Heap Bounded Assembly Language

David Aspinall (da@ed.ac.uk)
LFCS, Division of Informatics, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, U. K.

Adriana Compagnoni (abc@cs.stevens-tech.edu)
Department of Computer Science, Stevens Institute of Technology, Castle Point on Hudson, Hoboken, NJ 07030, U. S. A.

Abstract. We present 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 programming languages. We prove that HBAL is sound for a low-level untyped model of the machine, using a satisfiability relation which captures when a location correctly models a value of some type. This interpretation is closer to the machine than previous abstract machines used for typed assembly language models, and we separate typing of the store from an untyped operational semantics of programs, as would be required for proof-carrying code.

Our ultimate aim is to design a family of assembly languages which have high-level typing features which are used to express *resource bound* constraints. We want to link up the assembly level with high-level languages expressing similar constraints, to provide *end-to-end* guarantees, and a viable framework for proof-carrying code. HBAL is a first exemplifying step in this direction. It is designed as a target low-level language for Hofmann's LFPL (Hofmann, 2000b) language. Programs written in LFPL run in a bounded amount of heap space, and this property carries over when they are compiled to HBAL: the resulting program does not allocate store or assume an external garbage collector. Following LFPL, we include a special *diamond* resource type which stands for a unit of heap space of uncommitted type, similar in spirit to Tofte-Talpin's notion of *region*.

1. Introduction

Resource awareness is a crucial asset. Despite powerful computing hardware and effective optimizing compilers, efficiency concerns are often still paramount, particularly when computing resources are limited, such as in embedded and real-time systems or for applications to be run across the Internet. For embedded and real-time systems, programmers traditionally resorted to assembler code (or assembler-close fragments of 'C') to ensure a close control over resource consumption. For Internet applications, and lately for more powerful small devices, high-level languages such as Java *are* used, but it remains the responsibility of the programmer to argue that any resource bounds are met.

Programming with low-level languages is tedious and error prone. Recent research on *typed assembly languages* (Morrisett et al., 1999;

Crary and Morrisett, 1999; Crary et al., 1999; Xi and Harper, 1999), and type-safe versions of C (Necula et al., 2002; Jim et al., 2002) has begun to make the prospect of type-safe low-level programming a reality. But these frameworks provide no way of *guaranteeing* that resource bounds are met, and anyway, we would rather program in high-level languages to begin with.

Our research is into the design of type-systems for both high-level and low-level languages which can provide *end-to-end static guarantees* of resource-boundedness. We see a type system as a convenient and elegant way of expressing a *semantic constraint* such as particular forms of resource boundedness. For us, the *proof* in proof-carrying code is a *typing derivation* (or annotations sufficient to easily reconstruct it) in a type system for the low-level code. This typing derivation is generated by a compiler for the high-level language with a related type system.

The particular resource we are concerned with in this paper is *heap space*. To provide an end-to-end guarantee, we want to add resource-bounded type systems both to high-level languages and to low-level languages. Any compilation from high-level to low-level must be shown to preserve the resource bound.

In the first half of the paper we describe a first-order *heap-bounded assembly language*, dubbed HBAL. The type system of HBAL allows for safe reuse of heap space for elements of different types, which can be used to capture in-place update compilation of functional programming languages with similar type systems, providing a *static guarantee* of bounded heap-space usage. We illustrate this claim in the second half of the paper with a compiling function from Hofmann's LFPL (Hofmann, 2000b) into HBAL. The type system of HBAL was designed to marry well with that of LFPL, to provide a direct demonstration of an end-to-end guarantee.

1.1. Relation to other typed assembly languages

Before introducing HBAL, we will briefly relate our contribution to existing work. Typed assembly languages have been an active subject of study for several years now, and experimentation with different ideas is still taking place. Contributions include TAL (Morrisett et al., 1999; Crary et al., 1999), STAL (Morrisett et al., 1998), DTAL (Xi and Harper, 1999), and Alias Types (Smith et al., 2000; Walker and Morrisett, 2000). So far, new type systems have been introduced to type more programs than was possible in previous systems.

TAL began with higher-order functions and polymorphism, considering System F as a high-level language.[1] But TAL assumed a particular compilation technique, continuation-passing style, which is not used by every compiler. STAL addressed this by modelling 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 does not set out to type more low-level programs than previous systems. Instead, we begin with 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 in programming languages capturing complexity classes (Hofmann, 1999b; Hofmann, 2000a; Hofmann, 1999a; Hofmann, 2000b).

The motivation behind HBAL is to bring a type system approach to resource bounds to a low-level language. The restricted type system means that we can provide an end-to-end guarantee that space bounds are respected in the compiled code, by preservation of typing. Ultimately, we will be interested in richer typing constructs at the low-level, but only once we know how to deal with these in our higher-level language with resource-sensitive typing.

Whereas previous systems rely on a garbage collector to reclaim unreachable data on the heap, HBAL uses linear typing to prevent aliasing, and includes pseudo-instructions for safely altering the types of heap locations. The idea of linear typing is not new; even for typed low-level languages it was mentioned as a possibility for solving the alias problem by Crary and Morriset (Crary and Morrisett, 1999). Linear typing may seem rather restrictive for typing arbitrary low-level programs, but of course it is ideal for the low-level programs which arise from compiling our intended high-level language, since that is also linearly typed. (In fact, the system is not purely linear, and the combination of linear typing with the resource type $\diamond$ seems to offer a promising new direction for linear schemes; more is said later).

Linear typing means that we do not need to assume an external garbage collector (or other memory allocator), and we can provide a static guarantee that every program runs in a fixed amount of heap space. This is useful when space is limited, such as for embedded systems or smart card applications. Programs in those domains often follow the pattern of initializing memory at start-up and then working

[1] although the type system of System F already goes beyond most present day programming languages!

within that memory during execution, which fits well with our typing discipline.

As an aside, we mention that HBAL is certinaly not incompatible with garbage collection; it can easily be extended with `malloc` and `free` if desired, although necessarily sacrificing the guarantee of heap boundedness (see Section 5.2). One advantage of doing this would be that the memory-allocating initialization part of an application could be typed within the system.

Some more comparisons between HBAL and other typed assembly languages are given at the end of the paper, in Section 6.

1.2. The type structure of HBAL

HBAL has high-level types in its syntax, with fixed memory layout schemes and pseudo-instructions for constructors and destructors. The advantage of this is that we push data abstraction closer to the machine model: by restricting the assembly language to use type-safe pseudo-instructions we know that data abstraction cannot be violated, and our end-to-end guarantees are precise. The disadvantage of this method is that it may obscure some low-level optimizations. These could be deferred to a final untyped phase, but we would rather eventually include them within our analysis. See (Crary and Morrisett, 1999) for more discussion of this tradeoff.

As prototypical examples of high-level types we use lists and binary trees. We could easily instead adopt general notions of recursive data type and mechanisms for specifying layout conventions, following ideas like those used in FLINT (Shao, 1997) and other typed assembly languages. We use a uniform boxed representation for lists and trees, with the same layout for each case in a sum datatype. So a *leaf* always takes as much space in memory as a *node*. For lists, we have empty lists or cons cells:[2]

$$nil : \begin{array}{|c|}\hline 0 : \texttt{int} \\ \hline \_ : A \\ \hline \_ : [L(A)] \\ \hline \end{array} \qquad\qquad cons(a, l) : \begin{array}{|c|}\hline 1 : \texttt{int} \\ \hline a : A \\ \hline l : [L(A)] \\ \hline \end{array}$$

For trees, we use labelled leaves and labelled nodes:

$$leaf(a) : \begin{array}{|c|}\hline 0 : \texttt{int} \\ \hline a : A \\ \hline \_ : [T(A)] \\ \hline \_ : [T(A)] \\ \hline \end{array} \qquad\qquad node(a, l, r) : \begin{array}{|c|}\hline 1 : \texttt{int} \\ \hline a : A \\ \hline l : [T(A)] \\ \hline r : [T(A)] \\ \hline \end{array}$$

[2] This is not the most efficient representation, but it follows an easily generalized pattern. Using a slightly different formulation of LFPL, one can use the more familiar null pointers for *nil*, see Section 4 and (Aspinall and Hofmann, 2002).

HBAL types make the use of pointers explicit: one of the cells above has type $L(A)$ (list of A) or $T(A)$ (tree of A), whereas a pointer to such a cell has type $[L(A)]$ or $[T(A)]$.

The *resource type* $\Diamond$ (diamond) stands for some fixed amount of space on the heap, and it is motivated by its introduction in LFPL, where it represents an abstract notion of space manipulated by the programmer. A diamond is used to store a cell from one of the structured types (a list element or tree node). Diamonds need to be big enough to store the largest cell of any datatype; their size is statically determined from the program being assembled. If the largest type used is $T(\mathtt{int} \times \mathtt{int} \times \mathtt{int})$, for example, then $\mathit{size}(\Diamond) = 6$ (one word for the tag, one workd for each subtree pointer and 3 words for the data). Notice that there is an amount of wastage inherent in this scheme; in return for this we have a simple and direct scheme for managing heap space via the type system, and the *guarantee* of heap boundedness by using in-place update rather than multiple allocation. Section 5.3 has some ideas for reducing wastage by generalizing to allow different sizes of diamonds.

When some location m has type $\Diamond$ we know that we have a small region beginning at m, which can be used to store an element in some heap-allocated data-structure. If a register r_i contains a pointer to some heap space, its type is $[\Diamond]$ (pointer to a diamond). The instruction `use` r_i $L(A)$ then signifies the intention to use r_i for holding a pointer to an element in a list type. The `use` instruction is purely a typing directive which is erased by the assembler. It changes the type of r_i to be an *uninitialized* version of L(A), which exposes its structure as a product type.

The type structure also includes the idea of *initialization flags* as in TAL (Morrisett et al., 1999), so that $\mathtt{int}^0$ is the type of uninitialized integers, which cannot be read from, whereas $\mathtt{int}^1$ is the type of initialized integers. We also use initialization flags to help ensure the linearity constraints.

Here is a code fragment which constructs the list `2::nil`, using two diamonds. We show the register typing assumptions at each step, underlining the changes which should be thought of as the typing effect of executing the instructions:

$$
\begin{array}{ll}
r_1 : [\Diamond], r_2 : [\Diamond] & \mathtt{use} \quad r_1 \quad L(\mathtt{int}^1) \\
r_1 : \underline{[\mathtt{int}^0 \times \mathtt{int}^0 \times [L(\mathtt{int}^1)]^0]}, r_2 : [\Diamond] & \mathtt{fold\text{-}nil}_{\mathtt{int}^1} \quad r_1[0] \\
r_1 : \underline{[L(\mathtt{int}^1)]}, r_2 : [\Diamond] & \mathtt{arithi} \quad r_3 \longleftarrow r_0 + 2 \\
r_1 : [L(\mathtt{int}^1)], r_2 : [\Diamond], \underline{r_3 : \mathtt{int}} & \mathtt{use} \quad r_2 \quad L(\mathtt{int}^1) \\
r_1 : [L(\mathtt{int}^1)], & \\
\quad r_2 : \underline{[\mathtt{int}^0 \times \mathtt{int}^0 \times [L(\mathtt{int}^1)]^0]}, r_3 : \mathtt{int} & \mathtt{store} \quad r_2[1] \longleftarrow r_3 \\
r_1 : [L(\mathtt{int}^1)], & \\
\quad r_2 : [\mathtt{int}^0 \times \underline{\mathtt{int}^1} \times [L(\mathtt{int}^1)]^0], r_3 : \mathtt{int} & \mathtt{store} \quad r_2[2] \longleftarrow r_1 \\
r_2 : [\mathtt{int}^0 \times \mathtt{int}^1 \times \underline{[L(\mathtt{int}^1)]^1}, r_3 : \mathtt{int}] & \mathtt{fold\text{-}cons}_{\mathtt{int}^1} \quad r_2[0] \\
r_2 : \underline{[L(\mathtt{int}^1)]}, r_3 : \mathtt{int} & \ldots
\end{array}
$$

(we assume that r_0 always holds 0 and omit the assumption $r_0 : \mathtt{int}$). The first `use` instruction transforms a diamond into an uninitialized list of integers, which exposes its three-tuple representation. The pseudo-instruction `fold-nil` makes the list `nil`, by setting the tag to 0. The next four instructions prepare the cons cell. Notice that the second store causes r_1 to disapear from the context, preserving the single pointer property. The final `fold-cons` instruction sets the tag on the cons cell to 1, folding the components into an initialized list.

To inspect a structured type, we break it apart using `case` instructions which are inverse to the `fold` instructions. If we have a pointer to a node of structured type in r_i, we can use the instruction `discard` r_i to change the type back to $[\Diamond]$ again.

The full HBAL language includes pseudo-instructions for managing the stack and for doing procedure calls. We show HBAL code using all of these constructs in Section 4.

The rest of this paper is structured as follows. We describe HBAL in detail in Section 2. In Section 3 we define a machine model and establish type-soundness of well-typed HBAL programs with respect to the model. In Section 4 we briefly review LFPL and its compilation to HBAL. The compilation preserves typing, which means that the resource usage model in LFPL is preserved in HBAL. By the soundness of HBAL typing with respect to the machine model, this means that the resource constraints are really met by executing programs. Section 6 concludes the paper and mentions some related and future work.

