Implementing a Typed Assembly Language and its Machine Model

Ricardo Medel Matthieu Lucotte
Adriana Compagnoni*
Department of Computer Science,
Stevens Institute of Technology,
Castle Point on Hudson,
Hoboken, NJ 07030, U. S. A.

Abstract

We describe the implementation of a first-order linearly typed assembly language, HBAL, that allows the safe reuse of heap space for elements of different types. Linear typing ensures the single pointer property, disallowing aliasing, but allowing safe in-place-update compilation of a functional programming language. HBAL was designed as a target low-level language for Hofmann's LFPL programs [5] that run in a bounded amount of heap space.

Keywords: Programming Languages, Type Systems, Compilers, Typed Assembly Languages, Resource Awareness.

*This work was partially supported by the New Jersey Commission on Science and Technology and by the NSF Grant No. CCR-0093362 *CAREER: A formally verified environment for the production of secure software*. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

When computing resources are limited, such as in embedded and real-time systems or for applications to be run across the Internet, resource awareness is crucial. The particular resource we are concerned with in this paper is *heap space*. We describe the implementation of HBAL, a first-order *heap-bounded assembly language*, whose type system allows for safe reuse of heap space for elements of different types. HBAL is suitable for *resource-aware computing* using *proof-carrying code*.

In C even a malloc-free program can use more memory than what was assigned to the program initially. For example, a pointer can access an arbitrary memory location beyond the intended scope of the program. Since type systems are a convenient and elegant way of expressing a *semantic constraint* such as resource boundedness, in order to provide *end-to-end static guarantees* of resource-boundedness, we use type-systems for both high-level and low-level languages. In this context, the compiler translates typed high-level code into typed low-level code, where the *typing derivation* of the low-level code is the *proof* in proof-carrying code.

In our implementation the HBAL language is used to capture in-place update compilation of a functional programming language with similar type systems, providing a *static guarantee* of bounded heap-space usage. We implemented a compiling function from Hofmann's LFPL [3] into HBAL.

Typed assembly languages have been an active subject of study for several years. Contributions include TAL [7, 8], STAL [9], DTAL [12], and Alias Types [10, 11].

TAL began with higher-order functions and polymorphism, considering System F as a high-level language. But TAL assumed a particular compilation technique, continuation-passing style, which is not used by every compiler. STAL addressed this by modeling stacks with *stack polymorphism*. DTAL introduced the possibility for making some array bounds-checking optimizations, using dependent types. Alias Types allowed for areas of store to be reused in ways that TAL prohibited, tracking aliasing of locations to allow for safe memory management.

HBAL uses a deliberately restricted type system which reflects in the low-level the state-of-the-art of a type-system approach to dealing with resource bounds in a high-level language, following Hofmann's work ([3, 4, 2, 5]).

HBAL uses linear typing to prevent aliasing, and includes pseudoinstructions for safely altering the types of heap locations. This means that we do not need to assume an external garbage collector, and we can provide a static guarantee that every program runs in a fixed amount of heap space. However, HBAL is not incompatible with garbage collection, and it can be extended to include `malloc` and `free` operations if desired. This could be done so that the memory-allocation initialization part of an application could be typed within the system.

This section and Section 2, where HBAL's syntax and semantics are described, are extracted from [1]. The new contribution, i.e. the implementation of the compiler LFPL-to-HBAL and the machine model for HBAL, is described in Section 3. In Section 4 we provide some examples of HBAL programs. Conclusions and some future work are depicted in Section 5.

2 The HBAL language

HBAL is a first-order linearly typed assembly language. In the HBAL type system a *word type* (σ) is either an integer or a pointer to a structured type A (denoted as $[A]$), where A can be `code`, the type of instructions, a flagged word type (σ^z), a Cartesian product ($A \times A$), a list ($L(A)$), a tree ($T(A)$), or a diamond ($\diamond$). The flag (z) can be 0 (uninitialized) or 1 (initialized).

