Compilation of Functional Programming Languages

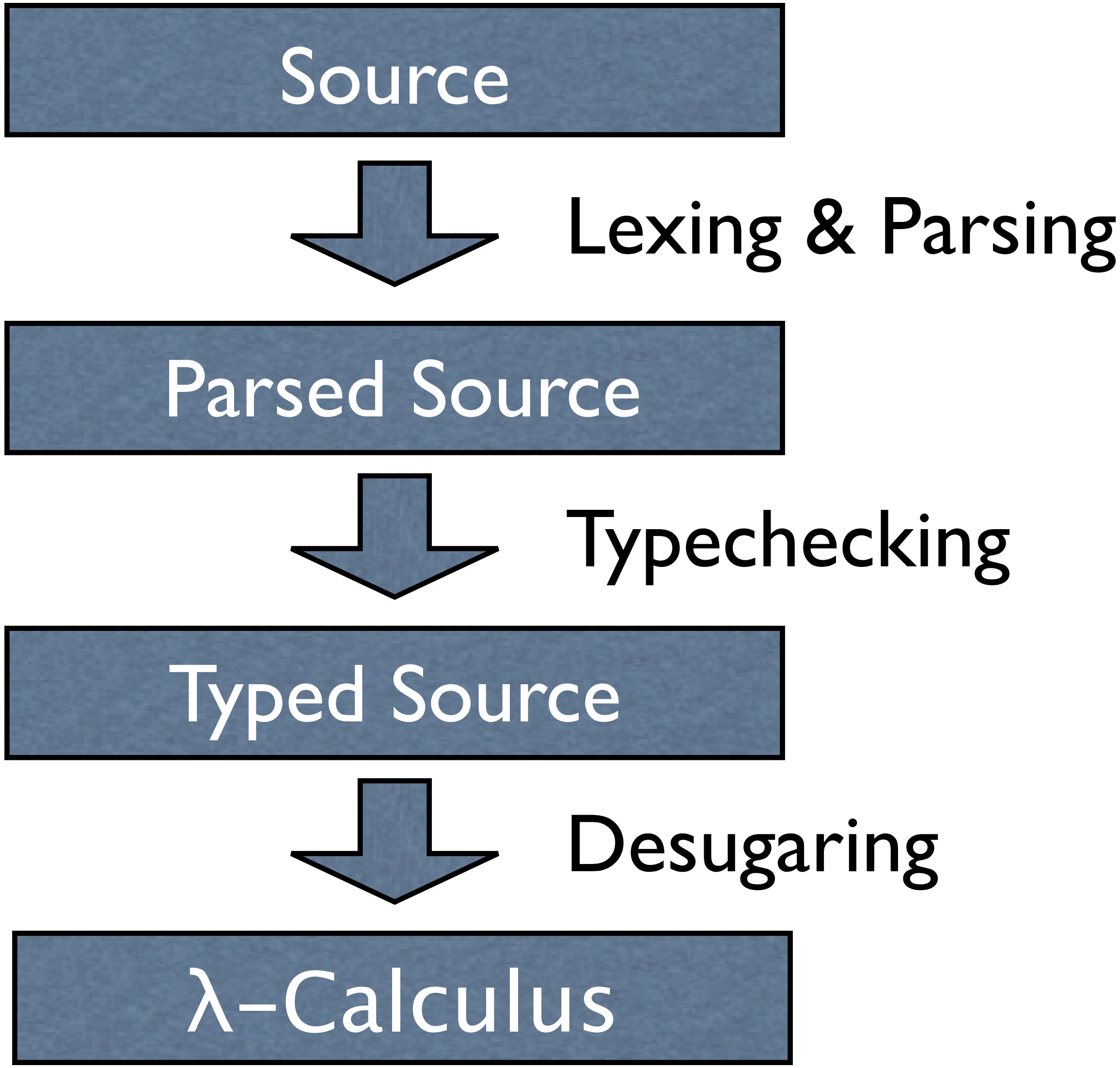
Wolfgang Thaller

presented for CAS 706, Winter 2005

Challenges

- Typechecking
- Memory Management
  (we <u>need</u> Garbage Collection)
- Polymorphism
- Higher Order Functions
- Lazy Evaluation

