Language-Based Security: Typed Assembly Language

Fritz Henglein

Software Security, DIKU

2025-09-26

All slides by Greg Morrisett

(Cornell University), 2001

Papers for this Lecture

G. Morrisett *et al.* Stack-Based Typed Assembly Language. *J. of Functional Programming (JFP), 12(1):43—88, 2002*.

G. Morrisett *et al.* From System-F to Typed Assembly Language. *ACM Transactions on Prog. Lang. and Systems (TOPLAS)*, 21(3):528—569, May 1999.

The Big Question?

From a least privilege perspective, many systems should be decomposed into separate processes. But if the overheads of communication (i.e., traps, copying, flushing TLB) are too great, programmers won’t do it.

Can we achieve isolation and cheap communication?

Cross Domain Calls

Must be paranoid when writing kernel code.

- consider a call to read a file
- is the argument a valid file descriptor?
- was the file opened for reading?
- is the file still open?

These checks are a real part of the cost:

- must keep state in the kernel (e.g., “real” file descriptors)
- must do checks at each system call

Consider:

```
signature FileDesc =
  sig
    type ‘a fd
    type read
    type write
    val open_read  : string -> read fd
    val open_write : string -> write fd
    val read       : read fd -> string
    val write      : write fd -> string
    val close      : ‘a fd -> unit
  end
```

ADTs as Capabilities

- Caller can’t spoof a file descriptor.
  - the types read, write, and fd are *abstract*
    - only the implementer of the interface can manufacture values of these types.
  - Only file descriptors opened via open_read can be passed to read.
    - since open_read is the only way to create a “read fd”.
  - polymorphism allows us to re-use code
    - close doesn’t care whether it’s given a read or write file descriptor.
- So fewer checks are needed.
    - close can’t tell that it’s given an *open* file descriptor
    - this is possible with a different signature...

Type Safety

To support type abstraction, we must also support *type safety*:

- at run-time, we shouldn't apply an operation to values of the wrong type.
  - programs must not "go wrong"
  - "wrong" = ??? "type" = ???
- formally:
  - construct a high-level semantics for the language where run-time values are tagged with their type.
  - operations must be *total* for values of the right type.
- notice that we haven't defined a "type"
  - we could label all values with the same type
  - that would necessitate making every operation total
  - that's what hardware does!

Types in Practice

Usually, we have a notion of types as predicates or sets of values.

- e.g., int, float, string, int -> int, etc.

Abstract types give us the power to extend the set of primitive types with uninterpreted predicates.

- so we can build domain-specific abstractions (e.g., fd, set, GUIcontext, etc.)

It's possible to take a more *semantic* view of types where we define the predicates in some logic or type theory.

- e.g., NuPRL, Coq, HOL, etc.
- we'll get back to this later on. For now....

Quote of the Day

“Type systems for programming languages are a syntactic mechanism for enforcing abstraction.”

J. Reynolds

Static vs. Dynamic Typing

- Dynamic typing (e.g., Scheme)
  - before executing an operation, check that the values have the right type.
  - requires run-time tags and run-time checks.
  - but doesn’t rule out a program unless it actually does something bad.
  - not unlike an IRM...
- Static typing (e.g., ML)
  - at compile/load time, check that the code is type-correct.
  - these checks are necessarily conservative in that they may rule out perfectly good code.
  - but doesn’t require run-time tags/checks
  - failures caught before code is deployed

Static vs. Dynamic

In practice, a mix of static and dynamic typing.

Scheme:

- static: well-formed S-expression
- dynamic: everything else

ML:

- static: most operations
- dynamic: div by zero, array bounds error, raise exn, some instances of pattern matching, etc.

Java:

- static: is it of the right “class”
- dynamic: div by zero, array bounds error, null pointer, downcasts, array assignment, etc.

