4 *Typed Assembly Language*

Greg Morrisett

How can we ensure that when we download some code that the code will not do something "bad" when we execute it? One idea is to leverage the principle of *proof-carrying code* introduced by Necula and Lee. The principle of PCC is that we can eliminate the need to trust a piece of code by requiring a formal, machine-checkable proof that the code has some desired properties. The key insight is that checking a proof is usually quite easy, and can be done with a relatively small (and hence trustworthy) proof-checking engine.

If we are to effectively use PCC to build trustworthy systems, then we must solve two problems:

1. What properties should we require of the code?
2. How do code producers construct a formal proof that their code has the desired properties?

The first question is extremely context and application dependent. It requires that we somehow rule out all "bad" things without unduly restricting "good" things, and make both "bad" and "good" formal. The second question is impossible to solve automatically for arbitrary code assuming non-trivial safety properties. So, how are we to take advantage of PCC?

One approach is based on *type-preserving* compilation. The idea here is to focus on some form of type safety as the desired property. The advantage of focusing on type safety is that programmers are willing to do the hard part—construct a proof that some code is type-safe. The way they do this is by writing the code in a high-level language (e.g., Java or ML). If the source they write doesn't type-check, then they rewrite the code until it does. In this respect, they are engaging in a form of interactive theorem proving.

Once the initial proof is done, a type-preserving compiler takes over, mapping the type-safe source code through a series of successively lower-level

intermediate languages to target code. As it transforms the code, it also (conceptually) transforms the proof. It is usually much easier to do this sort of transformation than to prove type safety directly on the generated code.

Of course, such a methodology demands that as part of the compilation process, we design a series of typed intermediate languages, culminating with typed machine code. Intermediate languages such as the Java Virtual Machine Language (JVML) and Microsoft's Common Language Infrastructure (CLI) are targets of type-preserving compilers for a number of high-level languages, including Java, C#, and ML.

However, both the JVM and CLI are relatively high-level "CISC-like" abstract machines. That is, they have pre-conceived notions of methods and objects which may be incompatible with an efficient encoding for a given language. For instance, the JVM does not support tail-calls, making the implementation of functional languages impractical. The CLI does support tail-calls, but there are other features that it lacks. For instance, arrays are treated as covariant in the type system, and thus require a run-time check upon update.

Of course, there will never be a universal, portable typed intermediate language (TIL) that supports all possible languages and implementation strategies. Nonetheless, we seek a principled approach to the design of TILs that minimizes the need to add new features and typing rules. In particular, we seek a more "RISC-like" design for type systems that makes it possible to encode high-level language features and to support a variety of optimizations.

4.1 TAL-0: Control-Flow-Safety

We begin our design for a "RISC"-style typed assembly language by focusing on one safety property, known as *control-flow safety*. Informally, we wish to ensure that a program does not jump to arbitrary machine addresses throughout its execution, but only to a well-defined subset of possible entry points. Control-flow safety is a crucial building block for building dynamic checks into a system. For instance, before performing a system call, such as a file read, we might need to check that the arguments to the call have the right properties (e.g., the file has been opened for reading and the destination buffer is sufficiently large.) Without control-flow safety, a malicious client could jump past these checks and directly into the underlying routine.

A focus on control-flow safety will also let us start with an extremely simple abstract machine and demonstrate the key ideas of adapting a type system to machine code. In subsequent sections, we will expand this machine and its type system to accommodate more features.

The syntax for our control-flow-safe assembly language, which we will call TAL-0, is given in Figure 4-1. We assume a fixed set of k general-purpose

r	::=	*registers:*	ι	::=	*instructions:*
	`r1` | `r2` | ··· | `rk`			r_d`:=`v	
v	::=	*operands:*		r_d`:=`r_s`+`v	
	n	*integer literal*		`if` r `jump` v	
	ℓ	*label or pointer*	I	::=	*instruction sequences:*
	r	*registers*		`jump` v	
				ι`;`I	

Figure 4-1: Instructions and operands for TAL-0

registers and a few representative instructions based on a subset of MIPS assembly language. To keep the language readable, we use a somewhat more familiar notation for instructions than the usual cryptic "`mov`," "`add`," etc. Intuitively, each instruction uses the value in a source register (r_s) and an operand to compute a value which is placed in the destination register (r_d). In this setting, an operand is either another register, a word-sized immediate integer,[1] or a label. We use "value" to refer to an operand that is not a register.

For our purposes, it is also useful to define instruction sequences (I) as lists of instructions terminated by an explicit, unconditional control transfer (i.e., a jump). Of course, when the assembly code is mapped down to machine code, jumps to adjacent blocks can be eliminated since the code will fall through. We could make the order of instruction sequences and hence fall-throughs explicit, but we prefer a simpler, more uniform assembly language to present the key ideas.

Here is an example TAL-0 code fragment that computes the product of registers `r1` and `r2`, placing the final result in `r3` before jumping to a return address assumed to be in `r4`.

```
prod: r3 := 0;          // res := 0
      jump loop

loop: if r1 jump done; // if a = 0 goto done
      r3 := r2 + r3;   // res := res + b
      r1 := r1 + -1;   // a := a - 1
      jump loop

done: jump r4          // return
```

1. As we are dealing with an assembly language, we ignore the issue of fitting a full word-sized integer into a single instruction.

R	::=	*register files:*	H	::=	*heaps:*
	$\{\mathtt{r}_1 = v_1, \ldots, \mathtt{r}_k = v_k\}$			$\{\ell_1 = h_1, \ldots, \ell_m = h_m\}$	
h	::=	*heap values:*	M	::=	*machine states:*
	$\mathtt{I}$	*code*		(H, R, I)	

Figure 4-2: TAL-0 abstract machine syntax

We model evaluation of TAL-0 assembly programs using a rewriting relation between *abstract* machine states. Rather than model the execution of a concrete machine, we use a higher-level representation for machine states which maintains certain distinctions. For instance, in a real machine, labels are resolved during program loading to some machine address which is also represented as an integer. In our abstract machine, we maintain the distinction between labels and arbitrary integers because this will allow us to easily state and then prove our desired safety property—that any control flow instruction can only branch to a valid, labelled entry point. Indeed, our abstract machine will get stuck if we try to transfer control to an integer as opposed to a label. So, the problem of enforcing the safety property now reduces to ensuring that our abstract machine cannot get stuck.

However, to support this level of abstraction, we must worry about the situation where a label is added to an integer, or where a test is done on a label. One possibility is to assume a coercion function `intof` which maps labels to integers, and use $\mathtt{intof}(\ell)$ whenever ℓ appears as an operand to an arithmetic instruction. Though a perfectly reasonable approach, there are a number of reasons one might avoid it: First, it violates the abstraction that we are attempting to provide which may lead to subtle information flows. In some security contexts, this could be undesirable. Second, such a coercion would make it harder to prove the equivalence of two program states, where labels are α-converted. In turn, this would restrict an implementation's freedom to re-arrange code or recycle its memory.[2] Finally, it simplifies the type system if we simply treat labels as abstract. In particular, if we included such a coercion, we would need some form of subtyping to validate the coercion. However, we emphasize that it is possible to expose labels as machine integers if desired.

The syntax for TAL-0 abstract machines is given in Figure 4-2. An abstract machine state M contains three components: (1) A heap H which is a finite,

2. The problem can be avoided by assuming a coercion relation between labels and integers that respects the alpha-equivalence class of labels.

partial map from labels to heap values (h), (2) a register file R which is a total map from registers to values, and (3) a current instruction sequence I. The instruction sequence is meant to model the sequence of instructions pointed to by the program counter in a concrete machine. One way to think of the machine's operation is that it pre-fetches sequences of instructions up to the nearest `jump` whenever a control-transfer is made.

We consider heaps and register files to be equivalent up to re-ordering. We also consider the labels in the domain of H to bind the free occurrences of those labels in the rest of the abstract machine. Finally, we consider abstract machine states to be equivalent up to alpha-conversion of bound labels.

The rewriting rules for TAL-0 are as follows:

$$\frac{H(\hat{R}(v)) = I}{(H, R, \texttt{jump}\ v) \longrightarrow (H, R, I)} \quad \text{(JUMP)}$$

$$(H, R, r_d\texttt{:=}v; I) \longrightarrow (H, R[r_d = \hat{R}(v)], I) \quad \text{(MOV)}$$

$$\frac{R(r_s) = n_1 \qquad \hat{R}(v) = n_2}{(H, R, r_d\texttt{:=}r_s\texttt{+}v; I) \longrightarrow (H, R[r_d = n_1 + n_2], I)} \quad \text{(ADD)}$$

$$\frac{R(r) = 0 \qquad H(\hat{R}(v)) = I'}{(H, R, \texttt{if}\ r\ \texttt{jump}\ v; I) \longrightarrow (H, R, I')} \quad \text{(IF-EQ)}$$

$$\frac{R(r) = n \qquad n \neq 0}{(H, R, \texttt{if}\ r\ \texttt{jump}\ v; I) \longrightarrow (H, R, I)} \quad \text{(IF-NEQ)}$$

The rules make use of $\hat{R}$ which simply lifts R from operating on registers to operands:

$$\hat{R}(r) = R(r)$$
$$\hat{R}(n) = n$$
$$\hat{R}(\ell) = \ell$$

Notice that for `jump` and a successful `if-jump` we simply load a new instruction sequence from the heap and begin executing it. Of course, this assumes that the destination operand evaluates to some label ℓ (as opposed to an integer), and that the heap provides a binding for ℓ. Otherwise, the machine cannot make the transition and becomes stuck. For instance, if we attempted to evaluate `jump 42`, then no transition could occur. In other words, if we can devise a type system that rules out such stuck machine states, then we can be assured that all control transfers must go to properly labelled instruction sequences.

Of course, there are other ways this particular abstract machine can get stuck. For instance, if we attempt to add an integer to a label, or test a label

using `if`, then the machine will get stuck. This reflects our choice to leave labels abstract.