The machine registers are called $r_0 \ldots r_n$ and there are two dedicated registers for the program counter pc and stack pointer sp. A *context* Γ is a finite mapping of registers r_i to word types σ, treated as a set of type assignments $r_i : \sigma$.

The initial context is denoted $\{\}$, but it contains the type assignment $r_0 : \mathtt{int}$. This register is used to always store the constant number 0, making it unusable by the programmer.

When we write the extended context $\Gamma, r_i : \sigma$, it is understood that r_i does *not* already appear in Γ. We assume that the program counter, register pc, never appears in any context. The context $\Gamma_{\backslash r_i}$ is defined to be the same as Γ except undefined on r_i.

size(A) is the size of the type A as it is laid out in memory and $A[c]$ denotes the type of the cth word in the layout of A in memory.

When a pointer is read form memory into a register, the memory is made inaccessible by uninitializing the pointer type. This prevents aliases, preserving the single pointer property. So, $A^{c:=1}$ denotes A with the initialization flag set on the cth word, and $A^{c:=0}$ denotes A with the cth initialization flag cleared. $A^{:=0}$ denotes the uninitialization of A.

There exists a subtyping relation on types, given by the contextual, reflexive, and transitive closure of the uninitialization operation, so $B(A) \leq B(A^{:=0})$.

A *program* P consists of a sequence of instructions p and labels l.

$$
\begin{array}{rcl}
P & ::= & \langle\rangle \;\mid\; p \;;\; P \;\mid\; l \;;\; P \\[6pt]
p & ::= & \mathtt{load}\;\; r \longleftarrow r[c] \;\mid\; \mathtt{store}\;\; r[c] \longleftarrow r \\
& \mid & \mathtt{arithi}\;\; r \longleftarrow r \odot c \;\mid\; \mathtt{arith}\;\; r \longleftarrow r \odot r \\
& \mid & \mathtt{bnz}\;\; r\;\; l \;\mid\; \mathtt{bez}\;\; r\;\; l \;\mid\; \mathtt{jmp}\;\; l \\
& \mid & \mathtt{call}\;\; l \;\mid\; \mathtt{ret}\;\; l \\
& \mid & \mathtt{salloc}\;\; A \;\mid\; \mathtt{sfree}\;\; c \\
& \mid & \mathtt{use}\;\; r\;\; A \;\mid\; \mathtt{discard}\;\; r \\
& \mid & \mathtt{fold\text{-}nil}_A\;\; r[c] \;\mid\; \mathtt{fold\text{-}cons}_A\;\; r[c] \\
& \mid & \mathtt{fold\text{-}leaf}_A\;\; r[c] \;\mid\; \mathtt{fold\text{-}node}_A\;\; r[c] \\
& \mid & \mathtt{caselist}_A\;\; r[c]\;\; l \;\mid\; \mathtt{casetree}_A\;\; r[c]\;\; l \\[6pt]
\odot & ::= & + \;\mid\; - \;\mid\; * \;\mid\; /
\end{array}
$$

HBAL has a small set of standard assembly instructions, together with several pseudo-instructions. There are three kinds of pseudo-instructions; they are either typechecking directives (`use` and `discard`) that will be erased by the assembler, datatype constructors and destructors (`fold-nil`, `fold-cons`, and `caselist` for lists, and `fold-leaf`, `fold-node`, and `casetree` for binary trees), or hiding pointer arithmetic operations (`salloc` and `sfree`), because it is not allowed to overwrite the stack pointer directly.

A program is a sequence of instructions and labels which defines a number of mutually recursive first-order functions. The typing rules define a judgment:

$$\Gamma \vdash P$$

which means that P is a well-typed assembly program in context Γ.

A program must be given together with a *signature*, Σ. The signature assigns *procedure types* of the form $A_1, \ldots, A_n \rightarrow A$ for $n \geq 0$ to subroutine labels and contexts to branch target labels.

