

The Lambda Calculus

Static Analysis 2009

Michael I. Schwartzbach
Computer Science, University of Aarhus

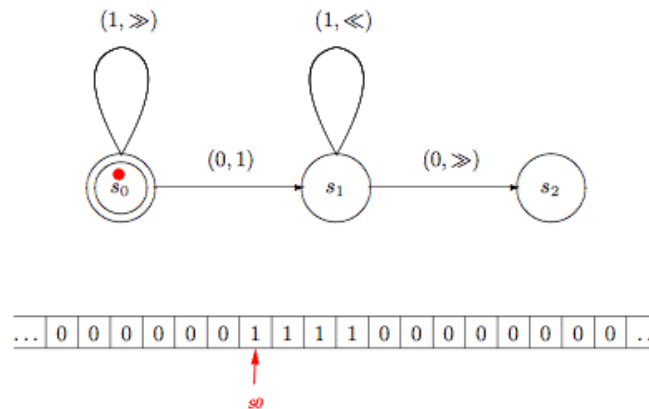
Models of Computation

- Turing machines (1936)
 - inspired by paper and pencil
- Lambda calculus (1936)
 - inspired by mathematical functions
- Cellular automata (1940)
 - inspired by life forms

- All such can emulate each other...

Turing Machines

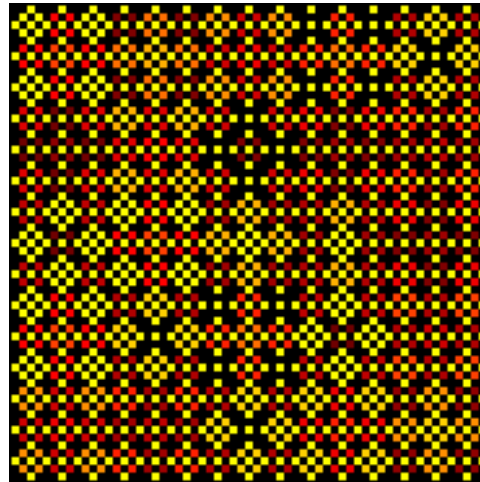
- A memory tape and a controlling program:



- Popular languages: C++, Java, C#, ...

Cellular Automata

- Living cells that interact with neighbors:



- No popular languages...

Lambda Calculus

- Functional expressions being rewritten:

```
λfx. f (f (f (f (f (f (f (f (fx))))))))
```

- Popular languages: Haskell, ML, Scheme...

The Role of The Lambda Calculus

- The origin of many fundamental programming language concepts
- The inner workings of all functional languages
- The "white lab mouse" of language research:
 - start with the lambda calculus
 - extend it with some novel features
 - experiment with the resulting language
- Just plain fun (if you love programming)



Syntax of the Lambda Calculus

- Only three grammar rules:

$E \rightarrow \lambda x.E$		(function definition)
$E_1 E_2$		(function application)
x		(variable reference)

- Example terms:

$\lambda x.x$	(identity function)
$\lambda f.\lambda g.\lambda x.f(gx)$	(function composition)
$\lambda x.xyz$	(???)

Syntactic Conventions

- We use currying as syntactic sugar:

$$\lambda x_1 x_2 x_3 \dots x_k. E \equiv \lambda x_1. \lambda x_2. \lambda x_3. \dots \lambda x_k. E$$

- Application is left-associative:

$$E_1 E_2 E_3 \dots E_k \equiv (\dots ((E_1 E_2) E_3) \dots E_k)$$

Bound and Free Variables

- A variable is **bound** to the nearest declaration:

$$\lambda x. \lambda y. xy (\lambda x. yx) x$$

- A variable that is not bound is called **free**:

$$\lambda x. \lambda y. xy (\lambda x. yz) x$$

Alpha Conversion

- Bound variables may be renamed:

$$\lambda x.x \equiv \lambda a.a$$
$$\lambda f.\lambda g.\lambda x.f(gx) \equiv \lambda g.\lambda f.\lambda z.g(fz)$$

which does not change the meaning of the term

- Free variables cannot be renamed:

$$\lambda x.xyz \equiv \lambda a.ayz$$

Beta Reduction

- The computational engine of the calculus
- Reductions correspond to single steps
- A **redex** is an opportunity to reduce:

$$(\lambda x. E_1) E_2$$

- The reduction **substitues** all free occurrences of the variable x in E_1 with a copy of E_2 :

$$E_1[x \setminus E_2]$$

Substitution

$(\lambda x. yxz x(\lambda x. yx)x)(abc)$

find redex



$(yxzx(\lambda x. yx)x)[x \backslash abc]$

reduce



$(yxzx(\lambda x. yx)x)[x \backslash abc]$

find free occurrences



$y(abc)z(abc)(\lambda x. yx)(abc)$

substitute

Example Beta Reduction

$(\lambda f g x. f(gx))(\lambda a. a)(\lambda b. bb)c \rightarrow$
 $(\lambda g x. (\lambda a. a)(gx))(\lambda b. bb)c \rightarrow$
 $(\lambda g x. gx)(\lambda b. bb)c \rightarrow$
 $(\lambda x. (\lambda b. bb)x)c \rightarrow$
 $(\lambda x. xx)c \rightarrow$
cc

Computing by Reduction

- Intuitively, beta reduction models computations by simplifying expressions:

$(\lambda x y. x(2 * y))(\lambda z. z + 1)5 \rightarrow$

$(\lambda y. (\lambda z. z + 1)(2 * y))5 \rightarrow$

$(\lambda z. z + 1)(2 * 5) \rightarrow$

$2 * 5 + 1 \rightarrow$

11

- However, we don't have numbers and such yet...

Variable Capture

- A simple reduction:

$$(\lambda x a.xa)(\lambda x.xa) \rightarrow \lambda a.(\lambda x.xa)a \rightarrow \lambda a.aa$$

- Now, first alpha convert $\lambda x a.xa$ to $\lambda x b.xb$:

$$(\lambda x b.xb)(\lambda x.xa) \rightarrow \lambda b.(\lambda x.xa)b \rightarrow \lambda b.ba$$

- The results are different, but alpha conversion should not change the meaning???