4.1.1 EXERCISE [⋆, ↛]: Taking H to be a heap that maps the labels `prod`, `loop`, and `done` to their respective instruction sequences above, and taking R_0 = `{r1=2,r2=2,r3=0,r4=exit}` where `exit` is some unspecified label, show that $(H, R_0,$ `jump prod`$)$ steps to a state $(H, R,$ `jump r4`$)$ with $R($`r3`$) = 4$. □

4.1.2 EXERCISE [RECOMMENDED, ⋆⋆⋆, ↛]: Build an interpreter for the TAL-0 abstract machine in your favorite programming language. □

4.1.3 EXERCISE [⋆⋆⋆, ↛]: Formulate a semantics for a concrete machine based on the TAL-0 instruction set. The concrete machine should manipulate only integers and have states of the form (M, R, pc) where M is a memory mapping 32-bit integers to 32-bit integers, R is a register file, and pc holds a 32-bit integer for the next instruction to execute. You should assume an isomorphism `encode` and `decode` that maps instructions to and from distinct integers.

Then, prove that the TAL-0 abstract machine is faithful to the concrete machine by establishing a simulation relation between abstract and concrete machine states and by showing that this relation is preserved under evaluation. □

4.2 The TAL-0 Type System

The goal of the type system for TAL-0 is to ensure that any well-formed abstract machine M cannot get stuck—that is, there always exists an M' such that $M \longrightarrow M'$. Obviously, our type system is going to have to distinguish labels from integers to ensure that the operands of a control transfer are labels. But we must also ensure that, no matter how many steps are taken by the abstract machine, it never gets into a stuck state (i.e., typing is preserved.) Thus, when we transfer control to a label, we need to know what kinds of values it expects to have in the registers.

To this end, we define our type syntax in Figure 4-3. There are four basic type constructors. Obviously, `int` will be used to classify integer values and `code` types will classify labels. Furthermore, `code`(Γ) will classify those labels which, when jumped to, expect to have values described by Γ in the associated register. Here, Γ is a *register file type*—a total function from registers to types. In this respect, we can think of a label as a continuation which takes a record of values described by Γ as an argument.

We also support universal polymorphism in TAL-0 through the addition of type variables (α) and quantified types ($\forall\alpha.\tau$). As usual, we consider types

τ	::=	*operand types:*	Γ	::=	*register file types:*
	`int`	*word-sized integers*		$\{r_1 : \tau_1, \ldots, r_k : \tau_k\}$	
	`code`(Γ)	*code labels*	Ψ	::=	*heap types:*
	α	*type variables*		$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$	
	$\forall\alpha.\tau$	*universal polymorphic types*			

Figure 4-3: TAL-0 type syntax

up to alpha-equivalence of bound type variables. Finally, we consider register file types and heap types to be equivalent up to re-ordering.

With these static constructs in hand, we can now formalize the type system using the inference rules in Figure 4-4. The first judgment, $\Psi \vdash v : \tau$, is used to determine the type of a value. Recall that a value is a register-free operand, so there is no need for a register context in the judgment. Integer literals are given type `int`, whereas labels are given the type assigned to them by the heap type context Ψ. Note that this rule only applies when the label is in the domain of Ψ.

The second judgment, $\Psi;\Gamma \vdash v : \tau$ lifts value typing to operands. A register is given the type assigned by the register file type Γ. In addition, a polymorphic operand can be instantiated with any type, in a fashion similar to ML.[3]

The next judgment, $\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$ is used to check instructions. The notation is meant to suggest that the instruction expects a register file described by Γ_1 on input, and produces a register file described by Γ_2 on output. Note that for the `if` instruction, we must ensure that the destination operand v is a code pointer that expects a register file described by the same Γ as any subsequent instruction. This ensures that, no matter which way the branch goes, the resulting machine state will be well-formed.

The judgment $\Psi \vdash I :$ `code`(Γ) assigns an instruction sequence I the type `code`(Γ) when the sequence expects to be given a register file described by Γ. In particular, a `jump` instruction's type is dictated by the type of its operand. The code type for a sequence of instructions is determined from composition. Most importantly, we can generalize the type of an instruction sequence by abstracting any type variables. Note that there is no need to prevent abstraction of type variables which occur free in the context Ψ, as is the case with generalization in ML. This is because Ψ will only contain closed types (see below). Thus, generalization is always possible.

3. We could also include instantiation for values, but to keep things simple, we have omitted that rule.

Values $\boxed{\Psi \vdash v : \tau}$

$$\Psi \vdash n : \texttt{int} \quad \text{(S-INT)}$$

$$\Psi \vdash \ell : \Psi(\ell) \quad \text{(S-LAB)}$$

Operands $\boxed{\Psi;\Gamma \vdash v : \tau}$

$$\Psi;\Gamma \vdash r : \Gamma(r) \quad \text{(S-REG)}$$

$$\frac{\Psi \vdash v : \tau}{\Psi;\Gamma \vdash v : \tau} \quad \text{(S-VAL)}$$

$$\frac{\Psi;\Gamma \vdash v : \forall\alpha.\tau}{\Psi;\Gamma \vdash v : \tau[\tau'/\alpha]} \quad \text{(S-INST)}$$

Instructions $\boxed{\Psi \vdash \iota : \Gamma_1 \to \Gamma_2}$

$$\frac{\Psi;\Gamma \vdash v : \tau}{\Psi \vdash r_d\texttt{:=}v : \Gamma \to \Gamma[r_d : \tau]} \quad \text{(S-MOV)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \texttt{int} \qquad \Psi;\Gamma \vdash v : \texttt{int}}{\Psi \vdash r_d\texttt{:=}r_s\texttt{+}v : \Gamma \to \Gamma[r_d : \texttt{int}]} \quad \text{(S-ADD)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \texttt{int} \qquad \Psi;\Gamma \vdash v : \texttt{code}(\Gamma)}{\Psi \vdash \texttt{if}\ r_s\ \texttt{jump}\ v : \Gamma \to \Gamma} \quad \text{(S-IF)}$$

Instruction Sequences $\boxed{\Psi \vdash I : \tau}$

$$\frac{\Psi;\Gamma \vdash v : \texttt{code}(\Gamma)}{\Psi \vdash \texttt{jump}\ v : \texttt{code}(\Gamma)} \quad \text{(S-JUMP)}$$

$$\frac{\Psi \vdash \iota : \Gamma \to \Gamma_2 \qquad \Psi \vdash I : \texttt{code}(\Gamma_2)}{\Psi \vdash \iota; I : \texttt{code}(\Gamma)} \quad \text{(S-SEQ)}$$

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall\alpha.\tau} \quad \text{(S-GEN)}$$

Register Files $\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\forall r.\Psi \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma} \quad \text{(S-REGFILE)}$$

Heaps $\boxed{\vdash H : \Psi}$

$$\frac{\forall \ell \in dom(\Psi).\Psi \vdash H(\ell) : \Psi(\ell) \qquad FTV(\Psi(\ell)) = \varnothing}{\vdash H : \Psi} \quad \text{(S-HEAP)}$$

Machine States $\boxed{\vdash M}$

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I : \texttt{code}(\Gamma)}{\vdash (H, R, I)} \quad \text{(S-MACH)}$$

Figure 4-4: TAL-0 typing rules

The judgment $\Psi \vdash R : \Gamma$ asserts that the register file R is accurately described by Γ, under the assumptions of Ψ. Similarly, the judgment $\vdash H : \Psi$ asserts that the heap H is accurately described by Ψ. This is essentially the same rule as a "letrec" for declarations in a conventional functional language: We get to assume that the labels have their advertised type, and then check for any inconsistencies within their definitions. This allows labels to refer to one another directly. Note also that we require the types in Ψ to be closed so that generalization remains valid.

Finally, the judgment $\vdash (H, R, I)$ puts the pieces together: We must have some type assignment Ψ that describes the heap H, a register file typing Γ that describes R consistent with Ψ, and I must be a continuation with a precondition of Γ on the register file, under the assumptions of Ψ.

Some Examples and Subtleties

As a simple example of the type system in action, let us revisit the `prod` example:

```
prod: r3 := 0;           // res := 0
      jump loop

loop: if r1 jump done;   // if a = 0 goto done
      r3 := r2 + r3;     // res := res + b
      r1 := r1 + (-1);   // a := a - 1
      jump loop

done: jump r4            // return
```

Let Γ be the register file type:

$$\{\texttt{r1},\texttt{r2},\texttt{r3}:\texttt{int},\texttt{r4}:\forall\alpha.\texttt{code}\{\texttt{r1},\texttt{r2},\texttt{r3}:\texttt{int},\texttt{r4}:\alpha\}\}$$

Registers `r1`, `r2`, and `r3` are assigned the type `int`, and `r4` is assigned a polymorphic code type for reasons revealed below. Let Ψ be the label type assignment that maps `prod`, `loop`, and and `done` to $\texttt{code}(\Gamma)$. Let I be the instruction sequence associated with `loop` and let us verify that it is indeed well-formed with the type that we have assigned it.

We must show that $\Psi \vdash I : \texttt{code}(\Gamma)$. It suffices to show that each instruction preserves Γ, since the final `jump` is to loop, which expects Γ. For the first instruction, we must show $\Psi \vdash \texttt{if r1 jump done} : \Gamma \to \Gamma$ using the S-IF rule:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{r1} : \Gamma(\texttt{r1}) = \texttt{int}}\,\text{S-REG} \quad \dfrac{\dfrac{}{\Psi \vdash \texttt{done} : \Psi(\texttt{done}) = \texttt{code}(\Gamma)}\,\text{S-LAB}}{\Psi;\Gamma \vdash \texttt{done} : \texttt{code}(\Gamma)}\,\text{S-VAL}}{\Psi \vdash \texttt{if r1 jump done} : \Gamma \to \Gamma}$$

Next, we must show that adding `r2` to `r3` preserves Γ:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{r2} : \Gamma(\texttt{r2}) = \texttt{int}}\,\text{S-REG} \quad \dfrac{}{\Psi;\Gamma \vdash \texttt{r3} : \Gamma(\texttt{r3}) = \texttt{int}}\,\text{S-REG}}{\Psi \vdash \texttt{r3 := r2 + r3} : \Gamma \to \Gamma}$$

Then, we must show that subtracting 1 from `r1` preserves Γ:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{r1} : \Gamma(\texttt{r2}) = \texttt{int}}\,\text{S-REG} \quad \dfrac{\dfrac{}{\Psi \vdash -1 : \texttt{int}}\,\text{S-INT}}{\Psi;\Gamma \vdash -1 : \texttt{int}}\,\text{S-VAL}}{\Psi \vdash \texttt{r1 := r1 + (-1)} : \Gamma \to \Gamma}$$

Finally, we must show that the `jump` which terminates the sequence has type $\texttt{code}(\Gamma)$, using the S-JUMP rule:

$$\dfrac{\dfrac{\dfrac{}{\Psi;\Gamma \vdash \texttt{loop} : \Psi(\texttt{loop}) = \texttt{code}(\Gamma)}\,\text{S-LAB}}{\Psi;\Gamma \vdash \texttt{loop} : \texttt{code}(\Gamma)}\,\text{S-VAL}}{\Psi \vdash \texttt{jump loop} : \texttt{code}(\Gamma)}$$

Stringing the sub-proofs together using the S-SEQ rule, we can thus confirm that $\Psi \vdash I : \texttt{code}(\Gamma)$.