Type-Safe Languages

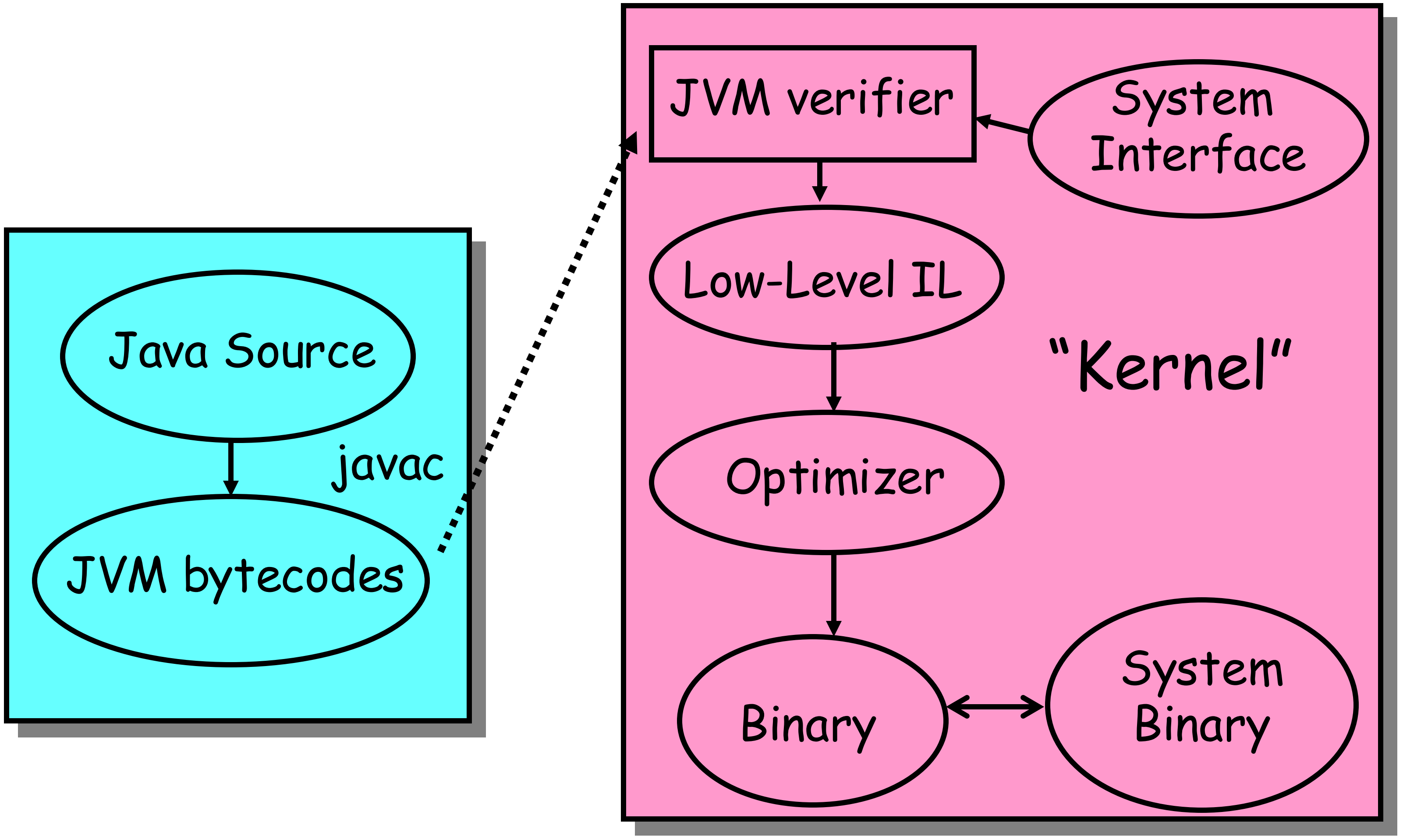
Traditional notions of type-safety come with a lot of benefits:

- no buffer overruns (an array subscript a[i] is only defined when i is in range for the array a.)
- no worries about self-modifying code or wild jumps, etc.

But proving type safety is *hard*.

- usually a very long, complicated, tedious proof.
- rarely (if ever) done for a full-scale language.
- never (to my knowledge) of an implementation.

Type-Based Protection (JVM)



Type-Safety & the JVM

The Java security model depends upon type-safety to ensure:

- you can build abstract types
- no wild reads/writes/jumps
- security manager's state is protected
- etc.

The type-safety of Java isn't the issue – rather, the type-safety of the JVM.

- check that the bytecodes are well-formed.
- could just as well be output of a Scheme compiler.
- bug in the type-checker or JVM implementation could lead to a security hole.

Compiling to the JVM

The JVM type-system isn't all that different from Java's.

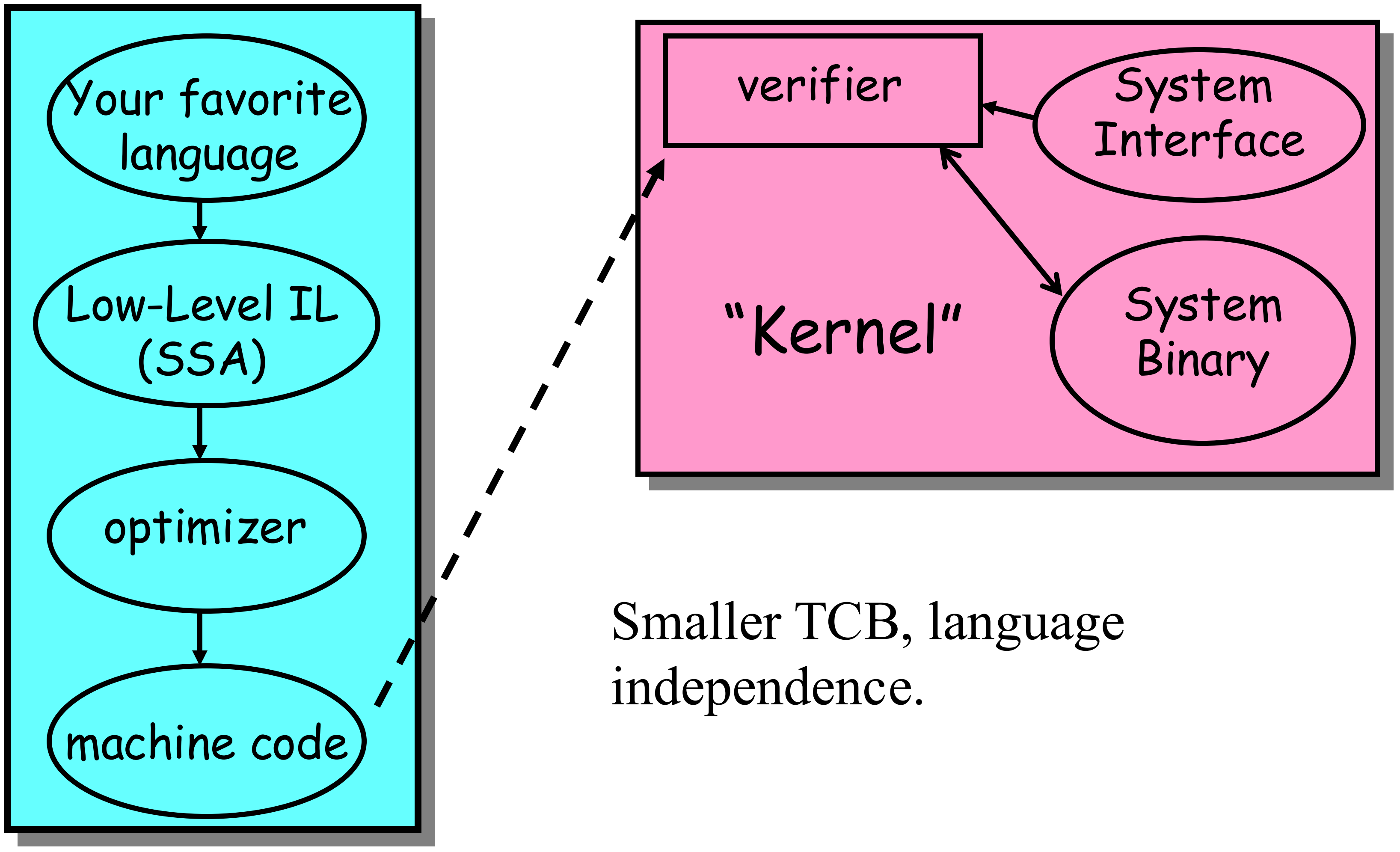
As a result, compiling other languages to Java isn't necessarily all that great.