Avoiding Variable Capture

- The problem occurs when a term with a free variable is copied into a term where that variable is already bound
- The solution is to implicitly alpha convert the bound variable into something harmless:

$$(\lambda x a.xa)(\lambda x.xa) \rightarrow \lambda b.(\lambda x.xa)b \rightarrow \lambda b.ba$$

Normal Forms and Termination

- A term with no more redexes is a **normal form**
- Normal forms correspond to the results of our computations (values in Haskell)
- Not all terms have normal forms:

$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \dots$

$(\lambda x.xxx)(\lambda x.xxx) \rightarrow (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow \dots$

Reduction Strategies

- More than one redex may be available:

$(\lambda fgx.f(gx))(\lambda a.a)(\lambda b.bb)c \rightarrow$
 $(\lambda gx.(\lambda a.a)(gx))(\lambda b.bb)c \rightarrow \dots$

- Which one should we reduce?

Confluence

- Fortunately, all strategies can only reach the same normal form:

```
(λfgx.f(gx))(λa.a)(λb.bb)c →  
(λgx.(λa.a)(gx))(λb.bb)c →  
(λgx.gx)(λb.bb)c →  
(λx.(λb.bb)x)c →  
(λx.xx)c →  
cc
```

```
(λfgx.f(gx))(λa.a)(λb.bb)c →  
(λgx.(λa.a)(gx))(λb.bb)c →  
(λx.(λa.a)((λb.bb)x))c →  
(λa.a)((λb.bb)c) →  
(λa.a)(cc) →  
cc
```

Call-By-Name Reduction

- Not all strategies are equally good at terminating
- But one strategy **always** terminates if possible:
call-by-name reduction selects the left-most available redex in the term
- This is also known as **lazy** evaluation (in Haskell)

Call-By-Value Reduction

- Call-by-name is often rather slow, since arguments may be evaluated several times for each use
- Call-by-value is an alternative that tries to evaluate the arguments only once
- This is known as **eager** evaluation in ML, Scheme, Java, and most other languages

CBN vs. CBV

- CBV sometimes fails to terminate:

$$(\lambda x.a)((\lambda y.yy)(\lambda y.yy)) \rightarrow_{\text{CBN}} a$$
$$(\lambda x.a)((\lambda y.yy)(\lambda y.yy)) \rightarrow_{\text{CBV}} (\lambda x.a)((\lambda y.yy)(\lambda y.yy)) \rightarrow \dots$$

- CBV is often faster (but not always)

The Lambda Tool

```
java Lambda -evaluate [-cbn|-cbv] [-<limit>] [-trace] [-full] [-stats]
-evaluate: Lambda normalization
  -cbn: call-by-name reduction
  -cbv: call-by-value reductions
  -<limit>: max number of reductions
  -trace: print after every reduction
  -full: print with all parentheses
  -stats: print alpha and beta statistics
```

- Terms are written with \backslash in place of λ
- Variables with more than one character are written as:
 $\backslash\langle\text{foo}\rangle.\backslash\langle\text{bar}\rangle.\langle\text{bar}\rangle\langle\text{bar}\rangle\langle\text{foo}\rangle$

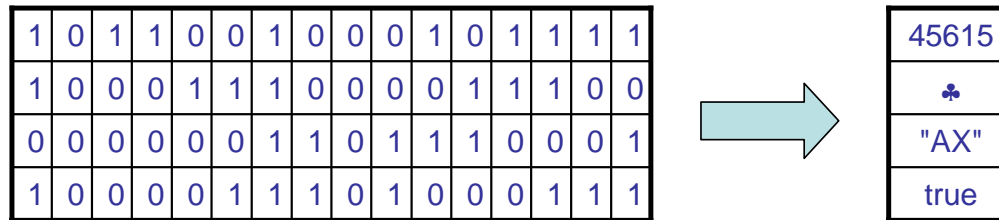
Abstract Values

- Normal forms are values, but they mean nothing in particular by themselves:
 - $\lambda x.xx$
 - `abc`
 - $\lambda xy.yyyyyyx$
- But similarly, bit patterns in memory mean nothing by themselves:

1	0	1	1	0	0	1	0	0	0	1	0	1	1	1	1
1	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0
0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	1
1	0	0	0	0	1	1	1	0	1	0	0	0	1	1	1

Encoding Values

- Interesting values must be **encoded** in the model:



- Programming is:
Information Representation Transformation

Church Numerals

- An encoding of natural numbers:
 - $0 \equiv \lambda fx.x$
 - $1 \equiv \lambda fx.fx$
 - $2 \equiv \lambda fx.f(fx)$
 - $3 \equiv \lambda fx.f(f(fx))$
 - ...
- An encoding of boolean values:
 - $true \equiv \lambda xy.x$
 - $false \equiv \lambda xy.y$

The Successor Function

- A term that computes $n+1$ given a number n :
 - $\text{succ} \equiv \lambda nfx.f(nfx)$

- Why does this work:

```
succ 3  ≡  (λnfx.f(nfx))(λfx.f(f(fx))) →  
          λfx.f((λfx.f(f(fx))))fx →  
          λfx.f((λx.f(f(fx))))x →  
          λfx.f(f(f(fx))) ≡ 4
```

- An induction proof shows that this always works

Testing for Zero

- $\text{iszero} \equiv \lambda n.n(\lambda x.(\lambda xy.y))(\lambda xy.x)$

```
iszero 0  ≡  (λn.n(λx.(λxy.y))(λxy.x))(λfx.x) →  
            (λfx.x)(λxxy.y)(λxy.x) →  
            (λx.x)(λxy.x) →  
            λxy.x ≡ true
```

```
iszero 3  ≡  (λn.n(λx.(λxy.y))(λxy.x))(λfx.f(f(fx))) →  
            (λfx.f(f(fx)))(λxxy.y)(λxy.x) →  
            (λx.(λxxy.y)((λxxy.y)((λxxy.y)x)))(λxy.x) →  
            (λxxy.y)((λxxy.y)((λxxy.y)(λxy.x))) →  
            λxy.y ≡ false
```

