Type Structure for Low-Level Programming Languages

Karl Crary[1] and Greg Morrisett[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15213
[2] Cornell University, Ithaca, NY 14850

Abstract. Providing type structure for extremely low-level programming languages, such as assembly language, yields a number of important benefits for compiler and systems projects, where reliability and security are crucial concerns. We discuss some of the issues involved in designing type systems for low-level languages, and identify some general principles that simplify constructing a proof of soundness.

1 Introduction

Over the past twenty years, there has been tremendous progress in the design and foundations of type systems for high-level programming languages culminating in the design of such languages as Modula-3, Standard ML, Haskell, and Java. The goal of much of the research was to strengthen the logic of a type system so that a richer class of abstractions and invariants could be stated and yet enforced automatically.

Recently, we (and others) have been exploring the design, applications, and foundations of type systems for extremely *low-level* languages. In particular, we have concentrated on the design and implementation of a statically *typed assembly language* (TAL) [14], suitable for execution on realistic machines. Our TAL consists of low-level instructions and data that correspond to a conventional, untyped assembly language, but augmented with annotations that support *static* type-checking. Once a program has been type-checked, the type annotations can be erased and the program can then be directly executed on a machine without fear that the program will violate the abstractions of the type system.

We have also been studying the benefits and drawbacks of compiling high-level languages in a type-preserving fashion to a typed assembly language. In a type-preserving compiler, high-level linguistic abstractions, such as variables, modules, closures, or objects, as well as abstractions introduced by the compiler, such as activation records or calling conventions, must be encoded using typing abstractions provided by TAL.

In our work thus far, we have found that using a typed assembly language within the framework of a type-preserving compiler has a number of compelling benefits:

- Compilers can use type information to support many optimizations. For instance, many code-motion transformations, such as loop invariant removal, require accurate information about whether two variables may contain aliases to the same data structure. A compiler based on TAL can easily determine that two variables do not alias if the types of the variables are incompatible.

- It is easy to find and eliminate a wide class of compiler bugs if the compiler emits TAL code. The reason is that at least some compilation bugs will manifest themselves as type-errors in the output code and be caught by the TAL type-checker. In practice, we have found that indeed, *most* bugs are caught by the checker. For instance, in the implementation of our Safe-C code generator [10], only one bug was not caught by type-checking.
- A current trend in systems software is to allow untrusted extensions to be installed in services, relying upon safe language technology to protect the integrity of the service. For example, Web browsers rely upon the type-safety of the Java Virtual Machine Language (JVML) to protect users from faulty or malicious applets. Because the JVML is not a real machine language, it cannot be directly executed; rather, a trusted interpreter or compiler must be used. The former costs performance, whereas the latter introduces a complicated component into the trusted computing base. By basing an extensible system on TAL, both drawbacks can be avoided.

While imposing a type structure on an assembly language has many benefits, designing the type system involves a number of complicated tradeoffs. On the one hand, we desire a very expressive type system so that we can encode more source-level abstractions, more compiler invariants, and more security properties. On the other hand, to realize the advantages of compiler debugging and security, we need decidable and relatively efficient algorithms for type-checking.

We have discovered that perhaps the most difficult aspect of the design is constructing a suitable formal model of the language and its type system so that we may easily establish a proof of soundness. On the one hand, the model should accurately capture the relevant low-level details of realistic machines. On the other hand, the model should abstract from as many low-level details as possible so that a rigorous proof of soundness is tractable.

Fundamentally, there is no optimal, general solution to these tradeoffs and therefore, we expect that different type systems will be needed in different contexts. It is therefore crucial that we identify general principles in the design of type systems for low-level languages, and useful proof techniques that simplify the construction of the model and the proof of soundness.

In this paper, we discuss some of the general principles that have emerged from our experience with designing, implementing, and using TAL. In particular, we discuss the rewriting model used for TAL, and how its syntactic structure is used to provide a faithful model of realistic machine states, and yet abstract enough details that we may use largely standard proof techniques, developed primarily for high-level languages, to establish soundness. We also state a general principle regarding types for memory states, and show how the type structure of TAL applies this principle in three different ways to achieve a desired level of flexibility. For the sake of brevity and clarity, we omit many of the technical

```
l_fact:   Λ[ρ].code{r1 : int, sp : ρ, ra : ∀[].code{r1 : int, sp : ρ}}.
          bnz r1,l_nzero[ρ] % if n ≠ 0 goto l_nzero
          mov r1,1          % result is 1
          jmp ra            % return
l_nzero:  Λ[ρ].code{r1 : int, sp : ρ, ra : ∀[].code{r1 : int, sp : ρ}}.
          sub sp,sp,2       % allocate stack space for n and the return address
          st sp(0),r1       % save n
          st sp(1),ra       % save return address
          sub r1,r1,1       % set argument to n − 1
          mov ra,l_cont[ρ]  % set return address to l_cont
          jmp l_fact[int :: ∀[].code{r1 : int, sp : ρ} :: ρ] % recursive call
l_cont:   Λ[ρ].code{r1 : int, sp : int :: ∀[].code{r1 : int, sp : ρ} :: ρ}.
          ld r2,sp(0)       % restore n
          ld ra,sp(1)       % restore return address
          add sp,sp,2       % deallocate stack space
          mul r1,r2,r1      % n × (n − 1)!
          jmp ra            % return
```