- e.g., no tail-calls in the JVM so Scheme and ML are hosed...
- no parametric polymorphism, no F-bounded subtyping, limited modules, etc.

In addition, the operations of the JVM are relatively high-level, CISC-like.

- method call/return are primitives
- interpreter or JIT is necessary

Ideally:



Smaller TCB, language independence.

Typed Assembly Language

Two goals:

- Get rid of the need for a trusted interpreter or compiler
  - type-check the actual code that will run.
  - try not to interfere with traditional optimizations.
- Provide generic *type constructors* for encoding high-level language type systems.
  - a more “RISC” philosophy at the type-level
  - better understanding of the process of compilation
  - better understanding of inter-language relationships.

What Was Done:

Theory:

- small RISC-style assembly language
- compiler from ML-like language to TAL
- soundness and preservation theorems

Practice:

- most of IA32 (32-bit Intel x86)
- more type constructors (array,+,μ,modules)
- prototype Scheme, Safe-C compilers

TAL-0

Registers: $r \in \{\mathbf{r1},\mathbf{r2},\mathbf{r3},...\}$

Labels: $L \in \mathit{Identifier}$

Integer: $n \in [-2^{k-1}..2^{k-1})$

Blocks: $B ::= \texttt{jmp}\ v \mid \iota\ ;\ B$

Instrs: $\iota ::= aop\ r_d,r_s,v \mid bop\ r,v \mid \texttt{mov}\ r,v$

Operands: $v ::= r \mid n \mid L$

Arithmetic Ops: $aop ::=$ **`add`** | **`sub`** | **`mul`** | ...

Branch Ops: $bop ::=$ **`beq`** | **`bne`** | **`bgt`** | **`bge`** | ...

Simple Program

```
; fact(n,a) = if (n <= 0) then a else fact(n-1,a*n)
;    r1 holds n, r2 holds a, r31 holds return address
;    which expects the result in r1
fact:  sub r3,r1,1     ; r3 := n-1
       ble r3,L2       ; if n < 1 goto L2
       mul r2,r2,r1    ; a := a*n
       mov r1,r3       ; n := n-1
       jmp fact        ; goto fact
L2:    mov r1,r2       ; result := a
       jmp r31         ; jump to return address
```

TAL-0 Abstract Machine

Model evaluation as a transition function mapping machine states to machine states: $M \to M$

Machine state: $M = (H, R, B)$

B is a basic block (corresponding to the current program counter.)

R maps registers to values

- ints n or labels L

H is a partial map from labels to labeled basic blocks (L:B)

Semantics

$(H, R, \texttt{mov}\ r_d,v\ ;\ B) \rightarrow (H, R[r_d{:=}\underline{R}(v)], B)$

where $\underline{R}(r) = R(r)$ else $\underline{R}(v) = v$

$(H, R, \texttt{add}\ r_d,r_s,v\ ;\ B) \rightarrow (H, R[r_d{:=}n], B)$

where $n = \underline{R}(v) \oplus R(r_s)$

$(H, R, \texttt{jmp}\ v) \rightarrow (H, R, B)$

where $\underline{R}(v) = L$ and $H(L) = B$

$(H, R, \texttt{beq}\ r,v\ ;\ B) \rightarrow (H, R, B)$

where $R(r) \neq 0$

$(H, R, \texttt{beq}\ r,v\ ;\ B) \rightarrow (H, R, B')$

where $R(r) = 0$, $\underline{R}(v) = L$, and $H(L) = B'$

Types for TAL-0

The abstract machine makes a distinction between addresses and integers.

