typed assembly
language

```
res = 0
while a != 0:
    res = res + b
    a = a - 1
return res
```

```
prod: r3 = 0
      jump loop
loop: if r1 jump done
      r3 = r2 + r3
      r1 = r1 + -1
      jump loop
done:
```

$$
\begin{aligned}
r &::= r1 \mid r2 \mid \cdots \mid rk \\
v &::= n \mid r \\
\imath &::= r_d = v \\
&\quad \mid r_d = r_s + v \\
&\quad \mid \texttt{if } r \texttt{ jump } l \\
\mathtt{I} &::= \texttt{jump } l \\
&\quad \mid \imath ; \mathtt{I}
\end{aligned}
$$

```
foo:  dest = bar
      jump dest
```

indirect jump

$$
\begin{aligned}
r &::= r1 \mid r2 \mid \cdots \mid rk \\
v &::= n \mid r \mid \boxed{l} \\
\imath &::= r_d = v \\
&\quad \mid r_d = r_s + v \\
&\quad \mid \texttt{if } r \texttt{ jump } \boxed{v} \\
I &::= \texttt{jump } \boxed{v} \\
&\quad \mid \imath ; I
\end{aligned}
$$

```
dest = 0
jump dest
```

need to prevent bad jumps

$$\tau ::= \texttt{int} \mid \texttt{code}$$

```
prod: r3 = 0
      jump loop
loop: if r1 jump done
      r3 = r2 + r3
      r1 = r1 + -1
      jump loop
done:
```

type of **loop**?

depends on registers.

$$\tau \quad ::= \quad \texttt{int} \mid \mathsf{code}(\Gamma)$$

$$\Gamma : r \rightarrow \tau$$

```
prod: r3 = 0
      jump loop
loop: if r1 jump done
      r3 = r2 + r3
      r1 = r1 + -1
      jump loop
done:
```

```
loop: code({
  r1 : int,
  r2 : int,
  r3 : int
})
```

```
loop: if r1 jump done
      r3 = r2
      jump loop
```

type of **loop**?

depends on r2 and r3…

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

```
loop: if r1 jump done
      r3 = r2
      jump loop
```

```
loop: ∀α.code({
  r1 : int,
  r2 : α,
  r3 : α
})
```

TAL-0

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

$$\frac{}{\Psi \vdash n : \texttt{int}}$$

$$\frac{}{\Psi, l : \tau \vdash l : \tau}$$

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

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

$$\frac{}{\Psi; \Gamma, r : \tau \vdash r : \tau}$$

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

$$\boxed{\Psi \vdash \imath : \Gamma_1 \to \Gamma_2}$$

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash r_d = v : \Gamma \to \Gamma, r_d : \tau}$$

$$\frac{\Psi; \Gamma \vdash r_s : \mathtt{int} \quad \Psi; \Gamma \vdash v : \mathrm{code}(\Gamma)}{\Psi \vdash \mathtt{if}\ r_s\ \mathtt{jump}\ v : \Gamma \to \Gamma}$$

$$\frac{\Psi; \Gamma \vdash r_s : \mathtt{int} \quad \Psi; \Gamma \vdash v : \mathtt{int}}{\Psi \vdash r_d = r_s + v : \Gamma \to \Gamma, r_d : \mathtt{int}}$$

$$\boxed{\Psi \vdash \mathtt{I} : \tau}$$

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

$$\frac{\Psi \vdash \imath : \Gamma \to \Gamma_2 \qquad \Psi \vdash \mathtt{I} : \mathtt{code}(\Gamma_2)}{\Psi \vdash \imath; \mathtt{I} : \mathtt{code}(\Gamma)}$$

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

why **polymorphism**?

(why not **subtyping**?)

```
foo:  r1 = 1
      jump baz
bar:  r1 = qux
      jump baz
baz:
```

type of **baz**?

```
baz: ∀α.code({
  r1 : α
})
```

```
foo:  r1 = 1
      jump baz
bar:  r1 = qux
      jump baz
baz:
```

type of **baz**?