Carrying on, we can show that each label's code has the type associated with it. However, there is a major subtlety with the last `jump r4` instruction and the type that we have assigned `r4` throughout the code. To understand this, consider the following:

```
foo: r1 := bar;
     jump r1

bar: ...
```

What type can we assign to `bar`? Without the polymorphism, it must be a code type $\texttt{code}(\Gamma)$ such that $\Gamma(\texttt{r1}) = \texttt{code}(\Gamma)$, since the label `bar` will be in register `r1` when we jump to it. But with only simple types (i.e., no subtyping, polymorphism, or recursive types), there is no solution to this equation.

With our support for polymorphism, the problem can be averted. In particular, we can assign `bar` a polymorphic type τ of the form $\forall\alpha.\texttt{code}\{\texttt{r1} : \alpha, \ldots\}$. At the `jump` instruction, we have a register file context of the form $\Gamma = \{\texttt{r1} : \tau, \ldots\}$ and we must show that $\Gamma \vdash \texttt{r1} : \texttt{code}(\Gamma)$. Using S-INST, we can instantiate the type of `r1`, which is τ, with τ to derive $\Gamma \vdash \texttt{r1} : \texttt{code}\{\texttt{r1} : \tau, \ldots\}$.

This explains why we have used a polymorphic type for `r4` in the `prod` example above. Of course, this problem can be solved in other ways. Clearly, adding recursive types provides a solution to the problem. An alternative is to add some type *Top* which is greater than or equal to all other types, and use this to forget the type of a register as we jump through it. Yet another solution is to treat register file types as partial maps, and provide a form of subtyping that lets you forget the type of a register, thereby making it unusable as an operand until it is assigned a value. This last approach was the one used by the original TAL. We prefer the approach based on polymorphism, because there are many other compelling uses of this feature.

4.2.1 EXERCISE [⋆, ↛]: Draw the derivation of well-formedness for the instruction sequences associated with `prod` and `done`. □

For instance, polymorphism can also be used to achieve a type for "join-points" in a control-flow graph. Consider the situation where we have two jumps from distinct contexts to the same label:

```
{r1:int,...}
jump baz

...
{r1:code{...},...}
jump baz
```

What type should `baz` require of register `r1`? Again, without support for some form of polymorphism or subtyping, we would be forced to make the types the same, in which case this code would be rejected. The problem could be worked around by, for instance, always loading an integer into `r1` before jumping to this label. But that would slow down the program with unnecessary instructions. Fortunately, polymorphism saves us again. In particular, we can assign `baz` a type of the form $\forall\alpha.$`code{r1:`α`,...}`. Then, at the first `jump`, we would instantiate α to `int`, whereas at the second `jump`, we would instantiate α with the appropriate `code` type. Of course, the addition of *Top* would also provide a convenient mechanism for typing join points.

One other feature that polymorphism provides which cannot be captured through simple subtyping, is the idea of *callee-saves* registers. A callee-save register is a register whose value should remain the same across a procedure call. If the procedure wishes to use that register, then it is responsible for saving and restoring its value before returning to the caller. Of course, we don't yet have a way to save and restore registers to memory, but a procedure could shuffle values around into different registers.

Suppose we wish to call a procedure, such as `prod`, and guarantee that the register `r5` is preserved across the call. We can accomplish this by requiring the procedure's entry label to have a type of the form:

$\forall\alpha.${`r5:`α`,r4:`$\forall\beta.$`code{r5:`α`,r4:`β`,...},...}`

where `r4` is the register which is meant to hold the return address for the procedure, and "..." does not contain a free occurrence of α. Note that the return address's type specifies that `r5` must have the same type (α) upon return as was originally passed in. Furthermore, the procedure is required to treat `r5` uniformly since its type is abstract. Since there is no way to manufacture values of abstract type, and since we've only passed in one α value, it must be that if the procedure ever returns, then `r5` has the same value in it as it did upon entry (see Exercise 4.2.5.) Note that the procedure is free to move `r5`'s value into some other register, and to use `r5` to hold other values. But before it can return, it must restore the original value.

So, it is clear that polymorphism can play a unifying role in the design of type systems for low-level languages. It provides a way for us to conveniently "forget" types, which is necessary for jumps through registers and join points. But it also provides an ability to capture critical compiler invariants, such as callee-saves registers.

4.2.2 EXERCISE [RECOMMENDED, ⋆, ↛]: Suppose we change the `done` instruction sequence from `jump r4` to `jump r1`. Show that there is no way to prove the resulting code is well-typed. □

4.2.3 EXERCISE [RECOMMENDED, ⋆⋆⋆, ↛]: Reformulate the type system by eliminating type variables and universal polymorphism in favor of a *Top* type and subtyping. Then show how the product example can be typed under your rules. □

4.2.4 EXERCISE [⋆⋆⋆, ↛]: Reformulate the type system by using recursive types in lieu of polymorphism. Then show how the product example can be typed under your rules. □

4.2.5 EXERCISE [⋆⋆⋆⋆, ↛]: Prove that the approach to callee-saves registers actually preserves values. Hint: one relatively easy way to prove this is suggested by Crary (1999). Another possible solution is to adapt Reynolds' relational semantics for the polymorphic lambda calculus (1983) to the TAL setting. The result should follow as a "free theorem" (Wadler, 1989). □

Proof of Type Soundness for TAL-0

We now wish to show that the type system given in the previous section actually enforces our desired safety property. In particular, we wish to show that, given a well-typed machine state M, then M cannot get stuck (i.e., jump to an integer or undefined label.) It suffices to show that a well-typed machine state is not immediately stuck (progress), and that when it steps to a new machine state M', that state is also well-typed (preservation). For then, by induction on the length of an evaluation sequence, we can argue that there is no stuck M' such that $M \longrightarrow^* M'$.

Our first step is to establish a set of substitution lemmas which show that derivations remain possible after substituting types for type variables:

4.2.6 LEMMA [TYPE SUBSTITUTION]: If:

1. $\Psi;\Gamma \vdash v : \tau_1$, then $\Psi;\Gamma[\tau/\alpha] \vdash v : \tau_1[\tau/\alpha]$.
2. $\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$ then $\Psi \vdash \iota : \Gamma_1[\tau/\alpha] \to \Gamma_2[\tau/\alpha]$.

3. $\Psi \vdash I : \tau_1$, then $\Psi \vdash I : \tau_1[\tau/\alpha]$.

4. $\Psi \vdash R : \Gamma$, then $\Psi \vdash R : \Gamma[\tau/\alpha]$. □

The register substitution lemma ensures that typing is preserved when we look up a value in the register file. It corresponds to the value substitution lemma in a soundness proof for a conventional lambda calculus.

4.2.7 LEMMA [REGISTER SUBSTITUTION]: If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$ and $\Psi;\Gamma \vdash v : \tau$ then $\Psi;\Gamma \vdash \hat{R}(v) : \tau$ □

As usual, we shall need a Canonical Values lemma that tells us what kind of value we have from its type:

4.2.8 LEMMA [CANONICAL VALUES]: If $\vdash H : \Psi$ and $\Psi \vdash v : \tau$ then:

1. If $\tau = \texttt{int}$ then $v = n$ for some n.

2. If $\tau = \texttt{code}(\Gamma)$ then $v = \ell$ for some $\ell \in dom(H)$ and $\Psi \vdash H(\ell) : \texttt{code}(\Gamma)$. □

This extends to operands as follows:

4.2.9 LEMMA [CANONICAL OPERANDS]: If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$, and $\Psi;\Gamma \vdash v : \tau$ then:

1. If $\tau = \texttt{int}$ then $\hat{R}(v) = n$ for some n.

2. If $\tau = \texttt{code}(\Gamma)$ then $\hat{R}(v) = \ell$ for some $\ell \in dom(H)$ and $\Psi \vdash H(\ell) : \texttt{code}(\Gamma)$. □

4.2.10 THEOREM [SOUNDNESS OF TAL-0]: If $\vdash M$, then there exists an M' such that $M \longrightarrow M'$ and $\vdash M'$. □

Proof: Suppose $M = (H, R, I)$ and $\vdash M$. By inversion of the S-MACH rule, there exists a Ψ and Γ such that (a) $\vdash H : \Psi$, (b) $\Psi \vdash R : \Gamma$, and (c) $\Psi \vdash I : \texttt{code}(\Gamma)$. The proof proceeds by induction on I.

case $I = \texttt{jump}\ v$*:* From (c) and inversion of S-JUMP, we have $\Psi;\Gamma \vdash v : \texttt{code}(\Gamma)$. From the Canonical Operands lemma, we know that there exists an I' such that $H(\hat{R}(v)) = I'$ and $\Psi \vdash I' : \texttt{code}(\Gamma)$. Taking $M' = (H, R, I')$, we can show $M \longrightarrow M'$ via the JUMP rule. We must now show $\vdash (H, R, I')$, but this follows immediately.

case $I = r_d\texttt{:=}v; I'$*:* From inversion of the S-SEQ rule, we have $\Psi \vdash r_d\texttt{:=}v : \Gamma \rightarrow \Gamma_2$ and $\Psi \vdash I' : \texttt{code}(\Gamma_2)$ for some Γ_2. Then, by inversion of the S-MOV rule, we have $\Psi;\Gamma \vdash v : \tau$ and $\Gamma_2 = \Gamma[r_d : \tau]$ for some τ. By the Register Substitution lemma, we have $\Psi;\Gamma \vdash \hat{R}(v) : \tau$. Taking $M' = (H, R[r_d = \hat{R}(v)], I')$, we see that $M \longrightarrow M'$ via the MOV rule. From the S-REGFILE rule, we conclude that $\Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d : \tau]$.