- This allows us to model jumping to an arbitrary location as an error.
- In particular, the machine is *stuck* if:
  - $(H, R,$ **`add`** $r_d, r_s, v\ ;\ B)$ and r_s or v aren't ints
  - $(H, R,$ **`jmp`** $v)$ and v isn't a label, or
  - $(H, R,$ **`beq`** $r, v\ ; B)$ and r isn't an int or v isn't a label
- So the type system needs a distinction between integers and labels.

Basic Type Structure

$t ::= int \mid \Gamma$

where $\Gamma = \{ r_1{:}t_1, r_2{:}t_2, \; r_3{:}t_3, \ldots\}$

A value with type $\{$ **`r1`** $: t_1,$ **`r2`** $: t_2,$ **`r3`** $: t_3, \ldots\}$ must be a label, which when you jump to it, expects you to at least have values of the appropriate types in the corresponding registers.

You can think of a label as a function that takes a record of arguments

- the function never returns – it always jumps off
- we assume record subtyping – we can pass a label more arguments than it needs

Simple Program with Types

```
fact:{r1:int,r2:int,r31:{r1:int}}
         ; r1 = n, r2 = accum, r31 = return address
         sub   r3, r1, 1    ; {r1:int,r2:int,r31:{r1:int},r3:int}
         ble   r3, L2
         mul   r2, r2, r1
         mov   r1, r3
         jmp   fact
L2:{r2:int, r31:{r1:int}}
         mov r1, r2
         jmp r31
```

Mis-Typed Program

```
fact:{r1:int,r31:{r1:int}}
        ; r1 = n, r2 = accum, r31 = return address
        sub   r3, r1, 1     ; {r1:int,r31:{r1:int},r3:int}
        bge   r1, L2
        mul   r2, r2, r1    ; ERROR!  r2 doesn't have a type
        mov   r1, r3
        jmp   L1
L2:{r2:int, r31:{r1:int}}
        mov r1, r2
        jmp r1            ; ERROR!  r1 isn't a valid label
```

Typing Contexts

- We need to keep track of:
  - the types of the registers at each point in the code (type-states)
  - the types of the labels on the code

Heap Types:

Ψ will map labels to label types.

Register Types:

Γ will map registers to types.

Typing Operands

- $\Psi;\Gamma \blacktriangleright n : int$ integer literals are ints
- $\Psi;\Gamma \blacktriangleright r : \Gamma(r)$ lookup register type in Γ
- $\Psi;\Gamma \blacktriangleright L : \Psi(L)$ lookup label type in Ψ
- $\Psi;\Gamma \blacktriangleright v : t_2$ if $\Psi;\Gamma \blacktriangleright v : t_1$ and $t_1 \leq t_2$

where

$$\{r_1{:}t_1,\ldots,r_n{:}t_n,r_{n+1}{:}t_{n+1}\} \leq \{r_1{:}t_1,\ldots,r_n{:}t_n\}$$

Typing Instructions

The judgment for instructions looks like:

$$\Psi \blacktriangleright \iota : \Gamma_1 \rightarrow \Gamma_2$$

where Γ_1 describes the registers on input to the instruction, and Γ_2 describes the registers on output.

The heap type (Ψ) stays invariant, reflecting the fact that we're not going to allow the types of heap objects to change over time.

Typing Rules: Instructions

$\Psi \blacktriangleright \mathit{aop}\ r_d, r_s, v : \Gamma \rightarrow \Gamma[r_d := \mathit{int}]$
    if $\Psi;\Gamma \blacktriangleright r_s : \mathit{int}$ and $\Psi;\Gamma \blacktriangleright v : \mathit{int}$

$\Psi \blacktriangleright \mathit{bop}\ r, v : \Gamma \rightarrow \Gamma$ if $\Psi;\Gamma \blacktriangleright r : \mathit{int}$ and $\Psi;\Gamma \blacktriangleright v : \Gamma$

$\Psi \blacktriangleright$ **`mov`** $r, v : \Gamma \rightarrow \Gamma[r_d := t]$ if $\Psi;\Gamma \blacktriangleright v : t$

$\Psi \blacktriangleright \iota : \Gamma_1 \rightarrow \Gamma_2$ if $\Psi \blacktriangleright \iota : \Gamma_3 \rightarrow \Gamma_4$
    and $\Gamma_1 \leq \Gamma_3$ and $\Gamma_4 \leq \Gamma_2$

Typing Rules: Basic Blocks

Jumps don't return, so we consider them to return anything (alternatively void)

$\Psi \blacktriangleright \texttt{jmp}\ v : \Gamma_1 \to \Gamma_2$ if $\Psi;\Gamma \blacktriangleright v : \Gamma_1$

Basic blocks are just compositions:

$\Psi \blacktriangleright \iota\,; B : \Gamma_1 \to \Gamma_3$ if $\Psi \blacktriangleright \iota : \Gamma_1 \to \Gamma_2$ and $\Psi \blacktriangleright B : \Gamma_2 \to \Gamma_3$

Subtyping:

$\Psi \blacktriangleright B : \Gamma_1 \to \Gamma_2$ if $\Psi \blacktriangleright B : \Gamma_3 \to \Gamma_4$ and $\Gamma_1 \leq \Gamma_3$ and $\Gamma_4 \leq \Gamma_2$

Typing Rules: Machine

$\blacktriangleright H : \Psi$ if

- dom(H) = dom(Ψ) and
- for all labels L such that $H(L) = B$, and $\Psi(L) = \Gamma_1$, there exists a Γ_2 such that $\Psi \blacktriangleright B : \Gamma_1 \to \Gamma_2$.

