
Typed Assembly Language

Greg Morrisett

Harvard University

`greg@eecs.harvard.edu`

References:

- G.Morrisett. Typed Assembly Language. In Benjamin C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, MIT Press, 2005.
- G.Morrisett, D.Walker, K.Crary, and N.Glew. From System-F to Typed Assembly Language, *ACM Transactions on Programming Languages and Systems*, 21(3):528-569, May 1999.
- See also <http://www.cs.cornell.edu/talc>

Type Safety

Type-safe programming languages, such as Scheme, ML, and Java enjoy memory isolation.

- That is, type-safety implies memory isolation.
- In fact, it's a much stronger property.

Furthermore, type-safety is all about *fine-grained* memory safety.

- Having a value of type `ptr(int)` means you can safely read or write the address that value corresponds to.
- Having a value of type $\tau_1 \rightarrow \tau_2$ means you can safely jump to the address that value corresponds to.

And languages that are *statically* typed, such as ML and Java, don't have to pay the performance costs of masking or checking values.

Question:

Why not force extensions to a service be written in ML or Java?

Performance?

- Ideally, we should beat the performance of SFI.
- That gives us a budget of 20% time overhead?
- In my experience, type-safe languages are much slower and fatter.
- Of course, we're enforcing a better policy...
- Key: if the overheads are too great, people won't use it.

Language-Independence

2. There are *many* type safe languages:

- SML, O'CamL, Haskell, Java, C#, Visual Basic, Mercury, Eiffel, Cyclone, Ccured, ...
- We should be able to pick a language suited to the task.
- A key advantage of OS/hardware and SFI is that they are language neutral.

Sun's JVM and Microsoft's .NET frameworks address these issues to some degree, but fall down in others:

- e.g., no support for tail-calls on the JVM
- e.g., improper treatment of array subtyping in both.
- forced object model.

Trust

3. Compilers and run-time systems for high-level, type-safe languages are complicated:

- SML/NJ: 100K lines of SML code, 50K lines of C code.
- It's very likely that the implementation has bugs.
- These points hold for the JVM and .NET worlds as well.

Typing Machine Code

- An alternative approach is to try to apply type-safety directly to *machine code*.
- That eliminates the need for a trusted compiler.
- Operationally, the machine code can efficiently support a wide variety of languages.
- The challenge is constructing a type system that is language-neutral.

TAL

The goals of Typed Assembly Language (TAL):

- as in SFI, memory & control-flow safety
- as in SFI, try to be language-neutral
- unlike SFI, support “swiss cheese” (i.e., least priv.)
- unlike SFI, static enforcement

The reality:

- it's not language-neutral
- (but it's better than the JVM or CLR)
- See the web page for TALx86

Outline

- TAL-0: control-flow isolation
- TAL-1: polymorphism
- TAL-2: data isolation
- TAL-3: allocation and initialization
- TAL-4: data abstraction
- TAL Wrapup

TAL-0: control-fbw isolation

TAL-0 Abstract Machine

machines $M ::= (H, R, I)$
heaps $H ::= \{\ell_1 = I_1, \dots, \ell_n = I_n\}$
register file $R ::= \{r_1 = v_1, \dots, r_k = v_k\}$
instr. sequences $I ::= \text{jump } v \mid \iota; I$

- Register file is a total map from registers to (register-free) operands.
- Heap is a partial map from labels to instruction sequences (code).
- Machine state is a code heap, register file, & instruction sequence.
- I plays the role of a program counter.

Instruction Syntax

registers	r	$::=$	$r_1 \mid r_2 \mid \dots \mid r_k$	
integers	n			
labels	ℓ			
operands	v	$::=$	$r \mid n \mid \ell$	
instructions	ι	$::=$	$r_d := v$	(move)
			\mid	
			$r_d := r_s + v$	(add,addi)
			\mid	
			ifeq r jump v	(beq)

