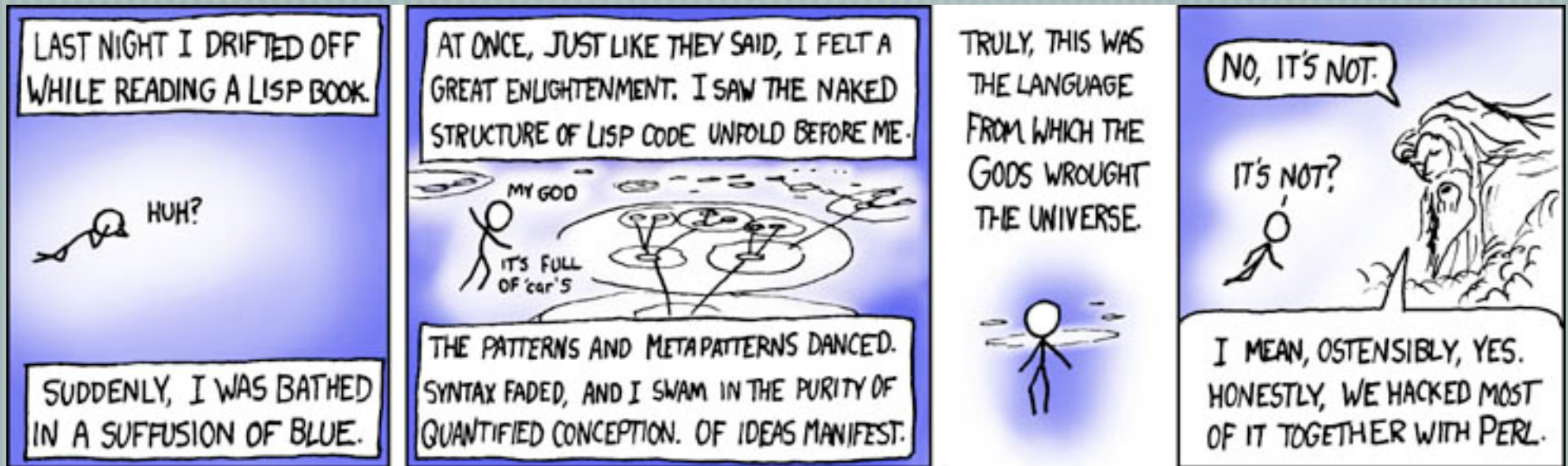


# A Dependently Typed Assembly Language

Cameron Desautels

# A Hint of Edutainment



<http://xkcd.com/224/> (with permission)

# Quick Review of TAL

— [ “The primary goal of the TAL type system is to provide a fully automatic way to verify that programs will not violate the primitive abstractions of the language.” [1]

— [ This is the basic goal of any type system.

# Quick Review of TAL

— [ TAL, however, does this at the assembly code level.

— [ But assembly language is untyped, so we need to supply typing annotations:

```
{r1:int, r2:int, r3:int, r4:code(...)}  
prod: mov r3, 0  
      jmp loop
```

# Quick Review of TAL

— [ “[I]n an untyped assembly language, there is only one abstraction: the machine word. In contrast, TAL provides a set of built-in abstractions, such as (word-sized) integers; pointers to tuples, and code labels, for each of which only some operations are applicable. For example, arithmetic is only permitted on integer values; dereferencing is only permitted for pointer values; and control transfer is only permitted for code labels.” [1]

# So What Can TAL Enforce?

— [ Basic operation sanity. (`add` expects integers)

— [ Jump target safety. (only `jump` to designated labels)

— [ Pointer safety.