2. HBAL

2.1. Types and contexts

The types of HBAL are given by the grammar:

$$\begin{array}{rcl} A & ::= & \texttt{code} \quad | \quad w^z \quad | \quad A \times A \quad | \quad L(A) \quad | \quad T(A) \quad | \quad \Diamond \\ w & ::= & \texttt{int} \quad | \quad [A] \\ z & ::= & 0 \quad | \quad 1 \end{array}$$

We use A to range over *structured types*, which may use many words of memory, and w to range over *word types*, which always use one word of memory. A word type (w) is either an integer or a pointer ($[A]$). Word types are tagged with an initialization flag, z, which can be either 0 or 1 indicating an uninitialized or an initialized value respectively. The special type `code` indicates the type of an assembly opcode; our system distinguishes code from data.

We assume the machine has some number of registers $r_0 \dots r_n$. We use r, r_i, r_j to range over registers, and assume 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 w, treated as a set of type assignments $r_i : w$. We write the initial context as $\{\}$, and we assume by convention that this context contains the type assignment $r_0 : \texttt{int}$.

When we write the extended context $\Gamma, r_i : w$, 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 $\Gamma_{\setminus r_i}$ is defined to be the same as Γ except undefined on r_i. Notice that there are no outermost initialization flags in contexts; the accessibility of a register is determined by whether it appears in Γ.[3]

2.2. Operations and notations for types

We define $size(A)$ to be the size of the type A as it is laid out in memory, given by induction on the structure of A: $size(\texttt{code}) = 1$, $size(w^z) = 1$, $size(A \times A') = size(A) + size(A')$, $size(L(A)) = 2 + size(A)$, and $size(T(A)) = 3 + size(A)$. As we mentioned before, $size(\Diamond)$ is a constant which can be determined statically from the program.

When A is a compound type and c is a non-negative integer, we use the notation $A[c]$ to denote the type of the cth word in the layout of A in memory. However, we want the type system to prevent accessing

[3] This is merely a design choice to emphasise a link with linear type systems; one could just as well include outermost initialization flags in contexts and include all registers. This approach would be more familiar to those who know TAL.

arbitrary words inside a structured type; instead we should only be allowed to access "top-level" words in a type. We formalize this by defining $A[c]$ as a partial function, by induction on A:

$$\begin{array}{lcll} w^z[0] & = & w^z & \\ (A \times A')[c] & = & A[c] & \text{if } c < \mathit{size}(A) \\ (A \times A')[c] & = & A'[c - \mathit{size}(A)] & \text{if } c \geq \mathit{size}(A) \end{array}$$

We allow access to memory locations only if they have a type of the form $A[c]$, precluding directly reading from locations with types `code`, $\Diamond$, or the high-level list and tree types; we can only manipulate pointers to those kinds of data. Additionally, we sometimes restrict to `code`-*free types*, which are types in which `code` appears only as [`code`], if at all.

Because the type system only allows fixed constant offsets to be used for c, forbidding pointer arithmetic, and because the same amount of space is used for each variant in a sum type, we can tell statically if $A[c]$ is defined. When $A[c]$ is defined, the notation $A^{c:=1}$ denotes A with the initialization flag set on the cth word, and $A^{c:=0}$ denotes A with the cth initialization flag cleared. Uninitialization of whole types is defined as follows:

$$\begin{array}{lcl} \mathtt{code}^{:=0} & = & \mathtt{code} \\ (w^y)^{:=0} & = & w^0 \\ (A \times B)^{:=0} & = & A^{:=0} \times B^{:=0} \\ (L(A))^{:=0} & = & \mathtt{int}^0 \times A^{:=0} \times [L(A)]^0 \\ (T(A))^{:=0} & = & \mathtt{int}^0 \times A^{:=0} \times [T(A)]^0 \times [T(A)]^0 \\ \Diamond^{:=0} & = & \Diamond \end{array}$$

Some examples may help: $L(\mathtt{int}^0)$ is the type of lists of uninitialized integers. If we had such a list, we should be able to traverse it and store integers into it, but not be able to read integers from it beforehand. An uninitialized list has an unfolded type, so $L(\mathtt{int}^0)^{:=0} = \mathtt{int}^0 \times \mathtt{int}^0 \times [L(\mathtt{int}^0)]^0$. The last component is not unfolded because it is a *pointer* to a structured type, not directly a structured type. Only this last component gives away the intended use of the type; compare with an uninitialized list of initialized integers, $L(\mathtt{int}^1)^{:=0} = \mathtt{int}^0 \times \mathtt{int}^0 \times [L(\mathtt{int}^1)]^0$.

2.3. Subtyping

We define a subtyping relation on types, given by the contextual, reflexive, and transitive closure of the uninitialization operation, so $B(A) \leq B(A^{:=0})$. The relation can be defined more explicitly by induction on types, as below.

DEFINITION 1 (Subtyping). *$A \leq A'$ is defined by the following rules.*

$$\texttt{code} \leq \texttt{code} \qquad \texttt{int}^0 \leq \texttt{int}^0 \qquad \texttt{int}^1 \leq \texttt{int}^1 \qquad \texttt{int}^1 \leq \texttt{int}^0$$

$$\Diamond \leq \Diamond$$

$$\frac{A \leq A'}{[A]^0 \leq [A']^0} \qquad \frac{A \leq A'}{[A]^1 \leq [A']^0}$$

$$\frac{A \leq A'}{[A]^1 \leq [A']^1} \qquad \frac{A \leq A' \qquad B \leq B'}{A \times B \leq A' \times B'}$$

$$\frac{A \leq A'}{L(A) \leq L(A')} \qquad \frac{A \leq A'}{T(A) \leq T(A')}$$

As particular cases we have, for example, that $w^1 \leq w^0$, $[\texttt{int}^1]^1 \leq [\texttt{int}^0]^1$, and $L(\texttt{int}^1) \leq L(\texttt{int}^0)$.

Subtyping is introduced in the type system by explicitly changing the context. We write $\mathbf{\Gamma}' \leq \mathbf{\Gamma}$ if for every typing $r_i : w$ in Γ there exists $r_i : w'$ in Γ', such that $w'^1 \leq w^1$.

2.4. Assembly language programs

A *program* P consists of a sequence of instructions p and labels l. We use a small set of standard instructions, together with several pseudo-instructions.

$$
\begin{array}{rcl}
P & ::= & \langle\rangle \;\mid\; p \;;\; P \;\mid\; l \;;\; P \\[1ex]
p & ::= & \texttt{load}\;\; r \longleftarrow r[c] \;\mid\; \texttt{store}\;\; r[c] \longleftarrow r \\
& \mid & \texttt{arithi}\;\; r \longleftarrow r \odot c \;\mid\; \texttt{arith}\;\; r \longleftarrow r \odot r \\
& \mid & \texttt{bnz}\;\; r\;\; l \;\mid\; \texttt{bez}\;\; r\;\; l \;\mid\; \texttt{jmp}\;\; l \\
& \mid & \texttt{call}\;\; l \;\mid\; \texttt{ret}\;\; l \\
& \mid & \texttt{salloc}\;\; A \;\mid\; \texttt{sfree}\;\; c \\
& \mid & \texttt{use}\;\; r\;\; A \;\mid\; \texttt{discard}\;\; r \\
& \mid & \texttt{fold-nil}_A\;\; r[c] \;\mid\; \texttt{fold-cons}_A\;\; r[c] \\
& \mid & \texttt{fold-leaf}_A\;\; r[c] \;\mid\; \texttt{fold-node}_A\;\; r[c] \\
& \mid & \texttt{caselist}_A\;\; r[c]\;\; l \;\mid\; \texttt{casetree}_A\;\; r[c]\;\; l \\[1ex]
\odot & ::= & + \;\mid\; - \;\mid\; * \;\mid\; /
\end{array}
$$

We use c to range over (non-negative) constant offsets used in the load and store instructions; the type system does not allow us to find the type of a negative offset from a location. As far as typing is concerned, there is no difference between the arithmetic operations, so we use the generic `arithi` instruction, where $\odot \in \{+,-,*,/\}$ ranges over the arithmetic operators.

The pseudo-instructions (`salloc` onwards) are either typing directives which are erased by the assembler, or macros which are expanded into short sequences of untyped ordinary instructions. The idea is to replace certain otherwise untypable instruction sequences with type-safe pseudo-instructions. To compile recursive functions, we use the pseudo-instructions `call` and `ret`, which expand to instructions which manipulate the stack and program counter. For heap and stack management, we have four pseudo-instructions: `use` and `discard`, which are are typing directives erased by the assembler, and `salloc` and `sfree` ("stack allocate", "stack free"), which the assembler replaces with addition or subtraction instructions which move the stack pointer. We also have pseudo-instructions for manipulating data with high-level types, as mentioned before; the fold instructions expand to instructions which set a tag field for the datatype and the case instructions expand to a test-and-branch sequence of code which allows the assembly language to process different cases of a sum datatype in a type-safe way. Expansions of the pseudo-instructions are shown in Section 3.1.

A program must be given together with a *signature*, Σ. The signature assigns either *procedure types* or contexts to labels. Labels are given for subroutines and branch targets. For subroutine labels l, $\Sigma(l)$ is a procedure type, which has the form $A_1, \ldots, A_n \rightarrow A$ for $n \geq 0$. A procedure type gives rise to a type for the stack pointer:

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

which supports a simple calling convention. 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. We begin typing the subroutine with $A^{:=0}$ on the stack, but when the return statement is hit we must find A.

For labels l which are branch targets, $\Sigma(l)$ is a context containing typing assumptions for live registers. When we write code fragments, we give $\Sigma(l)$ next to l itself. For type-checking we will assume that the signature is supplied with the program, although in reality a separate first pass could be made to infer some parts of it.

In the sequel we will usually assume a fixed program and associated signature.

2.5. Typing rules

Our rules define a judgement $\Gamma \ \vdash \ P$ which means that P is a well-typed assembly program in context Γ. By convention, $\Gamma \ \vdash \ \langle\rangle$. To find a context to begin type-checking P, we use the trivial operation $\mathsf{Ctxt}(P)$:

$$\begin{array}{lcll} \mathsf{Ctxt}(l \ ; \ P) & = & \Gamma_l & \text{when} \quad \Sigma(l) = \Gamma_l \\ \mathsf{Ctxt}(l \ ; \ P) & = & \{\} & \text{when} \quad \Sigma(l) = A_1, \ldots, A_n \ \rightarrow \ A \\ \mathsf{Ctxt}(\langle\rangle) & = & \{\} & \\ \mathsf{Ctxt}(P) & = & \text{undefined, otherwise} & \end{array}$$

Subroutine labels begin from the empty context; the type-checking rule for the label itself sets the context. Branch labels, on the other hand, check the current context is a subtype of $\Sigma(l)$, which is why we need this operation. We say that a program P is *well-typed* if $\mathsf{Ctxt}(P) \ \vdash \ P$.

Typing rules for load and store instructions

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

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

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

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

At the heart of the type system, there is the distinction between pointers and non-pointers, and the typing rules for the load and store instructions reflect this. Notice that by the conventions for contexts, each of the four rules assumes the source register is distinct from the destination register.

The first `load` rule allows us to read a non-pointer word (an integer) into a register r_i with impunity, provided that the word is a top-level word in the type of the source register r_j, and that it is flagged as initialized. The context is updated to assign r_i the non-pointer (`int`) type. The second `load` rule allows us to read any word, but it enforces a linearity constraint. Once the word has been read, we adjust the typing for the source register r_j to treat the location as uninitialized, so it cannot be read again. This prevents the aliasing which could occur by reading a pointer from memory into two different registers.

The first `store` rule lets us store an integer word at an offset from a register, again corresponding to a top-level word in type of r_j. When we type the following portion of program, we modify the context to treat the location $r_j + c$ as initialized. The second `store` lets us store any word, but enforces a linearity constraint. Once the word has been written, the typing assumption for the source register r_i is removed from the context before typing the following portion of program. This prevents the aliasing which could occur by writing a pointer from a register into two different memory locations. This rule cannot be used with the stack pointer; the stack is always uniquely referenced from sp.

To summarise the linearity constraints: once a pointer is read from memory, we can no longer refer to its location; once a pointer is stored into memory, we can no longer use the register containing it. Furthermore, we cannot copy a pointer in one register to another: our underlying instruction set could achieve pointer copying with an arithmetic instruction like `addi` $r_i \longleftarrow r_j + 0$, but this is not typable when r_j is a pointer, as we see next.

Typing rules for arithmetic instructions

$$\frac{\Gamma_{\backslash r_i}, r_i : \mathtt{int}, r_j : \mathtt{int} \vdash P}{\Gamma, r_j : \mathtt{int} \vdash \mathtt{arithi} \quad r_i \longleftarrow r_j \odot c \quad ; \quad P}$$

$$\frac{\Gamma_{\backslash r_i}, r_i : \mathtt{int}, r_j : \mathtt{int}, r_k : \mathtt{int} \vdash P}{\Gamma, r_j : \mathtt{int}, r_k : \mathtt{int} \vdash \mathtt{arith} \quad r_i \longleftarrow r_j \odot r_k \quad ; \quad P}$$

The arithmetic rules only allow us to type arithmetic on integers, preventing copying a register containing a pointer to another register, or pointer arithmetic in general.