Example

A procedure that computes the product of the integers in `r1` and `r2`, placing the result in `r3` before jumping to the return address 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
```

Dynamics

The rewriting rules for TAL-0 are as follows:

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

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

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

$$\frac{R(r) = 0 \quad H(\hat{R}(v)) = I'}{(H, R, \text{ifeq } r \text{ jump } v; I) \longrightarrow (H, R, I')}$$

$$\frac{R(r) = n \quad n \neq 0}{(H, R, \text{ifeq } r \text{ jump } v; I) \longrightarrow (H, R, I)}$$

Comments

- Think of each instruction as a function/combinator from register files to register files.
- Semi-colon is the “bind” i.e., sequencing.
- Jump is the function call.
- All sequences end with jump and there’s no nesting – CPS!
- It’s easy to show a step-for-step simulation with a concrete MIPS semantics.

Policy

No wild jumps (something we wanted for SFI).

- Positive re-statement: the only possible jumps are to the labelled instruction sequences.
- Baked into the formulation of the abstract machine!
- We get stuck if we try to jump to an integer or to an undefined label.
- Note that we also get stuck if we try to add a label and an integer.
- Moral: the policy is expressed by writing down an abstract machine. Configurations for which there is no transition are “bad”.

Statics

Clearly, need to distinguish labels from integers.

$$\begin{array}{l} \text{types } \tau ::= \text{int} \mid \text{code}(\Gamma) \\ \text{register file types } \Gamma ::= \{r_1:\tau_1, \dots, r_n:\tau_n\} \\ \text{heap types } \Psi ::= \{\ell_1:\tau_1, \dots, \ell_n:\tau_n\} \end{array}$$

- To keep the induction going, we need to know the types of the registers once we jump to a particular location (in case it jumps through a register.)
- $\text{code}(\Gamma)$ describes a label which when jumped to, expects register r_i to have a value of type $\Gamma(r_i)$.
- Ψ describes a heap (i.e., association of code labels to their code types.)

Operands

Typing Values & Operands: $\Psi \vdash v : \tau$, $\Psi; \Gamma \vdash v : \tau$:

$$\Psi \vdash n : \text{int}$$

$$\Psi \vdash \ell : \Psi(\ell)$$

$$\frac{\Psi \vdash v : \tau}{\Psi; \Gamma \vdash v : \tau}$$

$$\Psi; \Gamma \vdash r : \Gamma(r)$$

Instructions

Typing Instructions: $\Psi \vdash \iota : \Gamma_1 \rightarrow \Gamma_2$

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d : \tau]}$$

$$\frac{\Psi; \Gamma \vdash r_s : \text{int} \quad \Psi; \Gamma \vdash v : \text{int}}{\Psi \vdash r_d := r_s + v : \Gamma \rightarrow \Gamma[r_d : \text{int}]}$$

$$\frac{\Psi; \Gamma \vdash r_s : \text{int} \quad \Psi; \Gamma \vdash v : \text{code}(\Gamma)}{\Psi \vdash \text{if } r_s \text{ jump } v : \Gamma \rightarrow \Gamma}$$

Sequences

Typing Instruction Sequences: $\Psi \vdash I : \mathbf{code}(\Gamma)$

$$\frac{\Psi; \Gamma \vdash v : \mathbf{code}(\Gamma)}{\Psi \vdash \mathbf{jump} \ v : \mathbf{code}(\Gamma)}$$

$$\frac{\Psi \vdash \iota : \Gamma \rightarrow \Gamma_2 \quad \Psi \vdash I : \mathbf{code}(\Gamma_2)}{\Psi \vdash \iota; I : \mathbf{code}(\Gamma)}$$

Machines

Typing Heaps, Register Files, & Machines:

$$\frac{\forall \ell \in \text{Dom}(\Psi). \Psi \vdash H(\ell) : \Psi(\ell)}{\vdash H : \Psi}$$

$$\frac{\forall r. \Psi \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma}$$

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash I : \text{code}(\Gamma)}{\Psi \vdash (H, R, I)}$$

Soundness

Theorem: if $\vdash M$ then M cannot become stuck (i.e., do a wild read/write/jump.) (i.e., $M \longmapsto^* M_1$ implies there exists an M_2 s.t. $M_1 \longmapsto M_2$.)

Proof: Show $\vdash M$ implies there exists an M' such that $M \longmapsto M'$ and $\vdash M'$. That is, show a well-typed program is not immediately stuck, and that well-typedness is an invariant under evaluation.

Proof

Only one rule ends with $\vdash M$ so taking $M = (H, R, I)$, by inversion the derivation is of the form:

$$\frac{\frac{\mathcal{D}_H}{\vdash H : \Psi} \quad \frac{\mathcal{D}_R}{\Psi \vdash R : \Gamma} \quad \frac{\mathcal{D}_I}{\Psi \vdash I : \mathbf{code}(\Gamma)}}{\vdash (H, R, I)}$$

We proceed by cases on the last rule used in \mathcal{D}_I .

Proof, contd.

Case: \mathcal{D}_I ends with an instance of the jump rule:

$$\frac{\Psi; \Gamma \vdash v : \text{code}(\Gamma)}{\Psi \vdash \text{jump } v : \text{code}(\Gamma)}$$

Lemma [Canonical Operands]: If $\Psi \vdash R : \Gamma$ and $\Psi; \Gamma \vdash v : \tau$ then:

1. if $\tau = \text{int}$ then $\hat{R}(v) = i$ for some i , and
2. if $\tau = \text{code}(\Gamma')$ then $\hat{R}(v) = \ell$ for some $\ell \in \text{Dom}(\Psi)$.

So from (2), we know $v = \ell \in \text{Dom}(\Psi)$.