Arithmetic Operations

- Boolean operations:
 - $\text{and} \equiv \lambda xy.xy(\lambda xy.y)$
 - $\text{or} \equiv \lambda xy.x(\lambda xy.y)(\lambda xy.x)$
 - $\text{not} \equiv \lambda x.x(\lambda xy.y)(\lambda xy.x)$
- Integer operations:
 - $\text{pred} \equiv \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$
 - $\text{plus} \equiv \lambda mnfx.mf(nfx)$
 - $\text{mult} \equiv \lambda mnf.n(mf)$

The Fun Language

$E \rightarrow int \mid true \mid false \mid$	(literals)
$id \mid$	(variables)
$(E) \mid$	(parentheses)
$succ(E) \mid pred(E) \mid iszero(E) \mid$	(integers)
$plus(E_1, E_2) \mid mult(E_1, E_2) \mid$	(integers)
$not(E) \mid and(E_1, E_2) \mid or(E_1, E_2) \mid$	(booleans)
$pair(E_1, E_2) \mid first(E) \mid second(E) \mid$	(pairs)
$cons(E_1, E_2) \mid head(E) \mid tail(E) \mid$	(streams)
$if (E_1) E_2 \text{ else } E_3 \mid$	(conditionals)
$id (E_1, \dots, E_k) \mid$	(function call)
$let id = E_1 \text{ in } E_2 \mid$	(locals)
$let id(id_1, \dots, id_k) = E_1 \text{ in } E_2 \mid$	(functions)
$letrec id(id_1, \dots, id_k) = E_1 \text{ in } E_2$	(recursive functions)

The Factorial Function

```
letrec fac(n) = if (iszero(n)) 1  
                else mult(n, fac(pred(n)))  
in fac(6)
```

- The result is 720

A Higher-Order Function

```
let f(x,y) = succ(succ(plus(x,y))) in  
let g(h,z) = h(z,z) in  
g(f,4)
```

- The result is 10

Stream Programming

```
letrec inf(n) = cons(n,inf(succ(n))) in  
head(tail(tail(inf(7))))
```

- The result is 9

A Fibonacci Stream

```
letrec fib(x,y) =  
  (let z = plus(x,y) in cons(z,fib(y,z))) in  
letrec take(n,s) =  
  if (iszero(n)) 0  
  else pair(head(s),take(pred(n),tail(s))) in  
take(6,fib(0,1))
```

- The result is:
pair(1,pair(2,pair(3,pair(5,pair(8,pair(13,0))))))

Compiling From Fun To Lambda (1/3)

$$\lceil k \rceil \equiv \lambda fx.f^kx$$

$$\lceil \text{true} \rceil \equiv \lambda xy.x$$

$$\lceil \text{false} \rceil \equiv \lambda xy.y$$

$$\lceil id \rceil \equiv id$$

$$\lceil \text{succ}(E) \rceil \equiv (\lambda nfx.f(nfx)) \lceil E \rceil$$

$$\lceil \text{pred}(E) \rceil \equiv (\lambda nfx.n(\lambda gh.h(gf)))(\lambda u.x)(\lambda u.u) \lceil E \rceil$$

$$\lceil \text{iszero}(E) \rceil \equiv (\lambda n.n(\lambda x.(\lambda xy.y)))(\lambda xy.x) \lceil E \rceil$$

$$\lceil \text{plus}(E_1, E_2) \rceil \equiv (\lambda mnfx.mf(nfx)) \lceil E_1 \rceil \lceil E_2 \rceil$$

$$\lceil \text{mult}(E_1, E_2) \rceil \equiv (\lambda mnf.n(mf)) \lceil E_1 \rceil \lceil E_2 \rceil$$

$$\lceil \text{not}(E) \rceil \equiv (\lambda x.x(\lambda xy.y))(\lambda xy.x) \lceil E \rceil$$

$$\lceil \text{and}(E_1, E_2) \rceil \equiv (\lambda xy.xy(\lambda xy.y)) \lceil E_1 \rceil \lceil E_2 \rceil$$

$$\lceil \text{or}(E_1, E_2) \rceil \equiv (\lambda xy.x(\lambda xy.x)y) \lceil E_1 \rceil \lceil E_2 \rceil$$

Compiling From Fun To Lambda (2/3)

$$\lceil \text{pair}(E_1, E_2) \rceil \equiv (\lambda abx. xab) \lceil E_1 \rceil \lceil E_2 \rceil$$
$$\lceil \text{first}(E) \rceil \equiv (\lambda p. p(\lambda xy. x)) \lceil E \rceil$$
$$\lceil \text{second}(E) \rceil \equiv (\lambda p. p(\lambda xy. y)) \lceil E \rceil$$
$$\lceil \text{cons}(E_1, E_2) \rceil \equiv (\lambda abx. xab) \lceil E_1 \rceil \lceil E_2 \rceil$$
$$\lceil \text{head}(E) \rceil \equiv (\lambda p. p(\lambda xy. x)) \lceil E \rceil$$
$$\lceil \text{tail}(E) \rceil \equiv (\lambda p. p(\lambda xy. y)) \lceil E \rceil$$
$$\lceil \text{if } (E_1) E_2 \text{ else } E_3 \rceil \equiv \lceil E_1 \rceil \lceil E_2 \rceil \lceil E_3 \rceil$$
$$\lceil \text{id}(E_1, \dots, E_k) \rceil \equiv \lceil \text{id} \rceil \lceil E_1 \rceil \dots \lceil E_k \rceil$$
$$\lceil \text{let } id = E_1 \text{ in } E_2 \rceil \equiv (\lambda id. \lceil E_2 \rceil) \lceil E_1 \rceil$$
$$\lceil \text{let } id(id_1, \dots, id_k) = E_1 \text{ in } E_2 \rceil \equiv (\lambda id. \lceil E_2 \rceil)(\lambda id_1 \dots \lambda id_k. \lceil E_1 \rceil)$$