Phases



Source

Lexing & Parsing

Parsed Source

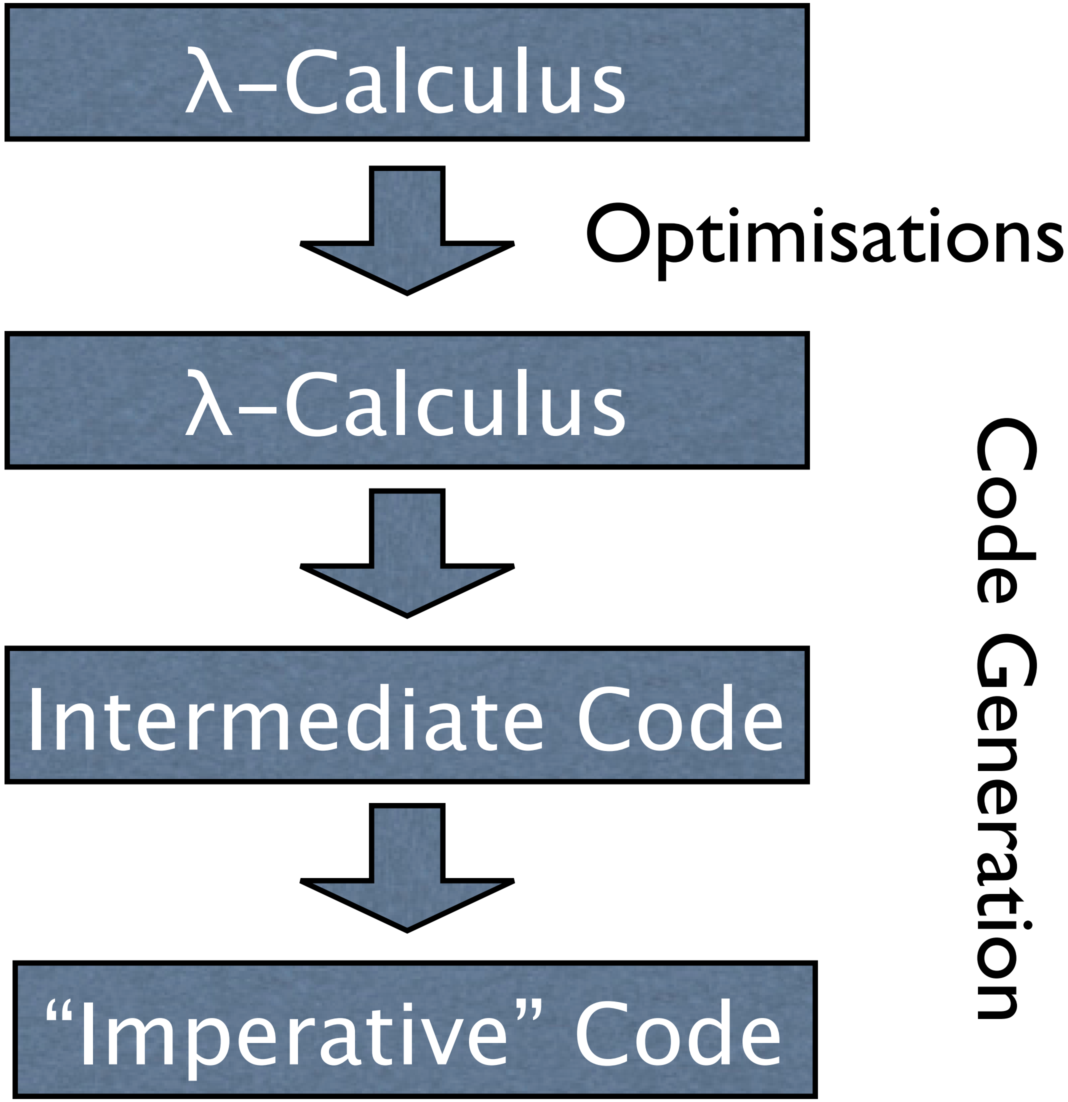
Typechecking

Typed Source

Desugaring

λ–Calculus

Phases (contd.)



λ–Calculus

Optimisations

λ–Calculus

Code Generation

Intermediate Code

“Imperative” Code

Enriched λ–Calculus

- Just the basics of functional programming
- Everything else is just syntactic sugar*.
- Let's "desugar" to a simpler language.