Fig. 1. TAL Factorial Example

details in our discussion and refer the interested reader to our more detailed reports [14, 11].

2 Basic Design

Figure 1 gives a simple example of TAL code that represents a recursive factorial function. The code consists of three labels, `l_fact`, `l_nzero`, and `l_cont`. Each of the labels is followed by a `code`-type annotation that is used during type-checking. Informally, the annotation specifies a typing pre-condition that must be satisfied before control may be transferred to the corresponding label. For example, the `l_fact` pre-condition requires that register `r1` contain an integer value (the argument to the function), and register `ra` contains a valid return address that itself expects an integer value in `r1` (the return value of the function). As we will discuss later, the pre-condition is polymorphic in the shape of the control stack. That is, the function may be called with any number and any type of values placed upon the stack. Furthermore, upon return, the size of the stack and indeed the values that were on the stack will remain unchanged.

In this example, all of the instructions are standard, RISC-style assembly language instructions with register, integer immediate, or label operands. However, some operands require typing annotations to support type-checking. For example, the operand of the `jmp` instruction in the basic block labelled by `l_nzero` is an explicit instantiation of the polymorphic label `l_fact`.

Whereas the example shown here uses only RISC-style instructions, in practice, we need a few unconventional instructions in order to simplify the type system. In our implementation, we implement these instructions as macros that are expanded at assembly time into real machine instructions. For example, to heap-allocate a data structure such as a record or tuple in TAL, a programmer must use the `malloc` macro instruction. The macro expands into a call to a runtime routine that allocates memory of the appropriate size and returns a pointer to the data. In this case, the macro is necessary because the type of the allocation routine is dependent upon the integer value passed as an argument, but the type system of TAL does not support such dependent types.

Thus, the designer of a typed assembly language does have some latitude in introducing new terms as macros in order to provide some abstraction and to simplify the type system. The price paid is that the macro terms must be translated before the code can be executed. Furthermore, the failure to expose the underlying primitive instructions prevents inter-macro optimizations such as instruction scheduling. It is therefore a delicate task to decide whether to add a new "instruction" or to add new typing constructs.

In the rest of this section, we further motivate and explain the design of the TAL language, and describe the model we use for the semantics of the language.

2.1 Programs and Values

In order to state and prove a type-soundness theorem for TAL code, we must construct a model of program evaluation that is faithful to the semantics of a real machine. We choose to model evaluation using a rewriting system that maps abstract machine states to abstract machine states in a fashion similar to the λ_{gc} machine [12, 13]. This style of abstract machine provides syntactic structure that reflects the intended abstractions provided by a type system, making it easy to state and prove soundness. For example, we draw a syntactic distinction between integer values and pointers. The abstract machine supports arithmetic on integers, but becomes "stuck" if we attempt to perform arithmetic upon pointers, because this would violate the intended abstraction. Thus, we can state that the type system is sound if it prevents programs from entering stuck states.

Though our model abstracts much of the inner workings of a real machine, we must make explicit more details than machines such as λ_{gc}. For example, we must distinguish between data that are stored in a register and data that are stored in memory, because the instructions for accessing data in these locations is quite different. In addition, for typing reasons that will become manifest later, we have found that it is profitable to break memory into two abstract areas: the heap and the control stack. Consequently, the syntax of the TAL abstract machine makes each of these stores explicit.

