ur e0, e2ur e1)
  | e2ur (ADD (e1, e2))
    = ADD_res (e2ur e1, e2ur e2)
```

The corresponding generating extension reads as follows:

```
(*  eval_gen : exp -> univ_res  *)
fun eval_gen (LIT i)
    = INT_res i
  | eval_gen (LAM f)
    = FUN_res (fn r => eval_gen (f (ur2e r)))
  | eval_gen (APP (e0, e1))
    = APP_res (eval_gen e0, eval_gen e1)
  | eval_gen (ADD (e1, e2))
    = ADD_res (eval_gen e1, eval_gen e2)
```

It should now be clear that `exp` and `univ_res` are isomorphic, since `ur2e` and `e2ur` are inverse functions, and that `eval_gen` computes the identity transformation up to this isomorphism. The resulting specialization is thus Jones optimal.

5 But Is It the Real Problem?

Jones's challenge, however, is not for any typed interpreter but for a typed self-interpreter. Such a self-interpreter, for example, is displayed in Taha, Makholm, and Hughes's article at PADO II [38]. We observe that the reading of Section 3 applies for this self-interpreter as well: its universal data type of values can be seen as a representation of higher-order abstract syntax.

6 A Self-Interpreter for Higher-Order Abstract Syntax

The second author has implemented a self-interpreter for higher-order abstract syntax in a subset of Haskell, and verified that its generating extension computes an identity transformation modulo an isomorphism [28].[1] Therefore, Jones's challenge is met.

7 Related Work

7.1 Specializing Lambda-Interpreters

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, modulo desugaring. 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.

[1] The self-interpreter is available from the authors' web page.

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.

7.2 Jones Optimality and Higher-Order Abstract Syntax

This article complements the first author's work on coercions for type specialization [8] and the second author's work on type specialization [29]. Our key insight is that a specialized interpreter is a higher-order abstract syntax representation of the source program. As pointed out by Taha in a personal communication to the first author (January 2003), however, this insight in itself is not new. Already in 2000, Taha and Makholm were aware that "A staged interpreter for a simply-typed lambda calculus can be modelled by a total map from terms to what is essentially a higher-order abstract syntax encoding." [37, Section 1.2]. Yet they took a different path and developed tag elimination and then tagless interpreters to achieve Jones-optimal specialization of typed interpreters.

7.3 Type Specialisation

The goal of type specialisation is to specialise both a source term and a source type to a residual term and a residual type. It was introduced by Hughes, who was inspired precisely by the problem of Jones optimality for typed interpreters [21, 20]. The framework of type specialisation, however, allows more than just producing optimal typed specialisers; traditional partial evaluation, constructor specialisation, firstification, and type checking are comprised in it (among other features). In contrast, we have solely focused on specializing (unannotated) typed interpreters here.

8 Conclusion

The statement of Jones optimality involves two ingredients:

1. an evaluator that is in direct style and compositional, i.e., that is defined by structural induction on the source syntax; and
2. a partial evaluator.

Our point is that if the partial evaluator, when it specializes the evaluator with respect to an expression,

- unfolds all calls to the evaluator,
- keeps the environment partially static, so that variables can be looked up at partial-evaluation time, and
- reconstructs everything else into a residual data type

then it computes a homomorphism, i.e., a compositional translation, from the source syntax (data type) to the target syntax (data type). When the source and target syntax are isomorphic, as in Section 4 and for lambda-Mix [18, 25], this homomorphism is an isomorphism and the partial evaluator is Jones optimal.

Acknowledgments. Our insight about higher-order abstract syntax occurred during the second author's presentation at ASIA-PEPM 2002 [29]. The topic of this article has benefited from discussions with Mads Sig Ager, Kenichi Asai, John Hughes, Neil Jones, Nevin Heintze, Julia Lawall, Karoline Malmkjær, Henning Korsholm Rohde, Peter Thiemann, and Walid Taha, with special thanks to Henning Makholm for a substantial as well as enlightening discussion in January 2003. We are also grateful to the anonymous reviewers for comments, with special thanks to Julia Lawall.