From $\vdash H : \Psi$ we can conclude $H(\ell) = I'$ for some I' such that $\Psi \vdash I' : \text{code}(\Gamma)$.

So $(H, R, I) \xrightarrow{(\quad)} H, R, I'$. Furthermore, $\vdash (H, R, I')$.

Proof, contd.

Case: \mathcal{D}_I ends with an instance of the sequencing rule, where the instruction at the beginning is a move:

$$\frac{\frac{\Psi; \Gamma \vdash v : \tau}{\Psi; \Gamma \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d:\tau]} \quad \frac{\mathcal{D}}{\Psi \vdash I' : \mathbf{code}(\Gamma[r_d:\tau])}}{\Psi; \Gamma \vdash r_d := v; I' : \mathbf{code}(\Gamma)}$$

where $I = r_d := v; I'$.

Clearly, $(H, R, I) \longmapsto (H, R[r_d = \hat{R}(v)], I')$.

It's easy to show from $\Psi; \Gamma \vdash v : \tau$ and $\Psi \vdash R : \Gamma$ that

$\Psi \vdash \hat{R}(v) : \tau$.

So it follows that $\Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d:\tau]$ and:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d:\tau] \quad \Psi \vdash I' : \mathbf{code}(\Gamma[r_d:\tau])}{\vdash (H, R[r_d = \hat{R}(v)], I')}$$

Proof, contd.

The other cases (where the instruction sequence starts with an add or a conditional branch) are similar.

Very easy proof.

However...

Oops!

You can't write a program that jumps through a register!
The only possible derivation for a jump through a register

looks like this:

$$\frac{\frac{\Psi \vdash \Gamma(r) : \mathbf{code}(\Gamma)}{\Psi; \Gamma \vdash r : \mathbf{code}(\Gamma)}}{\Psi \vdash \mathbf{jump} \ r : \mathbf{code}(\Gamma)}$$

The top line means that we need $\Gamma(r) = \mathbf{code}(\Gamma)$ but there is no solution to this equation, given our inductively defined types.

Fixes

There are lots of possible solutions to this “problem”.

- Introduce recursive types ($\mu\alpha.\tau = \tau\{(\mu\alpha.\tau)/\alpha\}$).
- Introduce subtyping on register file types (e.g., $\text{code}(\Gamma) \leq \text{int}$).
- Introduce universal polymorphism ($\forall\alpha.\tau$)

Adding all of these typing features is really necessary for a realistic TAL, but we can go a long way with just universal polymorphism.

TAL-1: polymorphism

Types revisited

types $\tau ::= \text{int} \mid \text{code}(\Gamma) \mid \alpha \mid \forall\alpha.\tau$

Values with type α must be treated abstractly.

Values with polymorphic types (e.g., $\forall\alpha.\tau$) can be instantiated by substituting a particular type for the bound type variable (e.g., $\tau\{\tau'/\alpha\}$). Formally:

$$\frac{\Psi; \Gamma \vdash v : \forall\alpha.\tau}{\Psi; \Gamma \vdash v : \tau\{\tau'/\alpha\}}$$

We can introduce polymorphic generalization only for instruction sequences:

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall \alpha. \tau}$$

- Polymorphism in the presence of mutable, shared data always causes trouble somewhere.
- By restricting generalization to code, we're avoiding some of the potential pitfalls (good rule of thumb.)
- We'll require that Ψ be closed so there will be no other free variables in the context.

Fixing the proof

Basically, introduce a bunch of substitution lemmas such as:

If $\Psi \vdash v : \forall\alpha.\tau$ then $\Psi \vdash v : \tau\{\tau'/\alpha\}$.

That's all there is to it!

How does it help?

If the target is polymorphic in the register we're jumping through, then we can specialize it to the current code type.

Let $\tau = \forall\alpha.\text{code}(\Gamma[r : \alpha])$.

$$\frac{\Psi \vdash \Gamma(r) : \tau}{\Psi \vdash \Gamma(r) : \text{code}(\Gamma[r : \tau])} \quad \frac{\Psi; \Gamma \vdash r : \text{code}(\Gamma[r : \tau])}{\Psi \vdash \text{jump } r : \text{code}(\Gamma[r : \tau])}$$

In essence, we're substituting τ for α within $\text{code}(\Gamma[r : \alpha])$ which is similar to what happens with recursive types.

Note that this requires *polymorphic recursion*—something SML does not support (but Haskell and O'CamI do.)

An 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 $\Gamma = \{r_1, r_2, r_3:\text{int}, r_4:\forall\alpha.\text{code}\{r_1, r_2, r_3:\text{int}, r_4:\alpha\}\}$.
- Let Ψ map `prod`, `loop`, and `done` to `code`(Γ).
- Claim that Ψ is type-consistent with the code that is given.

Other Uses for Polymorphism

Simulating subtyping:

- When compiling an “if-then-else”, we need to jump to a common join-point after the `then-` and `else-clauses`.
- Suppose the `then-clause` moves code into `r1` whereas the `else-clause` moves an integer into `r1`.
- By making the join-point polymorphic in `r1` (e.g., $\forall\alpha.\text{code}\{r_1:\alpha, \dots\}$) both paths can jump to the label. Here, α is treated as a “top”.
- Another alternative would be to add union types (e.g., $\tau_1 \vee \tau_2$) or direct support for subtyping.

Callee-saves registers

The advantage of universal polymorphism is that we can track input/output relationships. Consider a label ℓ with this type:

$$\ell : \forall \alpha. \text{code}\{r_1:\alpha, r_2:\text{code}\{r_1:\alpha, \dots\}, \dots\}$$

We can think of ℓ as a function that takes its argument in r_1 and its return address in r_2 , and when it returns, it returns its result in r_1 .

So, ℓ conceptually is the result of compiling something like: $\forall \alpha. \alpha \rightarrow \alpha$. That means that if we “call” ℓ , then it *must* return what we pass to it (if it returns at all.)

This is useful for capturing compiler idioms like callee-saves registers.

Moral

You can go a long way with just ints, code, and universal polymorphism. And the proofs remain remarkably simple. But, we still have a long way to go...

Checking vs. Inference

The right way to conceptualize the code consumer is that we're passing it the entire proof that the program type-checks. The consumer is just verifying the proof and extracting the program.

This is a relatively easy (and hence trustworthy) process.

In contrast, giving you the code and asking you to *infer* whether it is typeable is likely to be undecidable.

However, the size of real “proofs” would be tremendous compared to the code. So we have to rethink the representation of the proof.

We could note that most of the proof rules are syntax directed and so they can be run backwards.

The notable exceptions are polymorphic generalization and instantiation, as well as “guessing” the right Ψ .

One solution (used in TALx86) is to introduce term constructs that witness which rule to pick when the rules are not syntax directed.

For instance, TALx86 had an explicit instantiation syntax for polymorphic values, and allowed explicit types to be put on the labels.

Drawbacks: types are still large, must trust reconstruction engine.

TAL-2: data isolation

Data

Let's augment our abstract machine with data values:

memory values $m ::= I \mid \langle v_1, \dots, v_n \rangle$
heaps $H ::= \{ \ell_1 = m_1, \dots, \ell_n = m_n \}$

so now our memory (H) maps pointers to either code or sequences of word-sized values.

We could also imagine adding in load and store instructions:

instructions $\iota ::= \dots$
| $r_d := \text{Mem}[r_s + i]$ load
| $\text{Mem}[r_d + i] := r_s$ store

New Rules

$$\frac{R(r_s) = \ell \quad H(\ell) = \langle v_0, \dots, v_j, \dots, v_n \rangle \quad 4 * j = i}{(H, R, r_d := \mathbf{Mem}[r_s + i]; I) \longrightarrow (H, R[r_d = v_j], I)}$$

$$\frac{R(r_d) = \ell \quad H(\ell) = \langle v_0, \dots, v_j, \dots, v_n \rangle \quad 4 * j = i}{(H, R, \mathbf{Mem}[r_d + i] := r_s; I) \longrightarrow (H[\ell = \langle v_0, \dots, R(r_s), \dots, v_n \rangle], R, I)}$$

New Types

operand types $\tau ::= \text{int} \mid \text{code}(\Gamma) \mid \text{ptr}(\sigma) \mid \alpha \mid \forall\alpha.\tau \mid \forall\rho.\tau$

memory types $\sigma ::= \epsilon \mid \tau \mid \sigma_1; \sigma_2 \mid \rho$

- $\text{ptr}(\sigma)$ is the type of a data pointer, where the data are described by σ .
- We'll treat σ 's as equivalent modulo associativity of “;” with ϵ a left- and right-unit.
- Memory type variables (ρ) let us abstract over a whole sequence of types.
- We have to segregate abstraction over word-sized values (α) from abstraction over arbitrary-sized values (ρ).

Typing Load and Store

$$\frac{\Psi; \Gamma \vdash r_s : \text{ptr}(\sigma_1; \tau; \sigma_2) \quad \text{sizeof}(\sigma) = i}{\Psi \vdash r_d := \text{Mem}[r_s + i] : \Gamma \rightarrow \Gamma[r_d : \tau]}$$

$$\frac{\Psi; \Gamma \vdash r_d : \text{ptr}(\sigma_1; \tau; \sigma_2) \quad \text{sizeof}(\sigma) = i \quad \Psi; \Gamma \vdash r_s : \tau}{\Psi \vdash \text{Mem}[r_d + i] := r_s : \Gamma \rightarrow \Gamma}$$

where

$\text{sizeof}(\text{int}) = \text{sizeof}(\text{code}(\Gamma)) = \text{sizeof}(\text{ptr}(\sigma)) = \text{sizeof}(\alpha) = 4$

$\text{sizeof}(\sigma_1; \sigma_2) = \text{sizeof}(\sigma_1) + \text{sizeof}(\sigma_2)$

Note, we need to statically know the size of σ_1 to validate the load, so it can't have a ρ within it.

Close...

But alas, it falls short of what's needed:

- No way to allocate, initialize, or recycle memory.
 - But we need this for stack frames, records, objects, ...
- All loads and stores must use *constant* offsets relative to the base of an object.
 - This precludes arrays.
- There's no way to *refine* the type of a value.
 - Needed for datatypes, object downcasts, type-tests, etc.

TAL-3: allocation and initialization

High-Level Languages

In a high-level language, such as ML or Java, the allocation and initialization of objects is in some sense *atomic*.

For instance, for an ML record $\{x:\text{int}, f:\text{int}\rightarrow\text{int}\}$, the only way to create the record is to supply the initial values for x and f as in:

```
val r = {x = 3, f = (fn z => z + 1)}
```

This ensures that no one reads a field, thinking it's initialized, when it isn't.

But in TAL, we want to make the steps of allocation & initialization *explicit* to expose those steps to optimization (e.g., instruction scheduling, register allocation, tail-allocation, ...)

Allocation

To simplify things, let us add a new primitive instruction:

instructions $\iota ::= \dots \mid \text{malloc } n$

which allocates a fresh location ℓ to hold $n/4$ words, all initially zero:

$$(H, R, \text{malloc } (n * 4); I) \rightarrow (H[\ell \mapsto \underbrace{\langle 0, \dots, 0 \rangle}_n], R[r_1 \mapsto \ell], I)$$

$(\ell \notin \text{Dom}(H))$

The typing rule is then:

$$\Psi \vdash \text{malloc } (n * 4) : \Gamma \rightarrow \Gamma[r_1 : \underbrace{\text{ptr}(\text{int}, \dots, \text{int})}_n]$$

Example

Now we can compile an ML expression, such as

```
val r = {x = 3, y = 42}
```

to the following TAL code:

```
malloc 8;           // r1 : ptr(int,int)
r := r1;           // r : ptr(int,int)
Mem[r+0] := 3;
Mem[r+4] := 42;
```

Oops!

But this doesn't work when one of the components isn't an integer:

```
main:
  malloc 8;           // r1 : ptr(int,int)
  r := r1;           // r : ptr(int,int)
  Mem[r+0] := main;
  Mem[r+4] := 42;
```

because the rule for memory updates does not let us *change* the type of a location:

$$\frac{\Psi; \Gamma \vdash r_d : \text{ptr}(\sigma_1; \tau; \sigma_2) \quad \text{sizeof}(\sigma) = i \quad \Psi; \Gamma \vdash r_s : \tau}{\Psi \vdash \text{Mem}[r_d + i] := r_s : \Gamma \rightarrow \Gamma}$$

Why not?

Consider:

```
let val x : (int->int) ref = ref (fn x => x)
    val y : (int->int) ref = x
in
    x := 3;
    (!y)(42)
end
```

Aha! High-level languages do not let you change the type of memory locations because a *reader* expects the types to remain the same.

Consider:

```
let val nil :  $\forall 'a. 'a$  list
    val r :  $\forall 'a. ('a$  list) ref = ref nil
    val x : int list ref = r
    val y : (int->int) list ref = r
in
    x := [3];
    (hd(!y))(42);
end
```

Same problem—trying to use the same shared, updateable object at two different types.

Consider:

```
class Pt { int x,y; }
class Pt3d extends Pt { int z; }
void f(Pt3D[] A) {
    Pt[] B = A; // allowed since Pt3D <= Pt
    B[0] = new Pt(1,2);
    A[0].z; // oops!
}
```

Same problem—trying to use the same shared, updateable object at two different types.

But what about Java variables?

