Compilation of non-strict functional programming languages

Harald Steinlechner

UT Vienna

Seminar in Übersetzerbau

Motivation: Streams and recursive Definitions

```
[2^n | n <- [0..]]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
    0 1 1 2 3 5  8..
(+) 1 1 2 3 5 8  13...
===================
0 1 1 2 3 5 8 13 21...
```

```
 > take 10 fibs
 [0,1,1,2,3,5,8,13,21,34]
 let fib n = fibs!!n
```

Computing *n* fibs takes *n* additions.

What is required - some Semantics

```
-- function, which constantly returns 1
const1 :: Int -> Int
const1 _ = 1
bomb = bomb -- endless loop
> const1 bomb
```

Normal-order strategy

The leftmost, outermost redex is always reduced first

Call-by-name strategy

Normal order but no reductions inside abstractions. $\lambda x.id\ x \not\twoheadrightarrow$

Call-by-value strategy

Only outermost redexes are reduced and only when the right-hand-side has been reduced to a value.

A clever Implementation Technique: Lazy Evaluation

Call-by-need

Optimization of *call-by-name*. Instead of re-evaluating an argument each time it is used, overwrite all occurrences when evaluated once.

Call-by-need = Call-by-name + Sharing

- Only evaluate if really necessary
- Evaluate at most once

$$(\lambda x.\ And\ x\ x)\ (Not\ True) \Rightarrow And\ (Not\ True)\ (Not\ True)$$

Advantages

Composition [Hughes, 1989]

```
sort :: [Int] -> [Int]
sort = ... -- a clever written sort algorithm
minimumElement :: [Int] -> Int
minimumElement = head . sort
```

Programmer comfort

Lazy evaluation decouples evaluation of an expression from its binding. If never used - do not evaluate it. Short circuiting by default:

```
myIf p thenE elseE  = if p then thenE elseE
```

Recursive Values

Compute values only on demand. Infinite streams can be useful!

```
pow2s = 1 : map (*2) pow2s
```

Other issues

Functional languages are slim, but each element must be fast

- Heavy use of *Higher-Order-Functions*
- Currying should be `FAST`
- Pattern matching central. Must be `FAST`
- Many allocations

Today: Compilation of Lazy Evaluation

The Source Language

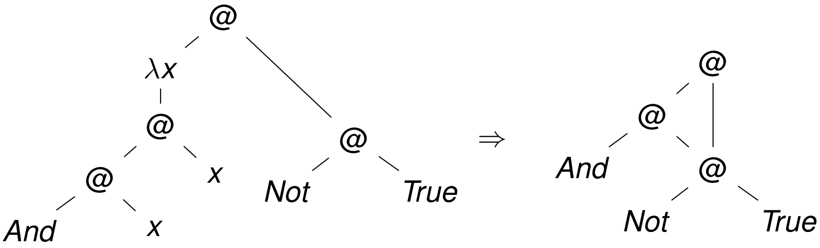
```
<exp> ::= <constant>                 -- Built-in constants
      |   <variable>                 -- Variable names
      |   <exp> <exp>                -- Applications
      |   Lam <variable> <exp>       -- Abstractions
      |   Let <variable> = <exp> -- Binding
          In <exp>
      |   If <exp> Then <exp> Else <exp>
      |   Plus, Minus, Not...        -- Built-in functions
```

Core-Language [Jones and Lester, 1992]

- λ-calculus enriched with *Builtins*, *Let-Bindings* and *Constants*.
- Small IR used in production Compilers (like *The Glasgow Haskell Compiler* [Marlow and Jones, 2012])

Graph-Reduction [Wadsworth, 1971]

$$(\lambda x.\ And\ x\ x)\ (Not\ True) \Rightarrow And\ (Not\ True)\ (Not\ True)$$



Pointer substitution

Instead of copying arguments onto occurrences of formal parameters
⇒ only update Pointers!

Let us generate code!

$$\lambda x.(\lambda y.x + y)$$

Free Variables

- Applications instantiate new expressions
- We need fixed code sequences for each sub-expression
- Treatment of free variables not clear
- Introduce *Environment* which holds free variables (like in the SECD-machine [Landin, 1964]).
- How to ensure sharing with these Environments?