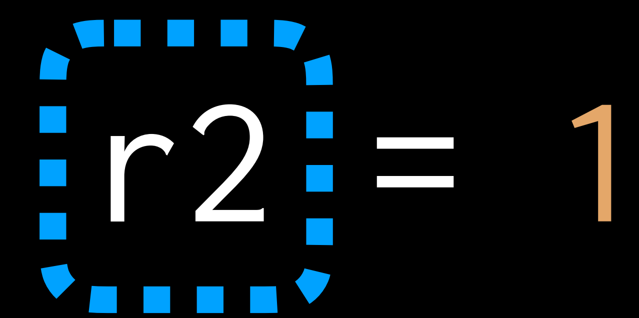
```
baz: code({
  r1 : ⊤
})
```

this does work…

```
bar:  r1 = r1 + 1
      ret
foo:  r1 = 1
      r2 = 1
      call bar
      r2 = r1 + 1
```

calling convention

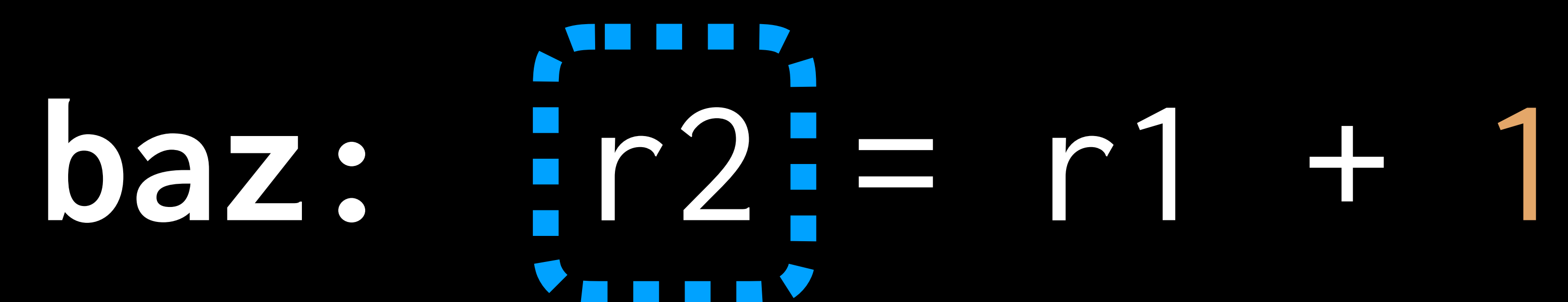
```
bar:  r1 = r1 + 1
      jump r3
foo:  r1 = 1
```



```
      r2 = 1
```

callee-saved registers

```
      r3 = baz
      jump bar
```



```
baz:  r2 = r1 + 1
```

```
bar:  r1 = r1 + 1
      jump r3
foo:  r1 = 1
      r2 = 1
      r3 = baz
      jump bar
baz:  r2 = r1 + 1
```

```
bar: ∀α.code({
  r1 : int,
  r2 : α,
  r3 : ∀β.code({
    r1 : int,
    r2 : α,
    r3 : β
  })
})
```

what about **memory**?

$$
\begin{aligned}
v &::= \cdots \mid p \\
\iota &::= \cdots \\
&\mid r_d = [r_s] \\
&\mid [r_d] = r_s \\
&\mid r_d = \mathtt{alloc}(\tau) \\
\tau &::= \cdots \mid \mathtt{ptr}(\tau)
\end{aligned}
$$

```
r3 = 0
[r1] = r3
r4 = [r1]
jump r4
```

will jump to 0 at runtime

```
r3 = 0          r3 : int
[r1] = r3       r1 : ptr(int)
r4 = [r1]       r4 : int
jump r4         invalid!
```

```
      r1 = alloc int
foo:  [r1] = foo          this is valid!
      r3 = 0
      r4 = [r1]           but the type
      [r1] = r3           of r1 changes.
      jump r4
```

```
      r1 = alloc int     r1 : ptr(int)
foo:  [r1] = foo         r1 : ptr(code(…))
      r3 = 0             r3 : int
      r4 = [r1]          r4 : code(…)
      [r1] = r3          r1 : ptr(int)
      jump r4            valid
```

```
r3 = 0
[r1] = r3
r4 = [r2]
jump r4
```

```
r1, r2 : ptr(code(…))
r3 : int
r1 : ptr(int)
r4 : code(…)
valid
```

```
r3 = 0
[r1] = r3
r4 = [r2]
jump r4
```

is this safe?

```
r2 = r1
r3 = 0
[r1] = r3
r4 = [r2]
jump r4
```

definitely not safe!

problem is **aliasing**

aliasing is hard

so, be more **restrictive**

$$v ::= r \mid n \mid l \mid u$$

$$\iota ::= \ldots \mid \texttt{commit}\ r_d$$

$$\tau ::= \cdots \mid \texttt{ptr}(\tau) \mid \texttt{uptr}(\tau)$$

unique pointers can't **alias**

$$\frac{}{\Psi \vdash r_d = \texttt{alloc}(\tau) : \Gamma \to \Gamma, r_d : \texttt{uptr}(\tau)}$$

$$\frac{\Psi; \Gamma \vdash r_d : \texttt{uptr}(\tau)}{\Psi \vdash \texttt{commit}\ \ r_d : \Gamma \to \Gamma, r_d : \texttt{ptr}(\tau)}$$