```
class Pt { int x,y; }
class Pt3d extends Pt { int z; }
void f(Pt3D A) {
    Pt B = A;
    B = new Pt(1,2);
    A.z;
}
```

Nothing goes wrong here even though we changed the type of B from Pt3D to Pt. Hmmmm....

And recall our rule for updating registers:

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d : \tau]}$$

There's no requirement that the type of r_d stay the same.

So, there's something different about Java variables and TAL registers compared to ML, Java, or TAL references.

Principle

The basic principle is:

If you're going to change the type of a location, then you need to change the types of all potential paths to that location (i.e., track aliasing precisely), or make sure that the change is compatible with all potential future accesses.

Principle

You can instantiate this principle a number of ways:

1. Allow unknown aliasing, reading, writing \Rightarrow type remains fixed.
2. Allow unknown read-only aliasing \Rightarrow co-variant subtyping.
3. Allow unknown write-only aliasing \Rightarrow contra-variant subtyping.
4. Don't allow aliasing, type can vary arbitrarily.

Option 4 is what we did with registers. Can we pull the same trick with memory?

Linearity

Let us make a distinction between *unique* pointers (no aliasing) vs. normal pointers.

We will allow you to change the type of a unique pointer:

$$\frac{\begin{array}{l} \Psi; \Gamma \vdash r_d : \text{uptr}(\sigma_1; \tau; \sigma_2) \\ \Psi; \Gamma \vdash r_s : \tau' \quad \text{sizeof}(\sigma_1) = i \end{array}}{\Psi \vdash \text{Mem}[r_d + i] := r_s : \Gamma \rightarrow \Gamma[r_d : \text{uptr}(\sigma_1; \tau'; \sigma_2)]}$$

and we will modify `malloc`'s type to reflect that it returns a unique pointer:

$$\Psi \vdash \text{malloc}(n * 4) : \Gamma \rightarrow \Gamma[r_1 : \underbrace{\text{uptr}(\text{int}, \dots, \text{int})}_n]$$

Furthermore, we can always convert a unique pointer to an unrestricted pointer, as long as we give up the ability to change the type of the pointer.

We will make this explicit by adding an instruction to *commit* a unique pointer:

$$\frac{\Gamma(r) = \text{uptr}(\sigma)}{\Psi \vdash \text{commit } r : \Gamma \rightarrow \Gamma[r : \text{ptr}(\sigma)]}$$

Example

For instance, the ML code:

```
val a = {x = 3, y = 42}
val b = {p = a, q = a}
```

could translate into:

```
malloc 8;           // r1:uptr(int, int)
r2 := r1;          // r2:uptr(int, int)
Mem[r2+0] := 3;    // r2:uptr(int, int)
Mem[r2+4] := 42;   // r2:uptr(int, int)
commit r2;         // r2:ptr(int, int)
malloc 8;          // r1:uptr(int, int)
Mem[r1+0] := r2;   // r1:uptr(ptr(int, int), int)
Mem[r1+4] := r2;   // r1:uptr(ptr(int, int), ptr(int, int))
commit r1;         // r1:ptr(ptr(int, int), ptr(int, int))
```

Good News/Bad News

That's the good news. The bad news is that we have to maintain the uniqueness invariant.

The approach used is to destroy access to old copies. For instance, the move rule for unique pointers looks like:

$$\frac{\Gamma(r_s) = \text{uptr}(\sigma)}{\Psi \vdash r_d := r_s : \Gamma \rightarrow \Gamma[r_d : \text{uptr}(\sigma), r_s : \text{int}]}$$

So, if you copy a unique pointer from r_s to r_d , then you can no longer use the copy in r_s .

Deallocation

We can also support an operation `free`:

$$(H[\ell \mapsto \langle v_1, \dots, v_n \rangle], R[r \mapsto \ell], \text{free } r; I) \rightarrow (H, R[r \mapsto 0], I)$$

but we'd better restrict the pointer to be unique!

$$\frac{\Psi; \Gamma \vdash r : \text{uptr}(\sigma)}{\Psi \vdash \text{free } r : \Gamma \rightarrow \Gamma[r : \text{int}]}$$

Without this restriction, we could get stuck trying to dereference a dangling pointer!

That is, there's no guarantee that the program state remains closed.

Problems with free

There are other problems with an explicit `free`:

- We may *recycle* the free'd location ℓ at a different type!
- That is, *free* locations can be captured by a subsequent `malloc`.
- This can lead to *extremely* subtle failures down the line.
- If the location contains unique pointers, then there's no way to regain access to them, resulting in a leak.
- In some sense, `malloc` and `free` are the ultimate in type-changing operations which is why they are fundamentally incompatible with type-safe languages.
- In turn, this tells you why garbage collection shows up in almost all type-safe languages.

Recycling

- Nonetheless, recycling of storage in a *controlled* fashion is important.
- In particular, most compilers want to recycle stack frames for different invocations of different procedures.
- It turns out that unique pointers are good enough for this as well.
- This is because we can think of sp as a register that holds a $uptr(\sigma)$.
- As we'll see, it's very nice to have abstract memory types (ρ) to abstract the “tail” of the stack.

Example

Example compilation:

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

Conventions:

- arguments are passed on the stack
- return address passed in `r4`
- results returned in `r1`
- callee pops arguments
- `r2` and `r3` are temps

But you can code your own convention if you like...

Type Translation

A procedure with type $(\tau_1, \dots, \tau_n) \rightarrow \tau$ maps down to:

$\forall \alpha, \beta, \gamma, \rho.$ code{
 $r_1 : \alpha$ // don't care about value on input
 $r_2 : \beta$ // don't care about value on input
 $r_3 : \gamma$ // don't care about value on input
 $sp : \text{uptr}(\tau_1, \dots, \tau_n, \rho)$ // τ_i 's on top of stack, rest abstract (ρ)
 $r_4 : \forall \delta, \epsilon, \iota.$ code{ // return addr.
 $r_1 : \tau$ // τ return value
 $r_2 : \delta$ // don't care about value on output
 $r_3 : \epsilon$ // don't care about value on output
 $sp : \text{uptr}(\rho)$ // arguments popped, rest intact
 $r_4 : \iota$ // abstract to support recursive type}}

Code Part I

fact:

$\forall a, b, c, s.$

code{r1:a, r2:b, r3:c, sp:uptr(int, s),
r4: $\forall d, e, f.$ 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, (All ...), int)

Mem[sp] := r2; // sp:uptr(int, (All ...), int)

r4 := cont;

jump fact // r1 := fact(z-1)

Code Part II

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

Aside

However that this is only sufficient for dealing with very simple procedural control info, as there's no way to get a pointer into the middle of the stack.

That is, we can't support something like `&x`, so we can't handle stack allocation, frame pointers, static links, etc.

See the “Stack-Based Typed Assembly Language” (JFP) paper for some ways around this, or look at the work on region allocation.

TAL-4: data abstraction

Motivation

If we're going to compile a modern OO or functional language, then we need support for objects and closures.

We could bake these types in, as the JVM and CLR do. But this forces you into a *particular* object/closure model.

Different languages and different compilers use wildly different encodings.

In the spirit of TAL, we should try to find more primitive type constructors that let us encode objects and closures...

Closures

Let's start with closures first since they are much simpler.

In most compilers, closures are represented as a (pointer to a) pair of a piece of code and an environment.

The environment is a data structure that holds the values corresponding to the free variables in the code.

Closure Conversion

For example, consider the ML source code:

```
val f : int -> (int -> int)
val f = fn x => let val g = fn y => x + y
                in g
            end
```

After closure conversion, the code might look like this:

```
gc(genv: {fenv=unit, x:int}, y:int) = genv.x + y

fc(fenv:unit, y:int) =
    let val genv = {fenv=fenv, x=x}
        val g = {code=gc, env=genv}
    in g
    end
```

Closure Conversion

In general, a function of type $\tau \rightarrow \tau'$ with free variables $x_1:\tau_1, \dots, x_n:\tau_n$ will be translated into a pair $\{\text{code}, \text{env}\}$ where:

- env is a record of type $\{x_1:\tau_1, \dots, x_n:\tau_n\}$.
- code is a code pointer that takes two arguments, the first of which is env and the second of type τ' , and returns a τ' .

So at the TAL-level $\tau \rightarrow \tau'$ becomes something like:

$$\text{ptr}((\text{code}(\tau_{\text{env}}, \tau) \Rightarrow \tau'), \tau_{\text{env}})$$

where “ \Rightarrow ” abstracts the calling convention of the generated code.

Oops!

Different $\tau \rightarrow \tau'$ values will have different environments.