---

* Syntactic sugar causes cancer of the semicolon.
-- Alan Perlis

λ–Calculus (contd.)

- Lambda Calculus
  - variables, constants
  - λ–abstraction, application
- Extended by:
  - let, letrec
  - algebraic datatypes, case
  - lots of built-in functions

Desugaring (1)

```
map f [] = []
map f (x:xs) = f x : map f xs
```

```
map = λ f . λ ys .
    case ys of
      Nil          -> Nil
      Cons x xs ->
        Cons (f x) (map f xs)
```

Desugaring (2)

```
class Show a where
    show :: a -> String

print :: Show a => a -> IO ()
print x = putStrLn (show x)
```

```
print :: (a -> String)
      -> a -> IO ()
print = λs. λx. putStrLn (s x)
```

Abstract Machines

- λ–calculus ≠ “real” computers
- define an “abstract machine” that matches FP more closely
- ... but still has “useful” operational semantics

Abstract Machines

- The G Machine
  (Augustson, Johnson, 1984)
- The Spineless Tagless G (STG) Machine
  (Peyton Jones, 1992)
- Eval/Apply STG (GHC ≥ 6.0)
  (Marlow, Peyton Jones, 2004)
- KAM (MLKit)
  (Elsman, Hallenberg 2002)
- And many more...

Garbage Collection

- No, we don’t want to call free().
- Heap allocation is **cheap**
  (with a copying collector).
- The Garbage Collector needs to
  - know all pointers
  - distinguish pointers from non-pointers

Polymorphism

- Monomorphisation
  (e.g. C++ templates, MLton)
- Pass extra information
  (e.g. qsort in C needs size of element)
- Uniform Representation
  (everything is a pointer; “boxed objects”)

Values in the Heap

- Heap object needs to contain information for the garbage collector
- Lazy evaluation: could be an unevaluated expression (a “thunk”)
- Maybe use a tag bit to distinguish values from thunks?
- Always need to check whether an object is evaluated