$\Psi \blacktriangleright R : \Gamma$ if

- for all registers r in dom(Γ), $\Psi;\{\} \blacktriangleright R(r) : \Gamma(r)$.

$\blacktriangleright (H,R,B)$ if

- there exists a Ψ such that $\blacktriangleright H : \Psi$, and
- there exists a Γ_1 such that $\Psi \blacktriangleright R : \Gamma_1$, and
- there exists a Γ_2 such that $\Psi \blacktriangleright B : \Gamma_1 \to \Gamma_2$

Theorems

Preservation (a.k.a. Subject Reduction):

if $\blacktriangleright M_1$ and $M_1 \rightarrow M_2$, then $\blacktriangleright M_2$.

Progress:

if $\blacktriangleright M_1$ then there exists an M_2 such that $M_1 \rightarrow M_2$.

Corollaries:

- a well-typed program can’t get stuck.
- all jumps are to valid labels.
- all arithmetic is done with integers (not labels).

Scaling It Up

The simple abstract machine and type system can be scaled in many directions:

- more primitive types & opn's (e.g., floats, jal, break, etc.)
- memory-allocated values (e.g., tuples and arrays)
- a control stack for procedures
- more polymorphism

We'll work backwards...

TAL-1: polymorphism

- Two changes to types:
  - add type variables to types: α
    - these are treated abstractly
    - but allow more code re-use
    - as we'll see, they come in handy elsewhere...
  - let labels be polymorphic over register types: $\forall\alpha,\beta.\{r1{:}\ \alpha,\ r2{:}\ \beta,\ r3{:}\ \{r1{:}\beta,\ r2{:}\alpha\}\}$
    - might describe a swap function that swaps the values in registers r1 and r2, for values of any two types. Note that register r3 contains the "return address" which expects the values to be swapped.
    - technical hitch: need to explicitly instantiate type variables due to potential recursion among labels.
- Dynamic semantics remains unchanged.

Example Polymorphism

```
swap:∀α,β.{r1:α, r2:β, r31:{r1:β,r2:α}}
      mov   r3, r1        ; {r1:α, r2:β, r31:{r1:β,r2:α}, r3:α }
      mov   r1, r2
      mov   r2, r3
      jmp   r31

swap_ints: {r1:int, r2:int, r31:{r1:int,r2:int}}
      jmp   swap[int,int]

swap_int_and_label: {r1:int,r2:{r2:int}}
      mov   r31, L
      jmp   swap[int,{r2:int}]
L: {r1:{r2:int},r2:int}
      jmp   r1
```

Callee-Saves Registers

```
f:∀α.{r1:int, r5:α, r31:{r1:int,r5:α}}

foo:   mov r5, 255     ; want to preserve r5 across call to f
       mov r1, 5
       mov r31, L
       jmp f[int]      ; f[int] : {r1:int, r5:int, {r1:int,r5:int}}
L:     {r1:int, r5:int}
       mul r3, r1, r5
       ...
```

Moral: polymorphism can be used for more than just code-reuse. It can also be used to force a procedure to "behave well" in some circumstances.

TAL-2: add a stack

$M = (H,R,S,B)$

S ::= nil | v::S

– model the stack as a list of values

ι ::= **`salloc`** n | **`sfree`** n | **`sld`** r_d,n | **`sst`** r_s,n

– new instructions: allocate *n* words on the stack, free *n* words, load the *nth* word from the top of the stack, store into the *nth* word.

– get stuck if we free too much or try to read/write locations too deep in the stack.

Simple Stack-Based Program

```
factrec:    sub r3,r1,1       ; x == 1 ?
            ble r3,L1         ; no, goto L1
            jmp r31           ; yes, return
L1:         salloc 2          ; allocate space for frame
            sst r31,0         ; save return address
            sst r1,1          ; save x
            mov r1,r3         ; x := x-1
            mov r31,RA        ; return address := RA
            jmp   factrec     ; do recursive call, result in r1
RA:         sld   r2,1        ; restore x into r2
            sld   r31,0       ; restore original return addr.
            mul   r1,r1,r2    ; res := x * fact(x-1)
            jmp r31           ; return
```

Semantics for Stack Opn’s

$(H, R, S, \texttt{salloc}\ 3\ ;\ B) \rightarrow (H, R, 0{::}0{::}0{::}S, B)$

$(H, R, v_1{::}v_2{::}v_3{::}S, \texttt{sfree}\ 3\ ;\ B) \rightarrow (H,R,S,B)$

$(H, R, S, \texttt{sld}\ r,3\ ;\ B) \rightarrow (H,R[r := v_3],S,B)$

where $S = v_1{::}v_2{::}v_3{::}S'$

$(H, R, S_1, \texttt{sst}\ r,3\ ;\ B) \rightarrow (H,R,S_2,B)$

where $S_1 = v_1{::}v_2{::}v_3{::}S'$

and $S_2 = v_1{::}v_2{::}R(r){::}S'$

Remarks

The stack operations have a 1-to-1 correspondence with RISC instructions.