case $I = r_d := r_s + v; I'$*:* From inversion of S-SEQ, we have $\Psi \vdash r_d := r_s + v : \Gamma \to \Gamma_2$ and $\Psi \vdash I' : \mathtt{code}(\Gamma_2)$ for some Γ_2. Then, by inversion of the S-ADD rule, we have $\Psi;\Gamma \vdash r_s : \mathtt{int}$, $\Psi;\Gamma \vdash v : \mathtt{int}$ and $\Gamma_2 = \Gamma[r_d : \mathtt{int}]$. From the and Canonical Operand lemma, we know that there exists integers n_1 and n_2 such that $R(r_s) = n_1$ and $\hat{R}(v) = n_2$. Taking $n = n_1 + n_2$ and $M' = (H, R[r_d = n], I')$, we see that $M \longrightarrow M'$ via the ADD rule. By the S-INT and S-VAL rules, $\Psi;\Gamma[r_d : int] \vdash n : int$. Thus, by the S-REGFILE rule, we conclude that $\Psi \vdash R[r_d = n] : \Gamma[r_d : \mathtt{int}]$.

case $I = \mathtt{if}\ r_s\ \mathtt{jump}\ v; I'$*:* From inversion of the S-SEQ rule, we have $\Psi \vdash \mathtt{if}\ r_s\ \mathtt{jump}\ v : \Gamma \to \Gamma_2$ and $\Psi \vdash I' : \mathtt{code}(\Gamma_2)$ for some Γ_2. Then, by inversion of the S-IF rule, we have $\Psi;\Gamma \vdash r_s : \mathtt{int}$, $\Psi;\Gamma \vdash v : \mathtt{code}(\Gamma)$ and $\Gamma_2 = \Gamma$. By the Canonical Operands lemma, there exists an ℓ and I_2 such that $\hat{R}(v) = \ell$, $H(\ell) = I_2$, and $\Psi \vdash I_2 : \mathtt{code}(\Gamma)$. Also by Canonical Operands, $R(r_s) = n$ for some integer n. If $n = 0$ then $M \longrightarrow (M, R, I_2)$ via IF-EQ. If $n \neq 0$ then $M \longrightarrow (M, R, I')$ via IF-NEQ. In either case, the well-formedness of the resulting machine state follows from the S-MACH rule. □

Proof Representation and Checking

It is not clear whether type inference for TAL-0 machine states is decidable. That is, given a machine state (H, R, I), does there exist a Ψ and Γ such that $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$, and $\Psi \vdash : \mathtt{code}(\Gamma)$? On the one hand, this seems possible since the type system is so simple. On the other hand, the system, as presented, supports polymorphic recursion for which inference is known to be undecidable in the context of the lambda calculus. Furthermore, as we progress to more advanced typing features, the decidability of type reconstruction will surely vanish. Thus, in any practical realization, we must require some help for constructing a proof that the code is indeed type-correct.

In the case of TAL-0, it is sufficient to provide types for the labels (i.e., Ψ). Indeed, it is even possible to omit types for some labels and keep reconstruction decidable. We really only need enough type information to cut each loop in the control-flow graph, or for those labels that are moved into registers. Minimizing the type information is an important goal in any practical system, since the size of the types can often be larger than the code itself! However, it is desirable to keep the type checker as simple as possible so that we can trust it is properly enforcing the type system.

One way to keep the type checker simple is to modify the syntax so that type reconstruction is entirely syntax directed. By this, we mean simply that for any given term, at most one rule should apply. Furthermore, the checker should not have to "guess" any of the sub-goal components. For TAL-0, this could be accomplished by (a) requiring types on all labels and (b) adding a form of explicit type instantiation to operands (e.g., $v[\tau]$).

An alternative approach is to force the code provider to ship an explicit representation of the complete proof of well-formedness, along with the code, and make sure that the proof and the code have the same instructions, labels, etc. Of course, these proof will tend to be much larger than the code itself, but we can use various techniques to reduce the size of the proof representation (see for instance Necula and Lee, 1998b).

Such a separation of proofs and code is advantageous because we can ship the *binary* machine code (as opposed to the assembly code), disassemble it, and then compare it against the assembly-level proof. If everything checks out, then we can load the binary and execute it directly. Such an approach is called *proof-carrying code* (Necula, 1997, 1998) and was first used by Necula's Touchstone compiler (Necula and Lee, 1998a), and the Special-J compiler (Colby et al., 2000), both of which are described more fully in the next chapter.

Indeed, we could even go so far as to pass along the proof of soundness for the entire type system, and a proof that the abstract machine is faithful to the concrete machine's semantics! This would ensure that the code consumer has to trust nothing but (a) the formalization of the concrete machine semantics, and (b) the proof checker. This is the approach proposed by Appel and Felty and is called *foundational proof-carrying code* (2000).

In the rest of this chapter, we will remain vague about how proofs are to be represented. The key thing to note is that we are not limited in the choice of type constructors by issues of inference. Rather, we will require that the code producer provide us with enough evidence that we can easily reconstruct and check the proof of well-formedness. Therefore, our only limitation will be the incompletenesses of the resulting proof system.

4.2.11 EXERCISE [★★★★, ↛]: Build a type-checker for TAL-0 in your favorite programming language. Assume that you are given as input a set of labels, their associated types, and instruction sequences. Furthermore, assume that operands are augmented with explicit syntax for polymorphic instantiation. □

4.3 TAL-1: Simple Memory-Safety

TAL-0 includes registers and heap-allocated code, but provides no support for allocated *data*. In this section, we will add primitive support for allocated objects that can be shared by reference (i.e., pointer) and extend our safety property to include a notion of object-level memory safety: No memory access should read or write a data object at a given location unless the program has been granted access to that location.

From a typing perspective, the critical issue will be how to accommodate locations that hold values of different types at different times during the execution of the program. We need such a facility to at least support the construction of compound values, such as tuples, records, datatype constructors, or objects. A high-level language, such as ML, provides mechanisms to allocate and initialize data structures as a single expression. For instance, `{x = 3, y = 4}` is an expression that builds a record with two components. At the assembly level, such high-level compound expressions must be broken into machine-level steps. We must first allocate space for the object, and then initialize the components by storing them in that space. To prevent someone from treating an uninitialized component as if it holds a valid value, we must use a different type. But obviously, once we initialize that component, its type should change to reflect that it is now valid for use.

Already, we have support for storing values of different types in registers. For instance, nothing prevents us from moving a `code` value into a register currently holdingan `int`. However, when we add allocated data objects, we can no longer track the changes easily due to *aliasing*. Let `ptr(`τ`)` denote a pointer to a data object of type τ and consider this sequence of instructions:

```
      {r1:ptr(code(...))}
1.    r3 := 0;
2.    Mem[r1] := r3;
3.    r4 := Mem[r1];
4.    jump r4
```

We assume upon entry that `r1` is a pointer to a data location that contains a code label. The first two instructions overwrite the contents of memory at the location in `r1` with the integer 0. The third instruction loads the value from the location in `r1` and jumps to it. Clearly, this code should be rejected by the type-checker as it violates our control-flow safety property. To ensure this, we might require that the type system update the type of `r1` whenever we store through it. For instance, after the second instruction, the type of `r1` would change from `ptr(code(...))` to `ptr(int)`. Then at instruction four, the code would be rejected because of an attempt to jump to an integer.

Now consider this sequence:

```
      {r1:ptr(code(...)),r2:ptr(code(...))}
1.    r3 := 0;
2.    Mem[r1] := r3;
3.    r4 := Mem[r2];
4.    jump r4
```

The code is exactly the same except that instead of loading the value pointed to by `r1`, we load the value pointed to by `r2`. Should this code type-check?

The answer depends on whether or not `r1` and `r2` hold the same value—that is, whether or not they are aliases for the same location. There is no problem when they are not aliases, but when they are, the code behaves the same as in the previous example (i.e., attempts to jump to the integer 0.) It becomes clear that to prevent this problem, whenever we update a memory location with a value of a different type, we must update the types of *all* aliases to that location. To do so, the type system must track whether or not two values are the same and, more generally, whether or not two code labels behave the same.

Of course, there is no complete logic for tracking the equalities of values and computations, but it is possible to construct a powerful type system that allows us to conservatively track equality of *some* values. For instance see the work on *alias types* (Smith, Walker, and Morrisett, 2000; Walker and Morrisett, 2001; DeLine and Fähndrich, 2001). But all of these systems are, in my opinion, technically daunting. Furthermore, certifying compilers for high-level languages rarely need such complex machinery.

Nonetheless, we need some support for (a) allocating and initializing data structures that are to be shared, and (b) stack-allocating procedure frames. Therefore, we will focus on typing principles that try to strike a balance between expressiveness and complexity. After all, our goal is to provide a simple but expressive structure for implementing type-safe, high-level languages on conventional architectures.

In particular, we will use the type system to separate locations into one of two classes: The first class, called *shared pointers*, will support arbitrary aliasing. However, the types of the contents of shared pointers must remain invariant. That is, we can never write a value of a different type into the contents of a shared location. This is the same basic principle that ML-style `ref`s and other high-level languages follow.

The second class of locations, called *unique pointers*, will support updates that change the type of the contents. However, unique pointers cannot be aliased. In particular, we will prevent unique pointers from being copied. Thus, they will behave much the same way as registers.

The combination of unique and shared pointers will provide us with a simple, but relatively flexible framework for dealing with memory. In particular, we will be able to use unique pointers to handle the thorny problem of allocating and initializing shared data structures. We will also be able to use unique pointers to model data structures whose lifetime is controlled by the compiler, such as stack frames.

r ::=		*registers:*	
	r1 | r2 | $\cdots$ | rk	*gp registers*	
	sp	*stack pointer*	
ι ::=		*instructions:*	
	$\cdots$	*as in TAL-0*	
	r_d := Mem[$r_s + n$]	*load from memory*	
	Mem[$r_d + n$] := r_s	*store to memory*	
	r_d := malloc n	*allocate n heap words*	
	commit r_d	*become shared*	
	salloc n	*allocate n stack words*	
	sfree n	*free n stack words*	
v ::=		*operands:*	
	r	*registers*	
	n	*integer literals*	
	ℓ	*code or shared data pointers*	
	uptr(h)	*unique data pointers*	
h ::=		*heap values:*	
	I	*instruction sequences*	
	$\langle v_1, \ldots, v_n \rangle$	*tuples*	

Figure 4-5: TAL-1 syntax additions

The TAL-1 Extended Abstract Machine

Figure 4-5 gives a set of syntactic extensions to TAL-0 which are used in the definition of TAL-1. We have added six new instructions: Two of the instructions can be used to load a value from memory into a register, or to store a register's value to memory respectively. The effective address for both instructions is calculated as a word-level offset from a base register. The other instructions are non-standard. The `malloc` instruction is used to allocate an object with n words. A (unique) reference to the object is placed in the destination register. Typically, `malloc` will be implemented by the concrete machine using a small sequence of inlined instructions or a procedure call. We abstract from these details here so that our abstract machine can support a wide variety of allocation techniques. The `commit` instruction is used to coerce a unique pointer to a shared pointer. It has no real run-time effect but it makes it easier to state and prove the invariants of the type system.