$$\frac{\Psi; \Gamma \vdash r_s : \texttt{ptr}(\tau)}{\Psi \vdash r_d = [r_s] : \Gamma \to \Gamma, r_d : \tau}$$

$$\frac{\Psi; \Gamma \vdash r_s : \texttt{uptr}(\tau)}{\Psi \vdash r_d = [r_s] : \Gamma \to \Gamma, r_d : \tau}$$

commit
compiles away
at runtime

$$\frac{\Psi;\Gamma \vdash r_s : \tau \quad \Psi;\Gamma \vdash r_d : \mathtt{uptr}(\tau') \quad \tau \neq \mathtt{uptr}(\sigma)}{\Psi \vdash [r_d] = r_s : \Gamma \to \Gamma, r_d : \mathtt{uptr}(\tau)}$$

$$\frac{\Psi;\Gamma \vdash r_s : \tau \quad \Psi;\Gamma \vdash r_d : \mathtt{ptr}(\tau) \quad \tau \neq \mathtt{uptr}(\sigma)}{\Psi \vdash [r_d] = r_s : \Gamma \to \Gamma}$$

$$\frac{\Psi;\Gamma \vdash v : \tau \quad \tau \neq \mathtt{uptr}(\sigma)}{\Psi \vdash r_d = v : \Gamma \to \Gamma, r_d : \tau}$$

TAL-1

- typing for **tuples** and **records**, memory offsets
- **stack** allocation and deallocation
- **objects** and **closures** using **existential types**
- **arrays** using **dependent types**
- TALx86 implemented and **formally verified**

real-world issues

- how to **compile** TAL (involves **alias analysis)**?
- how to generalize across **architectures**?
- how to build this into a secure **runtime** system?
- how to relate this to **proof-carrying code**?
- how successful is **bytecode verification** in the real world?

allocation

- how to manage **heap memory**?
- use-after-free and double-free are **unsound**
- **GCs** are slow, nondeterministic, antiparallel, hard to prove correct

allocation

- **unique pointers** also at high level language
- pointers unique ⇒ deallocation is **static**
- TAL naturally extends to **linear type systems**

weakening

$$\frac{\Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_1, x_1 : \tau_1, \Gamma_2 \vdash e : \tau}$$

“useless premises are okay”

contraction

$$\frac{\Gamma, x_2 : \tau_1, x_3 : \tau_1, \Gamma_2 \vdash e : \tau_2}{\Gamma_1, x_1 : \tau_1, \Gamma_2 \vdash [x_1/x_2][x_1/x_3]e : \tau_2}$$

“what can be done with many, can be done with one”

linear types

substructural logic lacking **weakening** and **contraction**

like SML with unused variable checking, **no sharing**

in practice: **affine types** allow weakening, not contraction

```
val x = []
val y = 1::x
val z = x          (* this isn't allowed! *)
```

```
fun clone (xs : 'a list) : ('a list * 'a list) =
  (xs, xs)
```

sharing isn't allowed

```
fun clone (xs : 'a list) (f : 'a -> 'a * 'a) =
  case xs of
    [] => ([], [])
  | x::xs =>
    let
      val (x1, x2) = f x
      val (xs1, xs2) = clone xs f
    in
      (x1::xs1, x2::xs2)
    end
```

why should cloning be **explicit**?

accurate **resource usage**

what does cloning a **file** do?

or a **network socket**?

zero allocation

linear type!

```
fun map (f : 'a -> 'b) (xs : 'a list) : 'b list =
  case xs of
    [] => []
  | x::xs => (f x)::(map f xs)
```

no allocations!

optimize to an in-place **for**-loop!

how many allocations?

```
val x = ref 1
val y = x
val z = y := 2
val 1 = !x          (* exception Bind *)
```

ML world

```
val x = ref 1
val y = x
val z = y := 2
val 1 = !x          Error: ‘x’ used after move
```

linear world

```
let l = [1, 2, 3];
l.push(4);
```

```
val l = [1, 2, 3]
val l = l @ [4]
```

imperative = functional?

- linearity difficult in practice
- how to recover **sharing**?
- system of **references**
- **sharing** xor **mutability**
- can pointers be **committed?**

further reading

ch. 1 (linear types), 2 (effect types), 4 (TAL), 5 (proof-carrying code), *Advanced Topics in Types and Programming Languages* by Pierce

"From System F to Typed Assembly Language", Morrisett, Walker, Crary, and Glew, 1999

"TALx86: A Realistic Typed Assembly Language", Morrisett et al., 1999

"Typed memory management in a calculus of capabilities", Crary et. al., POPL '99.

"Linear types can change the world", Wadler, 1990.