The only places we allow pointer arithmetic are for sp: we can move the stack pointer up with the pseudo-instruction `sfree` (which corresponds to an add instruction) and we can move it down with `salloc` (a subtraction).

$$\frac{\text{A is } \mathtt{code}\text{-free} \quad \Gamma, sp \ : \ [A^{:=0} \times A_{sp}] \vdash P}{\Gamma, sp \ : \ [A_{sp}] \vdash \mathtt{salloc} \quad A \quad ; \quad P}$$

$$\frac{\Gamma, sp \ : \ [A_{sp}] \vdash P \quad size(A) = c}{\Gamma, sp \ : \ [A \times A_{sp}] \vdash \mathtt{sfree} \quad c \quad ; \quad P}$$

The stack has a product type of varying length: each time we push something, an extra component is added to the left. The rule for `salloc` makes space for pushing something, ensuring that the space is treated as uninitialized by typing the rest of the program with the type $A^{:=0}$ for the new space. The rule for `sfree` removes c-words of space by matching the stack type with a type A that has size c to find the type of remainder, A_{sp}. Often we shall write `sfree` A as a short-hand for `sfree` $size(A)$.

The expansions of pseudo-instructions including `salloc` and `sfree` are shown later in Section 3.1.

Typing rules for jumps and branches

$$\frac{\Gamma \leq \Sigma(l) \quad \Sigma(l) \vdash P}{\Gamma \vdash l \;\; ; \;\; P}$$

$$\frac{\Gamma \leq \Sigma(l) \quad \mathsf{Ctxt}(P) \vdash P}{\Gamma \vdash \texttt{jmp} \;\; l \;\; ; \;\; P}$$

$$\frac{\Gamma, r_i : \texttt{int} \leq \Sigma(l) \quad \Gamma, r_i : \texttt{int} \vdash P}{\Gamma, r_i : \texttt{int} \vdash \texttt{bnz} \;\; r_i \;\; l \;\; ; \;\; P}$$

(The rule for `bez` r_i l is like that for `bnz` r_i l).

These rules let us type-check programs with non-linear control paths, and also introduce subtyping into the system. We rely on typing annotations already provided in the program signature (which might be inferred with the help of an extra pass and some live variable analysis). The first rule type-checks a label l in some context Γ. This rule is encountered when control drops in at l. The label context $\Sigma(l)$ is used to type-check the following code in P. For this to be sound, the register typings in $\Sigma(l)$ must already appear in the current context Γ, possibly with subtypes. The rules for jumps and branches are similar. If there is a possibility of control passing to a point labelled l, then all of the register typings assumed in the context Γ_l should be present in the current context Γ. In the case of a branch, we must assume that the branch might not be taken, and check the following instructions in context Γ again. In the case of a jump, control always passes elsewhere, so we begin checking the following code P with the appropriate context $\mathsf{Ctxt}(P)$.

Typing rules for subroutines

$$\frac{\begin{array}{l}\Sigma(l) = A_1, \ldots, A_n \;\rightarrow\; A \\ \Gamma \leq \{sp \;:\; [A_1 \times \cdots \times A_n \times [\mathsf{code}]^1 \times A^{:=0}]\} \\ sp \;:\; [A_1 \times \cdots \times A_n \times [\mathsf{code}]^1 \times A^{:=0}] \vdash P\end{array}}{\Gamma \vdash l \;\; ; \;\; P}$$

$$\frac{\Sigma(l) = A_1, \ldots, A_n \;\rightarrow\; A \quad sp \;:\; [[\mathsf{code}]^0 \times A \times A_{sp}] \vdash P}{\Gamma, sp \;:\; [A_1 \times \cdots \times A_n \times [\mathsf{code}]^0 \times A^{:=0} \times A_{sp}] \vdash \texttt{call} \;\; l \;\; ; \;\; P}$$

$$\frac{\Sigma(l) = A_1, \ldots, A_n \;\rightarrow\; A \qquad\qquad \mathsf{Ctxt}(P) \vdash P}{\Gamma, sp \;:\; [[\mathsf{code}]^1 \times A] \vdash \texttt{ret} \;\; l \;\; ; \;\; P}$$

The rule for subroutine labels checks the following code in a context which has a typing for the appropriate stack frame, preventing access beyond the frame. For a call instruction, the stack must be set up to match the subroutine label $\Sigma(l)$ following the calling convention. The space for the return value, $A^{:=0}$, is uninitialized. We don't know exactly what type the stack will have when we meet a call instruction, because we want to allow the call to work from any point, including recursively. So, as with the `sfree` rule, we let A_{sp} stand for the rest of the stack type. After the subroutine returns, control passes into P, which is type-checked in a context which assumes that the stack frame has been cleaned to just leave the return value, now set by the subroutine to have the correct type A.

At the end of the procedure body we hit the return instruction, where the stack must be cleaned to the point of the return location, and the return value has the correct type. We type-check the following code starting from the appropriate context, using $\mathsf{Ctxt}(P)$. The return instruction has to be annotated with the subroutine label it corresponds to match the types here.

The calling convention does not describe a way of saving registers across calls, but it can be achieved inefficiently by passing extra values into the function, and retrieving them as part of the return type A. (Another possibility would be to annotate subroutines with registers used, and allow some of Γ to be saved in the premise of the `call` rule). Notice that compared with STAL (Morrisett et al., 1998), we avoid stack polymorphism in favour of hard-wiring our calling convention with pseudo-instructions, keeping the type structure of HBAL simple.

Typing rules for memory pseudo-instructions

$$\frac{\text{A is } \mathtt{code}\text{-free} \quad r_j \neq sp \quad \Gamma, r_j \ :\ [A^{:=0}] \ \vdash\ P}{\Gamma, r_j \ :\ [\Diamond] \ \vdash\ \mathtt{use} \quad r_j \quad A \quad ; \quad P}$$

$$\frac{\text{A is } \mathtt{code}\text{-free} \quad r_j \neq sp \quad \Gamma, r_j \ :\ [\Diamond] \ \vdash\ P}{\Gamma, r_j \ :\ [A^{:=0}] \ \vdash\ \mathtt{discard} \quad r_j \quad ; \quad P}$$

The `use` instruction changes the type of a pointer r_j to be a pointer to some uninitialized space with a given type. Dually, if we have a diamond pointed to by r_j, we can reassign its type by using `discard`, which means we can `use` it again later with another type. Neither instruction works with the stack pointer. When the `discard` is encountered, the type system ensures that there are no other pointers to the location r_j, so it is safe to alter the type. Type soundness is crucial for the memory safety of this instruction, to prevent runtime type errors. The `use` instruction ensures that the new type is uninitialized, as $A^{:=0}$, so nothing can be read from it.

Typing rules for data pseudo-instructions

$$\frac{\Gamma, r_i \;:\; [C \times L(A) \times B] \;\vdash\; P \quad \mathit{size}(C) = c}{\Gamma, r_i \;:\; [C \times \mathtt{int}^0 \times A^{:=0} \times [L(A)]^0 \times B] \;\vdash\; \mathtt{fold\text{-}nil}_A \;\; r_i[c] \;\; ; \;\; P}$$

$$\frac{\Gamma, r_i \;:\; [C \times L(A) \times B] \;\vdash\; P \quad \mathit{size}(C) = c}{\Gamma, r_i \;:\; [C \times \mathtt{int}^0 \times A \times [L(A)]^1 \times B] \;\vdash\; \mathtt{fold\text{-}cons}_A \;\; r_i[c] \;\; ; \;\; P}$$

$$\frac{\begin{array}{l}\Gamma, r_i \;:\; [C \times \mathtt{int}^1 \times A^{:=0} \times [L(A)]^0 \times B] \;\vdash\; P \\ \Gamma, r_i \;:\; [C \times \mathtt{int}^1 \times A \times [L(A)]^1 \times B] \leq \Sigma(l_{\mathrm{cons}}) \quad \mathit{size}(C) = c\end{array}}{\Gamma, r_i \;:\; [C \times L(A) \times B] \;\vdash\; \mathtt{caselist}_A \;\; r_i[c] \;\; l_{\mathrm{cons}} \;\; ; \;\; P}$$

$$\frac{\Gamma, r_i \;:\; [C \times T(A) \times B] \;\vdash\; P \quad \mathit{size}(C) = c}{\Gamma, r_i \;:\; [C \times \mathtt{int}^0 \times A \times [T(A)]^0 \times [T(A)]^0 \times B] \;\vdash\; \mathtt{fold\text{-}leaf}_A \;\; r_i[c] \;\; ; \;\; P}$$

$$\frac{\Gamma, r_i \;:\; [C \times T(A) \times B] \;\vdash\; P \quad \mathit{size}(C) = c}{\Gamma, r_i \;:\; [C \times \mathtt{int}^0 \times A \times [T(A)]^1 \times [T(A)]^1 \times B] \;\vdash\; \mathtt{fold\text{-}node}_A \;\; r_i[c] \;\; ; \;\; P}$$

$$\frac{\begin{array}{l}\Gamma, r_i \;:\; [C \times \mathtt{int}^1 \times A \times [T(A)]^0 \times [T(A)]^0 \times B] \;\vdash\; P \\ \Gamma, r_i \;:\; [C \times \mathtt{int}^1 \times A \times [T(A)]^1 \times [T(A)]^1 \times B] \leq \Sigma(l_{\mathrm{node}}) \\ \mathit{size}(C) = c\end{array}}{\Gamma, r_i \;:\; [C \times T(A) \times B] \;\vdash\; \mathtt{casetree}_A \;\; r_i[c] \;\; l_{\mathrm{node}} \;\; ; \;\; P}$$

The typing rules for structured data operate at some arbitrary location within a type pointed to by r_i. The rules either make a new list or tree at $r_i[c]$, or decompose the existing one there. The fold rules match the type of r_i with fields which are suitably initialized for the constructor concerned. The pseudo-instruction itself is responsible for setting the tag (see Section 3.1), so this appears uninitialized before folding. The other uninitialized fields are needed in the nil and leaf case because each variant of a sum type has the same size and layout in memory.

The case rules begin with a list or tree at $r_i[c]$. The argument to a `caselist` or `casetree` is a label which is the destination of a branch instruction taken in the case of a cons or node constructor, respectively. The code immediately following the case instruction is followed for a nil or leaf constructor; it is typed in a context with a type for r_i corresponding to an unfolded nil or leaf. The label l_{cons} or l_{node} imposes a constraint on the branch context from Σ, to ensure it is compatible with the present context modified with a type for r_i for an unfolded cons or node.

It should be clear how to generalize the pattern here to deal with arbitrary recursive data types built with products and sums. For a datatype type with n summands, we would have n fold rules and a case rule with $n-1$ label arguments. For each datatype, we must design a uniform layout scheme which is the same for each constructor.

3. Type soundness of HBAL

Our goal in this section is to show a type soundness property for HBAL. We begin from a semantics for the untyped assembly instructions, operating on the register file R and a heap H in a machine model. Then we give an interpretation for HBAL types which captures the way datatypes are implemented in memory. With this we can express assertions like "r_1 contains a pointer to a list of integers". Finally, these ingredients allow us to prove a type soundness theorem that establishes that the execution of a typable HBAL program modifies structures on the heap correctly.

The interpretation for types is the central part of our proof. We define a relation $H \models_K m : A$ between locations m, heaps H (functions from locations to integers), types A, and a heap portions $K \subseteq \mathrm{dom}(H)$. The heap portion K is the set of locations which are reachable from m, according to the type A. We use K to enforce the single pointer property.

3.1. Translation of pseudo-instructions

The first thing to do is to explain the pseudo-instructions. Our pseudo-instructions are macros which abbreviate small pieces of code which are not typable directly in HBAL, or which would only have a weaker typing. We define a function $\mathsf{Asm}(p)$ on pseudo-instructions p, which defines the sequence of ordinary untyped instructions which results from assembling p. The code fragments below define $\mathsf{Asm}(p)$ by showing a pseudo-instruction on the left and its untyped expansion on the right.

The memory management instructions `use` and `discard` are simply erased. The stack instructions modify the stack pointer:

`salloc` A	`arithi` $sp \longleftarrow sp - (size(A))$
`sfree` A	`arithi` $sp \longleftarrow sp + (size(A))$

More realistically, we might include a check in `salloc` to prevent a stack overflow. In the machine model, we will assume that the stack is unbounded and never clashes with other locations in memory.

For the subroutine instructions, we assume that the assembler has resolved each label l to a memory location within the program, written $\mathsf{LAdr}(l)$. This is formalized in the machine model in the next section. The expansions are:

`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` $pc \longleftarrow sp[0]$

where $m = size(A_1) + \cdots + size(A_n)$ if $\Sigma(l) = A_1, \ldots, A_n \rightarrow A$, and we assume that the three machine instructions for call take 6 words of memory, so $pc + 6$ points to the instruction following the call.