# What Can't TAL Enforce?

— [ Let's write some code:

```
void init_array(int a[], int len)
{
    for (int i = 0; i < len; i++)
        a[i] = 0;
}
```

— [ ...and then call it:

```
int foo[5];
init_array(foo, 42);
```

# What Can't TAL Enforce?

— [ Since TAL is an AL, let's compile:

```
[r1: ptr(...), r2: int, r3:  $\alpha$ , r4: code(...)]  
init: mov    r3, 0
```

```
[r1: ptr(...), r2: int, r3: int, r4: code(...)]  
loop: be     r3, r2, done  
      store r1(r3), 0  
      add   r3, r3, 1  
      jmp  loop
```

```
[r1: ptr(...), r2: int, r3:int, r4: code(...)]  
done: jump  r4
```

# What Can't TAL Enforce?

What goes here?

— [ Since TAL is an AL, let's compile:

```
[r1: ptr(...), r2: int, r3: α, r4: code(...)]  
init: mov    r3, 0
```

```
[r1: ptr(...), r2: int, r3: int, r4: code(...)]  
loop: be     r3, r2, done  
      store r1(r3), 0  
      add   r3, r3, 1  
      jmp  loop
```

```
[r1: ptr(...), r2: int, r3:int, r4: code(...)]  
done: jump  r4
```

# What Can't TAL Enforce?

— [ Two questions:

— What type can we insert for the array?

— And what typing annotation can we insert to prevent the illegal memory access?

# What Can't TAL Enforce?

— [ Think hard...

# What Can't TAL Enforce?

— [ Think hard...ok, stop.

# Why Can't TAL Enforce This?

— [ We could have a static array type, but TAL wouldn't have any mechanism for understanding the length of the array.

— [ And TAL has no understanding that `a[ ]` and `len` have anything to do with one another.

— But they do!

— And we could use that information.

— Enter dependent types.

# What Are Dependent Types?

— [ They allow for types that depend on a (typed) value.

— [ So we take our original types:

$\tau ::= \alpha \mid \sigma \mid \textit{top} \mid \textit{unit} \mid \textit{int}$

# What Are Dependent Types?

— [ They allow for types that depend on a (typed) value.

— [ So we take our original types:

$\tau ::= \alpha \mid \sigma \mid top \mid unit \mid int$

type variable

program state

uninitialized

"don't use"

# What Are Dependent Types?

— [ They allow for types that depend on a (typed) value.

— [ So we take our original types, and add to them:

$\tau ::= \alpha \mid \sigma \mid \textit{top} \mid \textit{unit} \mid \textit{int}(x) \mid \tau \textit{array}(x)$

# What Are Dependent Types?

— [ They allow for types that depend on a (typed) value.

— [ So we take our original types, and add to them:

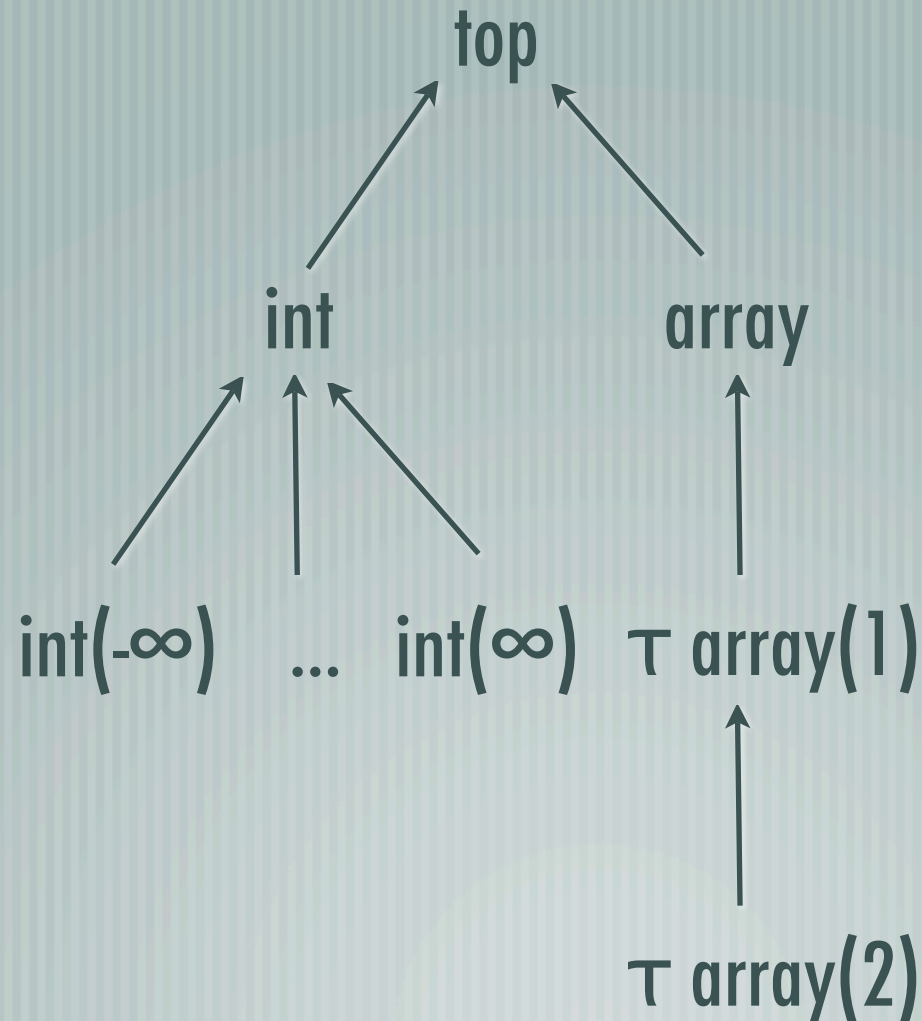
$\tau ::= \alpha \mid \sigma \mid \text{top} \mid \text{unit} \mid \text{int}(x) \mid \tau \text{ array}(x)$

— [ Where:

$x ::= a \mid i \mid x+y \mid x-y \mid x*y \mid x/y$

# Resultant Type Hierarchy

(Partial)



# What Do We Gain?

— [ Take our original function signature

```
void init_array(int a[], int len)
```

— [ In TAL we'd have needed a runtime check in the loop.

— [ In DTAL we can hoist this check out of the loop (and even out of the function) and employ typing checks:

```
{m:nat, n:nat | m <= n}
```

```
void init_array(int a[m], int(n) len)
```

# What Do We Gain?

— [ Take our original function signature

```
void init_array(int a[], int len)
```

— [ In TAL we'd have needed a runtime check in the loop.

— [ In DTAL we can hoist this check out of the loop (and even out of the function) and employ typing checks:

```
{m:nat, n:nat | m <= n}  
void init_array(int a[m], int(n) len)
```

**Now we just have to prove this.**

# Typing Annotations

- [ We've introduced more complicated typing annotations.

- Increases expressive power. Now we can correlate variables.

- Adds burden to programmer (sometimes).

- [ How to express and solve:

- System of linear equations.

- Run-time: NP complete.

# Let's Type Check Again

```
— [ [r1: int array, r2:  $\alpha$ , r3:  $\beta$ , r4: code(...)]  
init: mov      r3, 0  [... r3: int(0) ...]  
      arraysize r2, r1 [... r2: int(n) ...]
```

```
[r1: int array(m), r2: int(n), r3: int(0), r4: code(...)]  
{m:nat, n:nat | m <= n}  
loop: be      r3, r2, done  
      store r1(r3), 0  
      add   r3, r3, 1  
      jmp  loop
```

# Primary Applications

- [ Array bounds checking.

- [ Sum types:

- Recursive data structures.

- If we tag the data elements.