Compiling From Fun To Lambda (3/3)

$$\lceil \text{letrec } id(id_1, \dots, id_k) = E_1 \text{ in } E_2 \rceil \equiv \\ (\lambda id. \lceil E_2 \rceil)(Y(\lambda id. \lambda id_1 \dots \lambda id_k. \lceil E_1 \rceil))$$

where Y is a **fixed-point operator** such that $YX \rightarrow X(YX)$

- Y is used to enable recursive calls
- It provides dynamic unfoldings of the definition:

$$Y(\lambda id. \lambda id_1 \dots \lambda id_k. \lceil E_1 \rceil) \rightarrow \\ (\lambda id. \lambda id_1 \dots \lambda id_k. \lceil E_1 \rceil)(Y(\lambda id. \lambda id_1 \dots \lambda id_k. \lceil E_1 \rceil))$$

The Fixed-Point Operator in Action

- The program:

```
letrec f(n) = if (iszero(n)) 42 else f(pred(n)) in f(87)
```

is compiled into:

```
(λf.f[87])(YF)
```

where $F \equiv \lambda f. \lambda n. (\text{iszero } n) [42] (f (\text{pred } n))$

```
(λf.f[87])(YF) →  
(YF) [87] →  
F((YF)) [87] →  
(λf. λn. (iszero n) [42] (f (pred n))) ((YF)) [87] →  
(iszero [87]) [42] (((YF)) (pred [87])) →  
((YF)) (pred [87]) → (YF) [86] → ...
```

Concrete Fixed-Point Operators

- A famous fixed-point operator (1936):

$$Y \equiv ZZ \equiv (\lambda xy. y(xxy))(\lambda xy. y(xxy))$$

- It works:

$$YF \equiv (ZZ)F \rightarrow (\lambda y. y(ZZy))F \rightarrow F(ZZF) \equiv F(YF)$$

- There are infinitely many such operators

The Lambda Tool

```
java Lambda -evaluate [-cbn|-cbv] [-<limit>] [-trace] [-full] [-stats]
java Lambda -compile [-cbn|-cbv]
java Lambda -decompile
-evaluate: Lambda normalization
    -cbn: call-by-name reduction
    -cbv: call-by-value reductions
    -<limit>: max number of reductions
    -trace: print after every reduction
    -full: print with all parentheses
    -stats: print alpha and beta statistics
-compile: translate from Fun programs to Lambda
    -cbn: call-by-name code
    -cbv: call-by-value code
-decompile: translate from Lambda to Fun pairs and numerals
```

Compiling for CBV

- The previous compilation only works for CBN
- For CBV:
 - the Y operator loops
 - the if-expression always evaluates both branches and both things are bad for recursion
- We must delay some reductions:
 - $\lceil \text{if } (E_1) E_2 \text{ else } E_3 \rceil \equiv \lceil E_1 \rceil (\lambda a. \lceil E_2 \rceil a) (\lambda b. \lceil E_3 \rceil b)$
 - fixed-point operator: $\lambda g. (\lambda x. g(\lambda y. xxy)) (\lambda x. g(\lambda y. xxy))$
- Still, streams only work with CBN (as in Haskell)

CBN vs. CBV

```
let f(x) = pair(x,pair(x,pair(x,pair(x,pair(x,pair(x,0))))))  
in f(f(f(f(2))))
```

- CBN: 3,368 reductions, CBV: 53 reductions

```
let f(x) = pair(x,pair(x,pair(x,pair(x,pair(x,pair(x,0)))))) in  
let g(y) = 7 in  
g(f(f(f(f(2)))))
```

- CBN: 3 reductions, CBV: 55 reductions

Runtime Errors (1/2)

- Fun programs do not generate runtime errors:
 - no division operator
 - no empty list
 - no type checking
- A program is compiled into a Lambda term
- The term is reduced to a normal form, which may turn out to be pure nonsense:

`mult(true,pair(1,2))` \rightarrow $\lambda f.f(\lambda fx.f(fx)) \equiv ???$

Runtime Errors (2/2)

- Even worse, sometimes the nonsense actually happens to make sense:

```
let a = succ(first(1)) in  
let b = a(3) in           → 27  
b(cons(true,87))
```

- This means that errors are not detected!

Modelling Runtime Errors (1/2)

- To catch runtime errors, programs must be compiled in a more complicated manner
- Each value is modelled as a pair(*tag*, *val*) where:
 - *val* is the "old" value
 - *tag* is an integer interpreted as:
 - 0: runtime error
 - 1: integer
 - 2: boolean
 - 3: pair
 - 4: stream
 - 5: function with 1 argument
 - 6: function with 2 arguments
 - 7: function with 3 arguments
 - ...

Modelling Runtime Errors (2/2)

- $\llbracket E \rrbracket$ denotes the old compilation
- $\llbracket \llbracket E \rrbracket \rrbracket$ denotes the new compilation
- Examples:

```
 $\llbracket \llbracket k \rrbracket \rrbracket \equiv \llbracket \text{pair}(1,k) \rrbracket$ 
```

```
 $\llbracket \llbracket \text{plus}(E_1,E_2) \rrbracket \rrbracket \equiv$   
   $\llbracket \text{let } x = E_1 \text{ in}$   
     $\llbracket \text{let } y = E_2 \text{ in}$   
       $\llbracket \text{if } (\text{and}(\text{iszero}(\text{pred}(\text{first}(x))), \text{iszero}(\text{pred}(\text{first}(y))))$   
         $\llbracket \text{pair}(1, \text{plus}(\text{second}(x), \text{second}(y))) \rrbracket$   
       $\llbracket \text{else}$   
         $\llbracket \text{pair}(0,0) \rrbracket$   
     $\llbracket \rrbracket$   
   $\llbracket \rrbracket$ 
```

