Compilation of Functional Programming Languages

Wolfgang Thaller
presented for CAS 706, Winter 2005
Challenges

- Typechecking
- Memory Management (we need Garbage Collection)
- Polymorphism
- Higher Order Functions
- Lazy Evaluation
Phases

1. Source
2. Lexing & Parsing
3. Parsed Source
4. Typechecking
5. Typed Source
6. Desugaring
7. \( \lambda \)-Calculus
Phases (contd.)

- λ-Calculus
  - Optimisations
    - λ-Calculus
      - Intermediate Code
        - “Imperative” Code
          - Code Generation
Enriched $\lambda$-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)

\[
\text{map } f \; [] = [] \\
\text{map } f \; (x:xs) = f \; x : \text{map } f \; xs
\]

\[
\text{map} = \lambda f . \lambda ys . \\
\text{case } ys \text{ of} \\
\quad \text{Nil} \quad \rightarrow \quad \text{Nil} \\
\quad \text{Cons } x \; xs \rightarrow \\
\quad \text{Cons } (f \; x) \; (\text{map } f \; xs)
\]
class Show a where
    show :: a -> String
print :: Show a => a -> IO ()
print x = putStrLn (show x)

Desugaring (2)

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

• $\lambda$-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...
• 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 \( \lambda \)-calc, a function takes exactly one argument:
  \[
  \text{add} = \lambda x. \lambda y. x + y
  \]

• Handle multiple (curried) arguments at once for efficiency
  \[
  \text{add} = \lambda x \ y. x + y
  \]

• ... or just prefer to use tuples as parameters:
  \[
  \text{add} = \lambda (x, y). x + y
  \]
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

\[ \lambda y. x + y \]

\[ \text{add} = \lambda x. \lambda y. x + y \]

\[ \text{add } 42 = \lambda y. 42 + y \]

• 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

• 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:

\[
\text{add } 42 = \lambda y. \text{add } 42 \ y
\]

• A partial application node is itself a function closure.

\[
\text{add } = \lambda x \ y. \ x + y
\]
Push/Enter vs. Eval/Apply

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

\[ \text{map} = \lambda f . \lambda ys . \]
\[ \text{case } ys \text{ of} \]
\[ \text{Nil} \rightarrow \text{Nil} \]
\[ \text{Cons } x \text{ } xs \rightarrow \]
\[ \text{Cons } (f \ x) \ (\text{map } f \ xs) \]

\[ \text{map} = \]
\[ \backslash r \ [f \ ds] \]
\[ \text{case } ds \text{ of wild } \{ \]
\[ \text{Nil} \rightarrow \text{Nil } [] ; \]
\[ \text{Cons } x \text{ } xs \rightarrow \]
\[ \text{let } \{ \text{foo} = \backslash u \ [ ] \text{map } f \ xs ; \} \text{ in} \]
\[ \text{let } \{ \text{bar} = \backslash u \ [ ] f \ x ; \} \text{ in} \]
\[ \text{Cons } [\text{bar } \text{foo}] ; \]
\[ \} ; \]
The STG Language

\[
\begin{align*}
\text{map} &= \lambda r \ [f \ ds] \\
\text{case } ds \ \text{of } \text{wild } \{ \\
\quad \text{Nil } &\to \text{Nil } [] \\
\quad \text{Cons } x \ \text{xs } &\to \\
\begin{align*}
\text{let } &\{ \text{foo } = \lambda u \ [] \ \text{map } f \ \text{xs}; \} \ \text{in} \\
\text{let } &\{ \text{bar } = \lambda u \ [] \ f \ x; \} \ \text{in} \\
\text{Cons } &\text{[bar foo]};
\end{align*}
\}\end{align*}
\]

Parameters
- \(r\) = can be reentered (no update)
- \(u\) = requires update

Evaluate the first cell of the list

Allocate two thunks

Return a value
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 runtime system
- It performs the update (using an indirection), then returns to the next activation record on the stack.
• When you call a function whose return type is an ADT

• Instead of one return address, push one return address for each constructor