Typing rules

The typing rules ensure the single pointer property that says that every location can be reached from at most one live pointer in a register or on the heap. As an example, we give the typing rules for `load` and `store` instructions.

$$\frac{A[c] = \mathtt{int}^1 \quad \Gamma_{\backslash r_i}, r_j : [A], r_i : \mathtt{int} \vdash P}{\Gamma, r_j : [A] \vdash \mathtt{load} \quad r_i \longleftarrow r_j[c] \quad ; \quad P}$$

$$\frac{A[c] = [B]^1 \quad r_i \neq sp \quad \Gamma_{\backslash r_i}, r_j : [A^{c:=0}], r_i : [B] \vdash P}{\Gamma, r_j : [A] \vdash \mathtt{load} \quad r_i \longleftarrow r_j[c] \quad ; \quad P}$$

$$\frac{A[c] = \mathtt{int}^z \quad \Gamma, r_i : \mathtt{int}, r_j : [A^{c:=1}] \vdash P}{\Gamma, r_i : \mathtt{int}, r_j : [A] \vdash \mathtt{store} \quad r_j[c] \longleftarrow r_i \quad ; \quad P}$$

$$\frac{A[c] = [B]^z \quad r_i \neq sp \quad B \neq \mathtt{code} \quad \Gamma, r_j : [A^{c:=1}] \vdash P}{\Gamma, r_i : [B], r_j : [A] \vdash \mathtt{store} \quad r_j[c] \longleftarrow r_i \quad ; \quad P}$$

There are two rules for `load` and `store` because loading or copying a pointer may violate the single pointer property, while loading or copying an integer does not.

2.1 Semantics

Given an interpretation for HBAL types, which captures the way datatypes are implemented in memory, and an operational semantics for the untyped assembly instructions it was possible to prove a type soundness property. The safety preservation theorem establishes that the execution of a typeable HBAL program modifies structures on the heap safely.

First, it is necessary to define the function $\mathsf{Asm}(p)$ that translate pseudo-instructions to sequences of untyped instructions which results from assembling p. We assume that the assembler has resolved each label l to a memory location $\mathsf{LAdr}(l)$. Below we show its definition by giving a pseudo-instruction on the left and its untyped expansion on the right.

`salloc` A	`arithi` $sp \longleftarrow sp - (size(A))$
`sfree` A	`arithi` $sp \longleftarrow sp + (size(A))$
`call` l	`arithi` $r_1 \longleftarrow pc + 6$
	`store` $sp[m] \longleftarrow r_1$
	`arith` $pc \longleftarrow r_0 + \mathsf{LAdr}(l)$
`ret` l	`load` $pc \longleftarrow sp[0]$
`fold-nil`$_A$ $r_i[c]$	`store` $r_i[c] \longleftarrow 0$
`fold-cons`$_A$ $r_i[c]$	`store` $r_i[c] \longleftarrow 1$
`caselist`$_A$ $r_i[c]$ l_{cons}	`load` $r_1 \longleftarrow r_i[c]$
	`bnz` r_1 l_{cons}
`fold-leaf`$_A$ $r_i[c]$	`store` $r_i[c] \longleftarrow 0$
`fold-node`$_A$ $r_i[c]$	`store` $r_i[c] \longleftarrow 1$
`casetree`$_A$ $r_i[c]$ l_{node}	`load` $r_1 \longleftarrow r_i[c]$
	`bnz` r_1 l_{node}

Let $\mathsf{Loc} \subseteq \mathbf{Z}$ stand for the set of memory locations on the machine, $\mathsf{Reg} = \{0, 1, \dots, \mathsf{R}_{\max}\}$ be the register indices, Wrd be the set of machine words that can stand for integers or locations, and Code be the set of machine words which can stand for machine instructions, and Wrd is disjoint from Code.

A *machine configuration* M is a pair (R, H) where $H : \mathsf{Loc} \rightharpoonup \mathsf{Wrd} \uplus \mathsf{Code}$ is a heap configuration and $R : \mathsf{Reg} \to \mathsf{Wrd}$ is a register configuration, such that $R(0) = 0$.