More formally, a state of the TAL abstract machine (which we sometimes refer to as a "program") is a quadruple, (H, S, R, I), consisting of an abstract heap, stack, register file, and sequence of instructions. The first three correspond to memory components of a real machine; the last is a sequence of instructions to be executed by the machine and thus represents the machine's program counter. (To simplify the discussion, we will neglect the stack until Section 3.3.)

register names	r	$::=$	`r1` $\mid$ `r2` $\mid$ `r3` $\mid \cdots$
word values	w	$::=$	$i \mid \ell$
heap values	h	$::=$	$\langle w_1, \ldots, w_n \rangle$
small values	v	$::=$	$r \mid w$
heaps	H	$::=$	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	R	$::=$	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instruction sequences	I	$::=$	$\epsilon \mid \iota; I$
instructions	ι	$::=$	`mov` r_d, v $\mid$ `bnz` r, v $\mid$ `add` r_d, r_s, v $\mid$ `ld` $r_d, r_s[i]$
			$\mid$ `st` $r_d[i], r_s$ $\mid \cdots$
programs	P	$::=$	(H, R, I)

Fig. 2. Basic Syntax

The syntax includes strictures ensuring that each store contains only abstract values of appropriate size. In particular, registers may only contain word-sized values, whereas the heap may contain values of arbitrary size. Thus, the syntax makes a distinction between *word values* and *heap values*. Word values include integers (i) and labels (ℓ) (also referred to as pointers). Heap values include tuples and code blocks. Tuples are ordered pairs of word values, whereas code blocks are sequences of primitive instructions, prefixed with a typing annotation.

With these syntactic distinctions established, a heap is defined to be a finite mapping of labels (ℓ) to heap values (h), and a register file is defined to be a finite mapping of register names (r) to word values (w). Thus, if register `r1` is to "contain" the tuple $\langle 3, 4 \rangle$, the register file would map `r1` to some label ℓ, and the heap would map ℓ to $\langle 3, 4 \rangle$. These points are summarized in Figure 2.

Certain instructions, such as `mov`, may be used either with a register or literal operand. This gives rise to a third class of values called *small values* (v), which are either registers or word values. Thus the syntax for an instruction to move a small value v into a register `r` is `mov` r, v; the move's destination must be a register, but the source may be a register or a literal value. The distinction between word and small values must be drawn (instead of simply including register names among word values) because a register cannot contain another register.

It is worth noting that this syntactic structure is intentionally designed to be limiting. In particular, by viewing the heap as an abstract mapping of pointers to heap values, we hide additional structure that is present in a conventional machine, such as the relative locations of values in the heap. It is possible to expose such low-level details, but doing so makes it more difficult to state and reason about many properties relevant to type safety, compilers, or security. For example, in the contexts of both garbage collection and address-space isolation, it is crucial that we can easily identify or limit those heap objects that are "reachable" from the current program state. In our framework, doing so is trivial because pointers are abstract and do not admit operations such as pointer arithmetic. In other contexts, it might be necessary to expose lower-level details.

2.2 Types and Code

As stated earlier, the ultimate goal of the type system is to prevent the abstract machine from entering a "bad" or "stuck" state. In order to do so, the type system must track, at all program points, the types of all accessible data objects and prevent control from reaching an instruction that potentially violates the intended abstraction. For the most part, all accessible data objects are reached (directly or indirectly) through registers.[1]

Consequently, the primary burden of the type system is to track the types of the contents of every register. All this information is collected in a *register file type* (Γ), which is a finite mapping of register names to (ordinary) types. We say that the register file $R = \{r_1 \mapsto w_1, \ldots, r_m \mapsto w_m\}$ has type $\Gamma = \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$, if $m \geq n$ and w_i has type τ_i (for every $1 \leq i \leq n$). Note that we permit the forgetting of registers in order to match a register file type.

In a high-level programming language, an expression is type-checked relative to some context that specifies the types of its free variables. In TAL, registers serve the function of free variables, so the instructions of a code block are checked relative to a register file type. Thus, the judgement for instructions is of the form $\Gamma_1 \vdash \iota : \Gamma_2$ which reads informally as, given a register file satisfying the typing pre-condition Γ_1, execution of the instruction ι yields a register file satisfying the typing post-condition Γ_2.[2] For example, the rule instance:

$$\frac{\Gamma(r_1) = \mathtt{int} \qquad \Gamma(r_2) = \mathtt{int}}{\Gamma \vdash \mathtt{add}\ \ r_d, r_1, r_2 : \Gamma\{r_d : \mathtt{int}\}}$$

specifies that we may execute the add instruction in any context where registers r_1 and r_2 contains integer values, and that after the instruction is executed, the register file type will remain unchanged except that register r_d will contain an integer.