The `salloc` and `sfree` constructs manipulate a special unique pointer which is held in a distinguished register called `sp` (stack pointer). The instruction `salloc` attempts to grow the stack by n words, whereas `sfree` shrinks the stack by n words. The type system will prevent the stack from underflowing, so in principle, `sfree` could be implemented by a simple arithmetic operation (e.g., `sp := sp` + n.) Unfortunately, stack overflow will not be captured by this type system. Therefore, we assume that the `salloc` instruction checks for overflow and aborts the computation somehow.

As before, we will model machine states using a triple of a heap, register file, and instruction sequence. And, as before, register files will map registers to word-sized values, while heaps will map labels to heap-values. We extend

heap values to include tuples of word-sized values. Thus, a label can refer to either code or data. We could also use the heap to store unique data values, but this would make it more difficult to prove that the pointers to these values are indeed unique. Instead, we will extend operands with terms of the form $\texttt{uptr}(h)$ and use such a term to represent a unique pointer to a heap value h.

The rewriting rules for the instructions of TAL-1 that overlap with TAL-0 remain largely the same. However, we must prevent unique pointers from being copied. More precisely, we must prevent the situation where we have two references to the same unique data. Note that, for the addition and `if` instructions, the use of a unique pointer as an operand will cause the machine to get stuck since the operands must be integers to make progress. However, the typing for assignment ($r\texttt{:=}v$) must be changed to prevent copies of unique pointers:

$$\frac{\hat{R}(v) \neq \texttt{uptr}(h)}{(H, R, r_d\texttt{:=}v; I) \longrightarrow (H, R[r_d = v], I)} \quad \text{(MOV-1)}$$

This rule can only fire when the source operand is not a unique pointer.

We must now give the rewriting rules for the new instructions:

$$(H, R, r_d\texttt{:=}\,\texttt{malloc}\ n; I) \longrightarrow (H, R[r_d = \texttt{uptr}\langle m_1, \ldots, m_n\rangle], I) \quad \text{(MALLOC)}$$

$$\frac{r_d \neq \texttt{sp} \qquad \ell \notin dom(H)}{(H, R[r_d = \texttt{uptr}(h)], \texttt{commit}\ r_d; I) \longrightarrow (H[\ell = h], R[r_d = \ell], I)} \quad \text{(COMMIT)}$$

$$\frac{R(r_s) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle}{(H, R, r_d\texttt{:=}\,\texttt{Mem}[r_s + n]; I) \longrightarrow (H, R[r_d = v_n], I)} \quad \text{(LD-S)}$$

$$\frac{R(r_s) = \texttt{uptr}\langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle}{(H, R, r_d\texttt{:=}\,\texttt{Mem}[r_s + n]; I) \longrightarrow (H, R[r_d = v_n], I)} \quad \text{(LD-U)}$$

$$\frac{R(r_d) = \ell \qquad H(\ell) = \langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle \qquad R(r_s) = v \qquad v \neq \texttt{uptr}(h)}{(H, R, \texttt{Mem}[r_d + n]\ \texttt{:=}\ r_s; I) \longrightarrow (H[\ell = \langle v_0, \ldots, v, \ldots, v_{n+m}\rangle], R, I)} \quad \text{(ST-S)}$$

$$\frac{R(r_d) = \texttt{uptr}\langle v_0, \ldots, v_n, \ldots, v_{n+m}\rangle, \qquad R(r_s) = v \qquad v \neq \texttt{uptr}(h)}{(H, R, \texttt{Mem}[r_d + n]\ \texttt{:=}\ r_s; I) \longrightarrow (H, R[r_d = \texttt{uptr}\langle v_0, \ldots, v, \ldots, v_{n+m}\rangle], I)} \quad \text{(ST-U)}$$

$$\frac{R(\texttt{sp}) = \texttt{uptr}\langle v_0, \ldots, v_p\rangle \qquad p + n \leq \texttt{MAXSTACK}}{(H, R, \texttt{salloc}\ n) \longrightarrow (H, R[\texttt{sp} = \texttt{uptr}\langle m_1, \ldots, m_n, v_0, \ldots, v_p\rangle])} \quad \text{(SALLOC)}$$

$$\frac{R(\texttt{sp}) = \texttt{uptr}\langle m_1, \ldots, m_n, v_0, \ldots, v_p\rangle}{(H, R, \texttt{sfree}\ n) \longrightarrow (H, R[\texttt{sp} = \texttt{uptr}\langle v_0, \ldots, v_p\rangle])} \quad \text{(SFREE)}$$

The `malloc` instruction places a unique pointer to a tuple of n words in the destination register. We assume that the memory management subsystem

has initialized the tuple with some arbitrary integer values $m_1, \ldots, m_n$. Recall that the rewriting rules prevent these unique pointers from being copied. However, the `commit` instruction allows us to move a unique pointer into the heap where it can be shared. As we will see, however, shared pointers must have invariant types, whereas unique pointers' types can change.

The memory-load instruction has two variants, depending upon whether the source register holds a value that is shared or unique. If it is shared, then we must look up the binding in the heap to get the heap value. If it is unique, then the heap value is immediately available. Then, in both cases, the heap value should be a tuple. We extract the n^{th} word and place it in the destination register.

The memory-store instruction is the dual and is used to update the n^{th} component of a tuple. Note that the machine gets stuck on an attempt to store a unique pointer, thereby preventing copies from leaking into a data object.

As a simple example of the use of these constructs, consider the following code:

```
copy: {r1:ptr(int,int), r2,r3:int}
      r2 := malloc 2;
      r3 := Mem[r1];
      Mem[r2] := r3;
      r3 := Mem[r1+1];
      Mem[r2+1] := r3;
      commit r2;
      {r1:ptr(int,int), r2:ptr(int,int), r3:int }
```

The code is meant to do a deep copy of the data structure pointed to by `r1` and place the copy in register `r2`. Suppose that $r1$ holds a label ℓ_1 and $H(\ell_1) = \langle 3, 5\rangle$. After executing the malloc instruction, `r2` will hold a unique pointer to a pair of (arbitrary) integers of the form $\texttt{uptr}\langle m_1, m_2\rangle$. After the load and store, the first integer component of `r1` will have been copied into the first component of `r2`. Thus, the contents of ℓ_2 will have changed to $\texttt{uptr}\langle 3, m_2\rangle$. After the second load and store, the second component will have been copied, so `r2` will hold the value $\texttt{uptr}\langle 3, 5\rangle$. Finally, after the `commit` instruction, `r2` will hold a fresh, shared label ℓ_2 and the heap will have been extended so that it maps ℓ_2 to the heap value $\langle 3, 5\rangle$.

Here is an example program which uses `salloc` and `sfree`:

```
foo: {sp : uptr(int), r1 : code{...}}
      salloc 2;      // {sp : uptr(int,int,int)}
      Mem[sp] := r1; // {sp : uptr(code{...},int,int)}
      sfree 1        // {sp : uptr(int,int)}
```

τ	::=	*operand types:*	σ	::=	*allocated types:*
	$\cdots$	*as in TAL-0*		ϵ	*empty*
	$\texttt{ptr}(\sigma)$	*shared data pointers*		τ	*value type*
	$\texttt{uptr}(\sigma)$	*unique data pointers*		σ_1, σ_2	*adjacent*
	$\forall \rho.\tau$	*quantification over allocated types*		ρ	*allocated type variable*

Figure 4-6: TAL-1 types

On input, the stack has one integer element and `r1` has a code pointer. The first instruction grows the stack by two words. The second instruction stores the value in `r1` into the top of the stack. The third instruction frees one of the words. Note that `salloc` becomes stuck if we attempt to allocate more than `MAXSTACK` (total) words and that `sfree` becomes stuck if we attempt to shrink the stack by more words than are on the stack. Finally, note that our stacks grow "up" (indexing is positive) whereas the common convention is to have stacks that grow down. The only reason for this choice is that it unifies unique pointers to tuples with stacks. If we wanted to support downward stacks, then we could introduce a new kind of data structure (e.g., `stptr`.)

4.3.1 EXERCISE [RECOMMENDED, ⋆, ↛]: Show how stack-push and stack-pop instructions can be explained using the primitives provided by TAL-1. Explain how a sequence of pushes or a sequence of pops can be optimized. □

4.3.2 EXERCISE [⋆⋆⋆, ↛]: Modify the abstract machine so that unique pointers are allocated in the heap, just like shared pointers, but are represented as a tagged value of the form $\texttt{uptr}(\ell)$. Then show that the machine maintains the invariant that there is at most one copy of a unique pointer. □

4.4 TAL-1 Changes to the Type System

What changes and additions are needed to the type system to ensure that the new abstract machine won't get stuck? In particular, how do we ensure that when we do a load, the source register contains a data pointer (as opposed to an integer or code label), and the data pointer points to a heap value that has at least as many components as the offset requires? Similarly, how do we ensure that for a store, the destination register is a data pointer to a large enough heap value? And how do we ensure that the stack does not underflow? How do we ensure that we don't try to copy a unique pointer? In short, how do we ensure progress?

Heap Values $\boxed{\Psi \vdash v : \tau}$

$$\frac{\Psi;\Gamma \vdash v_i : \tau_i}{\Psi \vdash \langle v_1, \ldots, v_n \rangle : \tau_1, \ldots, \tau_n} \quad \text{(S-TUPLE)}$$

Operands $\boxed{\Psi \vdash v : \tau}$

$$\frac{\Psi;\Gamma \vdash h : \sigma}{\Psi;\Gamma \vdash \texttt{uptr}(h) : \texttt{uptr}(\sigma)} \quad \text{(S-UPTR)}$$

Figure 4-7: TAL-1 typing rules (heap values and operands)

Figure 4-6 gives a new set of types for classifying TAL-1 values. The τ types are used to classify values and operands, whereas the σ types are used to classify heap-allocated data. We have added three new operand types corresponding to shared pointers ($\texttt{ptr}(\sigma)$), unique pointers ($\texttt{uptr}(\sigma)$), and polymorphism over allocated types ($\forall\rho.\tau$.)