The data instructions for lists and trees expand as:

`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}

The fold instructions each have a stronger typing than their expansions would; the special way that they are used guarantees that a valid list or tree element is made at $r_i[c]$. Similarly, the case testing instructions have a stronger typing than the test and branch statements would achieve, allowing the list or tree to be unfolded and considered in two separate cases in the code. This technique allows sum datatypes to be handled without needing dependent types in the source language. (However, it has the disadvantage of needing additional abstract pseudo-instructions, preventing some optimizations being made directly on the typed code.)

3.2. The machine model

First, let $\mathsf{Loc} \subseteq \mathbf{Z}$ stand for the set of memory locations on our machine, $\mathsf{Reg} = \{0, 1, \ldots, \mathsf{R}_{\max}\}$ 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. To simplify the presentation, we will assume that Wrd is disjoint from Code; our model keeps code separate from data, which allows us to establish safety properties about non-modification of code, for example.

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 $R(0) = 0$ and some further conditions hold, as follows. Apart from $R(0)$, two other registers are distinguished: the program counter, $R(pc)$, and the stack pointer, $R(sp)$. We talk loosely of H as the "heap" configuration, but it actually covers all memory portions of interest, including the space where the program and stack are kept. We make the *unbounded stack assumption* that every machine has the space to grow its stack downwards indefinitely, which 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, we will assume that $R(sp) \leq 0$ while locations used for program and heap data are positive. Now we can define the effect of each machine instruction (i.e., an untyped assembly language instruction) on a machine configuration.

DEFINITION 2 (Machine transitions). *Given a machine $M = (H, R)$ we define $M \leadsto M'$, using the table below, by case analysis on the instruction at $H(R(pc)) \in \mathsf{Code}$:*

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

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. For the load and arithmetic instructions we assume that $i > 0$; for $i = 0$ the operations on r_0 have no effect.

Notice that the relation $M \rightsquigarrow M'$ is a *partial* function on machines M. There may be no valid instruction at $H(R(pc))$, or one of H' or R' may be undefined because of an attempt to access a location not in the domain of H. The type soundness result guarantees that a well-typed program only reads or writes memory locations which are already defined in H, and doesn't write to locations containing code.

Given a program P, a *machine assembled for* P is a machine configuration M which contains a representation of the assembly language program, with machine instructions are stored in some designated contiguous portion(s) of the heap. The assembly process is the obvious one: the assembler erases typing information from the program by expanding (or erasing) every pseudo-instruction p to the sequence $\mathsf{Asm}(p)$ described in Section 3.1. Supposing $P = \langle p_1, \ldots, p_n \rangle$, the assembly process defines a function $\mathsf{PAdr} : 1, \ldots, n \to \mathrm{dom}(H)$ which gives the destination location for the code when assembling the typed instruction p_u, where $1 \leq u \leq n$. (If p_u isn't erased, $\mathsf{PAdr}(u)$ will contain a machine instruction corresponding to or beginning p_u). For each of the locations m where P is stored, $H(m) \in \mathsf{Code}$. The assembly process also defines the function which yields the code location for each label, such that if $p_u = l$, then $\mathsf{LAdr}(l) = \mathsf{PAdr}(u)$. This value is used as the address in the machine instruction for jump and branch instructions (written as x in Definition 2). To slightly simplify the next definition and the proofs, we assume that every code location $\mathsf{LAdr}(l)$ has at least $\mathit{size}(\Diamond)$ code words following it in the heap, if necessary by expanding the program with dummy instructions.

3.3. Imposing types on the model

Given a machine configuration $M = (R, H)$, we define the satisfaction relation $H \models_K m : A$ which captures when the location m represents a valid element of type A in heap H.

DEFINITION 3 (Heap typing rules).

$$\frac{H(m) \in \mathsf{Code}}{H \models_{\{m\}} m : \mathtt{code}} \qquad \frac{H(m) \in \mathsf{Wrd}}{H \models_{\{m\}} m : \mathtt{int}^z}$$

$$\frac{H(m) \in \mathsf{Wrd}}{H \models_{\{m\}} m : [A]^0} \qquad \frac{H \models_K m : \Diamond/0}{H \models_K m : \Diamond}$$

$$\frac{c \leq \mathit{size}(\Diamond) \quad H(m+c) \in \mathsf{Wrd} \ \cdots \ H(m+\mathit{size}(\Diamond)-1) \in \mathsf{Wrd}}{H \models_{\{m+c,\ldots,m+\mathit{size}(\Diamond)-1\}} m : \Diamond/c}$$

$$\frac{\begin{array}{ll} H \models_K H(m) : A & m \notin K, K_d \\ H \models_{K_d} H(m) : \Diamond/\mathit{size}(A) & K \cap K_d = \{\} \end{array}}{H \models_{K \cup K_d \cup \{m\}} m : [A]^1}$$

$$\frac{H \models_{K_1} m : A_1 \quad H \models_{K_2} m + \mathit{size}(A_1) : A_2 \quad K_1 \cap K_2 = \{\}}{H \models_{K_1 \cup K_2} m : A_1 \times A_2}$$

$$\frac{H(m) = 0 \quad H \models_K m : \mathtt{int}^1 \times A^{:=0} \times [L(A)]^0}{H \models_K m : L(A)}$$

$$\frac{H(m) = 1 \quad H \models_K m : \mathtt{int}^1 \times A \times [L(A)]^1}{H \models_K m : L(A)}$$

$$\frac{H(m) = 0 \quad H \models_K m : \mathtt{int}^1 \times A \times [T(A)]^0 \times [T(A)]^0}{H \models_K m : T(A)}$$

$$\frac{H(m) = 1 \quad H \models_K m : \mathtt{int}^1 \times A \times [T(A)]^1 \times [T(A)]^1}{H \models_K m : T(A)}$$

Definition 3 formalizes the meaning of HBAL types. The heap is partitioned between code and data. Uninitialized pointers are unconstrained, but pointers which are initialized must point to a location with the correct type on the heap. Any pointer is associated with enough space for a diamond. The conditions on K ensure that the single pointer property holds, so there cannot be sharing or cycles in data-structures stored in H, and the spare space in diamonds is not referenced (the $\Diamond/c$ construct is used to pick out this spare space). Note that even locations with uninitialized word types are defined in H; the heap models the whole memory available to the program. (With the untyped machine of Definition 2, we can't expect to precisely track the behaviour of initialization flags in types.)

3.4. Type soundness

We begin from a notion of satisfiability $M \models \Gamma$, which expresses that a machine configuration M is consistent with a typing assignment Γ. Registers pointing into the heap must have types which are valid for the heap, and with the exception of the stack pointer, point somewhere where there is a diamond-sized portion of space available. Additionally, satisfiability requires a *heap separation* property, that there is no overlap between the portions of the heap accessible from the registers declared in Γ.

DEFINITION 4 (Satisfiability). *Given $M = (H, R)$, we define the relation $M \models_K r_i : w$ by cases:*

$$\frac{}{M \models_{\{\}} r_i : \mathtt{int}} \qquad \frac{H \models_K R(sp) : A \quad R(sp) \leq 0}{M \models_K sp : [A]}$$

$$\frac{\begin{array}{ll} H \models_K R(i) : A & R(i) > 0 \\ H \models_{K_d} R(i) : \Diamond/\mathit{size}(A) & K \cap K_d = \{\} \;\; r_i \neq sp \end{array}}{M \models_{K \cup K_d} r_i : [A]}$$

Then $M \models \Gamma$ holds iff for each $r_i \in \Gamma$, there exists a K_i such that $M \models_{K_i} r_i : A$, and moreover, the K_i are pairwise disjoint.

Fix a typable program $P = \langle p_1, \ldots, p_u, \ldots, p_n \rangle$. The derivation of $\mathsf{Ctxt}(P) \;\vdash\; P$ determines a series of contexts $\Gamma_1, ..., \Gamma_n$ which occur before each p_u in the conclusion of the typing rule for p_u, i.e., we have sub-derivations of the form $\Gamma_u \;\vdash\; p_u \;\; ; \;\; \langle p_{u+1}, \ldots, p_n \rangle$. The context Γ_u must be satisfied before executing p_u.

DEFINITION 5 (Type Safety). *Given a machine $M = (R, H)$ we say that M* is type safe (for P) at u *if:*

1. *M is assembled for P*
2. *$R(pc) = \mathsf{PAdr}(u)$* *(M is about to execute near p_u)*
3. *$M \models \Gamma_u$* *(M satisfies the typing context for p_u)*

The typing rules should preserve type safety. To state this formally, it helps to relate execution in the machine M with the control paths in the typed program P. We introduce a non-deterministic transition relation for the typed program.

DEFINITION 6 (Program transitions). *For a program $P = \langle p_1, \ldots, p_n \rangle$, we define a relation $p_u \rightsquigarrow p_v$ which holds between pairs of instructions indexed by the set:*

$$\begin{array}{l}
\{\, (i, i+1) \;\big|\; p_i \neq \mathtt{jmp}, \mathtt{call}, \mathtt{ret}, \quad \textit{and} \quad i < n \,\} \\
\cup \\
\left\{ (i,j) \left| \begin{array}{l} p_j = l,\ p_i = \mathtt{bnz}\, r_i\, l,\ \mathtt{bez}\ r_i\ l,\ \mathtt{jmp}\ l, \\ \mathtt{call}\ l, \mathtt{caselist}_A\ \ r_i[c]\ \ l,\ \textit{or}\ \mathtt{casetree}_A\ \ r_i[c]\ \ l \end{array} \right. \right\} \\
\cup \\
\{\, (i, j+1) \;\big|\; p_i = \mathtt{ret}\quad l, p_j = \mathtt{call}\quad l \quad \textit{and} \quad j < n \,\}.
\end{array}$$

The program transitions $p_u \rightsquigarrow p_v$ are an approximation to execution in the typed program; we don't have an operational semantics directly for the typed program but only for the untyped machine. Recall that $\mathsf{Asm}(p_u)$ stands for the sequence of ordinary untyped machine instructions which is the result of assembling the instruction p_u, expanding pseudo-instructions as described in Section 3.1. We will write $M \rightsquigarrow^{\mathsf{Asm}(p_u)} M'$ if M executes to M' through the instructions in $\mathsf{Asm}(p_u)$, by zero or more transitions in M. When p_u is a typing directive such as `use`, then $\mathsf{Asm}(p_u)$ is empty, and $M' = M$.

The following properties will be used in the main correctness theorem.

LEMMA 1 (Properties of heap typing).

1. ***Heap definedness.*** *If $H \models_K m : A$ then $\{m, \ldots, m + \mathit{size}(A) - 1\} \subseteq K \subseteq \mathrm{dom}(H)$. Moreover, if $A[c]$ is defined, then $H(m + c) \in \mathsf{Wrd}$, and if A is `code`-free, then $H(m + d) \in \mathsf{Wrd}$ for all $1 \leq d < \mathit{size}(A)$.*

2. ***Subword access.*** *Suppose $H \models_K m : A$ and $A[c]$ is defined. Then there exists K_c such that $H \models_{K_c} m + c : A[c]$, $K_c \subseteq K$ and $\{m, \ldots, m + c - 1\} \cap K_c = \{\}$.*

3. ***Integer modification.*** *Suppose $H \models_K m : A$ and $A[c] = \mathtt{int}^z$ and $\forall l \in (\mathrm{dom}(H) \setminus (m + c))$, $H(l) = H'(l)$. Then $H' \models_K m : A^{c:=z'}$.*

4. ***Pointer uninitialization.*** *Suppose $H \models_K m : A$ and $A[c] = [B]^1$. Then for some K_b, K_d, $H \models_{K \setminus K_b \setminus K_d} m : A^{c:=0}$, $H \models_{K_b} H(m+c) : B$ and $H \models_{K_d} H(m + c) : \Diamond/\mathit{size}(B)$, with $K_b \cap K_d = \{\}$.*

5. ***Pointer initialization.*** *Suppose $H \models_K m : A$, $A[c] = [B]^0$, and for some K_b, K_d st $K_b \cap K = \emptyset$, $K_d \cap K = \emptyset$, $H \models_{K_b} m_b : B$ and $H \models_{K_d} m_b : \Diamond/\mathit{size}(B)$. Then $H' \models_{K \cup K_b \cup K_d} m : A^{c:=1}$, where $H' = H((m + c) \mapsto m_b)$.*

6. ***Uninitialized types.*** *Suppose* $H(m) \in \mathsf{Wrd}, \ldots, H(m + \mathit{size}(A) - 1) \in \mathsf{Wrd}$ *and* A *is* `code`*-free.*
Then $H \models_{\{m,\ldots,m+\mathit{size}(A)-1\}} m : A^{:=0}$.

7. ***Product types.*** $H \models_K m : A \times B \times C$ *holds iff there exist pairwise disjoint sets* K_a, K_b, K_c *such that* $H \models_{K_a} m : A$, $H \models_{K_b} m + \mathit{size}(A) : B$, $H \models_{K_c} m + \mathit{size}(A) + \mathit{size}(B) : C$, *with* $K = K_a \cup K_b \cup K_c$.

PROOF: Each part is proved by induction or cases on the definitions of $H \models_K m : A$, $A[c]$, or the type concerned. □

LEMMA 2 (Subtyping).

1. *If* $H \models_K m : A$ *and* $A \leq A'$, *then* $H \models_{K'} m : A'$ *for some* $K' \subseteq K$.