The *unbounded stack assumption*, that is, that every machine has the space to grow its stack downwards indefinitely, is formalized by saying that H is defined to be data for all values below $R(sp)$, i.e., $\forall m \leq R(sp), H(m) \in \mathsf{Wrd}$. To ensure that the stack does not clash with the heap data or program code, it is established that $R(sp) \leq 0$ while locations used for program and heap data are positive.

The effect of each machine instruction (i.e., an untyped assembly language instruction) on a machine configuration is defined as:

Definition 1 (Machine transitions) *Given a machine $M = (R, H)$ we define $M \rightsquigarrow M' = (R', H')$, using the table below, by case analysis on the* *instruction at $H(R(pc)) \in \mathsf{Code}$:*

`load` $r_i \longleftarrow r_j[c]$	$R' = R[i \mapsto H(R(j) + c)]$
`store` $r_j[c] \longleftarrow r_i$	$H' = H[R(j) + c \mapsto R(i)]$
`arithi` $r_i \longleftarrow r_j \odot c$	$R' = R[i \mapsto R(j) \odot c]$
`arith` $r_i \longleftarrow r_j \odot r_k$	$R' = R[i \mapsto R(j) \odot R(k)]$
`jmp` x	$R' = R[pc \mapsto x]$
`bnz` r_i x	$R' = \begin{cases} R & \textit{if } R(i) = 0 \\ R[pc \mapsto x] & \textit{otherwise} \end{cases}$
`bez` r_i x	*similarly to* `bnz`

First, M' differs from M by incrementing $R(pc)$ according to the length of the instruction. Then the transformation given in the table above is applied, to give the new value H' or R' for an instruction that affects the register or heap configuration respectively.

In the definition above, the value of i for the load and arithmetic instructions is greater than 0. As we said, the operations on r_0 have no effect.

Given a machine configuration $M = (R, H)$, the satisfaction relation $H \models_K m : A$ can be defined in such way that captures when the location m represents a valid element of type A in heap H. This relation is defined in [1].

Memory M is type safe (for P) at u if and only if:

- the typed program $P = \langle p_1, \ldots, p_u, \ldots, p_n \rangle$ has been assembled in M by applying the $\mathsf{Asm}(p_i)$ function to each instruction $p_i \in p_1, \ldots, p_u$,
- M satisfies the typing constraints necessary to execute the next instruction p_u.

The next theorem establishes that a type safe machine can always progress to a new machine by executing the next typed instruction.

Theorem 1 (Progress) *Suppose M is type safe at u. Then there exists a machine M' such that $M \rightsquigarrow^{\mathsf{Asm}(p_u)} M'$.*

The second theorem establishes that whenever a type safe machine progresses to a new machine, the new machine is also type safe, provided we followed a typed path within the program P.

Theorem 2 (Safety preservation) *Suppose M is type safe at u and $M \rightsquigarrow^{\mathsf{Asm}(p_u)} M'$. Then either*

- *$\exists\, p_v$ such that $p_u \rightsquigarrow p_v$ and M' is type safe at v, or*
- *$R'(pc) \notin dom(\mathsf{PAdr})$ (the machine has left P).*

This theorem ends our presentation about the assembly language HBAL. The following sections are about the implementation of a compiler LFPL-to-HBAL and an interpreter for HBAL programs.

3 Implementation

As theorems 1 and 2 show, to guarantee safe execution of a HBAL program P we must:

- Typecheck P once ($\Gamma_{init} \vdash P$),
- Check that the memory satisfy the typing assumptions in the initial context before each execution ($\Gamma_{init} \models M$).

We implemented a compiler LFPL-to-HBAL and an HBAL machine model that executes HBAL programs provided by the compiler. In the following subsections we depict each module of our implementation, as shown in Figure 1. This system was implemented using Ocaml [6].

Compiler LFPL to HBAL

As shown in Figure 1, the compiler takes an LFPL program P as input and outputs an equivalent HBAL program P_H, an initial context Γ_{init}, and an environment Σ.