# A More Complicated Type

— [ Type of a function:

```
sprintf :  $\prod f:\text{Format}. \text{Data}(f) \rightarrow \text{String}$ 
```

— [ **Data(f):**

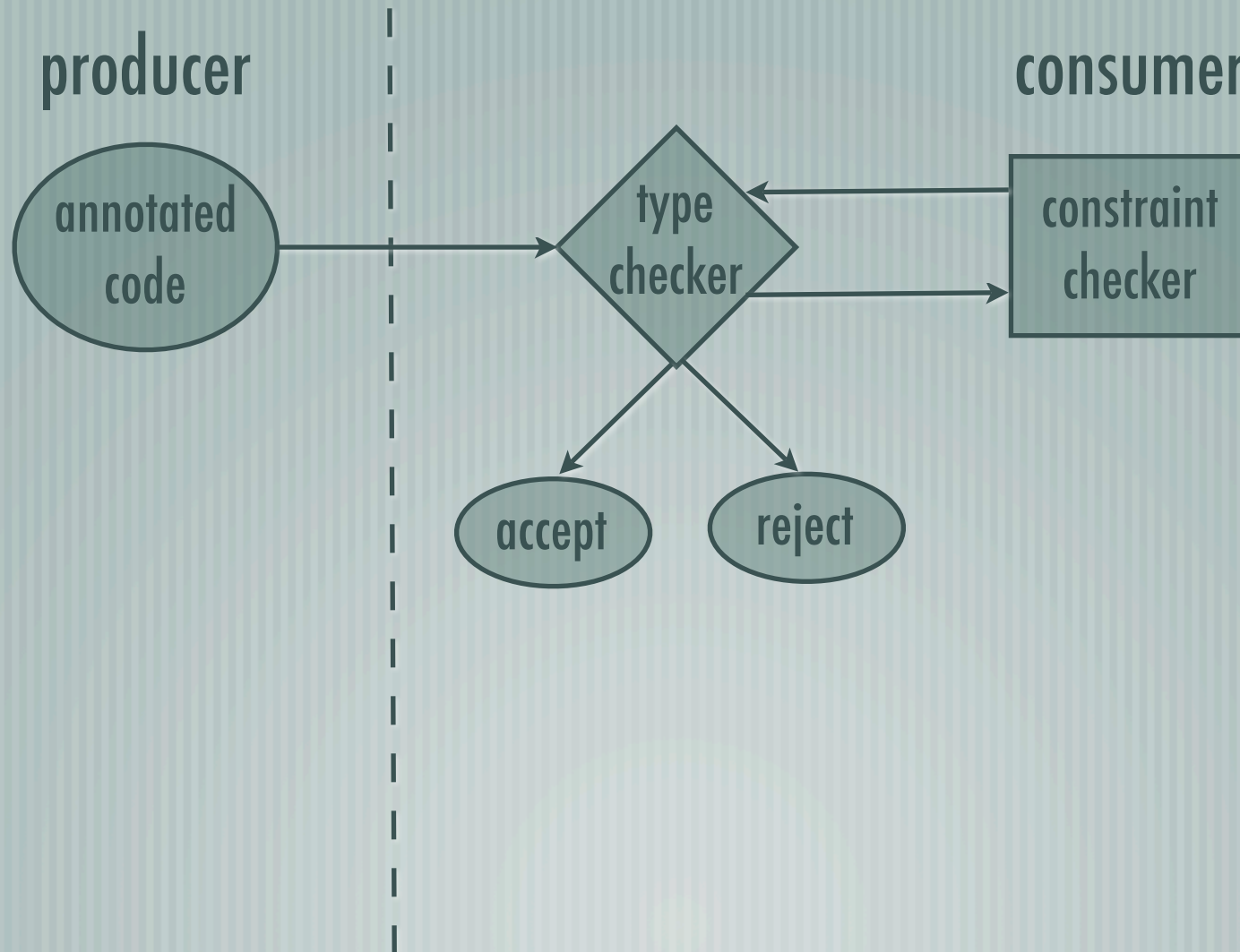
```
Data([]) = Unit
```

```
Data("%d"::cs) = Nat * Data(cs)
```

```
Data("%s"::cs) = String * Data(cs)
```

```
Data(c::cs) = Data(cs)
```

# Where Does This Fit In?



# References

- [1] G. Morrisett and D. Walker. From System F to Typed Assembly Language.
- [2] H. Xi and R. Harper. A Dependently Typed Assembly Language.
- [3] H. Xi. Facilitating Program Verification with Dependent Types.