Type-checking a sequence of instructions is achieved by using the post-condition of a previous instruction as the pre-condition for the next instruction. The judgement that a sequence of instructions I is valid is written $\Gamma \vdash I$. Note that in this judgement the sequence I is assigned no result type (*i.e.,* no post-condition). This is because all sequences are required to end with an unconditional control transfer (*i.e.,* jump or return). This aspect of TAL is similar to high-level functional languages that require programs to be written in *continuation-passing style* (CPS) [3, 15, 5].

[1] The exception to this, preallocated data objects in the heap, are handled by means very similar to those we now describe, as discussed in Morrisett *et al.* [14].

[2] In this paper we will deal with only a few of the TAL judgements and those only informally; full details appear in Morrisett *et al.* [14, 11]. In particular, the code validity judgement as formalized in Morrisett *et al.* includes other context components in addition to the register file type.

$$
\begin{array}{lrcl}
\textit{heap values} & h & ::= & \cdots \mid \mathsf{code}\, \Gamma.I \\
\\
\textit{types} & \tau & ::= & \mathtt{int} \mid \mathsf{code}\, \Gamma \\
\textit{register file types} & \Gamma & ::= & \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}
\end{array}
$$

Fig. 3. Type Syntax

The CPS nature of low-level code is important to understand the types that are assigned to code blocks. Because well-formed blocks end with an unconditional jump, control will never fall off the end of the block. Thus, logically speaking, the post-condition could specify *any* type for the registers since control will never reach that point.[3] We therefore omit the post-condition on well-formed blocks of code.

Within the abstract machine, each code block that is bound to a label must is prefixed with its typing pre-condition, written $\mathtt{code}\,\Gamma.I$. If I is valid assuming that register file type on entry (*i.e.,* if $\Gamma \vdash I$), then we say that the code block has type $\mathtt{code}\,\Gamma$. The new syntactic constructs are summarized in Figure 3. (For convenience we assume that code blocks are located in the heap. Types for tuples are omitted from the figure; they are the subject of Section 3.2.)

For example, in a high-level language such as ML, the factorial function would have type $\mathtt{int} \rightarrow \mathtt{int}$. In a CPS-based functional language, the type would be $(\mathtt{int} * (\mathtt{int} \rightarrow \mathtt{void})) \rightarrow \mathtt{void}$, reflecting the fact that the integer-accepting continuation is passed as a second argument. Similarly, in TAL the type of the function would be $\mathtt{code}\{\mathtt{r1}{:}\mathtt{int}, \mathtt{r2}{:}\,\mathtt{code}\{\mathtt{r1}{:}\mathtt{int}\}\}$ (assuming the argument and result are passed in `r1` and the return address in `r2`).

2.3 Quantified Types

In order to support polymorphism and data abstraction, TAL supplies universal and existential [9] types. This is important so that TAL may serve as a target language for compilers of high-level languages with these features, of course, but even monomorphic features of high-level languages often implicitly require type quantification. For example, existential types arise in objects [1] and closures [8], and universal types arise in stacks [11].

For simplicity, universal quantification is limited to code, since that is where the intended use of universal types lies. The introduction form (and the type) allows for the simultaneous abstraction of a series of type variables $\alpha_1, \ldots, \alpha_n$, collectively written Δ. Note that the elimination form, the instantiation of a value at a type (written $w[\tau]$ or $v[\tau]$), is considered a value, rather than a function call. This is permissible because we interpret TAL in a manner consistent with type erasure. That is, types are never relevant to execution and may be erased, and consequently there is no run-time difference between w and $w[\tau]$.

[3] Equivalently, we could use "false" (*i.e.,* a void or empty type) for the post-condition.

$$
\begin{array}{llcl}
\textit{word values} & w & ::= & \cdots \mid w[\tau] \mid \mathtt{pack}[\tau, w]\ \mathtt{as}\ \exists\alpha.\tau' \\
\textit{heap values} & h & ::= & \cdots \mid \Lambda[\Delta].\,\mathtt{code}\,\Gamma.I \\
\textit{small values} & v & ::= & \cdots \mid v[\tau] \mid \mathtt{pack}[\tau, v]\ \mathtt{as}\ \exists\alpha.\tau' \\
 & & & \\
\textit{types} & \tau & ::= & \cdots \mid \alpha \mid \forall[\Delta].\,\mathtt{code}\,\Gamma \mid \exists\alpha.\tau \\
\textit{tyvar series} & \Delta & ::= & \epsilon \mid \Delta, \alpha
\end{array}
$$

Fig. 4. Universal and Existential Types

The type-erasure interpretation makes TAL more faithful to real machines (which, of course, do not compute using types), and it makes some aspects of compilation substantially simpler, such as closure conversion [14]. Others, however, it makes more complicated, such as intensional polymorphism [6, 2]. An extensive discussion of type erasure, and of how type-passing languages may be compiled to type-erasure languages such as TAL, appears in Crary *et al.* [2].