Type Checking

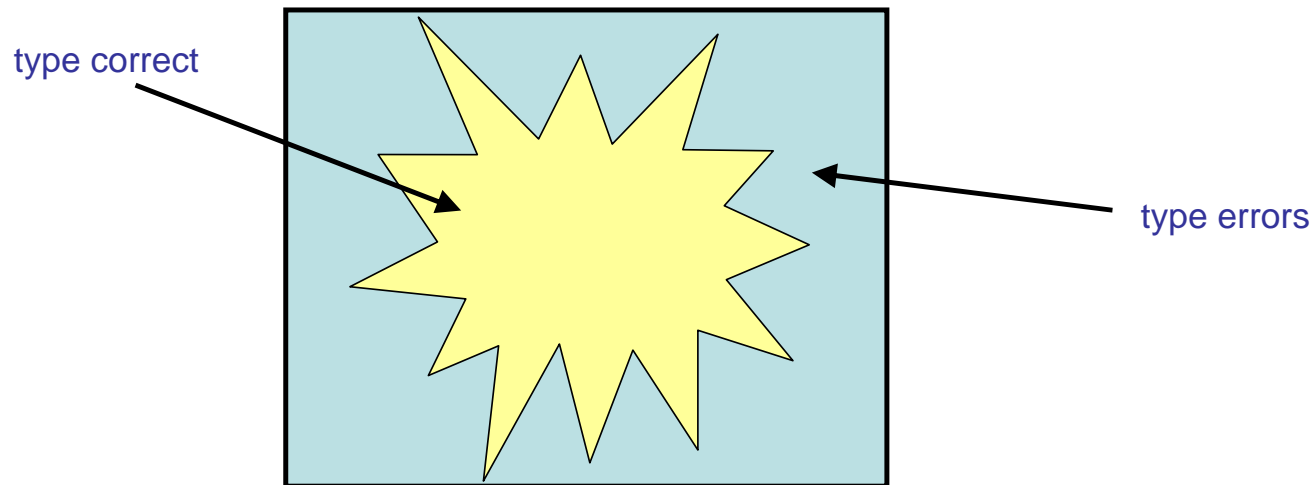
- Fun contains many "wild" terms:

```
let a = succ(first(1)) in  
let b = a(3) in  
b(cons(true,87))
```

- The compiler should detect such type errors:
 - errors are caught earlier in the development process
 - compile-time type checking avoids run-time type tags

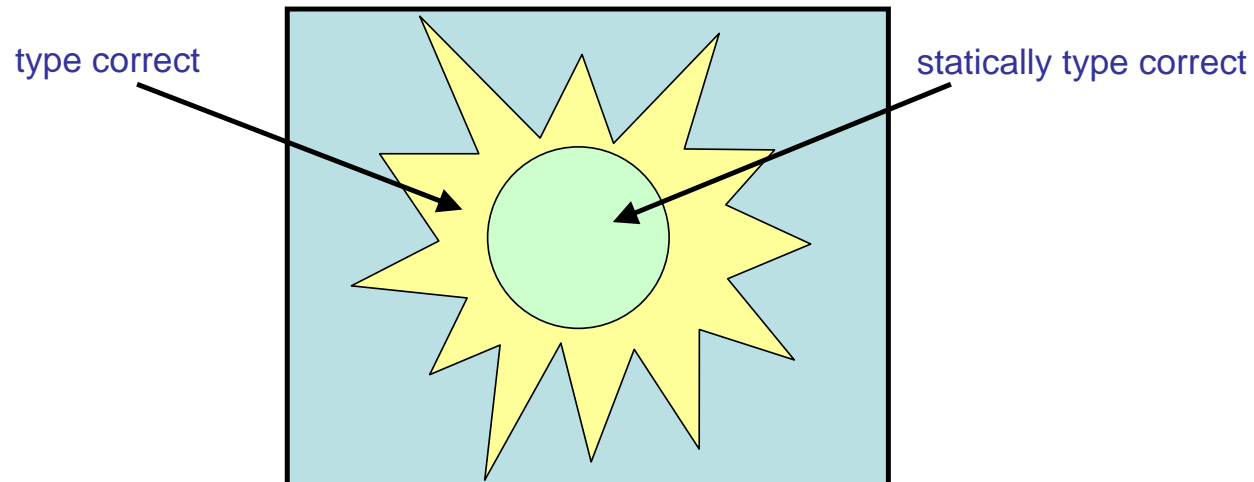
Undecidability

- It is not possible to decide if a program will cause a runtime type error during execution:



Static Type Checking

- Assign a type to every expression
- Check that certain type rules are satisfied
- If so, then no type errors will occur



Types for Fun

- A type is used to classify values:

```
 $\tau \rightarrow$  int |  
boolean |  
pair( $\tau_1, \tau_2$ ) |  
stream( $\tau$ ) |  
fun( $\tau_1, \dots, \tau_k, \tau$ )
```

Type Examples

- fac: fun(int, int)
- fib: fun(int,int,stream(int))

```
let f(x,y) = succ(succ(plus(x,y))) in  
let g(h,z) = h(z,z) in  
g(f,4)
```

- f: fun(int,int,int)
- g: fun(fun(int,int,int),int,int)

```
letrec inf(n) = cons(n,inf(succ(n))) in head(tail(tail(inf(7))))
```

- inf: fun(int,stream(int))

Type Checking

- Assign a type **variable** $[[E]]$ to every expression E
- Generate type **constraints** relating these variables to each other
- **Solve** the constraints using some algorithm

The generated constraints are solvable



The program is statically type correct



No type errors will occur at runtime

Symbol Checking

- We must first check that all used identifiers are also declared
- Fun has ordinary scope rules:

```
letrec fib(x,y) =  
    (let z = plus(x,y) in cons(z,fib(y,z))) in  
letrec take(n,s) =  
    if (iszero(n)) 0  
    else pair(head(z),take(pred(n),tail(s))) in  
take(6,fib(0,1))
```

Unique Identifiers

- We then rename all identifiers to be unique:

```
let f(f) = succ(f) in  
let f(f) = pair(f,let f = 17 in f)  
in f(10)
```

```
let f(f1) = succ(f1) in  
let f2(f3) = pair(f3,let f4 = 17 in f4)  
in f2(10)
```



Generating Constraints

iszero(E)

