Typed Assembly Language:
Type Theory for Machine Code

Karl Crary
Carnegie Mellon University

Certified code

- Goal: provide checkable evidence that a given program is "safe".
- Key issues in design of a certified code architecture:
  - What do we mean by safety?
  - What constitutes evidence of safety?
  - What limitations must we impose on programs for certification to work?
  - How do we construct such evidence?

Familiar theme in PL

"Well-typed programs cannot go wrong."

— Milner's adage

Type safety theorem:

- Progress

  From any well-typed (nonterminal) expression, one can take an execution step.

- Subject reduction

  From a well-typed expression, an execution step results in another well-typed expression.

- Corollary

  No well-typed expression can become "stuck".

Why is this safety?

- Design the operational semantics to exclude any "bad" operations.
  - Example: operational semantics provides no facility for arbitrary jumps.
  - Usually the case without any special effort.
- Thus, any program is safe so long as it stays within the operational semantics.
- Any program that "escaped" the semantics would formally become stuck.

$$P_1 \mapsto P_2 \mapsto P_3 \mapsto P_4 \not\mapsto \text{bad}$$

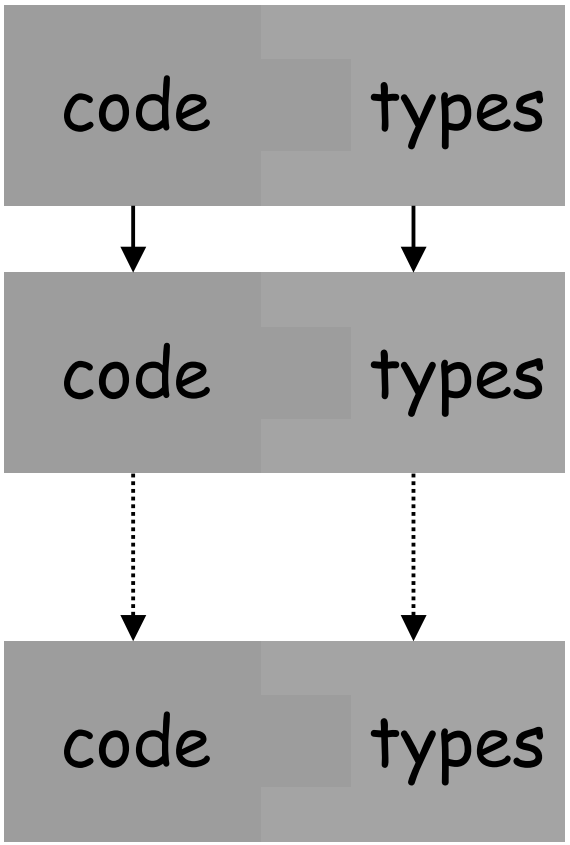
Typed Assembly Language

- Goals:
  - Type system for machine code.
  - Prove safety of TAL programs using the conventional PL techniques.
- Basis of a certified code architecture:
  - Type safety is the notion of safety.
  - The evidence is the program itself, plus typing annotations.
  - Type-preserving compiler constructs the evidence.

Type-preserving compilation

Preserve type information during compilation.

- Optimization
  - Use type information for enable additional optimization.
- Debugging
  - Typecheck intermediate representations to expose errors.



TAL allows us to reap the benefits of types throughout compilation.

Outline

- Overview of TAL
  - Focusing on similarities to conventional type theories
- Unique challenges facing low-level type system design
- Payoffs to using type theory

Example

Naive exponential function:

```
fun exp (n:int) =
   if n = 0 then
     1
   else
     2 * exp(n-1)
```

Continuation-passing style

Pass control with jumps, rather than function calls.

```
fun exp (n:int,k:int→0) =
   if n = 0 then
     k(1)
   else
     let x = n-1
     in
       exp(x,
           λy. let z = 2*y
               in
                  k(z)     )
```

Closed code blocks

```
fun exp[α](s:α,n:int,k:(α*int)→0) =
   if n = 0 then
     k(s,1)
   else
     let x  = n-1
     and s' = (k,s)
     in
       exp[((α*int)→0)*α]
          (s', x,
           λ(s',y). let z = 2*y
                    and k = π1(s')
                    and s = π2(s')
                    in
                      k(s,z)      )
```

TAL

Register-passing style (single argument functions) and assembly language notation.

```
exp: code[ρ]{sp:ρ,r1:int,r2:{sp:ρ,r1:int}→0}.
  bz r1,basecase[...]
  sub r1,r1,1           ; x  = n-1
  push r2               ; s' = (k,s)
  mov r2,cont[ρ]
  jmp exp[{sp:ρ,r1:int}→0::ρ]
cont: code[ρ]{sp:({sp:ρ,r1:int}→0::ρ),r1:int}.
  imul r1,r1,2          ; z = 2*y
  pop r2                ; k = π1(s'), s = π2(s')
  jmp r2
```

TAL's functional core

The heart of TAL is a lambda calculus constrained by:

- Continuation-passing style
- Closed code blocks
- Register-passing style

Of course, the devil is in the details . . .