2. *If* $\Gamma \leq \Gamma'$ *and* $M \models \Gamma$, *then* $M \models \Gamma'$.

PROOF: Part 2 follows from Part 1. Part 1 is proved by induction on the size of the derivation of $H \models_K m : A$, using the inductive definition of $A \leq A'$ (Definition 1). We show the case for pointers.

Consider proving the statement for $[A]^1 \leq [A']^z$. Suppose we have $H \models_{K \cup K_d \cup \{m\}} m : [A]^1$ derived from the assertions $H \models_K H(m) : A$ and $H \models_{K_d} H(m) : \Diamond/\mathit{size}(A)$, where $m \notin K, K_d$ and $K \cap K_d = \{\}$. For uninitialized pointers $z = 0$, we immediately have $H \models_{\{m\}} m : [A']^0$ and since $\{m\} \subseteq K \cup K_d \cup \{m\}$ we're done. For an initialized pointer $z = 1$, we use the induction hypothesis to derive $H \models_{K'} H(m) : A'$ for some $K' \subseteq K$ and since $\mathit{size}(A) = \mathit{size}(A')$, we still have $H \models_{K_d} H(m) : \Diamond/\mathit{size}(A')$ for the same K_d. Since $m \notin K'$ and $K' \cap K_d = \{\}$, we have $H \models_{K' \cup K_d \cup \{m\}} m : [A']^1$ with $K' \cup K_d \cup \{m\} \subseteq K \cup K_d \cup \{m\}$ as required. The other cases, for $A = A' = \texttt{code}$ or $[A]^0 \leq [A']^0$, are trivial. □

LEMMA 3 (Preservation of unreachable data).
Suppose M *is type-safe at* u *and* $M \leadsto^{\mathsf{Asm}(p_u)} M'$. *Suppose further that* $H \models_K m : A$ *for some heap region* K *which is disjoint from the regions* K_i *corresponding to* r_i *for registers which appear in* Γ_u. *Then* $H' \models_K m : A$ *also.*

PROOF: Similar to the proof of Theorem 1. □

Our first 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 $M \rightsquigarrow^{\mathsf{Asm}(p_u)} M'$.*

PROOF: By cases on the rule used to type-check p_u, using the fact that the context Γ_u for p_u is satisfied in M. The only way we may fail to reach an M' is if we attempt to access some undefined portion of the heap. We rely on a property of heap typing that $H \models_K m : A$ implies that $\{m, \ldots, m + \mathit{size}(A) - 1\} \subseteq \mathrm{dom}(H)$ to guarantee that portions of the heap accessed in reaching M' are always defined. □

The second theorem is the main result: 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 (a final return instruction may leave P, for example).

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*
- *$R'(pc) \notin \mathrm{dom}(\mathsf{PAdr})$ (the machine has left P).*

PROOF: Suppose that $R'(pc) \in \mathrm{dom}(\mathsf{PAdr})$ so we must establish the first case. By case analysis on p_u, we can show that M must have followed one of the typed paths $p_u \rightsquigarrow p_v$, and whichever one it followed, M' is sound at v.

To establish this last thing we must show the three items in Definition 5 are satisfied. The first part, that M' is also assembled for P, is satisfied because the type system prevents the program overwriting any code area: a register can never contain anything of type `code`. The second part, that $R'(pc) = \mathsf{PAdr}(v)$, is satisfied by the definition of the assembly process. The third part, that $M' \models \Gamma_v$, is proved by case analysis on the typing rule used to type p_u, and choice of p_v. This part of the proof relies on the previous lemmas which establish relations between $H \models \Gamma_u$ and $H' \models \Gamma_v$. See Appendix A for more details. □

The safety preservation theorem proves that executing sequences of instructions in the untyped machine which correspond to typed instructions from the HBAL program preserves type safety, most crucially, that the heap can be safely typed according to the typing assumptions from the program. This proves that run-time type errors cannot occur.

4. LFPL and its compilation

In (Hofmann, 2000b), Hofmann defines a linearly typed first-order functional language called LFPL. It 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
	$\mid \mathsf{nil}$	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
	$\mid \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{leaf}(a) \Rightarrow e_2 \mid$	
	$\quad \mathsf{node}(d_1, d_2, a, l, r) \Rightarrow e_3$	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. For example, a list cell is constructed by writing $\mathsf{cons}(d, h, t)$ where h and t are the head and tail as usual, and d is a value of type $\Diamond$ which is used to store the tail.[4] On the other hand, when a list e cell is decomposed with $\mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_n \mid \mathsf{cons}(h, t, d) \Rightarrow e_c$ the space that was used to store the tail is recuperated.

A simple LFPL example is the following program to reverse a list:

```
def list reverse_aux(list l,list acc) =
      match l with
              nil -> acc
```

[4] Here we follow the original presentation and compilation method of LFPL in (Hofmann, 2000b). An alternative scheme is described in (Aspinall and Hofmann, 2002) which uses the diamond d in $\mathsf{cons}(d, h, t)$ to store the cons cell, and uses a nullary pointer for empty lists. This alternative scheme is the more familiar one usually used for compiling lists, but involves a small change to the language.

```
                  | cons(d,h,t) -> reverse_aux(t,cons(d,h,acc))

def list reverse(list l) = reverse_aux(l, nil)
```

The compilation scheme translates this into assembly code which uses in-place update. Linearity in LFPL (and HBAL) ensures that after calling `reverse(l)`, the list `l` cannot be referred to again.

Programs in LFPL are compiled using a signature Σ which declares arities for a number of recursive functions. As an example of the typing rules, we consider those for list constructors and destructor:

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A)}$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)}$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d : \Diamond, h : A, t : \mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B}$$

Notice the way that contexts are handled: when Γ_1, Γ_2 is written it is understood that the domains of the Γ_1 and Γ_2 are disjoint. This prevents sharing of variables in sub-expressions of cons, for example. The rule for match, however, allows variables to be shared between the two branches, since only one branch will be evaluated.

LFPL can express non trivial functions such as insertion sort, quick sort, and breadth first traversal using queues, among many others, yet heap space of LFPL is bounded; in fact, it is fixed for any given program. In (Hofmann, 2000b), Hofmann describes a compilation scheme where programs in LFPL are translated into 'C' programs which do not use `malloc`, but instead perform all computation by in-place update. Here we follow a similar compilation scheme to compile LFPL into HBAL.

4.1. Compiling LFPL to HBAL

We define a type-preserving translation from well-typed LFPL programs into HBAL. Functions written in LFPL are compiled into assembly language procedures which run in constant heap space. To do computation on real data structures, we need some stub code to first define those structures and invoke the HBAL code. A larger assembly language program might make use of HBAL procedures as well-behaved library functions.

LFPL and HBAL are closely related, but have important differences (besides the obvious, that LFPL is a high-level functional language). In LFPL, structured types are treated as units which are copied atomically, but in the assembly language we need to instantiate their components piece-by-piece, and wrap-up the final result into a structured object. In the case of lists, for example, this is obtained by the pseudo-instructions `fold-nil` and `fold-cons` which fold a product into a structured type. LFPL uses linear typing rules and the notion of *heap-free* type to indicate whether memory space can be shared. A heap-free type is one which can reside solely on the stack and involves no pointers into the heap; for example, products of integers are heap free, but lists are not. In HBAL, we use a combination of linear typing and initialization flags to control sharing.

The interpretation $[\![e : A]\!]_\eta$ of a term e in an environment η is some piece of HBAL code which leaves a value of type A on top of the stack, and may use any registers. Allocating data structures on the stack allows the compilation to be described in a simplistic way as a stack-based evaluation. In practice, we would want to use a more sophisticated compilation scheme, which might need more powerful typing rules; see Section 5 for further consideration.

An environment η is a partial mapping from variables to natural numbers. For a variable x, if $\eta(x)$ is defined, it is an offset from sp pointing the location where the variable is stored on the stack. The environment $[x \mapsto c]\eta$ updates η to map x to c. When items are pushed on or off the stack (sp is decremented or incremented), we need to adjust the positions of variables in the corresponding environment η. We define η^{+c} as the environment η' where $\eta'(x) = \eta(x) + c$ for all x in the domain of η. Similarly, η^{-c} subtracts c from the offsets in η.

Interpretation of LFPL types

Types in LFPL map almost directly onto HBAL types:

$$\begin{aligned} [\![int]\!] &= \texttt{int}^1 \\ [\![\Diamond]\!] &= [\Diamond]^1 \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![L(A)]\!] &= L([\![A]\!]) \\ [\![T(A)]\!] &= T([\![A]\!]) \end{aligned}$$

We will often abuse the notation and write A when we mean $[\![A]\!]$.

A copy instruction
The translation uses a `copy` instruction to copy data items. To preserve the single pointer property, pointers in the source must be uninitialized after the copy. But we can't uninitialize the whole type because of *heap-free* types, which are treated non-linearly; variables of heap-free type may be referred to (hence copied) many times. We define an operation $A^{\mathsf{hp}:=0}$ to do the right amount of uninitializion:

$$\begin{array}{lcl}
\mathtt{code}^{\mathsf{hp}:=0} & = & \mathtt{code} \\
(\mathtt{int}^y)^{\mathsf{hp}:=0} & = & \mathtt{int}^y \\
([A]^y)^{\mathsf{hp}:=0} & = & [A]^0 \\
(A \times B)^{\mathsf{hp}:=0} & = & (A^{\mathsf{hp}:=0}) \times (B^{\mathsf{hp}:=0}) \\
L(A)^{\mathsf{hp}:=0} & = & L(A)^{:=0} \\
T(A)^{\mathsf{hp}:=0} & = & T(A)^{:=0} \\
\Diamond^{\mathsf{hp}:=0} & = & \Diamond
\end{array}$$

Here is the typing rule for `copy`:

$$\frac{\begin{array}{l}
\mathit{size}(A_1) = c \qquad \mathit{size}(B_1) = d \\
r_j[c] \text{ and } r_i[d] \text{ do not overlap for } \mathit{size}(A) \text{ words} \\
\{r_j \ : \ [A_1 \times A^{:=0} \times A_2],\ r_i \ : \ [B_1 \times A \times B_2]\} \subseteq \Gamma_1 \\
\{r_j \ : \ [A_1 \times A \times A_2],\ r_i \ : \ [B_1 \times A^{\mathsf{hp}:=0} \times B_2]\} \subseteq \Gamma_2 \\
\Gamma_{1\backslash r_i,r_j} \ \equiv \ \Gamma_{2\backslash r_i,r_j} \qquad \Gamma_2 \ \vdash \ P
\end{array}}{\Gamma_1 \ \vdash \ \mathtt{copy} \quad r_j[c] \ \longleftarrow \ r_i[d],\ A \quad ; \quad P}$$

The rule is complicated by allowing the source and destination registers to be the same (prohibited in the primitive rules), which is needed in the translation for copying up and down the stack. The side condition that there is no overlap between source and destination is statically decidable: if $r_i \neq r_j$ it holds immediately by the linearity of the system; if $r_i = r_j$ then we can compare the offsets c and d to ensure they differ by at least $\mathit{size}(A)$.

The `copy` instruction isn't a primitive in HBAL, but using induction over the type A, we can define it as a macro whose expansion is typable, and which copies from source to destination piecemeal, breaking apart structured types and ignoring uninitialized or unreadable types (see Appendix B for details). This shows that we can safely add `copy` to the system. In an implementation it would be more realistic to treat `copy` as a pseudo-instruction which expands into the obvious loop.

Interpretation of LFPL programs
A *program* in LFPL consists of a set of function definitions of the form

$$f(x_1 : A_1, \ldots, x_n : A_n) : A = e$$

where the typing

$$x_1 : A_1, \ldots, x_n : A_n \vdash_\Sigma e : A$$

holds in LFPL. Let the program

$$\begin{aligned} p = \{ \ & f_1(x_1 : A_1^1, \ldots, x_{n_1} : A_{n_1}^1) : A^1 = e_1, \\ & \ldots, \\ & f_k(x_1 : A_1^k, \ldots, x_{n_k} : A_{n_k}^k) : A^k = e_k \ \} \end{aligned}$$

Then the interpretation of p under environment η is

$$\begin{aligned} [\![p]\!]_\eta = \{ \ & [\![f_1(x_1 : A_1^1, \ldots, x_{n_1} : A_{n_1}^1) : A^1 = e_1]\!]_\eta, \\ & \ldots, \\ & [\![f_k(x_1 : A_1^k, \ldots, x_{n_k} : A_{n_k}^k) : A^k = e_k]\!]_\eta \ \} \end{aligned}$$

using the compilation of function definitions given next.

Compiling function definitions and calls

The stack management for function calls works as follows: the stack contains the actual parameters at the top, followed by return address, followed by space made for the return value.

$$\begin{aligned} & [\![f(x_1 : A_1, \ldots, x_n : A_n) : A = e]\!]_\eta = \\ & \quad l_f \\ & \quad [\![e]\!]_{[x_i \mapsto 1 + \Sigma_{j=1}^{i-1} size(A_j)]\eta} \\ & \quad \texttt{copy} \quad sp[size(A) + (\Sigma_{j=1}^{n} size(A_j)) + 1] \longleftarrow sp[0], \ A \\ & \quad \texttt{sfree} \quad A \\ & \quad \texttt{sfree} \quad A_1 \\ & \quad \ldots \\ & \quad \texttt{sfree} \quad A_n \\ & \quad \texttt{ret} \quad l_f \end{aligned}$$

In words: the code sequence evaluates the body of the function in a context containing the values for the arguments, copies the return value into the caller's frame, pops the result and arguments from the stack and returns.