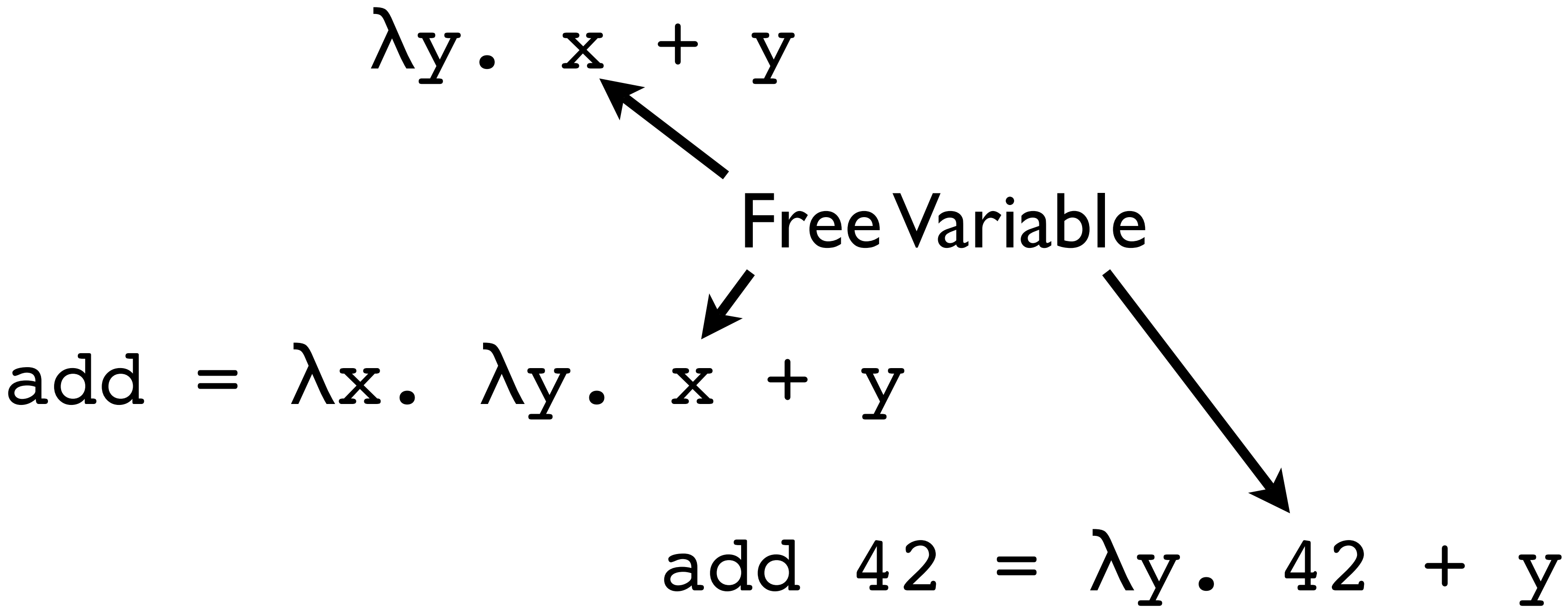
Functions

- In λ-calc, a function takes exactly one argument:
  `add = λx. λy. x + y`
- Handle multiple (curried) arguments at once for efficiency
  `add = λx y. x + y`
- ... or just prefer to use tuples as parameters:
  `add = λ(x,y). x + y`

Functions as Values

- Functions are first-class values
- A function is not just statically compiled code, it also “contains” some data
- represented by pointer to a “closure” (data structure with code pointer + data)
- calling a function directly remains simple

Free Variables



`λy. x + y`

Free Variable

`add = λx. λy. x + y`

`add 42 = λy. 42 + y`

- A pointer to a piece of code (like a C function pointer) is **not enough**
- We need to include the values for the free variables

Partial Application

```
add = λx y. x + y
```

- This function has “arity” 2
- The code expects two arguments
- If we call it with just one argument,
  we construct a “partial application node” on the heap:

```
add 42 = λy. add 42 y
```

- A partial application node is itself a function closure.

Push/Enter vs. Eval/Apply

- Who decides whether we passed enough arguments?
- The called function (push/enter)
- The caller (eval/apply)

Push/Enter

- Use a second, separate stack for argument passing
- At the beginning of a function, check whether there are enough arguments available
- If yes, take them from the stack, if no, construct a partial application node
- This method is traditionally used for lazy functional programming languages.

PUSH/ENTER

IS

DEAD

Eval/Apply

- The caller is responsible for:
- making sure the function itself is evaluated (not a thunk)
- checking how many arguments the function wants
- ... and proceeding accordingly
- This can be handled by code in the run-time system

Thunks

- a thunk represents an unevaluated expression in a lazy language
- ≈ a function without arguments:
  code pointer + free variables
- after evaluation is done, “update” the thunk (who is responsible for updating?)

Indirections

- If the result is no larger than the thunk was, just overwrite the thunk
- If the result is larger than the thunk was, allocate the result elsewhere and overwrite the thunk with an “indirection” that points to the value
- Indirections can be removed by the GC

The STG Machine

- “Spineless Tagless G Machine”
- Simon Peyton Jones, 1992
- intended for lazy languages
- used in the Glasgow Haskell Compiler

STG: Closures

- Uniform representation:
  a heap object *always* consists of...
  - A pointer to the "entry code"
  - Values for the free variables of that code
- If the object is already evaluated, the code will just "return" the value
- Indirections are trivial to implement
- No Tags necessary: Tagless

STG: The Stack

- The stack contains "activation records"
- An activation record is a return address plus values for free variables used by that code
- When eval/apply is used, this is almost like in C.