Challenges

"Language design in an uncooperative environment."

- In user-level language design, you want high-level abstractions that accomplish a lot.
  - Convenient for programmers.
  - Leads to nice type systems!
- For machine code, that's exactly what you do not want.
  - Machine code does not have large atomic operations.
- Safety of machine code can depend on very complicated invariants.

Basic strategy

- Address the requirements of low-level code with sophisticated type systems.
  - Need not be programmer usable.
  - But must be automatically checkable.
- Sacrifice some expressive power and limit how instructions may be used.
  - Ultimate fallback: atomic instruction sequences.

Art: designing tractable type systems that allow more expressive power.

- (Nothing very sophisticated in this talk.)

Typical problem: typing memory

- Need to know the type of contents of any memory location you can read.
- Due to unknown aliasing, this is very difficult to track.
- Can be unsound if you don't:

```
; suppose r1 : <{...}→0,int>
mov r2,r1    ; r2 : <{...}→0,int>
st r1(0),12  ; r1 : <int,int>, r2 unchanged
ld r3,r2(0)  ; r3 : {...}→0 but contains 12
jmp r3       ; BAD
```

Fallback solution

Borrow solution from ML references:

- Reference cells have a single, fixed type.
- ✔ Aliases are unimportant, can't change the type.
- ✘ To establish the invariant, need an atomic allocate/initialize sequence.
  - Reduce expressiveness, impede optimization.

```
...
malloc r1,2      ⎫
st r1(0),12      ⎬ atomic code
st r1(1),15      ⎭ sequence
; conclude r1:<int,int>
...
```

Initialization flags [TOPLAS 99]

Separate allocation from initialization.

- Memory cells have a single, fixed type.
- Initialization flags track whether they are filled yet.
- Aliases may have inaccurate initialization information, but only conservatively.
- Allocation is still atomic.

```
...
malloc r1,<int,int>  ; r1 : <int⁰,int⁰>
st r1(0),12          ; r1 : <int¹,int⁰>
st r1(1),15          ; r1 : <int¹,int¹>
...
```

Beyond initialization flags

- Allow the type of a memory cell to be changed.
  - Stacks [TIC98]
    - Re-use stack slots.
  - Alias types [Smith, Walker, Morrisett 2000]
    - Track aliased pointers and allow modification when all aliases are known.
- Allow explicit freeing of memory.
  - Capabilities [POPL99]
    - When memory is freed, revoke the capability to access it.
- Challenge: typecheck a garbage collector.

Soapbox

- You get nice payoffs from using type theory.
  - When you live right, good things happen.
- Key example: parametricity [Reynolds 83]
  - Prototypical instance:
    In the polymorphic lambda calculus, any function with type $\forall\alpha.\alpha\rightarrow\alpha$ must be the identity.
  - Provides the foundation for data abstraction.
  - Can use parametricity to establish important properties of TAL programs.
    - With no special design to obtain those properties!

Callee-saves registers

- Can specify that **r1** is callee-save with the type:

$$\forall\alpha.\{\mathtt{r1}:\alpha,\mathtt{r2}:\{\mathtt{r1}:\alpha\}\rightarrow\mathtt{0}\}\rightarrow\mathtt{0}$$

in (↑ α) out (↓ α)

- Naive reading allows the function to return a different value of type α.
- Parametricity says that the function must returns the *same* value.
  - Callee-saves is enforced, at no cost.

Problems in type system design

- Expressiveness
  (allow more correct code)
  - Memory flexibility
  - Sophisticated invariants
    (e.g., **r1** : if P then τ_1 else τ_2)
- Security
  (disallow more incorrect code)
  - Resource bounds [POPL00]
  - More complex safety policies [Walker 2000]
    (e.g., no network send after disk read)
  - Correctness properties
- Certified compilation

Moral

- Can do certified code using standard type-theoretic techniques.
- Although the type systems can be novel, the means of thinking about them are well-understood.
- Get the usual type-theoretic payoffs (*e.g.,* parametricity).
- Evidence is just the program plus type annotations.

For more information

- Morrisett *et al.*, 1999.
  From System F to Typed Assembly Language.
- Crary and Morrisett, 1999.
  Type Structure for Low-Level Programming Languages.
- Papers and software are available at http://www.cs.cornell.edu/talc