Because of the type-erasure interpretation, the introduction form for existential types (written $\mathtt{pack}[\tau, w]\ \mathtt{as}\ \exists\alpha.\tau'$ or $\mathtt{pack}[\tau, v]\ \mathtt{as}\ \exists\alpha.\tau'$) is also considered a value. The elimination form for existential types is the instruction $\mathtt{unpack}[\alpha, r], v$. The type variable α is bound in the remainder of the instruction sequence. When v contains the pack value $\mathtt{pack}[\tau, w]\ \mathtt{as}\ \exists\alpha.\tau'$, the unpack instruction executes by moving w into r and substituting τ for α is the remainder of the instruction sequence. Under the type-erasure interpretation, this is implementable at run-time as a simple move.

2.4 Scope and Alpha-Conversion

An important observation is the status of scope and alpha-conversion in TAL. So far we have seen three forms of "variable": type variables, register names, and pointers. Binding occurrences of type variables within code are introduced at the beginning of polymorphic code block or within an `unpack` instruction, and the scope of the variable extends to the end of the ensuing instruction sequence. As usual, we consider code blocks (and types) to be equivalent up to alpha-conversion of bound type variables.

Register names, however, do not alpha-convert; the purpose of making them explicit in the language is to be able to identify particular registers for calling conventions and for data placement. Nevertheless, there is ample evidence that variable names should not be important in well-behaved programming languages.

These two facts are reconciled by observing that registers are not really variables. Consider the code block `code{r1:int}.mov r2,r1;mov r3,r1;`... The argument to this code is not really the integer in `r1`, it is the *entire register file.* The register file is essentially a linear record [4, 17, 18]; each instruction consumes the register file and produces a new one. In essence the code amounts to the following representation with explicit register files:

$$
\begin{array}{l}
\lambda \mathit{rf}{:}\{\mathtt{r1{:}int}\}.\,\mathtt{let}\ \mathit{rf}' = (\mathtt{mov\ r2,r1})(\mathit{rf})\ \mathtt{in} \\
\qquad\qquad\qquad\quad \mathtt{let}\ \mathit{rf}'' = (\mathtt{mov\ r3,r1})(\mathit{rf}')\ \mathtt{in}\ \ldots
\end{array}
$$

Any of these register files names can be alpha-varied as usual. However, since the invariant holds that at every point there is exactly one register file available, we simply elide any mention of register files by name.

For pointers there is an open choice: we can make them alpha-vary, or we can view the entire memory as an enormous linear object as we did with the register file. Without a compelling case for the latter, we decide to make them alpha-vary for technical convenience. This means that when allocating memory, we need not specify the strategy for generating a new address; we may simply choose any unused pointer. However, as a consequence, it is impossible to perform any operations on pointers such as hashing them. Under some circumstances these operations may be useful, but it can also be argued that they often violate important abstraction properties: given objects of abstract type, it should be impossible to determine any properties about them, but it will sometimes be possible in the presence of pointer operations.

3 Memory and Aliasing

The most delicate aspect of the design of TAL lies in the way its type system handles data in memory. As we observed in Section 2.2, for the type system to work, it must track the types of all accessible data objects. As this applies to memory, it means that at every program point, the type system must have an accurate view of the contents of all accessible memory locations. An immediate consequence of this is that the type system cannot permit access to any memory location of which it does not have a view. In other words, arbitrary, unrestricted memory accesses are fundamentally incompatible with the type system.

Given that the type system permits memory accesses only to locations of which it has a view, a more subtle and important issue arises. The type system must ensure the invariant that its view of every accessible memory location is accurate. In the presence of aliasing, this invariant is not trivial to maintain: Suppose a memory location is accessible via two different paths, and the program, using one of those paths, modifies the memory location so that its type changes. It is easy enough to modify the view of that memory corresponding to the access path that was used, but to maintain the type system's invariants, we must also modify the view of that location along the other, aliasing, access path. Unfortunately, it is very difficult to determine statically whether two paths alias, and even harder to find statically all aliases of a given path.

The basic principle to be observed then is: *Any change to a memory location must not invalidate the type system's view of that location along any usable access path.* In the remainder of this section, we discuss several memory type mechanisms that adhere to this principle, each one being suitable for different problems.