The STG Language

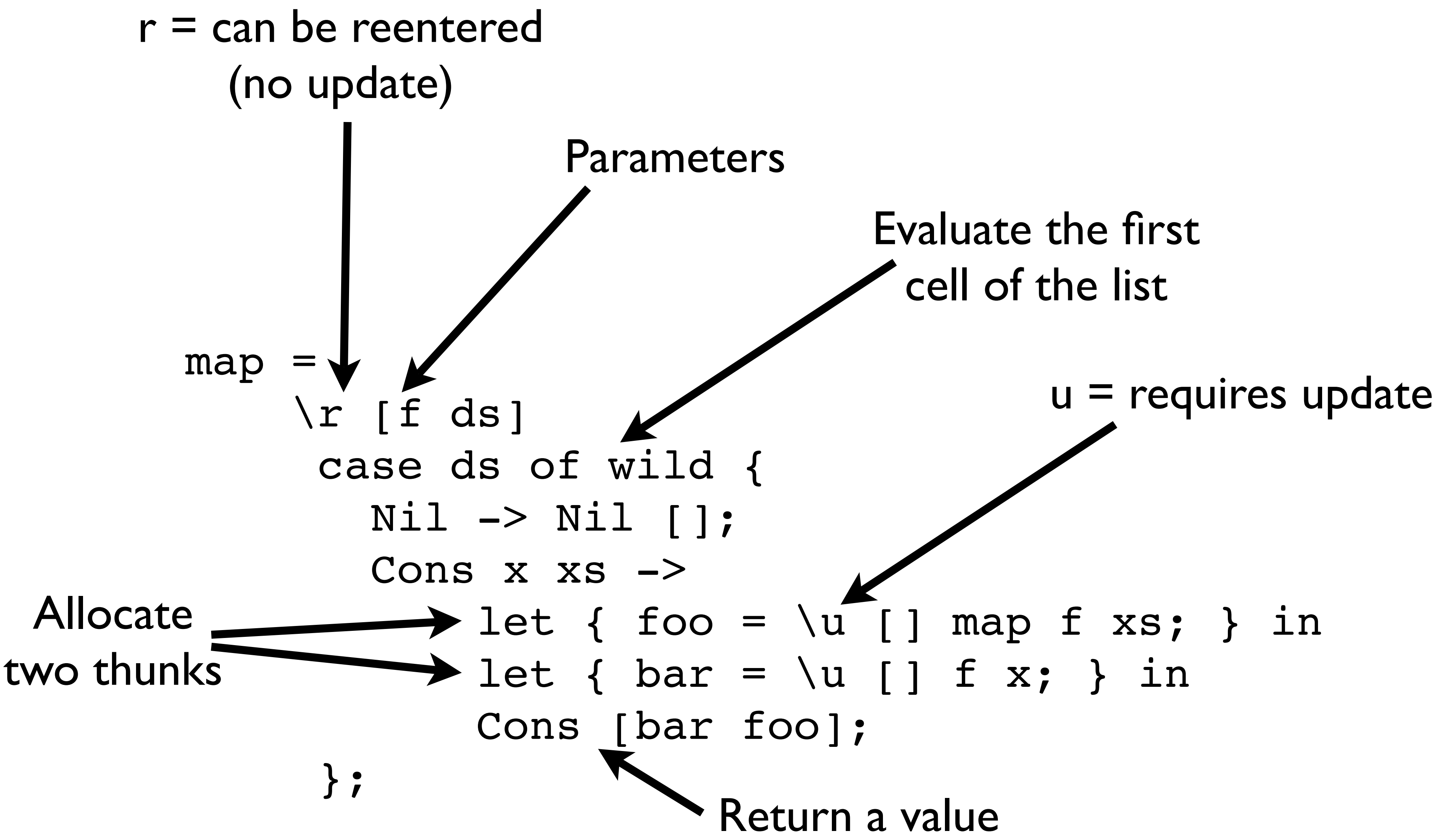
- Functional Intermediate code
- other abstract machines use instruction lists
- operational semantics:
  - let means allocate memory
  - case means evaluate something

The STG Language

```
  map = λ f . λ ys .
      case ys of
        Nil          -> Nil
        Cons x xs ->
          Cons (f x) (map f xs)
```

```
map =
    \r [f ds]
     case ds of wild {
       Nil -> Nil [];
       Cons x xs ->
           let { foo = \u [] map f xs; } in
           let { bar = \u [] f x; } in
           Cons [bar foo];
     };
```

The STG Language



r = can be reentered
(no update)

Parameters

Evaluate the first
cell of the list

u = requires update

Allocate
two thunks

Return a value

```
map =
    \r [f ds]
     case ds of wild {
       Nil -> Nil [];
       Cons x xs ->
           let { foo = \u [] map f xs; } in
           let { bar = \u [] f x; } in
           Cons [bar foo];
     };
```

STG: Updates

- At the beginning of the code that evaluates a thunk, push an “update frame”
- The update frame’s entry code is in the run-time system
- It performs the update (using an indirection), then returns to the next activation record on the stack.

STG: Vectored Returns

- When you call a function whose return type is an ADT
- Instead of one return address, push one return address for each constructor