- **`salloc`** corresponds to subtracting n from a stack pointer register (e.g., **`sub sp,sp,n`**)
- **`sfree`** corresponds to adding n to the stack pointer (e.g., **`add sp,sp,n`**)
- **`sst`** corresponds to writing a value into offset n from the stack pointer (e.g., **`st sp(n),r`**)
- **`sld`** corresponds to reading a value from offset n relative to the stack pointer (e.g., **`ld r,sp(n)`**)

CISC-like instructions (e.g., push/pop) can be synthesized.

Typing the Stack

Stack types:

$$s ::= nil \mid t::s \mid \rho$$

- The *nil* type represents the empty stack.
- The type *t::s* represents a stack *v::S* where *t* is the type of *v* and *s* is the type of *S.*
- The type ρ is a stack type variable that describes some unknown “tail” in the stack.
  - In addition, we’ll let label types be polymorphic over stack types.

Typing Factrec (Bug)

```
factrec:∀ρ.{sp:ρ, r1:int, r31:{r1:int,sp:ρ}}
          sub r3,r1,1        ; r3:int
          bne r3,L1[ρ]
          jmp r31
L1:∀ρ.{sp:ρ, r1:int, r3:int, r31:{r1:int,sp:ρ}}
          salloc 2           ; sp: int::int::ρ
          sst r31,0          ; sp: {r1:int,sp:ρ}::int::ρ
          sst r1,1
          mov r1,r3
          mov r31,RA[ρ]      ; r31: {sp:{r1:int,sp:ρ}::int::ρ,r1:int}
          jmp factrec[{r1:int,sp:ρ}::int::ρ]
RA:∀ρ.{sp:{r1:int,sp:ρ}::int::ρ, r1:int}
          sld   r2,1         ; r2:int
          sld   r31,0        ; r31:{r1:int,sp:ρ}
          mul   r1,r1,r2
          jmp r31
```

Typing Factrec Corrected

```
factrec:∀ρ.{sp:ρ, r1:int, r31:{r1:int,sp:ρ}}
      sub r3,r1,1       ; r3:int
      bne r3,L1[ρ]
      jmp r31
L1:∀ρ.{sp:ρ, r1:int, r3:int, r31:{r1:int,sp:ρ}}
      salloc 2          ; sp: int::int::ρ
      sst r31,0         ; sp: {r1:int,sp:ρ}::int::ρ
      sst r1,1
      mov r1,r3
      mov r31,RA[ρ]     ; r31: {sp:{r1:int,sp:ρ}::int::ρ,r1:int}
      jmp factrec[{r1:int,sp:ρ}::int::ρ]
RA:∀ρ.{sp:{r1:int,sp:ρ}::int::ρ, r1:int}
      sld   r2,1        ; r2:int
      sld   r31,0       ; r31:{r1:int,sp:ρ}
      mul   r1,r1,r2
      sfree 2           ; sp:ρ
      jmp r31
```

The Theorems Carry Over

- Typing ensures we don't get stuck
  - e.g., try to write off the end of the stack
  - But it doesn't ensure the stack stays within some quota...
- We can adequately encode procedures.
- With a bit more complication, we can deal with exceptions, threads, and continuations (see the STAL paper for details.)

Things to Note

- We didn't have to bake in a notion of procedure call/return. Jumps were good enough.
  - side effect: tail-calls are a non-issue
- Polymorphism and polymorphic recursion are crucial for encoding standard procedure call/return.
- When combined with the callee-saves trick, we can code up calling conventions.
  - arguments on stack or in registers?
  - results on stack or in registers?
  - return address? caller pops? callee pops?
  - caller saves? callee saves?
- It's the orthogonal combination of typing features that makes things scale well.

Tuples and Arrays

The register file and stack gives us some local storage for word-sized values.

- that space can be recycled for values of different types.
- critical trick: can’t create pointers to these values (i.e., aliases)

What about aggregates?

- e.g., tuples, records, arrays, objects, datatypes, etc.
- TAL (and JVM) model is to place these “large” values in the heap and refer to them via pointer.
- This introduces the potential for aliasing.
  - issue already present for label values
- Recycling this memory won’t be as easy

Tuples are Easy

Let heap H map labels to either blocks of code or tuples of values: $\langle v_0, v_1, \ldots, v_{n-1} \rangle$

- note that the values are either ints or labels, and that we consider labels to be abstract pointers.

Add instructions for creating tuples, and for accessing the ith component of a tuple:

$\iota ::= \mathbf{\texttt{talloc}}\ r_d, n \mid \mathbf{\texttt{ld}}\ r_d, r_s(n) \mid \mathbf{\texttt{st}}\ r_d(n), r_s$

Add tuple types: $\langle t_0, t_1, \ldots, t_{n-1} \rangle$

Semantics for Tuple Opn's

$(H, R, v_0::v_1::v_2::S,$ `talloc` $r_d,3\ ;\ B) \rightarrow$

$\quad (H[L=\langle v_0,v_1,v_2\rangle],R[r := L],S,B)$

$\quad$ where L is a fresh label (i.e., not in Dom(H)).