- *"The argument must be of type int and the result is of type boolean"*
- $[[E]] = \text{int} \wedge [[\text{iszero}(E)]] = \text{boolean}$

if (E₁) E₂ else E₃

- *"The condition must be of type boolean, both branches must have the same type, and the whole expression has the same type as the branches"*
- $[[E_1]] = \text{boolean} \wedge$
 $[[\text{if } (E_1) E_2 \text{ else } E_3]] = [[E_2]] = [[E_3]]$

Type Constraints (1/3)

<i>k</i> :	$[[k]] = \text{int}$
true:	$[[\text{true}]] = \text{boolean}$
false:	$[[\text{false}]] = \text{boolean}$
<i>id</i> :	<i>no constraints</i>
succ(E):	$[[E]] = [[\text{succ}(E)]] = \text{int}$
pred(E):	$[[E]] = [[\text{pred}(E)]] = \text{int}$
iszero(E):	$[[E]] = \text{int} \wedge [[\text{iszero}(E)]] = \text{boolean}$
plus(E ₁ ,E ₂):	$[[\text{plus}(E_1, E_2)]] = [[E_1]] = [[E_2]] = \text{int}$
mult(E ₁ ,E ₂):	$[[\text{mult}(E_1, E_2)]] = [[E_1]] = [[E_2]] = \text{int}$
not(E):	$[[E]] = [[\text{not}(E)]] = \text{boolean}$
and(E ₁ ,E ₂):	$[[\text{and}(E_1, E_2)]] = [[E_1]] = [[E_2]] = \text{boolean}$

Type Constraints (2/3)

$\text{or}(E_1, E_2)$:	$[[\text{or}(E_1, E_2)]] = [[E_1]] = [[E_2]] = \text{boolean}$
$\text{pair}(E_1, E_2)$:	$[[\text{pair}(E_1, E_2)]] = \text{pair}([[E_1]], [[E_2]])$
$\text{first}(E)$:	$[[E]] = \text{pair}([[\text{first}(E)]], \alpha)$
$\text{second}(E)$:	$[[E]] = \text{pair}(\alpha, [[\text{second}(E)]])$
$\text{cons}(E_1, E_2)$:	$[[\text{cons}(E_1, E_2)]] = \text{stream}([[E_1]]) = [[E_2]]$
$\text{head}(E)$:	$[[E]] = \text{stream}([[\text{head}(E)]])$
$\text{tail}(E)$:	$[[E]] = [[\text{tail}(E)]] = \text{stream}(\alpha)$
$\text{if } (E_1) E_2 \text{ else } E_3$:	$[[E_1]] = \text{boolean} \wedge$ $[[E_2]] = [[E_3]] = [[\text{if } (E_1) E_2 \text{ else } E_3]]$
$\text{id}(E_1, \dots, E_k)$:	$[[\text{id}]] = \text{fun}([[E_1]], \dots, [[E_k]], [[\text{id}(E_1, \dots, E_k)]])$

α is a "fresh" type variable

Type Constraints (3/3)

let $id = E_1$ in E_2 :

$$[[id]] = [[E_1]] \wedge [[\text{let } id = E_1 \text{ in } E_2]] = [[E_2]]$$

let $id(id_1, \dots, id_k) = E_1$ in E_2 :

$$[[id]] = \text{fun}([[id_1]], \dots, [[id_k]], [[E_1]]) \wedge$$
$$[[\text{let } id(id_1, \dots, id_k) = E_1 \text{ in } E_2]] = [[E_2]]$$

letrec $id(id_1, \dots, id_k) = E_1$ in E_2 :

$$[[id]] = \text{fun}([[id_1]], \dots, [[id_k]], [[E_1]]) \wedge$$
$$[[\text{letrec } id(id_1, \dots, id_k) = E_1 \text{ in } E_2]] = [[E_2]]$$

Constraints Are Equalities

- All type constraints are of the form:

$$T_1 = T_2$$

where T_i is a type term with variables:

```
T → int |  
    boolean |  
    pair(T1, T2) |  
    stream(T) |  
    fun(T1, ..., Tk, T) |  
    α
```

General Terms

- Constructor symbols:

- 0-ary: a, b, c
- 1-ary: d, e
- 2-ary: f, g, h
- 3-ary: i, j, k

- Terms:

- a
- d(a)
- h(a,g(d(a),b))

- Terms with variables:

- d(X,b)
- h(X,g(Y,Z))

The Unification Problem

- An equality between two terms with variables:

$$k(X,b,Y) = k(f(Y,Z),Z,d(Z))$$

- A solution (a unifier) is an assignment from variables to terms that makes both sides equal:

$$X = f(d(b),b)$$

$$Y = d(b)$$

$$Z = b$$

Unification Errors

- Constructor error:

$$d(X) = e(X)$$

- Arity error:

$$a = a(X)$$

The Unification Algorithm

- Paterson and Wegman (1976)
- In time $O(n)$:
 - finds a most general unifier
 - or decides that none exists
- This is used as a backend for type checking

The Lambda Tool

```
java Lambda -symbol
java Lambda -type
java Lambda -unify
-symbol: check Fun programs and make identifiers unique
-type: generate type constraints for Fun programs
-unify: solve general constraints by unification
```

The Factorial Function (1/3)

```
letrec fac(n) = if (iszero(n)) 1
                  else mult(n,fac(pred(n)))
in fac(6)
```

The Factorial Function (2/3)

```
[[fac]] = fun([[n]],[[1:if (iszero(n)) 1 else mult(n,fac(pred(n)))]])
[[letrec fac(n) = if (iszero(n)) 1 else mult(n,fac(pred(n))) in fac(6)]] = [[fac(6)]
[[iszero(n)]] = boolean
[[if (iszero(n)) 1 else mult(n,fac(pred(n)))]] = [[1]]
[[if (iszero(n)) 1 else mult(n,fac(pred(n)))]] = [[mult(n,fac(pred(n)))]]
[[1]] = int
[[mult(n,fac(pred(n)))]] = int
[[n]] = int
[[fac(pred(n))]] = int
[[fac]] = fun([[pred(n)]],[[fac(pred(n))]])
[[pred(n)]] = int
[[n]] = int
[[fac]] = fun([[6]],[[fac(6)]])
[[6]] = int
```