$[\![f(e_1,\ldots,e_n):A]\!]_\eta =$

$\quad$ `salloc` A
$\quad$ `salloc` $[$`code`$]$
$\quad [\![e_n : A_n]\!]_{\eta + size(A)+1}$
$\quad \vdots$
$\quad [\![e_i : A_i]\!]_{\eta + size(A)+1+\Sigma_{j=i}^{n} size(A_j)}$
$\quad \vdots$
$\quad [\![e_1 : A_1]\!]_{\eta + size(A)+1+\Sigma_{j=1}^{n} size(A_j)}$
$\quad$ `call` l_f
$\quad$ `sfree` $[$`code`$]$

The call first makes space for return value and return address, calculates arguments from last to first, stores the return address onto the stack, calls the function, and finally pops the return address from the stack.

Compiling constants and variables

$[\![c : \texttt{int}]\!]_\eta =$

$\quad$ `salloc` `int`
$\quad$ `addi` $r_1 \longleftarrow r_0 + c$
$\quad$ `store` $sp[0] \longleftarrow r_1$

$[\![x : A]\!]_\eta =$

$\quad$ `salloc` $[\![A]\!]$
$\quad$ `copy` $sp[0] \longleftarrow sp[\eta(x)],\ [\![A]\!]$

Integer constants are straightforward. To compile a variable we allocate space for a copy and copy the value of the variable from higher up in the stack. To prevent aliasing, this may make parts of the original copy unreadable, but the linear typing scheme in the source language ensures that variables which contain pointers are used at most once.

Compiling pairs

$[\![(e_1, e_2) : A_1 \times A_2]\!]_\eta =$
  $[\![e_2 : A_2]\!]_\eta$
  $[\![e_1 : A_1]\!]_{\eta + size(A_2)}$

$[\![\mathsf{match}\ e\ \mathsf{with}\ (x, y) \;\Rightarrow\; e_p : A]\!]_\eta =$
  `salloc` A
  $[\![e : A_1 \times A_2]\!]_\eta$
  $[\![e_p]\!]_{[y \mapsto size(A_1)][x \mapsto 0]\eta + size(A_1 \times A_2)}$
  `copy` $sp[size(A) + size(A_1) + size(A_2)] \longleftarrow sp[0],\ A$
  `sfree` A
  `sfree` $A_1 \times A_2$

To translate a pair, first evaluate the second component, then the first one, which leaves their values on the stack. To match a pair, first calculate the pair, and then compute body of match in the new context.

Compiling lists

$[\![\mathsf{nil} : L(A)]\!]_\eta =$
  `salloc` $\mathtt{int} \times A \times [L(A)]$
  $\mathtt{fold\text{-}nil}_A\ sp[0]$

$[\![\mathsf{cons}(h, t, d) : L(A)]\!]_\eta =$
  `salloc` $[L(A)]$
  $[\![h : A]\!]_\eta$
  `salloc int`
  $[\![t : L(A)]\!]_\eta$
  $[\![d : \Diamond]\!]_\eta$
  `load` $r_2 \longleftarrow sp[0]$
  `use` $r_2\ \ L(A)$
  `copy` $r_2[0] \longleftarrow sp[1],\ L(A)$
  `sfree` $[\Diamond]^0$
  `sfree` $L(A)$
  `store` $sp[size(A) + 1] \longleftarrow r_2$
  $\mathtt{fold\text{-}cons}_A\ sp[0]$

$[\![\mathsf{match}\ e\ \mathsf{with\ nil}\ \Rightarrow\ e_n \mid \mathsf{cons}(h,t,d)\ \Rightarrow\ e_c : A']\!]_\eta =$

$\mathtt{salloc}\quad A'$
$[\![e : L(A)]\!]_{\eta^{+size(A')}}$
$\mathtt{caselist}_A\quad sp?[0]\quad mcons$
$\mathtt{sfree}\quad \mathtt{int} \times A \times [L(A)]$
$\mathtt{sfree}\quad A'$
$[\![e_n : A']\!]_\eta$
$\mathtt{jmp}\quad mdone$
$mcons$
$\mathtt{load}\quad r_2 \longleftarrow sp[1 + size(A)]$
$\mathtt{salloc}\quad L(A)$
$\mathtt{copy}\quad sp[0] \longleftarrow r_2[0],\ L(A)$
$\mathtt{discard}\quad r_2$
$\mathtt{salloc}\quad [\Diamond]$
$\mathtt{store}\quad sp[0] \longleftarrow r_2$
$[\![e_c]\!]_\delta$
$\mathtt{copy}\quad sp[size(A') + 1 + 2 * size(L(A))] \longleftarrow sp[0],\ A'$
$\mathtt{sfree}\quad A'$
$\mathtt{sfree}\quad [\Diamond]$
$\mathtt{sfree}\quad L(A)$
$\mathtt{sfree}\quad L(A)$
$mdone$

where
$\delta = [d \mapsto 0][t \mapsto 1][h \mapsto 1 + size(L(A)) + 1]\eta^{+x}$ and $x = size([\Diamond] \times L(A) \times L(A) \times A')$.

Compiling a nil cell first allocates space on the stack for a list and creating a nil cell with `fold-nil`, which initializes the tag to 0.

Compiling a cons cell consists of allocating space on the stack for a list, computing the head and copying it in the stack dedicated space. Followed by computing the tail, finding the space to store the tail ($[\![d : \Diamond]\!]_\eta$), copying the tail to the heap, and storing the pointer to the tail. Once all the components are in place, the compilation ends by folding the prefix of the stack onto a list, which consists of setting the tag to 1 and folding the corresponding part of the type of *sp* into a list.

Notice that because the compilation scheme leaves a list *cell* at the top of the stack, as opposed to a pointer, the diamond argument to the cons is used to store the tail of the list t, not the whole cell; this follows the scheme in (Hofmann, 2000b) (see the footnote on page 25).

Compiling a match starts by computing the list, and branching according to the `caselist` instruction, the nil case follows `caselist` the cons case follows at label *mcons*.

The compilation of trees is similar to the compilation of lists.

4.2. Type correctness of compilation

DEFINITION 7.
Given $\Gamma = x_1 : A_1, ..., x_n : A_n$

1. $[\![\Gamma]\!] = [\![A_1]\!] \times ... \times [\![A_n]\!]$
2. $\eta \models (\Gamma, \Delta)$ *if* $x : A$ *in* Γ *implies* $\Delta[\eta(x)] = [\![A]\!]$*, where* $\Delta[n]$ *is the* n^{th} *projection of the product* Δ*.*
3. *The initial and final contexts of a program follows* $\{\Gamma\}P\{\Gamma'\}$ *are defined by:*
   a) $\{\Gamma\}\langle\rangle\{\Gamma\}$
   b) $\{\Gamma\}(i;P)\{\Gamma'\}$ *if* $\Gamma \vdash i;P$*,* $\Gamma'' \vdash P$*, and* $\{\Gamma''\}(i;P)\{\Gamma'\}$*.*

We consider $r_0 : \texttt{int}$ as an implicit assumption for HBAL contexts, since r_0 always contains 0. We also assume an implicit Σ that contains typing information for labels and subroutines.

THEOREM 3 (Type correctness).
If $\Gamma \vdash_\Sigma e : A$ *in LFPL then*
for all η *and for all* Δ *such that* $\eta \models (\Gamma, \Delta)$ *and* $\Delta = B \times [\![\Gamma]\!] \times C$ *(where* B, C *can be empty), it follows that*
$\{sp : [\Delta]\}[\![e : A]\!]_\eta\{sp : [[\![A]\!] \times \Delta]\}$

The theorem says that after executing the translation of an expression, at the top of the stack there is an element of the corresponding type, and whatever was on the stack before remains untouched.

The proof is by induction on the derivation of $\Gamma \vdash_\Sigma e : A$ in LFPL. The general Δ as opposed to $[\![\Gamma]\!]$ is necessary in the case of function applications to be able to apply the induction hypothesis on the arguments.

A corollary of the previous theorem is that the translation of a typable LFPL program(set of function definitions) is typable in HBAL.

Although here we prove only type preservation of the compilation scheme, it would be straightforward to prove *correctness* of the scheme, using a variation of Definition 3 which also relates a semantic value in a denotational interpretation of LFPL. In fact, we consider the possibility of proving full correctness as an advantage of our approach: we maintain the abstractions of the high-level language all the way to the low-level. Of course, establishing correctness would be harder in the case of a more sophisticated compilation scheme or optimization stages.

5. Variants of HBAL

We have deliberately kept the presentation of HBAL simple, mainly for the sake of exposition. We provided a type structure and inference rules which suffice for a naive type-safe translation from the high-level language LFPL, and which illustrate our method of pushing the high-level type abstractions firmly into the low-level code. For other applications of HBAL, or for more sophisticated compilation schemes involving optimizations or representation changes, we would need a more flexible language and type system. In this section we sketch some desirable improvements to HBAL, many of which are straightforward, or which could be adopted from other typed assembly languages. Some others require further research.

5.1. More expressive types and subtyping

As we mentioned earlier, a real system would include a general scheme for defining high-level types apart from just lists and trees, perhaps using a `datatype` mechanism mirroring facilities in the high-level languages. Furthermore, we could use additional constructors to allow more flexible layout in memory, including `null` pointers in representation with an extra `caseptr` descriminator.

The subtyping relation could also be made richer. For example: at present, we are strict about following the scheme to dedicate memory to a particular type using pseudo-instructions, before storing data with the correct type. This could be relaxed by adding subtyping assertions such as $\Diamond \leq A^{:=0}$ and $[A] \leq [\Diamond]$ for structured types A, which would allow us to omit the `use` and `discard` instructions. Adding subtypings $w^1 \leq w'^0$, would allow us to freely change word types in a safe way. We could also include product-length subtyping $A \times B \leq A$. These changes are not technically difficult, but they make the connection between the high-level types and low-level ones in the assembly language harder to track; this is a motivation for retaining a simpler, more verbose system.

As more substantial extensions of the type language, one could consider polymorphic and higher-order types as in TAL (Morrisett et al., 1999), dependent types as in DTAL (Xi and Harper, 1999), and object types. It is a matter for further research to integrate these notions into

HBAL, considering the resource usage implications and connection with high-level languages. A recent result in (Hofmann, 2002) shows that a large class of functions on lists definable in a system with higher-order functions can be computed in bounded space.

5.2. Dynamic memory management

It isn't in keeping with our approach, but we mention that it is straightforward to add `malloc` and `free` instructions to LFPL which interface with an external memory manager. The `malloc` instruction fetches a diamond from the memory manager, and a `free` returns a diamond.

$$\frac{\Gamma, r_j \;:\; [\Diamond] \;\vdash\; P}{\Gamma \;\vdash\; \texttt{malloc} \quad r_j \quad ; \quad P} \qquad \frac{\Gamma \;\vdash\; P}{\Gamma, r_j \;:\; [\Diamond] \;\vdash\; \texttt{free} \quad r_j \quad ; \quad P}$$

The single-pointer property guarantees the safety of `free`. Using `malloc` allows for a range of possibilities from schemes which rely wholly on garbage collection (without `free`) to schemes which rely wholly on programmer-level memory management (using `free`). In between, schemes for compile-time garbage collection would insert `free` at automatically determined places.

5.3. Different sized diamonds

In the high-level language LFPL, diamond types are an abstract way of dealing with heap space: one which is consistent with functional semantics, yet allows efficient in-place update. The simplification to a "one size fits all" diamond means that we can easily write functions which map between different kinds of datatype. The disadvantage is a potentially large waste of space, in case the program contains any large datatypes; the needed space would be multiplied by some constant factor. For example, if we mostly computed with integer lists but once used lists of 10-tuples of integer lists, then (following the translation of types used in Section 4), $size(\Diamond) = 32$, so most cells would contain wasted space. We view this as a trade-off which one might accept in return for an absolute guarantee of heap-boundedness which is provided by the system.

To improve the wastage we could introduce diamonds in a range of sizes: $\Diamond_k$ occupying k words, in some way mimicking the schemes of traditional memory allocators. A k-sized diamond would be large enough to store any type A such that $size(A) \leq k$. To keep track of the size of diamond used for a structured type, we would also need to annotate the type with the size of the diamond used, writing $L(A)^{\Diamond_k}$.