3.1 Registers

The simplest possible means to track aliasing is to prevent it from happening. One common mechanism to do this is to use linear types [4,17,18], which prevents there from being multiple access paths to any piece of data. An even more draconian mechanism is to disallow any indirect access to data at all, allowing only a single, unit-length access paths. However, this mechanism is entirely appropriate for machine registers, which cannot be accessed indirectly anyway. Thus, registers can be modified freely, updating their types in the register file type, without fear that any aliases to those registers would be given an inaccurate view.

3.2 The Heap

The heavy-handed mechanism that works for registers is clearly unacceptable for the heap, where it is essential to permit multiple access paths to data objects. Nevertheless, it is impractical to track aliasing in the heap. Instead, we assume that any memory location may have any number of unknown aliases, and employ a typing regime in which a modification cannot invalidate the type system's view of any of those aliases.

This regime works as follows: When a memory location is first allocated, it is stamped with a type. From that point on, stores into the memory location may only write values of the type with which the location is stamped. Since the stamping occurs when the memory location is first allocated, all aliases will view the memory location as stamped in the same way. Therefore, any stores will write only values appropriate to what is expected along other access paths.

This mechanism is familiar as the typing regime for references in Standard ML [7] and other ML dialects. However, a complication arises in TAL because when first allocated, memory locations are uninitialized and conceptually contain junk. In contrast, in Standard ML an initial value is provided for references, so an uninitialized state is avoided.

An attractive alternative is to give uninitialized memory locations a *nonsense* type and refine that type when it is initialized. But this mechanism is barred by the solution to the aliasing problem. (This mechanism will be suitable in the next section when dealing with stacks.)

The solution we have adopted to this problem is to flag every memory location with its initialization state. Suppose a two-word block is allocated, intended to contain a pair of integers. The newly allocated location is given the type $\langle \mathtt{int}^0, \mathtt{int}^0 \rangle$. The 0 flags indicate that both fields of the tuple are uninitialized and may not be read from. When a field of a tuple is initialized, the flag is changed to a 1, indicating that it is permissible to read from the field. Thereafter, interference [16] is possible, in that an initialized field may be changed using alternative access paths, but any such changes can only write integers, so all views of that field remain valid.

One form of inaccurate view can arise in this regime; it is possible for a field to be initialized using one access path while another path still believes it to be uninitialized. However, this sort of inaccuracy is benign; it does no harm for an alias to believe it may not read from a field that happens to be initialized. Thus, a better way to read the "0" flag is that the corresponding field is *possibly* uninitialized.

$$\begin{array}{lll} \textit{types} & \tau & ::= \; \cdots \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \\ \textit{initialization flags} & \varphi & ::= \; 0 \mid 1 \end{array}$$

Fig. 5. Heap Typing Mechanisms

There are two essential limitations on TAL code that make the above mechanism work. First, it is not permissible to reuse a memory location at a different type. Once memory is allocated, it remains stamped with the same type until it is reclaimed by the garbage collector (at which point the collector has verified that there are no access paths to that memory).

Second, we cannot expose the internal workings of the memory allocation process. Instead, memory allocation is viewed as an atomic operation that finds the needed memory and stamps it with a type. In the semantics, allocation is performed by a primitive `malloc` instruction; in practice the `malloc` instruction macro-expands (after typechecking) to an appropriate fixed code sequence. This limitation is necessary because any finer view of the memory allocation process would expose that allocated space is drawn from a pool of untyped memory. Our approach for dealing with aliasing depends essentially on the fact that no location ever exists (as far as the type system is concerned) without being typed.

3.3 The Stack

To supports stacks in Typed Assembly Language, we must consider a third approach to dealing with the aliasing problem. The typing regime discussed in the previous section for heap-allocated space is unsuitable for stack-allocated space, because the pattern of access to the stack is quite different than to the heap. For the heap we made the significant restriction that no space could ever be re-used at a different type. This restriction is unacceptable for stack space: the entire purpose of the stack is to re-use space repeatedly during dynamic growing and shrinking of the stack.

Recall that the basic principle to observe is that any change to a memory location must not invalidate the type system's view of that location along any *usable* access path. For the heap we adopted a regime ensuring no change could invalidate the view along any path. For the stack we take a different approach: it will be possible to invalidate the view of a stack slot along some access paths, but such paths will become unusable.

Before we can discuss the regime that enforces this, we must begin with some preliminaries. First, we introduce the notion of a *stack type*, which describes the shape of the stack. Stack types are either `nil`, indicating that the stack is empty, or $\tau{::}\sigma$, indicating that the first value on the stack has type τ and the remainder is described by σ. The stack is accessed by an identified stack pointer register (`sp`), and the register file type accordingly assigns `sp` a stack type, rather than a regular type. This is summarized in Figure 6.