$(H, R, S,$ `ld` $r_d,r_s(2)\ ;\ B) \rightarrow (H,R[r_d := v_2],S,B)$

$\quad$ where $H(R(r_s)) = \langle v_0,v_1,v_2\rangle$

$(H[L=\langle v_0,v_1,v_2\rangle], R, S,$ `st` $r_d(2),r_s\ ;\ B) \rightarrow$

$\quad (H[L=\langle v_0,v_1,R(r_s)\rangle],R,S,B)$

$\quad$ where $R(r_d) = L$

Typing for Tuple Opn's

$\Psi \blacktriangleright$ **`tmalloc`** $r_d, n : \Gamma \rightarrow \Gamma[sp{:=}s,\ r_d{:=}\langle t_1, t_2, \ldots, t_n\rangle]$

where $\Gamma(sp) = t_1{::}t_2{::}\cdots{::}t_n{::}s$

$\Psi \blacktriangleright$ **`ld`** $r_d, r_s(n) : \Gamma \rightarrow \Gamma[r_d{:=}\ t]$

if $\Psi;\Gamma \blacktriangleright r_s : \langle t_0, t_1, \ldots, t_{m-1}\rangle$ and $0 <= n < m$.

$\Psi \blacktriangleright$ **`st`** $r_d(n), r_s : \Gamma \rightarrow \Gamma$

if $\Psi;\Gamma \blacktriangleright r_d : \langle t_0, t_1, \ldots, t_{m-1}\rangle$, $0 <= n < m$, and

$\Psi;\Gamma \blacktriangleright r_s : t_n$.

Remarks

- The load and store operations correspond to conventional RISC instructions.
- The talloc does not – typically, this would be implemented by a call into the runtime to atomically allocate and initialize the tuple.
- There's no tfree...
  - rather, we'll rely upon a garbage collector to reclaim heap storage.
- And you can't update a component in a tuple with a value of a different type...
  - rather the types of tuples are *invariant*.
  - same is true for code and other heap objects.

Why no tfree?

When we update a register or stack slot, we update the corresponding register type or stack slot type.

- only one occurrence of the type so it's easy to update it.
- such values are called *linear*.

But we can create many copies of pointers. If we recycle heap memory, then we need to update the types of *all* copies of the pointers.

- This would require accurately tracking the actual *value* of a label in it's type, which is hard to do since we can dynamically create new pointers via tmalloc at run-time.
- There are approaches along these lines that work (c.f., Regions of Tofte et al., Capabilities, Alias types, etc.)
- But they're awfully complicated.
- So a simple approach (only registers/stack linear) seems nice.

Arrays

Hard issues:

- need to allocate and initialize storage of unknown size.
- each array subscript operation must be in bounds.
- in turn, this implies we need size information at run-time.

Typical solution: special operations

- new_array: (int->α) -> α array
- asub, aupd: built-in bounds checks
- arrays carry size information with them

Advanced solutions use *dependent types* (e.g., Xi & Pfenning's DTAL.)

- the size of an array can depend upon the value of some other variable
- refinements or comprehensions let us relate values:
  r1:{i:int | i < j}, r2:array(int)[j]

More Language Features

Closures, classes, objects, modules, etc?

- delicate combinations of advanced typing constructs such as existential types, F-bounded polymorphism, translucent sums, and recursive types.
- these are well-studied in the semantics literature.
- but semantic encodings don't always match with efficient implementations.
- much current research tries to address this.

TAL and the JVM

The principles behind TAL and the JVM (or Microsoft's CLR) aren't too different.

TAL concentrated on orthogonal typing components and a principled approach to compiling "type-theoretic" languages.

The JVM (and CLR) focused on a particular class of OO-based languages with very definite implementation strategies.

To be fair...

The JVM and CLR provide many more features than TAL that constrain things:

- accurate garbage collection, debugging, and security manager all need to be able to walk the stack
- threads, synchronization
- class loaders
- security managers
- tons of libraries
- details about file formats, etc.

Some Practical Issues

As you go from a high-level language to a low-level TAL-like language, the types (predicates) become *BIG.*

Careful engineering is required:

The Popcorn Compiler (PII266)

object code: 0.55MB, 39 modules

naïve encoding: 4.50MB checking: 750s

optimized encoding: 0.27MB checking: 22s

Example from Popcorn

Source Type: int -> bool

TAL Type:

```
All a:T,b:T,c:T,r1:S,r2:S,e1:C,e2:C.
   {ESP: {EAX:bool, M:e1+e2, EBX:a,ESI:b,EDI:c,
              ESP:int::r1@{EAX:exn,ESP:r2,M:e1+e2}::
              r2
             }::int::r1@{EAX:exn,ESP:r2,M:e1+e2}::r2,
    EBP: sptr{EAX:exn,ESP:r2,M:e1+e2}::r2,
    EBX:a, ESI:b, EDI:c, M:e1+e2}
```

Compressing Types

- Gzip: no help during verification
- Tailor types to language/compiler
  - e.g., fix the calling convention..
  - but has obvious drawbacks.
- Higher-order type constructors
  - fairly effective, good for readability
- Hash-cons (i.e., use graphs)
  - extremely effective, fast type equality

Compressing Types, cont'd

- Reconstruction
  - can be very effective w.r.t. both space & time
  - but must be careful not to increase TCB
  - see Necula's LF-i for a good example of how to do things right.
- Avoid substitutions & reductions
  - memoize reductions (doesn't work for TILT)
  - keep track of free variables
  - see Shao's FLINT compiler

One lesson: unlike provers, optimize for success.