The rules for `use` and `discard` would become:

$$\frac{k' \leq k \quad A = L(-)^{\Diamond_{k'}}, T(-)^{\Diamond_{k'}} \quad \Gamma, r_j \ : \ [A^{:=0}] \ \vdash \ P}{\Gamma, r_j \ : \ [\Diamond_k] \ \vdash \ \mathtt{use} \quad r_j \quad A \quad ; \quad P}$$

$$\frac{A = L(-)^{\Diamond_k}, T(-)^{\Diamond_k} \quad \Gamma, r_j \ : \ [\Diamond_k] \ \vdash \ P}{\Gamma, r_j \ : \ [A] \ \vdash \ \mathtt{discard} \quad r_j \quad ; \quad P}$$

Types for subroutines will also need need extra constraints, for example a function which maps lists to trees using in place update may not have enough space to work unless the list cells are big enough to store the result tree cells. Some static inference could be performed on the source program to determine the distribution of diamonds needed to make sure functions have enough space to compute their result; we suspect that most programs could be compiled to use a small range of diamond sizes according to the flow of data between functions.

5.4. Allowing limited aliasing

Linear type systems are restrictive in practice. Some high-level languages mix both worlds, distinguishing linear and non-linear variables by static inference or by programmer control. Our methodology is to provide *static guarantees* as part of the language; instead of relaxing linearity completely we want to restrict to programs which can be implemented safely (i.e., consistently with the functional interepretation) using in-place update.

HBAL already allows unrestricted copying of non-pointer values, but we would like to allow limited aliasing of types containing pointers too. A promising approach is to use a relaxed linear system which allows sharing of data when it is used in a non-destructive context. In (Aspinall and Hofmann, 2002) a mechanism is given for LFPL, based on the idea of *usage aspect*. This allows a variable to be used several times in a read-only fashion, before being used in a destructive way. Like linear typing, we assume a single-threaded execution, but we take into account a particular order of evaluation as well.

Other related work includes as (Smith et al., 2000; Kobayashi, 1999; Wilhelm et al., 2000).

5.5. Stack bounds and iteration

HBAL provides a static guarantee about heap usage, but no bound on the size of the stack. In virtual machines and runtime systems stacks are often allocated on the heap, and in real machines, stack space is hardly unbounded. To be serious about resource bounded programming we need to address this.

It might be imagined that the `salloc` could be treated just like a `malloc` during compilation; however this is forbidden if we insist on a compilation strategy which produces a program consisting of a collection of properly terminated subroutines, since the typing rules ensure that the stack contains the same number of items at the end of the subroutine as it did at the start (just like the constraints imposed in the Java Virtual Machine (Stata and Abadi, 1999)).

However, stack space is still not bounded since arbitrary recursion is allowed. A idea then would be to restrict to the *tail recursive* fragment in which general recursion is prohibited, at the expense of restricting the computational power of the system. This might involve either optimization after HBAL typing, or better, improvement of the HBAL typing rules to allow typing of iterative algorithms.

An alternative strategy is to consider inferring *concrete bounds*, where the number of recursive invocations (and space usage) of a function can be described as a function of its input, along the lines described by Crary and Wierich (Crary and Weirich, 2000).

6. Conclusions, Related Work and Future Research

HBAL allows for safe reuse of memory without relying on a garbage collector or a block structure region policy. A type system with linearity constraints ensures type safety, by preventing aliasing of heap locations. Special pseudo-instructions allow types to be altered at isolated points in the code. HBAL's type system prevents self-modifying programs using the special type `code`. To prove type soundness, we defined a machine model and proved that execution of a program typable in HBAL preserves the type safety of machine configurations; this is our main result (Theorem 2).

We showed how to translate Hofmann's LFPL into HBAL.

The mechanism for memory reuse was motivated by the functional programming language LFPL defined by Hofmann. In HBAL, the control and re-typing of memory is made explicit in a simple protocol. In HBAL extended with `malloc`,`free`, a memory location is first allocated and assigned type diamond (`malloc`); it is then assigned a type and declared uninitialized (`use`); then it is initialized (`store`,`fold`) and uninitialized (`case`,`load`) repeatedly during execution; finally it may be discarded (`discard`), which causes it to lose its type to become a diamond. Once it is a diamond, it can be reused at a different type, or freed (`free`). This explicit tracking mechanism is a novel aspect of

HBAL. The linearity constraints of the type system make it safe to discard memory space, because any location is accessible through at most one register.

Another novel aspect of HBAL is its treatment of structured types. Recursive types such as list and tree, are considered different from their unfoldings; other accounts of structured types explain how such types are laid out in memory, but do not consider these types as abstract data types. Lists in HBAL can only be constructed with `fold-cons` and `fold-nil`, and destructed with `caselist`.

In Section 5 we discussed possible variants of HBAL with more expressive typing and subtyping relations, different size diamonds, dynamic memory management, controlled aliasing, stack bounds and iteration.

6.1. Related work

Typed assembly languages have been an active subject of study for several years now. Contributions already mentioned include TAL (Morrisett et al., 1999; Crary et al., 1999), DTAL (Xi and Harper, 1999), STAL (Morrisett et al., 1998), and Alias Types (Smith et al., 2000; Walker and Morrisett, 2000).

Both HBAL and Alias Types use substructural typing to control aliasing, but the systems differ substantially. For example, HBAL has traditional assembly language control flow intructions such as conditional branches and jumps, and allows subroutines in a similar way to STAL; Alias Types, like TAL, relies on a transformation into continuation passing style. More importantly, HBAL preserves the single pointer property well-known from abstract machines for linear lambda calculi: every location can be reached from at most one live pointer in a register or on the heap. Alias Types allows pointer aliasing, using a type system close to O'Hearn and Pym's logic of bunched implications (O'Hearn and Pym, 1999). Alias Types therefore allows more efficient data representations using sharing, but does not come with a guarantee of bounded space usage. (In future we plan to extend HBAL to allow limited sharing, as mentioned in Section 5.4.) HBAL's restriction to first-order types is also important for guaranteed space bounds, since (unless we assume continuation-passing style) dealing with higher-order types may require storing closures on the heap.

Wider afield, a central idea among related work is to use typing information at different stages of compilation such as in the study of type systems for the Java Virtual Machine (Stata and Abadi, 1999). The work done in the Fox project on typed intermediate languages (Harper and Morrisett, 1995; Morrisett, 1995; Tarditi et al., 1996) was a major contribution, which inspired the FLINT (Shao, 1997) typed intermediate representation. FLINT was designed to make the intermediate representation sufficiently general to support not only ML but a wide variety of other programming languages, such as Java and C, and with attention to engineering issues (Shao et al., 1998). By contrast, our approach is to design a deliberately restricted assembly language, which is customized for LFPL in a similar way to the way that JVML is customized for Java.

The semantic model of types presented in (Appel and Felty, 2000) describes high-level types like those in HBAL, allowing the traversal, allocation, and initialization of values. Again heap space is never updated, assuming the presence of a garbage collector.

Tofte-Talpin's region calculus (Tofte and Talpin, 1997; Tofte and Birkedal, 1998) proposes an alternative to traditional garbage collection by dividing the heap into a list of regions which are allocated and deallocated according to a stack discipline derived from the block structure of the program. A type system ensures that the deallocation of a region does not destroy accessible data. (Banerjee et al., 1999) shows a translation of a version of the region calculus into a variant of the polymorphic lambda calculus, justifying region re-use. The diamond type used in HBAL is similar in spirit to regions, but does not involve a stack discipline. Diamonds are a first-class type, and can be stored and retrieved from data structures.

6.2. Typed-assembly languages for proof-carrying code

Another difference between HBAL and the TAL family of languages is in the handling of the store. In TAL, the contents of the store is formalized as part of the static semantics, and the type safety of a program is only tested once the contents of memory are known.

By contrast, HBAL has a clear distinction between its syntax and its operational semantics: the store is not mentioned in our syntax or static semantics and the well typedness of a HBAL program is independent of the memory contents. The store is introduced later, in an untyped machine model (Section 3.2), and we prove that any well typed HBAL program preserves memory safety in the machine model (Section 3.4).

Typed assembly languages have been advocated for proof-carrying code, but 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. Our approach splits the concerns as one might expect: the code producer can build the safety proof for execution on *any* safe memory, and the consumer only needs to check that the initial memory is safe.

To be fair to TAL, the appearance of types in the operational semantics forms part of the proof technique used to prove the type preservation property. (Indeed, we also considered presenting a typed-operational semantics for our model and might need to do that if we added polymorphism.) It seems straightforward to erase types from the TAL operational semantics, and also to quantify a typing proof over all heaps which satisfy some initial typing property, instead of including a heap in the typing judgement. Nevertheless, we believe it is useful to draw attention to this point, as well as to introduce a slightly different formulation of the typed assembly paradigm.

Acknowledgements We thank Martin Hofmann, Healfdene Goguen, Andrew Appel, Amal Ahmed, and Peter O'Hearn for help, comments, and suggestions. Members of the New Jersey Programming Languages and Systems Seminar made helpful remarks, including Kathleen Fisher, Dave MacQueen, Michael Hicks, and Dan Wang. Matthieu Lucotte provided helpful feedback and implemented a HBAL typechecker. DA was partially supported by UK EPSRC grant no. GR/N28436. AC was partially supported by the New Jersey Commission on Science and Technology and by the NSF project *CAREER:A formally verified environment for the production of secure software.*

References

Appel, A. W. and A. P. Felty: 2000, 'A Semantic Model of Types and Machine Instructions for Proof-Carrying Code'. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. pp. 243–253.

Aspinall, D. and M. Hofmann: 2002, 'Another Type System for In-place Update'. In: *Proceedings ESOP 2002 - European Symposium on Programming*. To appear.

Banerjee, A., N. Heintze, and J. G. Riecke: 1999, 'Region analysis and the polymorphic lambda calculus'. In: *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*. pp. 88–97.

Crary, K., N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic.: 1999, 'TALx86: A Realistic Typed Assembly Language.'. In: *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. pp. 25–35.

Crary, K. and G. Morrisett: 1999, 'Type Structure for Low-Level Programming Langauges'. In: *Proceedings of the International Colloquium on Automata, Languages, and Programming, Prague, Czech Republic*. pp. 40–54.

Crary, K. and S. Weirich: 2000, 'Resource bound certification'. In: *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*. pp. 184–198.

Harper, R. and G. Morrisett: 1995, 'Compiling Polymorphism Using Intensional Type Analysis'. In: *Proceedings of the ACM Symposium on Principles of Programming Languages, San Francisco, pages 130-141.*

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

Hofmann, M.: 1999b, 'Typed lambda calculi for polynomial-time computation'. Habilitation thesis, TU Darmstadt, Germany. Edinburgh University LFCS Technical Report, ECS-LFCS-99-406.

Hofmann, M.: 2000a, 'Programming languages capturing complexity classes'. *SIGACT News Logic Column* **9**. 12 pp.

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

Hofmann, M.: 2002, 'The strength of non size-increasing computation'. In: *Proceedings ACM Principles of Programming Languages*.

Jim, T., G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang: 2002, 'Cyclone: A Safe Dialect of C'. In: *USENIX Annual Technical Conference, Monterey CA*.

Kobayashi, N.: 1999, 'Quasi-Linear Types'. In: *Proceedings ACM Principles of Programming Languages*. pp. 29–42.

Morrisett, G.: 1995, 'Compiling with Types'. Ph.D. thesis, Carnegie Mellon University, Pittsburgh. Tech Report CMU-CS-95-226.

Morrisett, G., K. Crary, N. Glew, and D. Walker: 1998, 'Stack-Based Typed Assembly Language'. In: *Second International Workshop on Types in Compilation*. Kyoto, pp. 95–117. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

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

Necula, G. C., S. McPeak, and W. Weimer: 2002, 'CCured: Type-safe retrofitting of legacy code'. In: *Proceedings ACM Principles of Programming Languages*.

O'Hearn, P. W. and D. J. Pym: 1999, 'The Logic of Bunched Implications'. *Bulletin of Symbolic Logic* **5**(2), 215–243.

Shao, Z.: 1997, 'Typed Common Intermediate Format'. In: *1997 USENIX Conference on Domain-Specific Languages*. Santa Barbara, CA.

Shao, Z., C. League, and S. Monnier: 1998, 'Implementing Typed Intermediate Languages'. In: *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. Baltimore, Maryland, pp. 313–323.

Smith, F., D. Walker, and G. Morrisett: 2000, 'Alias Types'. In: G. Smolka (ed.): *Ninth European Symposium on Programming*, Vol. 1782 of *lncs*. pp. 366–381, Springer-Verlag.

Stata, R. and M. Abadi: 1999, 'A Type System for Java Bytecode Subroutines'. In: *ACM Transactions on Programming Languages and Systems 21*, Vol. 21(1).

Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, , and P. Lee.: 1996, 'TIL: A Type-Directed Optimizing Compiler for ML'. In: *Proceedings of 1996 SIGPLAN Conference on Programming Language Design and Implementation*. pp. 181–192.

Tofte, M. and L. Birkedal: 1998, 'Region Inference Algorithm'. *ACM Transactions on Programming Languages and Systems* **20**(5), 724–767.

Tofte, M. and J.-P. Talpin: 1997, 'Region-based memory management'. *Information and Computation* **132**(2), 109–176.

Walker, D. and G. Morrisett: 2000, 'Alias Types for Recursive Data Structures'. In: *Third International Workshop on Types in Compilation*. Montreal, Canada.

Wilhelm, R., M. Sagiv, and T. Reps: 2000, 'Shape Analysis'. In: *Proceedings Compiler Construction, CC 2000*.