The type system maintains the invariant that the stack type assigned to `sp` by the current register file type is always correct. For example, allocating

$$\begin{array}{lrcl}
\textit{types} & \tau & ::= & \cdots \mid \texttt{ns} \\
\textit{stack types} & \sigma & ::= & \texttt{nil} \mid \tau :: \sigma \\
\textit{register file types} & \Gamma & ::= & \{\mathrm{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \\
 & & & \\
\textit{stacks} & S & ::= & \texttt{nil} \mid w :: S \\
\textit{programs} & P & ::= & (H, S, R, I)
\end{array}$$

Fig. 6. Stack Typing Mechanisms

$$\begin{array}{lrcl}
\textit{word values} & w & ::= & \cdots \mid \texttt{ptr}(i) \\
\textit{types} & \tau & ::= & \cdots \mid \texttt{ptr}(\sigma)
\end{array}$$

Fig. 7. Stack Indirection

one word of space on the stack (in practice performed by subtracting the word size from the stack pointer) when sp has type σ, changes sp's type to $\texttt{ns} :: \sigma$, indicating that the top stack word is nonsense (garbage), and the remainder of the stack is as before. Unlike the regime for heaps, we may store a value of any type into any stack slot, and the type of sp will change accordingly. Thus, if the sp has type $\texttt{int} :: \texttt{ns} :: \sigma$, then storing an integer into the second stack slot (*e.g.,* mov sp$(1), 12$) results in the type $\texttt{int} :: \texttt{int} :: \sigma$.

To this point, we have trivially avoided any aliasing problems by providing only a single access path into the stack, through sp. We now wish to consider what happens we add pointers into the stack. Syntactically, we represent pointers into the stack by $\texttt{ptr}(i)$, which points to the ith word of the stack. Pointers into the stack are indexed from the bottom, so they need not change when the stack grows or shrinks. The type of a pointer into the stack is given by $\texttt{ptr}(\sigma)$, which describes the segment of the stack lying below that pointer. This is summarized in Figure 7.

A pointer into the stack is first obtained by copying the stack pointer into another register. At that point, that register has type $\texttt{ptr}(\sigma)$, where σ is the type of sp. However, further computation may change the stack's type, and this will not be reflected in the type of the copy (to do so would require tracking of aliasing). For example:

```
;; begin with sp : $\langle\texttt{int}^1\rangle :: \sigma$
mov r1, sp        ;; r1 : ptr($\langle\texttt{int}^1\rangle :: \sigma$)
mov sp(0), 12     ;; sp : int $:: \sigma$, but still r1 : ptr($\langle\texttt{int}^1\rangle :: \sigma$)
```

After the second line, the type for r1 is no longer consistent with what it points to. If the type system permits loading from r1, the resulting value will be believed to have the tuple type $\langle\texttt{int}^1\rangle$, but in fact will be the integer 12.

The problem is that the store to sp(0) has invalidated the view of the stack held by r1. To solve this problem, we disallow loading from any pointer whose view has been invalidated. It is easy to determine statically when such an invali-

$$\begin{array}{lrcl}
\textit{word values} & w & ::= & \cdots \mid w[\sigma] \\
\textit{small values} & v & ::= & \cdots \mid v[\sigma] \\
\\
\textit{stack types} & \sigma & ::= & \cdots \mid \rho \mid \sigma_1 \cdot \sigma_2 \\
\textit{tyvar series} & \Delta & ::= & \cdots \mid \Delta, \rho
\end{array}$$

Fig. 8. Stack Hiding Mechanisms

dation has occurred because `sp`'s type is ensured always to be valid. Thus, when loading from a pointer into the stack, that pointer's type is compared against `sp`'s type and the load is rejected if the pointer's type is inaccurate.

Stack Polymorphism The problem with the type discipline discussed to this point is that stack types always fully describe the shape of the stack. In practice, however, it is extremely important to be able to ignore irrelevant portions of the stack. Typically, nearly all the stack is irrelevant. Recall that a function with type $\texttt{code}\{\texttt{sp}{:}\sigma, \ldots\}$ can only be called with stacks described by σ. If σ fully describes the stack, then calls to that function are limited to cases in which the stack has the indicated depth and components. One of many negative consequences of this is that recursion is impossible.

We settle this problem by adding stack type variables (ρ) and polymorphism over stack types. Thus, if the only relevant element of the stack is the top value (say, an integer argument), then we might use the type $\forall[\rho].\,\texttt{code}\{\texttt{sp}{:}\texttt{int}{::}\rho, \ldots\}$. By a suitable instantiation of ρ, this function could be called with any stack whose top element is an integer.