This work is supported by the ESPRIT Working Group APPSEM II (`http://www.tcs.informatik.uni-muenchen.de/~mhofmann/appsem2/`).

The second author is a PhD student at the University of Buenos Aires and is partially funded by grants from the ALFA-CORDIAL project Nr. ALR/B73011/94.04-5.0348.9 and the FOMEC project of the Computer Science Department, FCEyN, at the University of Buenos Aires.

References

1. Lars Birkedal and Morten Welinder. Handwriting program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.
2. Anders Bondorf. Compiling laziness by partial evaluation. In Simon L. Peyton Jones, Guy Hutton, and Carsten K. Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in Computing, pages 9–22, Glasgow, Scotland, 1990. Springer-Verlag.
3. Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 308–317, Copenhagen, Denmark, June 1993. ACM Press.
4. Wei-Ngan Chin, editor. *ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Aizu, Japan, September 2002. ACM Press.
5. Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.
6. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
7. Charles Consel and Siau-Cheng Khoo. Semantics-directed generation of a Prolog compiler. *Science of Computer Programming*, 21:263–291, 1993.

8. 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*, number 1443 in Lecture Notes in Computer Science, pages 908–917. Springer-Verlag, 1998.
9. Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as the technical report BRICS-RS-96-13.
10. 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*, pages 54–65, La Jolla, California, June 1995. ACM Press.
11. Dirk Dussart, John Hughes, and Peter Thiemann. Type specialization for imperative languages. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 204–216, Amsterdam, The Netherlands, June 1997. ACM Press.
12. Andrei P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
13. Andrei P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
14. Andrei P. Ershov, Dines Bjørner, Yoshihiko Futamura, K. Furukawa, Anders Haraldsson, and William Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987*, New Generation Computing, Vol. 6, No. 2-3. Ohmsha Ltd. and Springer-Verlag, 1988.
15. Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.
16. Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
17. Robert Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In Chin [4], pages 9–19.
18. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.
19. Carsten K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.
20. John Hughes. An introduction to program specialisation by type inference. In *Functional Programming*, Glasgow University, July 1996. Published electronically.
21. John Hughes. Type specialisation for the lambda calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 183–215, Dagstuhl, Germany, February 1996. Springer-Verlag.
22. John Hughes. A type specialisation tutorial. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 293–325, Copenhagen, Denmark, July 1998. Springer-Verlag.

23. John Hughes. The correctness of type specialisation. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 215–229, Berlin, Germany, March 2000. Springer-Verlag.
24. Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.
25. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.
26. 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, 1989.
27. Siau Cheng Khoo and Sundaresh. Compiling inheritance using partial evaluation. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 211–222, New Haven, Connecticut, June 1991. ACM Press.
28. Pablo E. Martínez López. *Type Specialisation for Polymorphic Languages*. PhD thesis, Department of Computer Science, University of Buenos Aires, Buenos Aires, Argentina, 2003. Forthcoming.
29. Pablo E. Martínez López and John Hughes. Principal type specialisation. In Chin [4], pages 94–105.
30. Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Walid Taha, editor, *Proceedings of the First Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, number 1924 in Lecture Notes in Computer Science, pages 129–148, Montréal, Canada, September 2000. Springer-Verlag.
31. Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of OOPSLA'91, the ACM SIGPLAN Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 300–315, Austin, Texas, October 1995. SIGPLAN Notices 30(10).
32. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
33. Torben Æ. Mogensen. Constructor specialization. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, Copenhagen, Denmark, June 1993. ACM Press.
34. 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.
35. 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*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.
36. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

37. Walid Taha and Henning Makholm. Tag elimination or type specialization is a type-indexed effect. In *Proceedings of the 2000 APPSEM Workshop on Subtyping and Dependent Types in Programming*, Ponte de Lima, Portugal, July 2000. `http://www-sop.inria.fr/oasis/DTP00/`.
38. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001*, number 2053 in Lecture Notes in Computer Science, pages 257–275, Aarhus, Denmark, May 2001. Springer-Verlag.
39. Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.
40. Peter Thiemann. Higher-order code splicing. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 243–257, Amsterdam, The Netherlands, March 1999. Springer-Verlag.
41. Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.