The Factorial Function (3/3)

```
[[n]] = int
[[1]] = int
[[fac]] = fun(int,int)
[[mult(n,fac(pred(n)))]] = int
[[6]] = int
[[pred(n)]] = int
[[fac(6)]] = int
[[iszero(n)]] = boolean
[[fac(pred(n))]] = int
[[if (iszero(n)) 1 else mult(n,fac(pred(n)))]] = int
[[letrec fac(n) = if (iszero(n)) 1 else mult(n,fac(pred(n))) in fac(6)]] = int
```

The Nonsense Program (1/3)

```
let a = succ(first(1)) in  
let b = a(3) in  
b(cons(true,87))
```

The Nonsense Program (2/3)

```
[[a]] = [[succ(first(1))]]
[[let a = succ(first(1)) in let b = a(3) in b(cons(true,87))]] =
  [[let b = a(3) in b(cons(true,87))]]
[[succ(first(1))]] = int
[[first(1)]] = int
[[1]] = pair([[4:first(1)]],[[6]])
[[1]] = int
[[b]] = [[a(3)]]
[[let b = a(3) in b(cons(true,87))]] = [[b(cons(true,87))]]
[[a]] = fun([[3],[[7:a(3)]]])
[[3]] = int
[[b]] = fun([[cons(true,87)],[[b(cons(true,87))]])]
[[cons(true,87)]] = [[87]]
[[cons(true,87)]] = stream([[true]])
[[87]] = int
[[true]] = boolean
```

The Nonsense Program (3/3)

```
*** unification constructor error  
pair(int,#v1)  
int
```

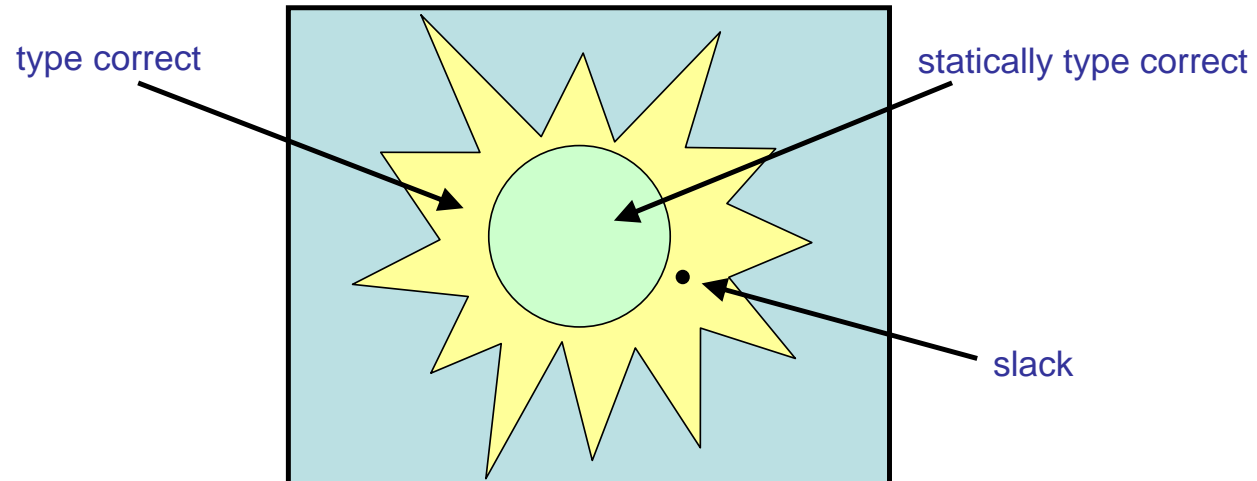
- The type error is caught at compile-time

Efficiency Through Types

- Untyped programs must use the expensive compilation strategy: $\llbracket E \rrbracket$
- Typed programs can use the much cheaper strategy: $\lceil E \rceil$
- Errors must be caught at either runtime or at compile-time

Slack

- A type checker will unfairly reject some programs:



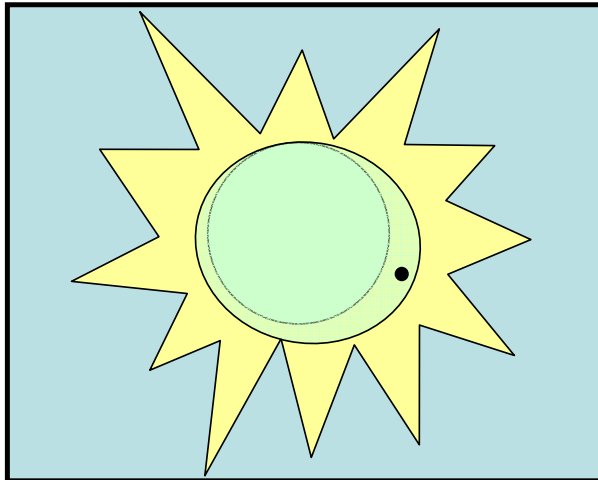
Concrete Slack

```
letrec fac2(n,foo) =  
  if (iszero(n)) 1 else mult(n,foo(pred(n),foo))  
in fac2(6,foo)
```

```
letrec f(n) = pair(n,f(pred(n)))  
in first(second(second(f(7))))
```

Fighting Slack

- Make the type checker a bit more clever:



- An eternal struggle...

