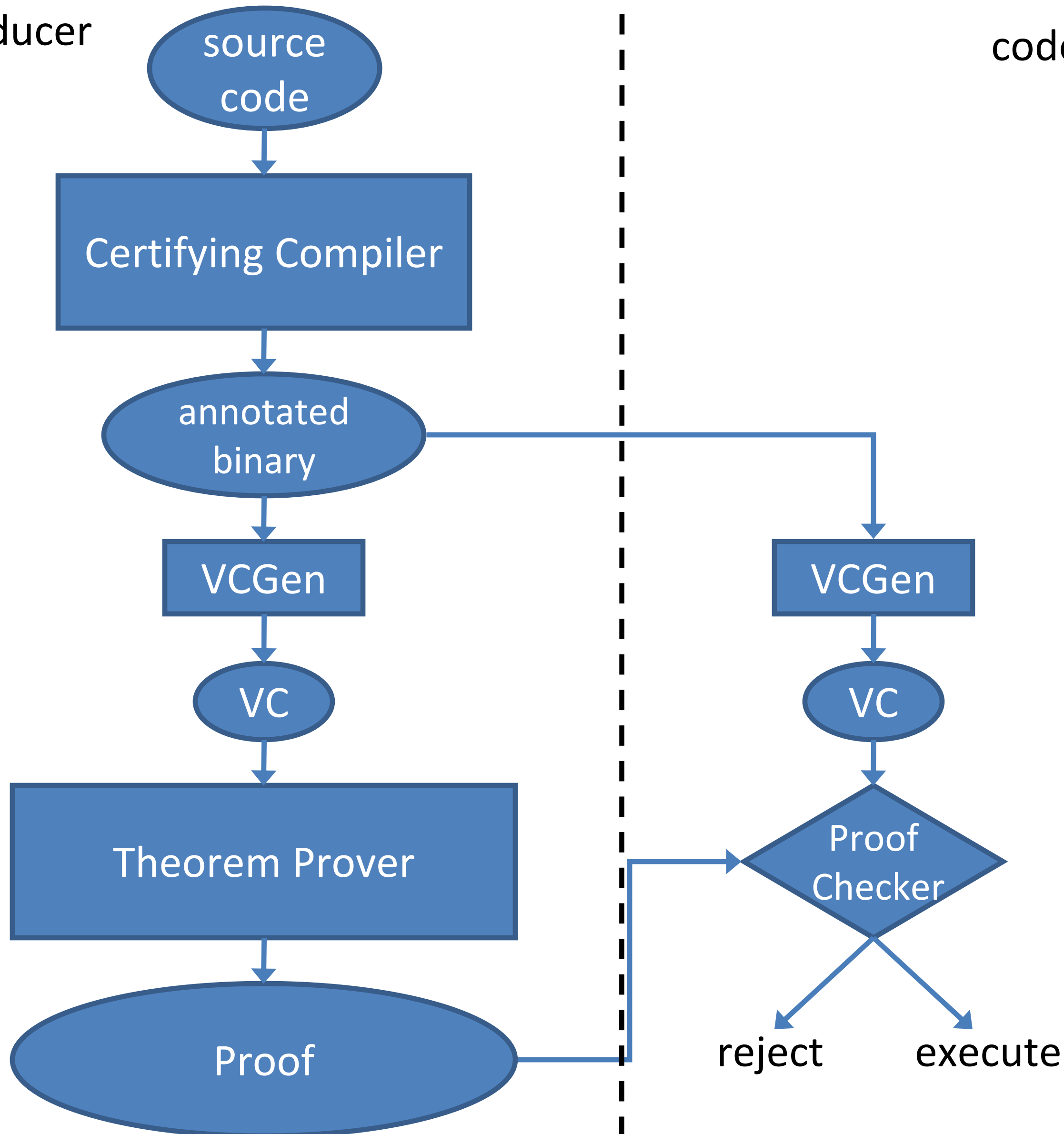
Typed Assembly Language

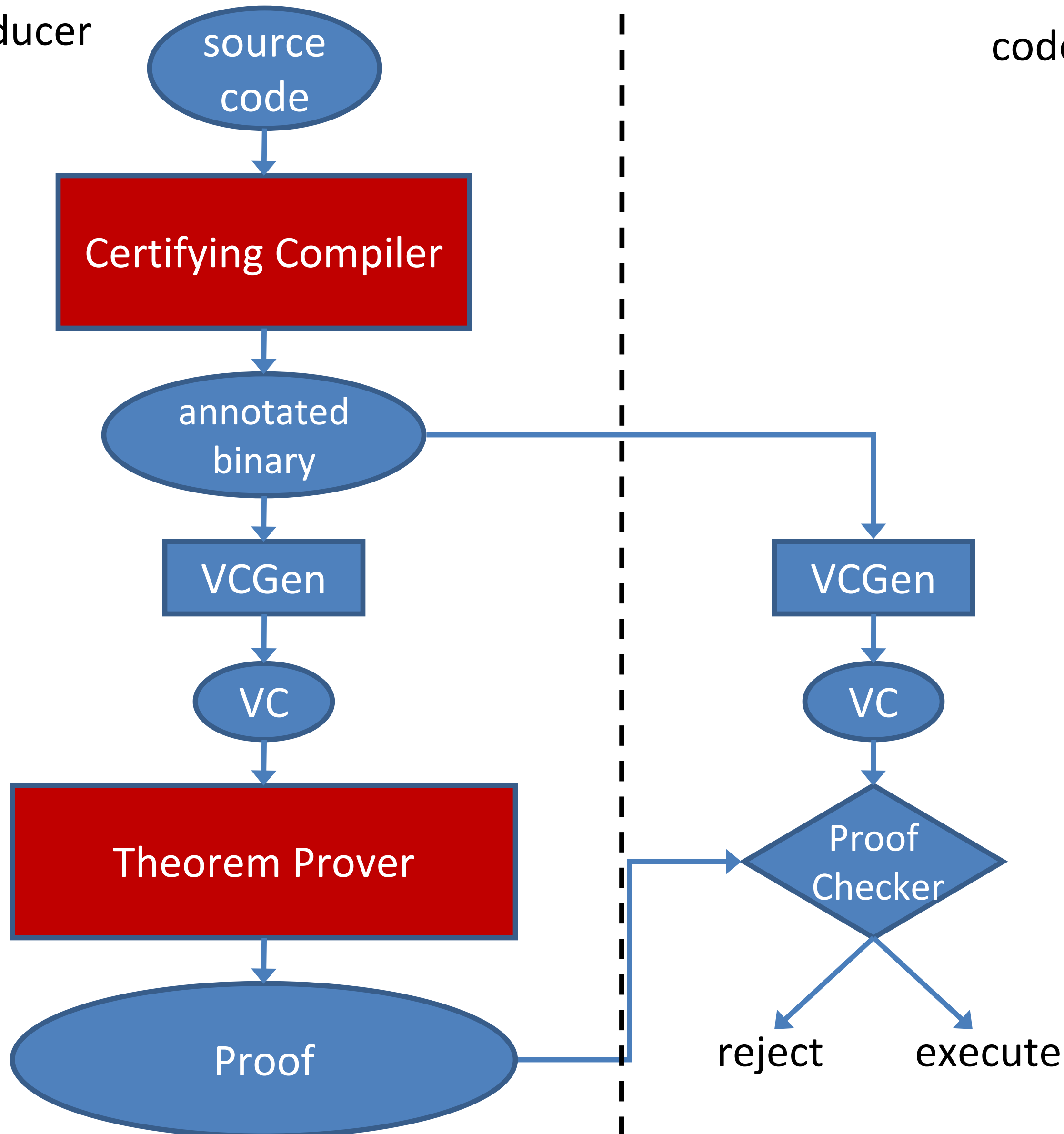
Language-based Security

Spring 2008

PCC



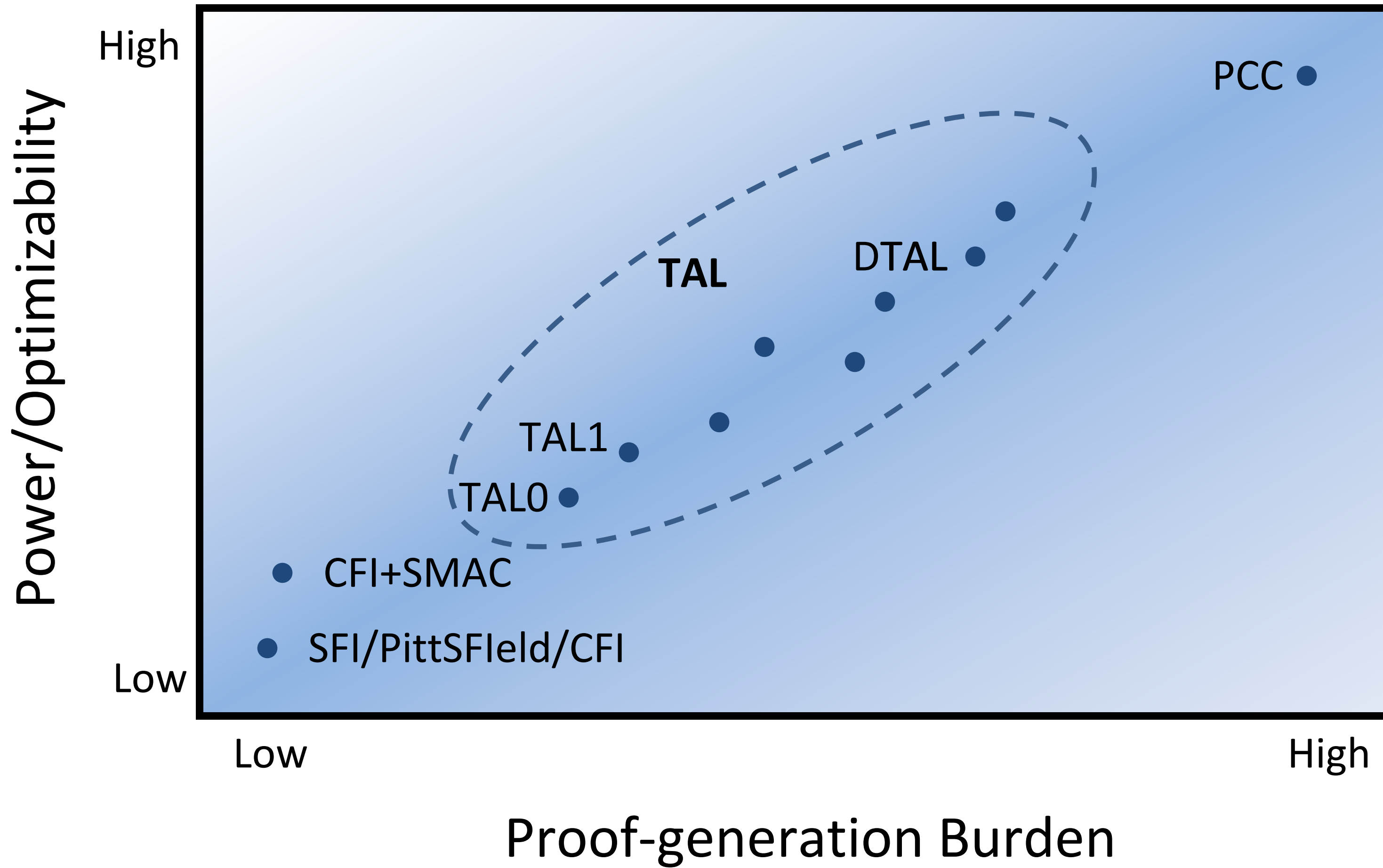
PCC



PCC

- Pro: Extremely Powerful
  - Can enforce any property provable from axioms of first-order logic (in principle)
  - Can support any optimization provably sound from architectural axioms (in principle)
- Cons:
  - hard to implement certifying compiler
    - how to discover loop invariant annotations?
    - how to prove aggressive code optimizations sound?
  - proofs are large and complex even for small code