```
let val f1 = fn s:string => fn x:int => length(s)
      val f2 = fn z:int => fn y:int => z+y
in
  if flip() then
    (f1 "foo")
  else
    (f2 42)
end
```

But they need to have the same type to get a compositional translation.

Existential Types

The solution is to use an *existential* to abstract the type of the environment:

$$\mathcal{T}[\tau \rightarrow \tau'] = \exists\alpha.\text{ptr}((\text{code}(\alpha, \tau) \Rightarrow \tau'), \alpha)$$

Existentials are introduced and eliminated as follows:

$$\frac{\Psi; \Gamma \vdash v : \tau[\tau'/\alpha]}{\Psi; \Gamma \vdash v : \exists\alpha.\tau}$$

$$\frac{\Psi; \Gamma \vdash r : \exists\alpha.\tau}{\Psi \vdash \text{unpack } r : \Gamma \rightarrow \Gamma[r : \tau]} (\alpha \text{ fresh})$$

The freshness constraint ensures that we treat α abstractly.

Application

So an application of a closure f of type $\text{int} \rightarrow \text{int}$ to an argument 42 might get compiled to something like this:

```
// r1 :  $\exists a. (\text{code}(a, \text{int}) \Rightarrow \text{int}, a)$ 
  unpack r1;           // r1 :  $(\text{code}(b, \text{int}) \Rightarrow \text{int},$ 
  r4 := Mem[r1+0];     // get out code of closure
  renv := Mem[r2+4];  // get out environment
  ra := cont;         // set the return address to
  r1 := 42;           // put argument in r2
  jump r4;            // call code
cont: ...              // return here.
```

Closures as Objects

Closures are in fact a degenerate case of objects:

$$\mathcal{T}[\tau \rightarrow \tau'] = \exists \alpha. \text{ptr}((\text{code}(\alpha, \tau) \Rightarrow \tau'), \alpha)$$

We can generalize this to:

$$\exists \rho_1, \rho_2. \text{ptr}(\text{ptr}(\sigma_{vt}, \rho_1), \tau_1, \dots, \tau_n, \rho_2)$$

where σ_{vt}, ρ_1 is a sequence of code pointers representing the method table.

Here, τ_1, \dots, τ_n correspond to “public” fields (accessible outside the object) and ρ_2 corresponds to “private” fields, (accessible only to the methods of the object.)

Similarly, ρ_1 abstracts a super-class’s methods, so they are available within the methods, but not outside.

Example

Ignore vtables for a second and just consider instance variables for two classes:

```
class Pt { int x,y; }  
class Pt3d extends Pt { int z; }
```

Concretely, they would map down to:

- $Pt = ptr(int, int)$
- $Pt3D = ptr(int, int, int)$

We want to be able to pass a $Pt3D$ (or any future extension of Pt) to functions that expect a Pt . Both can be abstracted to:

$$\exists \rho. ptr(int, int, \rho)$$

For Pt , we pick $\rho = \epsilon$ and for $Pt3D$, we pick $\rho = int$.

Object Encodings

This style of encoding:

$$\exists \rho_1, \rho_2. \text{ptr}(\text{ptr}(\sigma_{vt}, \rho_1), \tau_1, \dots, \tau_n, \rho_2)$$

is similar to Pierce & Turner’s “objects as existentials”.

- In practice, we need to also add support for recursive types (and closures) to support the “self” parameter within methods (see Bruce.)
- Using this style of encoding, we can compile both Java-style OO languages as well as (call-by-value) functional languages such as ML.
- As with calling conventions, we are not forcing a particular representation on the compiler.

Recap

- Aliasing: the root of all evil.
- Uniqueness (a.k.a. linearity) provides convenient power-to-weight.
- Abstraction (\forall and \exists) gets us surprisingly far.
- At this point, we can compile mini-Java and mini-ML.

Beyond...

People like Karl Crary, Zhong Shao, and I have worked to develop more complete TAL's that scale up to realistic languages.

- arrays: need support for reasoning about *dynamic* offsets in loads and stores. See DTAL by Xi & Harper.
- datatype tests: need support for refining the type of a disjoint union based on a run-time test. See TALTwo by Crary.
- downcasts: need support similar to the above. See Neal Glew's thesis.
- exceptions: need support for stack-unwinding. See STAL.

The resulting type systems are fairly complicated. Should we trust them?

Up next: Proof-carrying code.