Xi, H. and R. Harper: 1999, 'A Dependently Typed Assembly Language'. Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology.

Appendix

A. Proof of Theorem 2

The theorem states: supposing 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
- $R'(pc) \notin \mathrm{dom}(\mathsf{PAdr})$ (the machine has left P).

Here we prove that in the case $p_u \rightsquigarrow p_v$, $M' \models \Gamma_v$, where Γ_v is the context for the next instruction p_v. The proof is by case analysis on the typing rule used to type p_u, and the possible next instruction p_v.
Let $M = (H, R)$ and $M' = (H', R')$.
Recall M is type safe at u implies that $M \models \Gamma_u$, where Γ_u is the context for p_u. This means that there are pairwise disjoint sets K_i such that $M \models_{K_i} r_i : A_i$ for each $r_i : A_i$ in Γ_u. We must show that there are pairwise-disjoint sets K'_i such that $M' \models_{K'_i} r_i : [A'_i]$ for each $r_i : [A'_i]$ in Γ_v. Notice that $M' \models_{\emptyset} r_i : \mathtt{int}$.

We consider each of the cases below, in varying detail.

LOAD NON-LINEAR

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

In this case, $H' = H$ and $R' = R[k \mapsto H(R(j) + c)]$. The next instruction is the next instruction in P'.

We take $K'_i = K_i$ for all $i \neq k$.

LOAD LINEAR

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

Again, $H' = H$ and $R' = R[k \mapsto H(R(j)+c)]$. The next instruction is the next instruction in P'.

To prove $M' \models_{K'_i} r_i : [C]$, we have to consider the following cases.

[$i \notin j, k$] By the definition of satisfiability we have to distinguish between $r_i = sp$ and $r_i \neq sp$

$r_i \neq sp$ We have to prove:

1. $H' \models_{K'} R'(i) : C$

2. $H' \models_{K'_d} R'(i) : \Diamond/\mathit{size}(C)$
3. $R'(i) > 0$
4. $K' \cap K'_d = \{\}$

and take $K'_i = K' \cup K'_d$
Since $M \models_{K_i} r_i : [C]$ we have that $H \models_K R(i) : C$ implies $H' \models_K R'(i) : C$ because $H = H'$ and $R(i) = R'(i)$. Also, $H \models_{K_d} R(i) : \Diamond/\mathit{size}(C)$ implies $H' \models_{K_d} R'(i) : \Diamond/\mathit{size}(C)$. Furthermore, $r_i \neq \mathit{sp}$ and $K_d \cap K = \emptyset$ by assumption. Finally, take $K'_i = K_d \cup K = K_i$, which were already pairwise disjoint.

[$r_i = \mathit{sp}$] Similar to the previous case, take $K'_i = K_i$, given that $M \models_{K_{sp}} \mathit{sp} : [C]$.

[$i = j$]

[$r_j \neq \mathit{sp}$] We know from $M \models_{K_j} r_j : [A]$ by satisfiability, that $K_j = K^A \cup K^A_d$, where $H \models_{K^A} R(j) : A$, $H \models_{K^A_d} R(j) : \Diamond/\mathit{size}(A)$, $K^A \cap K^A_d = \emptyset$ and $R(j) > 0$.
By Lemma 1(4), $H \models_{K^A} \setminus K_b \setminus K^B_d R(j) : A^{c:=0}$, $H \models_{K^B_d} H(R(j) + c) : \Diamond/\mathit{size}(B)$, $H \models_{K_b} H(R(j) + c) : B$, $H(R(j) + c) > 0$, and $K_b \cap K^B_d = \emptyset$.
We have to prove $M' \models_{K'_j} r_j : [A^{c:=0}]$ for some K'_j. Take $K'_j = K' \cup K'_d$. By definition of satisfiability, we have to prove:

$H' \models_{K'} R'(j) : A^{c:=0}$ Since $H = H'$ and $R'(j) = R(j)$, take $K' = K^A \setminus K_b \setminus K^B_d$.

$R'(j) > 0$ Because $R'(j) = R(j) > 0$.

$H' \models_{K'_d} R'(j) : \Diamond/\boldsymbol{\mathit{size}}(A)$ Since $H = H'$, $R'(j) = R(j)$ and $\mathit{size}(A) = \mathit{size}(A^{c:=0})$, take $K'_d = K^A_d$.

$K' \cap K'_d = \emptyset$ $K_A \setminus K_b \setminus K^B_d \subseteq K^A$ implies $K' \cap K'_d \subseteq K^A \cap K^A_d = \emptyset$. Then $K'_j = (K^A \setminus K_b \setminus K^B_d) \cup K^A_d$.

It only remains to prove that $K'_j \cap K'_l = \emptyset$ for all $l \notin \{j, k\}$. Since $K_j \cap K_l = \emptyset$ for all such l, then $K'_j \cap K_l = \emptyset$.

[$r_j = \mathit{sp}$] We have to prove $M' \models_{K'_{sp}} \mathit{sp} : [A]$ for some K'_{sp}. This follows by definition of satisfiability, since $H = H'$ and $R(\mathit{sp}) = R'(\mathit{sp})$.

[$i = k$] We know $r_k \neq \mathit{sp}$. We have to prove that $M' \models_{K'_k} r_k : [B]$. By definition of satisfiability we have to prove:

1. $H' \models_{K_b} R'(k) : B$.
2. $R'(k) > 0$.
3. $H' \models_{K^B_d} R'(k) : \Diamond/\mathit{size}(B)$.

4. $K_b \cap K_d^B = \emptyset$.

Since $R'(k) = H(R(j)+c)$ and $H = H'$ then, by Lemma 1(4), all four conditions are satisfied.

Take $K'_k = K_b \cup K_d^B$. Disjointness of K'_j and K'_k is immediate from the definition of K'_j and K'_k. Furthermore, $K'_k \cap K'_l = \emptyset$ for all $l \notin \{j,k\}$, follows from $K'_l = K_l$,$K'_k = K_k$, and $K_k \cap K_l = \{K_b \cup K_d^B\} \cap K_l = \emptyset$ for all $l \notin \{j,k\}$.

STORE NON-LINEAR

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

In this case, $H' = H[R(j) + c \mapsto R(k)]$ and $R' = R$.

We take $K'_i = K_i$ for all i. By Lemma 1(3), $H' \models_{K_j} R(j) : A^{c:=1}$.

STORE LINEAR

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

The untyped semantics is the same as the previous case, so $H' = H[R(j) + c \mapsto R(k)]$ and $R' = R$.

We take $K'_i = K_i$ for $i \neq k, j$. By assumption, we have $M \models_{K_k} r_k : B$, and $K_j \cap K_k = \emptyset$. Since $R(j) + c \in K_j$, we have $H' \models_{K_k} R'(k) : B$ too, where $K_k = K \cup K_d$ with $H' \models_{K_d} R'(k) : \Diamond/size(B)$ and $K_k \cap K_d = \{\}$.

By Lemma 1(5), we can take $K'_j = K_j \cup K_k$ to get $H' \models_{K'_j} R'(j) : A^{c:=1}$.

ARITH C,ARITH Easy, since satisfaction for registers with type `int` is trivial.

SALLOC By the definition of the pseudo-instruction, we have $R' = R[sp \mapsto (R(sp) - size(A))]$.

By the unbounded stack assumption, we have $H(R'(sp)) \in \mathsf{Wrd}, \ldots, H(R'(sp) + size(A) - 1) \in \mathsf{Wrd}$. By Lemma 1(6), $H' \models_K R'(sp) : A^{:=0}$ for K st $K \cap K_{sp} = \{\}$. Then we have $H' \models_{K'_{sp}} R'(sp) : A^{:=0} \times A_{sp}$ for $K'_{sp} = K_{sp} \cup K$.

SFREE Straightforward.

BRANCH LABEL The underlying machine does not change when a label is “executed”. The soundness of the context change follows from Lemma 2 and definition of satisfaction.

JMP In this case we must consider a different flow of control. The next instruction p_v is the instruction with context $\Sigma(l)$. By Lemma 2 again.

BNZ There are two possible next instructions, depending on $R(i)$. The typing rule guarantees that the context for either one is sound, by Lemma 2 or trivially.

CALL The next instruction executed will be the instruction after the subroutine label l. We must prove that the context

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

is satisfied. This is straighforward by the assumed type for sp, and the effect of the pseudo-instruction which sets the return pointer.

RET The next instruction executed may be any instruction which follows a call to this subroutine. We must prove that the context for typing any instruction following each use of `call` l is satisfied. To do this we need some additional reasoning about the preservation of the stack during execution of the subroutine, proved with the help of Lemma 3.

USE This instruction does not change the machine. Soundness is straightforward using Lemma 1(6).

DISCARD This instruction does not change the machine. Its soundness follows from Lemma 1(1).

FOLD-NIL,FOLD-CONS,FOLD-LEAF,FOLD-NODE Each of the fold instructions is straightforward to prove sound, using the assumption about r_i and the effect of the pseudo-instruction, to use the corresponding heap typing rule, with the help of Lemma 1(7).

CASELIST, CASETREE Each case instruction has two possible next instructions. By the assumption, we know that the tag $H(m) = 0$ or $H(m) = 1$ and the assumption used to prove the heap typing justifies the unfolded type used in either branch, with the help of Lemma 1(7). □

B. Copy as a macro

To define the compilation of LFPL in Section 4, we used a `copy` instruction for copying elements of an arbitrary type. Here we show that this instruction can be defined, somewhat tediously, as a macro. The point of demonstrating this is to show that the additional pseudo-instruction can be considered as a derived instruction.

The `copy` has this typing rule:

$$\frac{\begin{array}{l} \mathit{size}(A_1) = c \qquad \mathit{size}(B_1) = d \\ r_j[c] \text{ and } r_i[d] \text{ do not overlap for } \mathit{size}(A) \text{ words} \\ \{r_j \ : \ [A_1 \times A^{:=0} \times A_2],\ r_i \ : \ [B_1 \times A \times B_2]\} \subseteq \Gamma_1 \\ \{r_j \ : \ [A_1 \times A \times A_2],\ r_i \ : \ [B_1 \times A^{\mathsf{hp}:=0} \times B_2]\} \subseteq \Gamma_2 \\ \Gamma_{1 \backslash r_i, r_j} \ \equiv \ \Gamma_{2 \backslash r_i, r_j} \qquad \Gamma_2 \ \vdash \ P \end{array}}{\Gamma_1 \ \vdash \ \texttt{copy} \quad r_j[c] \ \longleftarrow \ r_i[d],\ A \quad ; \quad P}$$

Here is the definition of `copy` as a macro, given inductively on the structure of the type it copies. The macro does not copy uninitialized or unreadable fragments of data of a given type. So for the types `code`, $\Diamond$, and w^0, copy has no effect (the macro expands to an empty intstruction sequence). The expansion of `copy` for other types is shown below:

$\texttt{copy} \quad r_j[c] \ \longleftarrow \ r_i[d],\ w^1 =$
$\quad \texttt{load} \quad r_1 \ \longleftarrow \ r_i[d]$
$\quad \texttt{store} \quad r_j[c] \ \longleftarrow \ r_1$

$\texttt{copy} \quad r_j[c] \ \longleftarrow \ r_i[d],\ A_1 \times A_2 =$
$\quad \texttt{copy} \quad r_j[c] \ \longleftarrow \ r_i[d],\ A_1$
$\quad \texttt{copy} \quad r_j[c + \mathit{size}(A_1)] \ \longleftarrow \ r_i[d + \mathit{size}(A_1)],\ A_2$

$\texttt{copy} \quad r_j[c] \ \longleftarrow \ r_i[d],\ L(A) =$
$\quad \texttt{caselist}_A \ \ r_i[d] \ \ \mathit{cpcons}_d$
$\quad \texttt{fold-nil}_A \ \ r_j[c]$
$\quad \texttt{jmp} \quad \mathit{cpdone}_d$
$\quad \mathit{cpcons}_d$
$\quad \texttt{copy} \quad r_j[c+1] \ \longleftarrow \ r_i[d+1],\ A \times [L(A)]^1$
$\quad \texttt{fold-cons}_A \ \ r_j[c]$
$\quad \mathit{cpdone}_d$

$\texttt{copy}\quad r_j[c] \longleftarrow r_i[d],\ T(A) =$
$\texttt{casetree}_A\ \ r_i[d]\ \ \mathit{cpnode}_d$
$\texttt{copy}\quad r_j[c+1] \longleftarrow r_i[d+1],\ A$
$\texttt{fold-leaf}_A\ \ r_j[c]$
$\texttt{jmp}\quad \mathit{cpdone}_d$
cpnode_d
$\texttt{copy}\quad r_j[c+1] \longleftarrow r_i[d+1],\ A \times [T(A)]^1 \times [T(A)]^1$
$\texttt{fold-node}_A\ \ r_j[c]$
cpdone_d

Where $\Sigma(\mathit{cpdone}_d)$ assigns r_i a type of the form $A_1 \times L(A)^{:=0} \times A_2$ or similarly for trees. Subtyping is needed to type-check jmp cpdone_d when copying a list, to unify the types of the head element (which is uninitialized as $A^{:=0}$ in the nil case but only $A^{\mathsf{hp}:=0}$ after copying, in the cons case).