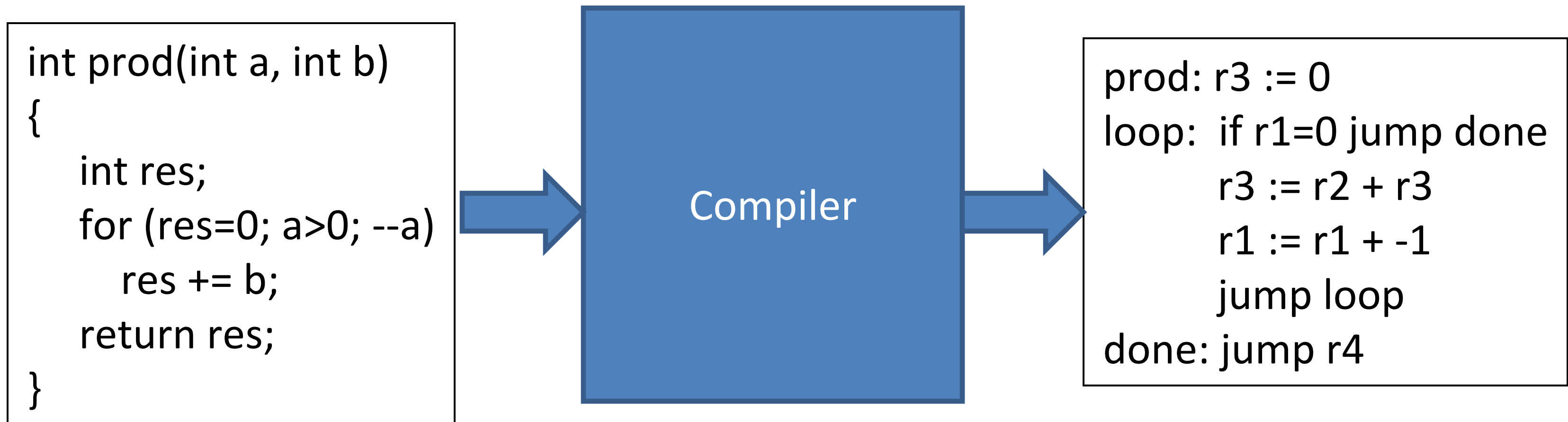
Power vs. Proof



Typed Assembly Language

- Certifying Compiler
  - assigns a type to each register/memory value at each code point
  - typing annotations replace logic annotations
- Proof of Policy Adherence
  - **any program that is well-typed is safe to execute**
  - proof of safety is just the program's type derivation
  - no VC, no VCGen, no theorem prover
- Verification
  - verifier is a type-checker (instead of a proof-checker)

Example Compilation



- Informally…
  - r1 = “a”
  - r2 = “b”
  - r3 = “res”
  - r4 = return address
  - function result returned through r3

Break Into Basic Blocks

```
prod: r3 := 0
loop:  if r1=0 jump done
       r3 := r2 + r3
       r1 := r1 + -1
       jump loop
done: jump r4
```

```
prod: r3 := 0
       (jump loop)

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

done: jump r4
```

- Code-producer labels all valid jump targets
- CFI Policy: No jump goes to an unlabeled address
- In type-theory terminology…
  - Values prod, loop, and done have type “code”
  - To prove: Every argument to “jump” has type “code”

Naïve TAL

- Two types: τ ::= int | code
- State Γ:r$\rightarrow\tau$ maps registers to types

{r1:int, r2:int, r3:int, r4:code}	prod: r3 := 0 (jump loop)
{r1:int, r2:int, r3:int, r4:code}	loop: if r1=0 jump done r3 := r2 + r3 r1 := r1 + -1 jump loop
{r1:int, r2:int, r3:int, r4:code}	done: jump r4

Naïve TAL

- Two types: τ ::= int | code
- State Γ:r$\to\tau$ maps registers to types

{r1:int, r2:int, r3:int, r4:code}	prod: r3 := 0
{r1:int, r2:int, r3:int, r4:code}	(jump loop)
{r1:int, r2:int, r3:int, r4:code}	loop: if r1=0 jump done
	r3 := r2 + r3
	r1 := r1 + -1
	jump loop
{r1:int, r2:int, r3:int, r4:code}	done: jump r4

Naïve TAL

- Two types: τ ::= int | code
- State Γ:r$\to\tau$ maps registers to types

{r1:int, r2:int, r3:int, r4:code}	prod: r3 := 0
{r1:int, r2:int, r3:int, r4:code}	(jump loop)
{r1:int, r2:int, r3:int, r4:code}	loop: if r1=0 jump done
{r1:int, r2:int, r3:int, r4:code}	r3 := r2 + r3
{r1:int, r2:int, r3:int, r4:code}	r1 := r1 + -1
{r1:int, r2:int, r3:int, r4:code}	jump loop
{r1:int, r2:int, r3:int, r4:code}	done: jump r4

Naïve TAL

- Two types: τ ::= int | code
- State Γ:r$\rightarrow\tau$ maps registers to types

{r1:int, r2:int, r3:int, r4:code}	prod: r3 := 0
{r1:int, r2:int, r3:int, r4:code}	(jump loop)
{r1:int, r2:int, r3:int, r4:code}	loop: if r1=0 jump done
{r1:int, r2:int, r3:int, r4:code}	r3 := r2 + r3
{r1:int, r2:int, r3:int, r4:code}	r1 := r1 + -1
{r1:int, r2:int, r3:int, r4:code}	jump loop
{r1:int, r2:int, r3:int, r4:code}	done: jump r4