The first-order functional language LFPL (Linear Functional Programming Language) has the following grammar of types and terms:

$$A ::= \mathsf{N} \mid \diamond \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A_1 \otimes A_2 \mid A_1 + A_2$$

$e ::=$	x	(variable)
$\mid$	$f(e_1, \ldots, e_n)$	function application
$\mid$	c	integer constant
$\mid$	$e_1 \star e_2$	infix op., $\star \in \{+, -, \times, =, \leq\ \ldots\}$
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$	conditional
$\mid$	$\mathsf{inl}(e)$	left injection
$\mid$	$\mathsf{inr}(e)$	right injection
$\mid$	$e_1 \otimes e_2$	pairing
$\mid$	nil	empty list
$\mid$	$\mathsf{cons}(e_1, e_2, e_3)$	cons with res. arg.
$\mid$	$\mathsf{leaf}(e)$	leaf constructor
$\mid$	$\mathsf{node}(e_1, e_2, e_3, e_4, e_5)$	node constr. w. two res. args.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ nil \Rightarrow e_2 \mid \mathsf{cons}(d, h, t) \Rightarrow e_3$	list elimination
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{leaf}(a) \Rightarrow e_2 \mid$ $\mathsf{node}(d_1, d_2, a, l, r) \Rightarrow e_3$	tree elim.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ x \otimes y \Rightarrow e_2$	pair elim.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow e_2 \mid \mathsf{inr}(x) \Rightarrow e_3$	sum elim.

Heap space in LFPL is explicitly manipulated through high-level constructs; to construct a value of a recursive datatype, the programmer must supply an argument of type $\diamond$ for every sub-instance of the recursive type. More about the LFPL language can be found in [3].

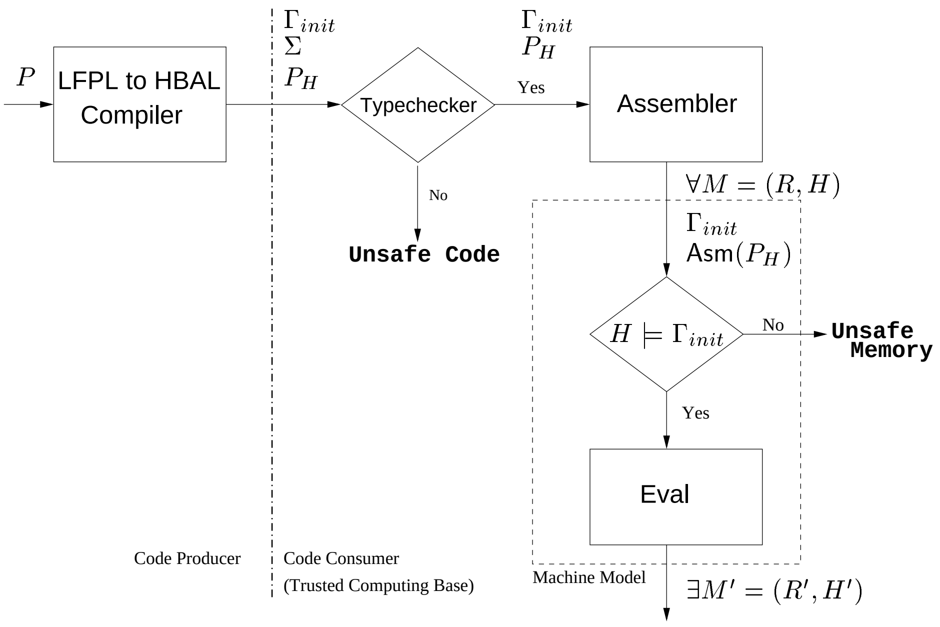

Figure 1: Compiler and Machine Model implementation for HBAL.

The compiler translates a LFPL program P into a HBAL program P_H and constructs the initial context Γ_{init} and environment Σ. All of these are sent to the typechecker as input.