A Subtlety with TAL

We could place typing annotations on each code label (the compiler knows them.)

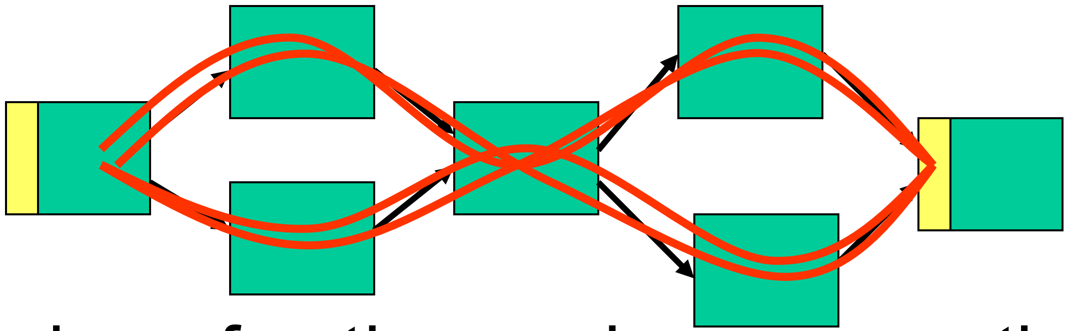
But we really only need annotations on:

- labels that are back-edges of loops
- labels that escape
- the rest can be inferred by flowing typing pre-conditions through all (forward) control-flow paths.

For TAL, leaving the other annotations off results in a 15% space savings.

However...

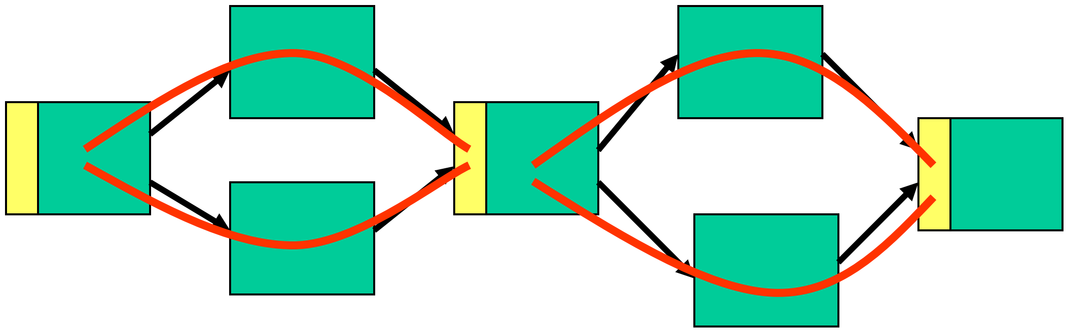
Verification must [re-]type-check each instruction on every control-flow path up to an annotation.



The number of paths can be exponential in the size of the code…and this bad case happens in real code (22 vs. 750 secs)

Bad News/Good News

Bad News: an optimal placement of invariants is NP complete.



Good News: a simple greedy heuristic seems to work well in practice.

TAL: Summary

- Types provide a syntactic framework for enforcing abstraction.
  - static typing holds the promise of cheap security enforcement.
- But until recently, had to buy into high-level languages to get static typing.
  - performance issues
  - TCB issues
- Systems like TAL or the JVM try to provide the advantages of types without the disadvantages.
- TAL concentrates on orthogonal typing constructs that can be used in concert to encode high-level language or compiler invariants.

Beyond TAL

- TAL logic:
  - Memory assertions via memory typing
  - Memory typing:
    - Abstract types (e.g. integers versus labels)
    - Each part of memory is typed independently
      - No relational properties between parts, or rather
      - Weak relational properties
        - » Equality (via parametric polymorphism, alias typing)
        - » Subtyping
        - » Refinement typing
    - No reasoning about arithmetic
      - Array bounds checking, semantic invariants, etc.

Beyond TAL

- Proof-Carrying Code (PCC)
  - Memory assertions via first-order logic
    - Relational constraints between different memory parts
    - Includes type abstraction (e.g. integers vs. labels)
  - Floyd-Hoare Logic (axiomatic semantics) for constructs; e.g. {Pre} `ld r2,r1(i)` {Post}
    - Formal proof representation: Verification conditions and inlined pre/post-conditions
    - Efficient verification of formal proof during code loading
  - Execution of ``raw'' code

Beyond TAL

- Separation Logic (SL)
  - Memory assertions via (first-order) separation logic
    - Built-in ‘separated conjunction P * Q’, asserting that P and Q hold for nonoverlapping memory
  - Automatic and interactive tools for program analysis
    - E.g. Infer (Facebook)

Beyond TAL

- Ownership types
  - Memory assertions via memory typing
    - With strong updates to typing assertions (as in TAL)
    - .. and admitting aliasing of updatable references (unlike TAL)
  - Example: Rust
    - Also has expressive static type system (type abstraction)
    - Extended with powerful interactive proof system (Project RustBelt, Project Iris)

Beyond TAL

- Certified compilation
  - Formal specification of axiomatic semantics of source language and of machine language
  - Formal (mechanized) proof that compiler always generates code with the same semantics
    - Cerifying compiler: Generate code, then check generated code whether it is sound
    - Certified compiler: One proof (of compiler), no subsequent checking of generated code required
  - Example: CompCert C-to-PowerPC compiler