This mechanism allows us to ignore all of the stack after some initial segment. However, it is also often useful to ignore a non-terminal segment of the stack. For instance, we might care about some material at the top of the stack and some in the middle, but none in between. This often happens with exception handlers or displays placed amidst the stack. To support this, we add a facility for concatenating stack types. Thus, $(\texttt{ns} :: \texttt{nil}) \cdot (\texttt{int} :: \texttt{nil})$ is equivalent to $\texttt{ns} :: \texttt{int} :: \texttt{nil}$. These mechanisms are summarized in Figure 8.

Stack concatenation is useful in conjunction with stack type variables. Suppose we are interested in a value with type τ located amidst the stack, but are not interested in any other part. Then we may specify the stack to have type $\rho_1 \cdot (\tau :: \rho_2)$, where ρ_1 and ρ_2 are universally quantified. By suitable instantiation of ρ_1 and ρ_2, this type will apply to any stack that contains a τ value.

Note that since the length of ρ_1 is unknown, there is no way to locate the τ value of interest. It is in this setting that it is important to support pointers into the middle of the stack, raising the aliasing issues discussed above. Thus, a function that wished to receive a τ value passed in the middle of the stack would be given a type such as $\forall[\rho_1, \rho_2].\,\texttt{code}\{\texttt{sp}{:}\rho_1 \cdot (\tau :: \rho_2), \texttt{r1}{:}\texttt{ptr}(\tau :: \rho_2)\}$. This type typically arises when τ is the type of an exception handler and `r1` is the exception register [11].

Limitations The principal limitation of this approach to typing stacks is that an undesirable amount of information must be explicitly specified. Although polymorphism can be used to hide portions of the stack that are irrelevant, all stack components that are of interest must be specified in the stack's type, even when those components are accessed using a pointer. In contrast, when data is stored on the heap, only a well-typed heap pointer is necessary.

Another limitation is that when the stack contains multiple values of interest, the stack type must specify their relative order, if not their precise locations. This difficulty can be ameliorated by adding intersection types to provide multiple views of the same stack. For example, when the stack contains two interesting values (with types τ_1 and τ_2) in no particular order, the stack could be given type $(\rho_1 \cdot \tau_1{::}\rho'_1) \wedge (\rho_2 \cdot \tau_2{::}\rho'_2)$. We have not yet explored all the ramifications of this enhancement, as experience has not yet demonstrated a convincing need for it.

A third limitation stems from having two different type systems, one for heaps and one for stacks. Since the data in the heap are given different types than data in the stack (tuple types versus `ptr` types), interfaces must specifically state in which their data will lie. Because of this, and because of the limitations of stacks discussed above, programmers prefer to keep data on the heap unless there is a particular reason to place it on the stack. In practice, then, TAL stacks are not really a first-class storage medium.

4 Conclusion

There are a number of difficult but interesting tradeoffs in the design of type systems for low-level languages and, in all likelihood, there is no "ultimate" type system. Rather, a language designer must weigh the costs and benefits of expressiveness versus simplicity. In the design of TAL, we opted for the latter when reasonable.

In this paper, we have stated a general principle regarding types for memory locations that is useful in the design of any programming language. In essence, the principle states that we may change the contents of a location in memory, as long as we track the changes in the type system along all accessible paths. Within the context of Typed Assembly, we showed how this principle is used in three different ways to achieve some needed expressiveness while retaining as much simplicity as possible. In particular, we argued that a linear discipline was appropriate for register files, but was too inflexible for heaps and stacks. Similarly, we argued that the standard reference discipline of high-level languages is appropriate for most heap objects, but that a validation approach may be more appropriate for objects allocated on the stack.

References

1. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, Sendai, Japan, September 1997.

2. Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998. Extended version published as Cornell University technical report TR98-1721.
3. M.J. Fischer. Lambda calculus schemata. In *ACM Conference on Proving Assertions About Programs, SIGPLAN Notices 7(1)*, pages 104–109, 1972.
4. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
5. Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, January 1993.
6. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.
7. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.
8. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.
9. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
10. Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999. To appear.
11. Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science.* Springer-Verlag, March 1998. Extended version published as CMU technical report CMU-CS-98-178.
12. Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, California, June 1995.
13. Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1997.
14. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 1999. To appear. An earlier version appeared in 1998 Symposium on Principles of Programming Languages.
15. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972.
16. John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978.
17. Philip Wadler. Linear types can change the world! In *IFIP Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North-Holland.
18. Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.