This part of our implementation could be consider as the *code producer*, the untrusted part of the generation of mobile code.

Typechecker

The compiler's output is given to a typechecker that decides if the HBAL program P_H is *type safe* in the environment Σ with an initial context Γ_{init}. If it is safe, the program and the context are given to an assembler.

Assembler

This module applies the function $\mathsf{Asm}(p)$ to each pseudoinstruction in the program P_H, obtaining an *expanded* untyped assembly program. After that, each instruction of the expanded program is assembled, i.e. stored in a cell of an array, and each label is replaced by its address.

The initial context Γ_{init} and the assembled program $\mathsf{Asm}(P_H)$, are provided to the machine model.

Machine Model

The first task of the Machine Model is to check if the heap configuration of the machine where the program will be executed verifies the restrictions imposed by the initial context ($H \models \Gamma_{init}$). If the initial machine $M = (R, H)$

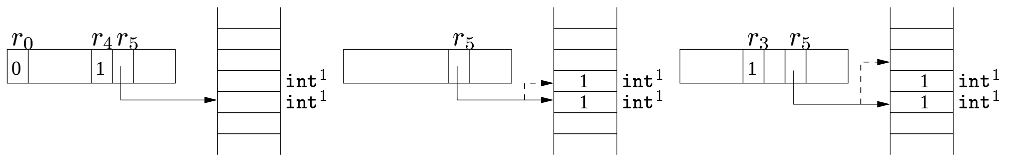

Figure 2: Intended execution of the first example.

is *memory safe*, then the evaluation module is called to execute the program $\mathsf{Asm}(P)$ on M.

The execution of $\mathsf{Asm}(P)$ on the machine $M = (R, H)$ is *safe*, because since $H \models \Gamma_{init}$, by definition, M is safe for P at the first instruction, and by theorems 1 and 2, each step in the execution of $\mathsf{Asm}(P)$ progresses to a new *safe* machine.

The typechecker, assembler, and machine model modules are considered as the *code consumer*, and they are part of the TCB (*Trusted Computing Base*). The code provided by the compiler is safe, if it passes the analyses for *safe code* and *safe memory.*

It is worthy of mention that once the program P_H is determined as *code safe* and assembled, each new execution in the same TCB only requires the *memory safe* verification, which only involves checking the types of live registers against the current memory.

4 Examples

In this section we give some examples of HBAL programs that show the kind of errors that can be avoided by applying our typechecker.

Note that in this implementation the flagged word type notation was changed from σ^1 to σ+ (initialized) and from σ^0 to σ- (uninitialized).

Preventing reading of uninitialized locations

The following HBAL program stores an integer (1) in the heap address pointed by r_5 and the next address, and the same integer in the register r_3. But an attempt to copy in the register r_3 the value stored in the $r_5[2]$ is detected by the typechecker as a violation of the first `load` typing rule.

The notation $r_5[c]$ is used to denote the value stored in the heap at the address pointed by the register r_5 with offset c. A graphical explanation of this program intended execution is given in Figure 2.

```
sign
main:{sp:[[code]+]+,r5:[int+ * int+]+}
<>
```

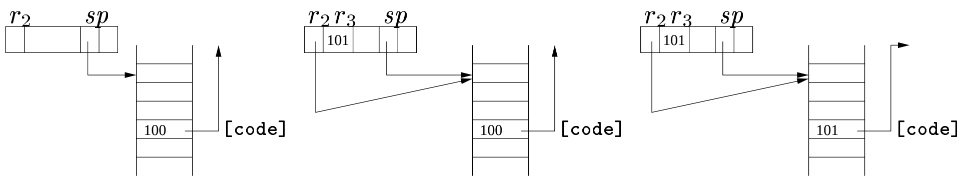

Figure 3: Intended execution of the second example.

```
main:
arithi r4 <- r0 + 1;
store r5[0] <- r4;
store r5[1] <- r4;
load r3 <- r5[1];
load r3 <- r5[2];
ret main;
<>
```

The value stored in $r_5[2]$ can be confidential information, then the problem can be considered as a *privacy* violation, or junk, in such case the problem can be seen as a *safety* concern.