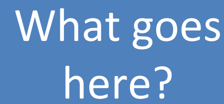
- Problem: What if there is a block somewhere that looks like…

{r1:code, r2:int, r3:int, r4:int}	foo: jump r1

- Code type-checks but is unsafe! (type system is not sound)
- Not enough to just say r4 has type “code”

Slightly Less Naïve TAL

- Two types: τ ::= int | code(Γ)
- State Γ:r$\to\tau$ maps registers to types
- Annotations assign types to code se (usually code types)
  - Example: prod has type code(r1,r2,))



code(r1,r2,r3:int, r4:code(r1,r2,r3:int, r4:code(…))}

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

Slightly Less Naïve TAL

- Two types: τ ::= int | code(Γ)
- State Γ:r$\rightarrow\tau$ maps registers to types

```
code(r1,r2,r3:int,
      r4:code(r1,r2,r3:int,
               r4:code(r1,r2,r3:int,
                        r4:code( …

This isn't looking good…
```

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

- Observations:
  - Only caller knows register state expected at r4
  - Different callers might have different states!
  - This callee's safety does NOT depend on register state at r4.
- Need a type for "any type is okay here"
  - Solution: Polymorphism!

TAL-0

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α
- State Γ:r$\rightarrow\tau$ maps registers to types

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

TAL-0

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α
- State Γ:r$\rightarrow\tau$ maps registers to types

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

- At jumps, check that source state is a **subtype** of the destination state

TAL-0

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α
- State Γ:r$\rightarrow\tau$ maps registers to types

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	prod: r3 := 0
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	(jump loop)
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	loop: if r1=0 jump done
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	r3 := r2 + r3
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	r1 := r1 + -1
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	jump loop
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))	done: jump r4

- At jumps, check that source state is a **subtype** of the destination state

TAL-0

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α
- State Γ:r$\rightarrow\tau$ maps registers to types

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

- At jumps, check that source state is a **subtype** of the destination state
- At computed jumps, destination state comes from the destination register's type

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α)) < $\forall\alpha$.code(r1,r2,r3:int, r4:α)

TAL-0

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α
- State Γ:r$\rightarrow\tau$ maps registers to types

code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

- At jumps, check that source state is a **subtype** of the destination state
- At computed jumps, destination state comes from the destination register's type

code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) < ∀α.code(r1,r2,r3:int, r4:α)

TAL-0

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α
- State Γ:r$\rightarrow\tau$ maps registers to types

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))
code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

code(r1,r2,r3:int, r4:$\forall\alpha$.code(r1,r2,r3:int, r4:α))

```
prod: r3 := 0
      (jump loop)

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

done: jump r4
```

- At jumps, check that source state is a **subtype** of the destination state
- At computed jumps, destination state comes from the destination register's type

TAL-0

code(r1,r2,r3,r4:int) code(r1,r2,r3:int, r4:code(r1:code(…), r2,r3,r4:int))	bad: r4 := foo jump prod
code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α)) code(r1,r2,r3:int, r4:∀α.code(r1,r2,r3:int, r4:α))	prod: r3 := 0 (jump loop) loop: if r1=0 jump done r3 := r2 + r3 r1 := r1 + -1 jump loop done: jump r4
code(r1:code(…), r2,r3,r4:int)	foo: jump r1

- Problem fixed:
  - caller may not call prod with r4=foo because no state with r4=foo is a subtype of prod's type

    code(r1:code(…), r2,r3,r4:int) ≮ ∀α.code(r1,r2,r3:int, r4:α)

Memory Safety

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α | ptr(τ_1,…,τ_n)
- State Γ:r$\rightarrow\tau$ maps registers to types

r1,r2,r3:$\forall\alpha.\alpha$, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

r1,r2:$\forall\alpha.\alpha$, r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

r1:$\forall\alpha.\alpha$, r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

```
prod: r3 := 0
      (jump loop)

loop: r1 := Mem[sp]
      if r1=0 jump done
      r2 := Mem[sp+1]
      r3 := r1 + r2
      r1 := r1 + -1
      Mem[sp] := r1
      jump loop

done: Mem[sp] := r3
      r1 := Mem[sp+2]
      jump r1
```