The allocated types (σ) consist of sequences of operand types. The syntax supports nesting structure (i.e., trees) but we implicitly treat adjacency as associative with ϵ as a unit. So, for instance:

$$\texttt{ptr}(\texttt{int}, (\rho, (\texttt{int}, \epsilon))) = \texttt{ptr}((\texttt{int}, \rho), \texttt{int})$$

Allocated types also support variables (ρ) which are useful for abstracting a chunk of memory. That is, α can be used to abstract a single word-sized type, whereas ρ can be used to abstract a type of arbitrary size. As we will see, polymorphism over allocated types is the key to efficient support for procedures.

Figures 4-7 and 4-8 give the new typing rules. As in TAL-0, we take Γ to be a total map from registers to operand types. We are also assuming that Ψ is a finite partial function from labels to allocated operand types (i.e., `code` or `ptr` types.)

The well-formedness rules for tuples and unique pointers are straightforward. The S-MOV-1 rule defines the new type for the move instruction. It has as a pre-condition that the value being moved should not be a unique pointer.

The typing rule for `malloc` requires a non-negative integer argument, and updates the destination register's type with a unique pointer of an n-tuple of integers. The `commit` instruction expects a unique pointer in the given register, and simply changes its type to a shared pointer.

The load and store instructions require two rules each, depending upon whether they are operating on unique pointers. Notice that for the store rules, we are not allowed to place a unique pointer into the data structure. Notice also that that when storing into a unique pointer, there is no requirement that the new value have the same type as the old value. In contrast, for shared pointers, the old and new values must have the same type.

Instructions $\boxed{\Psi \vdash \iota : \Gamma_1 \to \Gamma_2}$

$$\frac{\Psi;\Gamma \vdash v : \tau \qquad \tau \neq \mathtt{uptr}(\sigma)}{\Psi \vdash r_d := v : \Gamma \to \Gamma[r_d : \tau]} \quad \text{(S-MOV-1)}$$

$$\frac{n \geq 0}{\Psi \vdash r_d := \mathtt{malloc}\ n : \Gamma \to \Gamma[r_d : \mathtt{uptr}\langle \underbrace{\mathtt{int}, \ldots, \mathtt{int}}_{n} \rangle]} \quad \text{(S-MALLOC)}$$

$$\frac{\Psi;\Gamma \vdash r_d : \mathtt{uptr}(\sigma) \qquad r_d \neq \mathtt{sp}}{\Psi \vdash \mathtt{commit}\ r_d : \Gamma \to \Gamma[r_d : \mathtt{ptr}(\sigma)]} \quad \text{(S-COMMIT)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \mathtt{ptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash r_d := \mathtt{Mem}[r_s + n] : \Gamma \to \Gamma[r_d : \tau_n]} \quad \text{(S-LDS)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \mathtt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash r_d := \mathtt{Mem}[r_s + n] : \Gamma \to \Gamma[r_d : \tau_n]} \quad \text{(S-LDU)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \tau_n \qquad \tau_n \neq \mathtt{uptr}(\sigma') \qquad \Psi;\Gamma \vdash r_d : \mathtt{ptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{Mem}[r_d + n] := r_s : \Gamma \to \Gamma} \quad \text{(S-STS)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \tau \qquad \tau \neq \mathtt{uptr}(\sigma') \qquad \Psi;\Gamma \vdash r_d : \mathtt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{Mem}[r_d + n] := r_s : \Gamma \to \Gamma[r_d : \mathtt{uptr}(\tau_1, \ldots, \tau, \sigma)]} \quad \text{(S-STU)}$$

$$\frac{\Psi;\Gamma \vdash \mathtt{sp} : \mathtt{uptr}(\sigma) \qquad n \geq 0}{\Psi \vdash \mathtt{salloc}\ n : \Gamma \to \Gamma[\mathtt{sp} : \mathtt{uptr}(\underbrace{\mathtt{int}, \ldots, \mathtt{int}}_{n}, \sigma)]} \quad \text{(S-SALLOC)}$$

$$\frac{\Psi;\Gamma \vdash \mathtt{sp} : \mathtt{uptr}(\tau_1, \ldots, \tau_n, \sigma)}{\Psi \vdash \mathtt{sfree}\ n : \Gamma \to \Gamma[\mathtt{sp} : \mathtt{uptr}(\sigma)]} \quad \text{(S-SFREE)}$$

Figure 4-8: TAL-1 typing rules (instructions)

The rules for `salloc` and `sfree` are straightforward. For `sfree` n we check that there are at least n *values* on the stack to avoid underflow. Note that the rule does not allow allocated type variables (ρ) to be eliminated, reflecting the fact that, in general, we do not know how many words are occupied by ρ. For instance, ρ could be instantiated with ϵ in which case there are no values. A similar restriction holds for loads and stores—we must at least know the sizes up through the word component we are projecting or storing.

To prevent the machine from becoming stuck, `salloc` would ideally check that adding n words to the stack would not exceed `MAXSTACK`. But alas, we cannot always determine the current length of the stack. In particular, if its type contains an allocated variable ρ, then we are in trouble. One way around this problem is to change the abstract machine so that it does not get stuck upon overflow by defining a transition (e.g., by jumping to a pre-defined label). This would correspond to a machine trap due to an illegal access.

It is possible to extend our soundness proof for TAL-0 to cover the new constructs in TAL-1 and show that a well-formed machine state cannot get stuck, except when the stack overflows. In the proof of soundness, one critical property we must show is that any typing derivation remains valid under an extension of the heap. In particular, if $\vdash H : \Psi$ and $\Psi \vdash h : \tau$, then $\vdash H[\ell = h] : \Psi[\ell : \tau]$. Another critical property is that we would have to show that a given label ℓ that occurs in the heap has exactly one type throughout the execution of the program. In other words, once we have committed a pointer so that it can be shared, its type must remain invariant.

4.4.1 EXERCISE [★★★★, ↛]: Extend the proof of soundness to cover the new features in TAL-1. □

4.5 Compiling to TAL-1

At this point, TAL-1 provides enough mechanism that we can use it as a target language for the compiler of a polymorphic, procedural language with integers, tuples, records, and function pointers (but not yet lexically nested closures.) As a simple example, let us start with the following C code:

```
int prod(int x, int y) {
  int a = 0;
  while (x != 0) {
    a = a + y;
    x = x - 1;
  }
  return a;
}

int fact(int z) {
  if (z != 0) return prod(fact(z-1),z);
  else return 1;
}
```

and show how it may be compiled to TAL-1, complete with typing annotations on the code labels. We assume a calling convention where arguments are

passed on the stack, and the return address is passed in r4. We also assume that results are returned in register r1, and arguments are popped off the stack by the callee. Finally, we assume that registers r2 and r3 are freely available as scratch registers.

Let us first translate the prod function under these conventions:

```
prod: ∀a,b,c,s.
       code{r1:a,r2:b,r3:c,sp:uptr(int,int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      r2 := Mem[sp];    // r2:int, r2 := x
      r3 := Mem[sp+1];  // r3:int, r3 := y
      r1 := 0           // r1:int, a := 0
      jump loop

loop: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
              r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      if r2 jump done;  // if x ↔ 0 goto done
      r1 := r1 + r3;    // a := a + y
      r2 := r2 + (-1);  // x := x - 1
      jump loop

done: ∀s.code{r1,r2,r3:int,sp:uptr(int,int,s),
              r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      sfree 2;         // sp:uptr(s)
      jump r4
```

The code itself is rather straightforward. What is most interesting is the types we have placed on the labels. Note that, upon input to prod, r1, r2, and r3 can have any types, since we have abstracted their types. Note also that the stack pointer has type uptr(int,int,s) and thus has two integers at the front, but can have any sequence of values following, since we have abstracted the tail with an allocated type variable s. The return address in r4 is polymorphic for r2 and r3 to allow values of any type in those registers upon return. As discussed earlier, the type of r4 is abstracted by the return address to allow jumping through that register. Furthermore, the return address demands that the stack have type uptr(s), reflecting that the callee should pop the arguments before jumping to the return address. Thus, the contents of the rest of the stack is guaranteed to be preserved since s is abstract.

In general, a source-level procedure that takes arguments of types $\tau_1, \ldots, \tau_n$ and returns a value of type τ would translate to a label with the same type as prod's, except that the stack pointer would have type uptr($\tau_1, \ldots, \tau_n, s$), and r4's return code type would expect r1 to have type τ.

The types for the loop and done labels are similar to prod's, except that registers r1,r2, and r3 must hold integers. The reader is encouraged to check that the resulting code is well-formed according to the typing rules for TAL-1.

Now let us translate the recursive factorial procedure using the same calling conventions:

```
fact: ∀a,b,c,s.
       code{r1:a,r2:b,r3:c,sp:uptr(int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      r1 := Mem[sp];    // r1:int, r1 := z
      if r1 jump retn   // if z = 0 goto retn
      r2 := r1 + (-1);  // r2:int, r2 := z-1
      salloc 2          // sp:uptr(int,int,int,s)
      Mem[sp+1] := r4;  // sp:uptr(int,(∀d,e,f.code{...}),int,s)
      Mem[sp] := r2;    // sp:uptr(int,(∀d,e,f.code{...}),int,s)
      r4 := cont;
      jump fact         // r1 := fact(z-1)

cont: ∀c,s',d,e,f.
       code{r1:int,r2:d,r3:e,r4:f,
            sp:uptr(∀d,e,f.code{...},int,s')}
      r4 := Mem[sp];   // restore original return address
      Mem[sp] := r1;   // sp:uptr(int,int,s')
      jump prod        // tail call prod(fact(z-1),z)

retn: ∀b,c,s.
       code{r1:int,r2:b,r3:c,sp:uptr(int,s),
            r4:∀d,e,f.code{r1:int,r2:d,r3:e,r4:f,sp:uptr(s)}}
      r1 := 1;
      sfree 1;       // sp:uptr(s)
      jump  r4       // return 1
```

The first couple of instructions load the argument from the stack, test if it is zero, and if so, jump to the `retn` block where the argument is popped off the stack and the value 1 is returned. If the argument `z` is non-zero, then we must calculate `z-1`, pass it in a recursive call to `fact`, and then pass the result along with `z` to the `prod` function.