Preventing malicious diversion of control

In procedure calls the calling convention gives this type to the stack pointer:

$$sp \ : \ [A_1 \times \cdots \times A_n \times [\texttt{code}]^1 \times A^{:=0}]$$

The stack frame contains space for the return value of type A, followed by the return pointer, and then the subroutine arguments at the top. Figure 3 shows graphically the stack after a procedure call with three arguments.

The next program illustrates an attack attempting to divert control to a different return address.

```
sign
main:{sp:[int+*int+*int+*[code]+*int-]+}
<>

main:
load r2 <- sp[0];
arithi r3 <- r2 + 1;
store r2[3] <- r3;
ret main;
<>
```

This program is rejected by the typechecker because the store operation attempts to alter a code address.

5 Conclusions

In this work we describe the implementation of a first-order linearly typed assembly language, HBAL. The HBAL language allows safe reuse of heap space for elements of different types. Linear typing ensures the single pointer property, disallowing aliasing, but allowing safe in-place-update compilation of programming language. HBAL was designed as a target low-level language for Hofmann's LFPL programs that run in a bounded amount of heap space.

The described implementation includes a LFPL-to-HBAL compiler, the typechecker, and a machine model that execute HBAL programs. The system was implemented using the Ocaml language. The typechecker and machine model can be used as a Trusted Computing Base by a code consumer in an environment that uses mobile code.

The implementation is simplified by the fact that HBAL has a clear distinction between its syntax and its operational semantics: the store is not mentioned in its syntax or static semantics and the well typedness of a HBAL program is independent of the memory contents. Since it is unrealistic to expect to know the content of the memory where mobile code will be run to be able to establish its safety, this approach splits the concerns as one might expect. The typechecker can build a safety proof for execution on *any safe* memory, and the machine model only needs to check that the initial memory is safe for each run.

The compiler and typechecker can be used on-line, with examples of well behaved programs and programs with errors, at the web page:

`http://www.cs.stevens-tech.edu/~abc/hbal/`

Further research could integrate into HBAL a general scheme for defining high-level types apart from just lists and trees. Besides, some constructors could be added to allow a more flexible layout in memory, such as *null* pointers and a `caseptr` discriminator.

The implementation presented in this paper could be extended to compile different languages to HBAL. This extension may include languages without *resource awareness* that produce programs that are not *memory safe*. In such case the typechecker will work as a filter accepting only *safe* programs from the high-level language.

6 Acknowledgments

We would like to thank David Aspinall for sharing with us the sources of the implementation of LFPL. We also want to thank anonymous referees for suggestions and comments on an earlier version of this paper.

References

[1] David Aspinall and Adriana Compagnoni. Heap bounded assembly language. Technical Report 2002-2, Stevens Institute of

Technology. Department of Computer Science, 2002. Available at http://www.cs.stevens-tech.edu/˜abc/publications/hbal.ps.

[2] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of the 14th Symposium on Logic in Computer Science (LICS '99)*, 1999.

[3] Martin Hofmann. Typed lambda calculi for polynomial-time computation, 1999. Habilitation thesis, TU Darmstadt, Germany. Edinburgh University LFCS Technical Report, ECS-LFCS-99-406.

[4] Martin Hofmann. Programming languages capturing complexity classes. *SIGACT News Logic Column*, 9, 2000. 12 pp.

[5] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. An extended abstract appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[6] X. Leroy, D. Doligez, J. Garrigue, D. Rèmy, and J. Vouillon. The objective caml system release 3.04: Documentation and user's manual. Technical Report on-line manual, Institut National de Recherche en Informatique et en Automatique, December 2001.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[8] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *In the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[9] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[10] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *lncs*, pages 366–381. Springer-Verlag, April 2000.

[11] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Third International Workshop on Types in Compilation*, Montreal, Canada, September 2000.

[12] Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology, July 1999.