Memory Safety

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α | ptr(τ_1,…,τ_n)
- State Γ:r$\to\tau$ maps registers to types

r1,r2,r3:$\forall\alpha.\alpha$, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2:$\forall\alpha.\alpha$, r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

r1,r2:$\forall\alpha.\alpha$, r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

r1:$\forall\alpha.\alpha$, r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

```
prod: r3 := 0
      (jump loop)

loop: r1 := Mem[sp]
      if r1=0 jump done
      r2 := Mem[sp+1]
      r3 := r1 + r2
      r1 := r1 + -1
      Mem[sp] := r1
      jump loop

done: Mem[sp] := r3
      r1 := Mem[sp+2]
      jump r1
```

Memory Safety

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α | ptr(τ_1,…,τ_n)
- State Γ:r$\rightarrow\tau$ maps registers to types

r1,r2,r3:∀α.α, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2:∀α.α, r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))

r1,r2:∀α.α, r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r2:∀α.α, r1,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r2:∀α.α, r1,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))

r1:∀α.α, r2,r3:int, sp:ptr(int,int,∀α.code(r1:α, r2,r3:int, sp:ptr(int)))

```
prod: r3 := 0
      (jump loop)

loop: r1 := Mem[sp]
      if r1=0 jump done
      r2 := Mem[sp+1]
      r3 := r1 + r2
      r1 := r1 + -1
      Mem[sp] := r1
      jump loop

done: Mem[sp] := r3
      r1 := Mem[sp+2]
      jump r1
```

Memory Safety

- Four types: τ ::= int | code(Γ) | $\forall\alpha.\tau$ | α | ptr(τ_1,…,τ_n)
- State Γ:r$\rightarrow\tau$ maps registers to types

r1,r2,r3:$\forall\alpha.\alpha$, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2:$\forall\alpha.\alpha$, r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

r1,r2:$\forall\alpha.\alpha$, r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r2:$\forall\alpha.\alpha$, r1,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r2:$\forall\alpha.\alpha$, r1,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1,r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

r1:$\forall\alpha.\alpha$, r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1:$\forall\alpha.\alpha$, r2,r3:int, sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))
r1:$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)), r2:$\forall\alpha.\alpha$, r3:int,
sp:ptr(int,int,$\forall\alpha$.code(r1:α, r2,r3:int, sp:ptr(int)))

```
prod: r3 := 0
      (jump loop)

loop: r1 := Mem[sp]
      if r1=0 jump done
      r2 := Mem[sp+1]
      r3 := r1 + r2
      r1 := r1 + -1
      Mem[sp] := r1
      jump loop

done: Mem[sp] := r3
      r1 := Mem[sp+2]
      jump r1
```

Problem: Aliasing

Types	Code
r1:ptr(code(…)), r2:ptr(code(…)), r3:int, r4:int	r3 := 0
r1:ptr(code(…)), r2:ptr(code(…)), r3:int, r4:int	Mem[r1] := r3
r1:int, r2:ptr(code(…)), r3:int, r4:int	r4 := Mem[r2]
r1:int, r2:ptr(code(…)), r3:int, r4:code(…)	jump r4

Problem: Aliasing

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

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

- What if r1=r2?
  - code jumps to 0!
  - type system is unsound (type-checks but unsafe)
  - safety depends on aliasing
  - aliasing analysis is undecidable in general
- Solution: regular-pointers vs. unique pointers

TAL-1

- Introduce two kinds of pointer: ptr(…) & uptr(…)
- Regular pointers: ptr(…)
  - Storing into a regular pointer must never change its type
  - Example: Mem[r1]:=0 when r1:ptr(code(…)) is illegal (fails type-checking)
- Unique pointers: uptr(…)
  - Storing into unique pointer MAY change its type
  - Unique pointers may not alias
  - Example: {r1:uptr(…)} r2:=r1 {r1:int, r2:uptr(…)}
- Common Usage
  - Stack treated as register store on x86, so sp:uptr(…)
  - Heap data structures have static types, so use ptr(…)
  - Some safe programs conservatively prohibited!
  - Not all languages compile to TAL! (example: unsafe C)

Summary

- Advantages of Type Annotations
  - natural translation from source-level types to low-level types
  - easier Certifying Compiler implementation
  - Proof-checking is just type-checking
- Which policies are enforceable?
  - enforcement power = expressiveness of type-system
  - An extreme: have a type for every first-order logical predicate (makes TAL=PCC)
- TAL Research: Find a type-system that…
  - is expressive enough to enforce most security policies
  - is expressive enough to allow most code optimizations
  - is a natural compilation target for most well-typed source langs
- Next Time: Adding dependent types to TAL (DTAL)