To do the recursive call, we must allocate stack space for the return address (`r4`) and the argument `z-1`, and save those values on the stack. Then we must load a new return address into `r4`, namely `cont` and finally jump to `fact`. When the recursive call returns, control will transfer to the `cont` label. Notice that `cont` expects the stack to hold the original return address and the value of `z`. We restore the original return address by loading it from the stack. We then overwrite the same stack slot with the result of `fact(z-1)`. Finally, we do a tail-call to `prod`.

It is important to recognize that the calling conventions we chose are not specific to the abstract machine. For instance, we could have chosen a conven-

tion where the return address is pushed on the stack, or where arguments are passed in registers, introduced callee-saves registers, etc. In contrast, virtual machines languages such as the JVML and CLI bake the notion of procedure and procedure call into the language. To add support for different calling conventions (e.g., tail-calls, or a tailored convention for leaf procedures) requires additions and changes to the abstract machine and its type system. In contrast, by focusing on a more primitive set of type constructors (e.g., $\forall$, `code`, and `uptr` types), we are able to accommodate many conventions without change to the type system.

4.5.1 EXERCISE [RECOMMENDED, ⋆⋆, ↛]: Rewrite the `fact` procedure with a calling convention where the arguments and return address are placed on the stack. Include typing annotations on the labels and convince yourself that the code type-checks. □

4.6 Scaling to Other Language Features

TAL-1 only supports simple tuple or record-like data structures. Thus, it is insufficient for compiling real-world high-level languages which provide data abstraction mechanisms such as closures, algebraic datatypes, objects, and/or arrays.

Simple Objects and Closures

Support for closures and simple forms of objects is readily accommodated by adding *existential* abstraction for both operand and allocated types:

$$\tau \quad ::= \quad \ldots \mid \exists\alpha.\tau \mid \exists\rho.\tau$$

The rules for introducing and eliminating existentials on operands are extremely simple:

$$\frac{\Psi;\Gamma \vdash v : \tau[\tau'/\alpha]}{\Psi;\Gamma \vdash v : \exists\alpha.\tau} \qquad \text{(S-PACK)}$$

$$\frac{\Psi;\Gamma \vdash v : \exists\alpha.\tau \qquad \alpha \notin FTV(\Gamma)}{\Psi;\Gamma \vdash v : \tau} \qquad \text{(S-UNPACK)}$$

(There are two similar rules for existentials that abstract allocated types.) The first rule allows us to abstract a common type for some components of a data structure. For instance, if r has type

```
ptr(code{r1:int,r2:int,...},int)
```

then we can use the S-PACK rule to treat the value as if it has type

$$\exists\alpha.\texttt{ptr(code\{r1:}\alpha\texttt{,r2:int,...\},}\alpha\texttt{)}.$$

Now, such a value can only be used by eliminating the existential using the S-UNPACK rule. However, we are required to continue treating α as abstract and distinct from any other type that may be in our context.

As suggested by Pierce and Turner (1993), we can use existentials to encode very simple forms of objects. For example, consider an object interface that looks like this, written in a Java-style:

```
interface Point {
  int getX();
  int getY();
}
```

and consider two classes that implement this interface:

```
class C1 implements Point {
  int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
}

class C2 implements Point {
  int x = 0, y = 0 , n = 0;
  int getX() { n++; return x; };
  int getY() { n++; return y; }
}
```

We can think of objects as pairs of a method table and an instance variable frame. The methods take the instance variable frame as an implicit "self" argument. For instance, the C1 class would have an instance frame that holds two integers, whereas the C2 class would have an instance frame that holds three integers. Thus, at an intermediate language level, C1's get operations would have type:

$$\texttt{ptr(int,int)} \to \texttt{int}$$

while C2's operations would have type:

$$\texttt{ptr(int,int,int)} \to \texttt{int}$$

When we build a C1 object or a C2 object, we need to hide the type of the instance frame so that only the methods can gain access to and manipulate the

values of the instance variables. We also need to hide the type of the instance frames so that we can give the objects a common type. We can achieve this by using an existential to abstract the type of the instance frame and pairing the methods with the instance frame. At an intermediate level, the type of a `Point` object would thus be something like:

$$\exists\alpha.\texttt{ptr}(\texttt{ptr}(\alpha \to \texttt{int}, \alpha \to \texttt{int}), \alpha)$$

Note that for any value with this type, we cannot directly access the instance variables because their type is abstract (i.e., α). However, once such an object is unpacked, we can project out a method and the instance frame, and pass the frame to the method because the methods expect an α value as an argument.

Closures are simple forms of objects where the instance frame holds the environment, and there is a single method for applying the closure to its arguments. Thus, with the simple addition of existential types, we have the ability to encode the primary features of modern languages, notably closures and objects. Indeed, the original TAL paper (Morrisett, Walker, Crary, and Glew, 1999) showed how a polymorphic, core functional language could be mapped to a version of typed assembly with support for existentials. This translation was based on previous work of Minamide, Morrisett, and Harper (1996).

Of course, in a more realistic implementation, we might represent objects without the level of indirection on instance variables, and instead of passing only the instance variables to methods, we could pass the whole object to a method. With the addition of recursive types ($\mu\alpha.\tau$) and the usual isomorphism ($\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$), this becomes possible:

$$\exists\rho.\mu\alpha.\texttt{ptr}(\texttt{ptr}(\alpha \to \texttt{int}, \alpha \to \texttt{int}), \rho)$$

Notice that in the above encoding, we have abstracted an allocated type (ρ) which is used to describe the rest of the object after the method pointer. Furthermore, the methods in the method table expect to take values of type α which is isomorphic to the type of the (unpacked) object. That is, the methods take in the whole object (including the method table) instead of just the instance variables.

This last encoding of objects is closely based on ideas of Bruce (1995; 2002). There are many other potential encodings (see Bruce, Cardelli, and Pierce, 1997, for a nice overview.) In short, it is possible to draw upon the wealth of literature on object and closure encodings to find a small set of re-usable type constructors, such as F-bounded existentials, to provide your typed assembly language with enough power to support compilation of modern object-oriented or functional languages, without baking in a particular

object model. Again, this contrasts with the JVM and CLI which fix on a single object model and provide poor support for encoding languages outside that model.

One problem with these object encodings is that they do not readily support a form of "down-casting" where we perform a run-time test to determine whether an object implements a given interface. Such operations are common in languages such as Java, where the lack of parametric polymorphism and the erroneous addition of covariant arrays requires dynamic type tests. In general, dynamic type tests can be accomplished by using some form of representation types (Crary, Weirich, and Morrisett, 1998), but these encodings are relatively heavyweight and do not support the actual representations used by implementations. Glew (1999) suggested and implemented extensions to TALx86 that better supports practical implementations.

Arrays, Arithmetic, and Dependent Types

In TAL-1, we are restricted to using *constant* offsets to access the data components of an object. Similarly, we can only allocate objects whose size is known at compile time. Thus, we cannot directly encode arrays.

The simplest way to add arrays is to revert to high-level, CISC-like instructions. We could imagine adding a primitive r_d`:= newarray(`r_i, r_s`)` which would allocate an array with r_i elements, and initialize all of the components with the value in r_s, returning a pointer to the array in register r_d. To read components out of an array, we might have an operation r_d`:= ldarr(`r_s, r_i`)` which would load the r_i^{th} component of array r_s, placing the result in r_d. Dually, the operation `starr(`r_s, r_d, r_i`)` would store the value in r_s into the r_i^{th} component of array r_d.

To ensure type safety for the `ldarr` and `starr` operations, we would need to check that the element offset r_i did not exceed the number of elements in the array and jump to an exception handler if this constraint is not met. In turn, this would demand that we (implicitly) maintain the size of the array somewhere. For instance, we could represent an array with n elements as a tuple of $n + 1$ components with the size in the first slot.

4.6.1 EXERCISE [RECOMMENDED, ⋆⋆, ↛]: Extend the TAL-1 abstract machine with rewriting rules for arrays as described above and provide typing rules for the new instructions. □

The advantage of the approach sketched above is that it leaves the type system simple. However, it has a number of drawbacks: First, there is no way to eliminate the check that an offset is in bounds, even if we know and can prove that this is the case. Second, this approach requires that we maintain

the size of the array at runtime, even though the size might be statically apparent. Third, the real machine-level operations that make up the primitive subscript and update operations would not be subject to low-level optimizations (e.g., instruction scheduling, strength reduction, and induction variable elimination.)

An alternative approach based on *dependent types* was suggested by Xi and Harper (2001) and implemented (to some degree) in TALx86. The key idea behind the approach, called DTAL, is to first add a form of *compile-time* expressions to the type system:

$$e \ ::= \ n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \mathtt{i} \mid \cdots$$

A compile-time expression is made up of constants (n), arithmetic operations, and compile-time integer variables (`i`). We then allow type constructors to depend upon (i.e., be indexed by) a compile time expression. For instance, the type $\mathtt{arr}(\tau, 30 + 12)$ would classify those arrays that have 42 components, each of which is a value of type τ. Similarly, the type `int(36)` would classify those integer values that are equal to 36—in other words, `int(36)` is a singleton type.

To support integers whose value is unknown, or arrays whose number of elements are unknown, we can use a suitably quantified compile-time variable. For instance, the type `∃i.int(i)` would classify any integer value, and the type $\exists \mathtt{i}.\mathtt{arr}(\tau,\mathtt{i*2})$ would classify arrays of τ values with an even number of components. More importantly, the type

$$\forall \mathtt{i}_1,\mathtt{i}_2.\mathtt{code\{r1:int}(\mathtt{i}_1),\mathtt{r2:arr}(\mathtt{T},\mathtt{i}_1),\mathtt{r3:int}(\mathtt{i}_2)\}$$

would classify code that expects `r2` to hold an array with i_1 elements, `r1` to hold an integer equal to i_1, and `r3` to hold some other integer equal to i_2. Thus, we can use this limited form of dependent types to track an important relation between two values—that one register holds the number of elements in the array pointed to by another register.