Regular Types

$\tau \rightarrow$ int |
boolean |
pair(τ_1, τ_2) |
stream(τ) |
fun($\tau_1, \dots, \tau_k, \tau$)

- We have assumed finite types
- But we can accept more program if allow also infinite, but **regular** types

Regular Terms

- Infinite but (eventually) repeating:

- $e(e(e(e(e(e(\dots))))))$
- $d(a, d(a, d(a, \dots)))$
- $f(f(f(f(\dots), f(\dots)), f(f(\dots), f(\dots))), f(f(f(\dots), f(\dots)), f(f(\dots), f(\dots))))$

- A non-regular term:

- $f(a, f(d(a), f(d(d(a)), f(d(d(d(a))), \dots))))$

Regular Unification

- Paterson and Wegman (1976)
- The unification problem can be solved in $O(n\alpha(n))$
- $\alpha(n)$ is the inverse Ackermann function:
 - smallest k such that $n \leq \text{Ack}(k,k)$
 - this is never bigger than 5 for any real value of n

Regular Types in Action

```
letrec fac2(n,foo) =  
  if (iszero(n)) 1 else mult(n,foo(pred(n),foo))  
in fac2(6,foo)
```

- This function now has a type:

```
[[fac2]] = fun(int,fun(int,fun(int,...,int),int),int)
```

Polymorphism

- We still cannot type check a polymorphic function:

```
let f(x) = pair(x,0) in pair(f(42),f(true))
```

```
*** unification constructor error  
int  
boolean
```

- The function f must have two different types

Polymorphic Expansion (1/3)

- Expand all non-recursive functions before generating the type constraints

```
let f(x) = pair(x,0) in pair(f(42),f(true))
```



```
pair(let f(x) = pair(x,0) in f(42),let f(x) = pair(x,0) in f(true))
```



```
pair(let f(x) = pair(x,0) in f(42),  
      let f1(x1) = pair(x1,0) in f1(true))
```

Polymorphic Expansion (2/3)

```
[[pair(let f(x) = pair(x,0) in f(42),let f1(x1) = pair(x1,0) in f1(true))]] =  
  pair([[let f(x) = pair(x,0) in f(42)],[[let f1(x1) = pair(x1,0) in f1(true)]]]  
[[f]] = fun([[x],[[pair(x,0)]]]  
[[let f(x) = pair(x,0) in f(42)]] = [[f(42)]]  
[[pair(x,0)]] = pair([[x],[[0]])]  
[[0]] = int  
[[f]] = fun([[42],[[f(42)]]]  
[[42]] = int  
[[f1]] = fun([[x1],[[pair(x1,0)]]]  
[[let f1(x1) = pair(x1,0) in f1(true)]] = [[f1(true)]]  
[[pair(x1,0)]] = pair([[x1],[[0]])]  
[[0]] = int  
[[f1]] = fun([[true],[[f1(true)]]]  
[[true]] = boolean
```

Polymorphic Expansion (3/3)

```
[[true]] = boolean
[[f1]] = fun(boolean,pair(boolean,int))
[[x]] = int
[[pair(x,0)]] = pair(int,int)
[[f1(true)]] = pair(boolean,int)
[[f(42)]] = pair(int,int)
[[0]] = int
[[pair(let f(x) = pair(x,0) in f(42),let f1(x1) = pair(x1,0) in f1(true))]] =
    pair(pair(int,int),pair(boolean,int))
[[pair(x1,0)]] = pair(boolean,int)
[[f]] = fun(int,pair(int,int))
[[x1]] = boolean
[[let f1(x1) = pair(x1,0) in f1(true)]] = pair(boolean,int)
[[let f(x) = pair(x,0) in f(42)]] = pair(int,int)
[[42]] = int
```

A Polymorphic Explosion (1/2)

```
let f1(y) = pair(y,y) in
let f2(y) = f1(f1(y)) in
let f3(y) = f2(f2(y)) in
let f4(y) = f3(f3(y)) in
f4(0)
```

- This is polymorphically typable!
- But what is the type of f4?

The Full Lambda Tool

```
java Lambda -evaluate [-cbn|-cbv] [-<limit>] [-trace] [-full] [-stats]
java Lambda -compile [-cbn|-cbv]
java Lambda -symbol
java Lambda -decompile
java Lambda -type
java Lambda -polymorph
java Lambda -unify
```

-evaluate: Lambda normalization
-compile: translate from Fun programs to Lambda
-symbol: check Fun programs and make identifiers unique
-decompile: translate from Lambda to Fun pairs and numerals
-type: generate type constraints for Fun programs
-polymorph: expand non-recursive functions in Fun programs
-unify: solve general constraints by regular unification

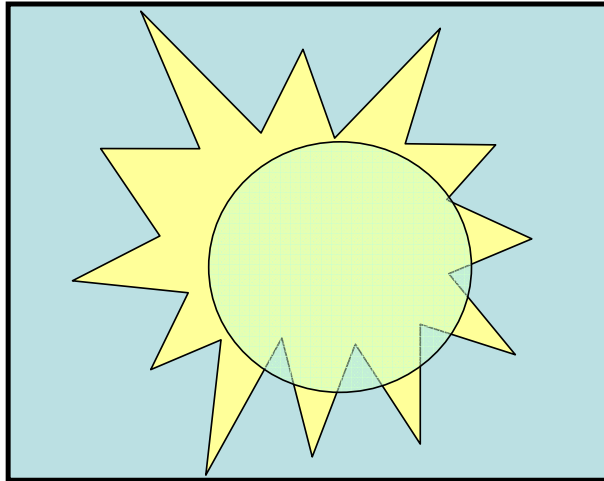
Always More Slack

- Regular types and polymorphism is not enough:

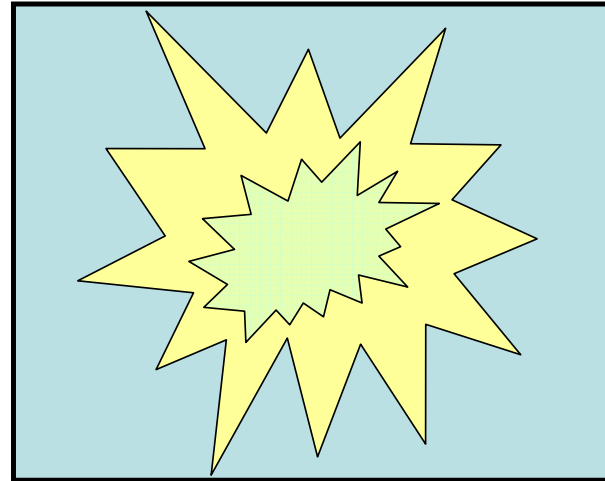
```
letrec f(n) = if (iszero(n)) 0
           else pair(f(pred(n)),f(pred(n))) in f(4)
```

is unfairly rejected by the type checker

Things to Worry About



The type checker is unsound



The type checker is undecidable