To support the elimination of array bounds checks, we need to go beyond equality relations and track refinements of values as we perform tests. For instance, in a context with the code type above, if we wanted to use `r3` to index into array `r2`, it should be sufficient to check that the value in `r3` is greater than or equal to 0, and less than `r1`:[4]

```
sub:  ∀i1,i2.code{r1:int(i1),r2:arr(T,i1),r3:int(i2), ...}
  if r3<0 jump L;
  rt := r3 - 1;
  if rt>=0 jump L;
  rd := ldarr(r2,r3)   // rd := Mem[r2+r3]
```

4. In practice, this can be determined using a single unsigned comparison.

In other words, the `ldarr` operation above should type-check since we are in a context where `r2` is an array of size i_1, `r3` is an integer equal to i_2, and the predicate $(i_2 \geq 0) \wedge (i_2 < i_1)$ is true. To support this validation, DTAL checked instructions under a typing context of the form $(\Gamma; P)$ where P was a predicate that was assumed to be true. Tests, such as the `blt r3, ERROR` instruction, would typically add conjuncts to the context's predicate.

For example, type-checking for the fragment above would proceed as follows:

```
1. sub:  ∀i1,i2.(...; true)
2.   if r3<0 jump L;  // ({...}; true ^ (i2 >= 0))
3.   rt := r3 - 1;    // ({...,rt:int(i2-i1)};(i2 >= 0))
4.   if rt>=0 jump L; // ({...,rt:int(i2-i1)};(i2 >= 0)^(i2-i1 < 0))
5.   rd := ldarr(r2,r3)
```

After checking line 2, the context's predicate (`true`) has been refined by adding the conjunct i_2 `>= 0` since `r3` has type `int(`i_2`)` and the test can only fall through when `r3` is greater than or equal to zero. At line 3, `rt` is given the type `int(`i_2`-`i_1`)` since `r1` has type `int(`i_1`)` and `r3` has type `int(`i_2`)` and the operation places `r3 - r1` into `rt`. Then, at line 4, the test adds the conjunct i_1`-`i_1 `< 0` to the fall-through continuation's context. Finally, at line 5, we are able to satisfy the pre-condition of the `ldarr` construct since we are in a context where the index lies between 0 and the size of the array.

In DTAL, the predicates used for refinements were restricted to linear inequalities so as to keep type-checking decidable. But in general, we could add support for arbitrary predicates, and simply require that the code producer provide an explicit proof that the current (inferred) predicate implied the necessary pre-conditions. In other words, we can fall back to a very general proof-carrying code framework, as described in the following chapter.

However, as the designer of the type system, we would still be responsible for providing the code producer a set of sound (and relatively complete) inference rules for proving these relations. Though this is possible, it is nowhere as easy as the simple proofs that we have presented here.

4.7 Some Real World Issues

Clearly, TAL-1 and the extensions described earlier provide the mechanisms needed to implement only very simple languages. Furthermore, most of the operations of the machine have a one-to-one correspondence with the operations of a typical RISC-style machine. Some operations, such as `commit` could be pushed into the proof representation since they have no run-time effect.

And abstracting other operations, such as `malloc`, insulates us from details of the memory management runtime.

Of course, there are a number of simple extensions that could be made to make the type system a little more useful. For instance, we could annotate primitive memory type components with flags to control whether that component supports read-only, write-only, or read-write access.

4.7.1 EXERCISE [$\star$, $\rightarrowtail$]: Assuming you added type qualifiers for read-only and write-only access to tuple components. How would you change the abstract machine so that you captured their intended meaning? □

We could also add support for subtyping in a number of ways. For instance, we could take: $\texttt{ptr}(\sigma, \sigma') \leq \texttt{ptr}(\sigma)$. That is, we can safely forget the tail of a sequence of values, for both `ptr` and `uptr` types. We can also consider a read-write component to be a subtype of a read-only or a write-only component. For shared, read-only components, we can support covariant deep subtyping, and for write-only components, we can have contra-variant subtyping. Interestingly, it is sound to have covariant subtyping on read-write components of unique pointers. All of these extensions were supported in some form for TALx86.

An issue not addressed here is support for primitive values of sizes less than a machine word (e.g., a `char` or `short`). But this too is relatively easy to accommodate. The key thing is that we need some function from operand types to their sizes so that we can determine whether or not a `ld` or `st` is used at the right offset. A slightly more troublesome problem is the issue of alignment. On many architectures (e.g., the MIPS, SPARC, and Alpha), primitive datatypes must be naturally aligned. For instance, a 64-bit value (e.g., a `double`) should be placed on a double-word boundary. Of course, the compiler can arrange to insert padding to ensure this property, if we assume that `malloc` places objects on maximally aligned boundaries. Still, we might need to add a well-formedness judgment to memory types to ensure that they respect alignment constraints.

The approach to typing the stack is powerful enough to accommodate standard procedure calls, but cannot handle nested procedures, even if they are "downward-only" as in Pascal. To support this, we would, in general, need some form of static pointers back into the stack (or display.) The STAL type system supports a limited form of such pointers which also provides the mechanisms needed to implement exceptions (Morrisett et al., 2002). These extensions were used in the TALx86 implementation. An alternative approach, based on intersection types, is suggested by Crary's TALT (Crary, 2003) which has the advantage that it better supports stack-allocated objects. A more general approach is to integrate support for *regions* into the

type system in the style of Cyclone (Grossman, Morrisett, Jim, Hicks, Wang, and Cheney, 2002) or one of the other region-based systems described in Chapter 3.

The original TAL used a different mechanism, based on initialization flags and subtyping, to support shared-object initialization. More recently, the work of Petersen et al. (2003) provides an approach to initialization based on a "fuse" calculus. The approach based on initialization flags has the advantage that uninitialized objects are first class, which is useful in some contexts, such as "tail-allocation" (Minamide, 1998). Neither our approach based on unique pointers nor the fuse calculus supports this. Furthermore, neither approach supports the initialization of circular data structures, which is important when building recursive closures. These concerns motivated the design of alias types (Smith, Walker, and Morrisett, 2000) which handles all of these concerns and more, and which were implemented in TALx86. Recently, Ahmed and Walker (2003) have suggested yet another approach based on an embedding of the logic of bunched implications within a type system.

It is possible to add a `free` instruction to TAL-1 which takes a unique pointer and returns it to the runtime for re-use. In some sense, `free` is the ultimate type-changing operation, for it is simply a way to recycle memory so that it can later be used to hold values of a different type. Unfortunately, it is not so easy to provide a `free` operation for shared pointers.

4.7.2 EXERCISE [RECOMMENDED, ⋆, ↛]: Given an example program that could "go wrong" if we allowed `free` to operate on shared pointers. □

As noted earlier, the type system is closed under extensions to the heap, but not necessarily a shrinking heap. It can be shown that if a location is not reachable from the registers (or from reachable code) then a heap value can be safely thrown away. But discovering that this property is true requires more than a single machine instruction. Indeed, it requires a run-time examination of the pointer-graph to determine the unreachable objects.

Therefore, with the introduction of shared data pointers, we are essentially buying into the need for a garbage collector to effectively recycle memory. Of course, to support accurate garbage collection requires that we provide enough information that the collector can determine pointers from other data values at run-time. Furthermore, the collector requires knowledge of the size of heap objects. Finally, many collectors require a number of subtle properties to hold (e.g., no pointers to the interior of heap objects) before they can be invoked. Capturing all of these constraints in a useful type system is still somewhat of a challenge.

TALx86 and the proposed TALT use a *conservative* collector to recycle memory. Conservative collectors do not require precise information about

which objects are pointers. However, they tend to have leaks, since they sometimes think an integer is a pointer to an unreachable object. Like other collectors, a number of invariants must hold in order for the collection to be sound. The TALT system formalizes these constraints as part of its type system.

Another possibility is to integrate the Capability types which provide a general support for regions at the TAL level (Walker, Crary, and Morrisett, 2000). With this type system, it is possible to code up a copying collector within the language as suggested by Wang and Appel (2001). However, doing an efficient copying collector requires a bit more technical machinery. Some of these issues are resolved by Monnier, Saha, and Shao (2001).

Finally, note that, at this point, a paper and pencil proof of the soundness of a system that incorporates these extensions becomes quite large (and tedious) and we are therefore likely to make mistakes. To avoid such pitfalls, we would be wise to encode the abstract machine and type system in some formal system where the proof can be verified. For example, Crary encodes his TALT abstract machine and typing rules using the LF framework (2003) whereas Hamid et al. have done this using Coq (2002). Another approach championed by Appel and Felty (2000) is called Foundational Proof Carrying Code, whereby the types are semantically constructed using higher-order logic in such a way that they are by definition sound with respect to the (concrete) machine's semantics.

4.8 Conclusions

Type systems for low-level code, including compiler intermediate languages and target languages, are an exciting area of research. In part, this is because the "human constraint" is lifted since the typing annotations are produced and consumed by machines instead of humans. That is, we do not have to worry about the type system being too complicated for the average programmer, or that it requires too many typing annotations. These concerns often dominate the design of type systems for high-level languages. Of course, it is still important to keep the design as simple and orthogonal as possible so that we can construct proofs of soundness and have confidence in the implementation. Ideally, proofs should be carried out in a machine-checked environment.

Low-level languages also present new challenges to type system designers. For instance, the issues of initialization and memory recycling are of little concern in high-level languages, since these details are meant to be handled by the compiler and run-time system. Yet, the nitty-gritty details of the run-time system are crucial for the proper functioning of the system.

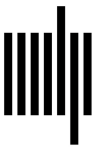
This is a section of doi:10.7551/mitpress/1104.001.0001

Advanced Topics in Types and Programming Languages

Edited by: Benjamin C. Pierce

Citation:

Advanced Topics in Types and Programming Languages
Edited by: Benjamin C. Pierce
DOI: 10.7551/mitpress/1104.001.0001
ISBN (electronic): 9780262281591
Publisher: The MIT Press
Published: 2024



The MIT Press

This book was set in Lucida Bright by the editor and authors using the LaTeX document preparation system.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Advanced topics in types and programming languages / Benjamin C. Pierce, editor.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-16228-8 (hc.: alk. paper)
1. Programming languages (Electronic computers). I. Pierce, Benjamin C.
QA76.7.A36 2005
005.13—dc22

200457123

10 9 8 7 6 5 4 3 2 1