A fundamentally different approach: *SKI*-Combinators

This approach was used in the SASL-Implementation [Turner, 1979].

$$\begin{aligned} S\ f\ g\ x &= f\ x\ (g\ x) && \textbf{(S)} \\ K\ x\ y &= x && \textbf{(K)} \\ I\ x &= x && \textbf{(I)} \end{aligned}$$

- Any computable function can be expressed in λ-calculus as well as in SKI-Combinators
- Any λ term can be transformed into *SKI* (and vice versa)
- Transform term in (enriched) λ-calculus into equivalent *SKI-Combinator*.
- *SKI-Terms* have no free variables!

How to compile to *SKI*

$$\lambda x.\ x \rightarrow I \quad \textbf{(I-Transformation)}$$
$$\lambda x.\ c \rightarrow K\ \ c \quad \textbf{(K-Transformation)}$$
$$\lambda x.\ e_1 e_2 \rightarrow S(\lambda x.\ e_1)(\lambda x.\ e_2) \quad \textbf{(S-Transformation)}$$

$$(\lambda x. + x\ x)\ 5$$
$$\rightarrow^S\ \ S\ (\lambda x. + x)(\lambda x.x)\ 5$$

How to compile to *SKI*

$$\lambda x.\ x \to I \quad \textbf{(I-Transformation)}$$
$$\lambda x.\ c \to K\ \ c \quad \textbf{(K-Transformation)}$$
$$\lambda x.\ e_1 e_2 \to S(\lambda x.\ e_1)(\lambda x.\ e_2) \quad \textbf{(S-Transformation)}$$

$$\begin{aligned} & (\lambda x. + x\ x)\ 5 \\ \to^S\ & S\,(\lambda x. + x)(\lambda x.x)\ 5 \\ \to^S\ & S\,(S\,(\lambda x.+)\,(\lambda x.x))\,(\lambda x.x)\ 5 \end{aligned}$$

How to compile to *SKI*

$$\lambda x.\ x \rightarrow I \quad \textbf{(I-Transformation)}$$
$$\lambda x.\ c \rightarrow K\ \ c \quad \textbf{(K-Transformation)}$$
$$\lambda x.\ e_1 e_2 \rightarrow S(\lambda x.\ e_1)(\lambda x.\ e_2) \quad \textbf{(S-Transformation)}$$

$$\begin{aligned} & (\lambda x. + x\ x)\ 5 \\ \rightarrow^{S}\ \ & S\ (\lambda x. + x)(\lambda x.x)\ 5 \\ \rightarrow^{S}\ \ & S\ (S\ (\lambda x.+)\ (\lambda x.x))\ (\lambda x.x)\ 5 \\ \rightarrow^{I}\ \ & S\ (S\ (\lambda x.+)\ I)\ (\lambda x.x)\ 5 \end{aligned}$$

How to compile to *SKI*

$$\lambda x.\ x \rightarrow I \quad \textbf{(I-Transformation)}$$
$$\lambda x.\ c \rightarrow K\ \ c \quad \textbf{(K-Transformation)}$$
$$\lambda x.\ e_1 e_2 \rightarrow S(\lambda x.\ e_1)(\lambda x.\ e_2) \quad \textbf{(S-Transformation)}$$

$$
\begin{aligned}
& (\lambda x. + x\ x)\ 5 \\
\rightarrow^{S}\ & S\ (\lambda x. + x)(\lambda x.x)\ 5 \\
\rightarrow^{S}\ & S\ (S\ (\lambda x.+)\ (\lambda x.x))\ (\lambda x.x)\ 5 \\
\rightarrow^{I}\ & S\ (S\ (\lambda x.+)\ I)\ (\lambda x.x)\ 5 \\
\rightarrow^{I}\ & S\ (S\ (\lambda x.+)\ I)\ I\ 5
\end{aligned}
$$

How to compile to *SKI*

$$\lambda x.\ x \rightarrow I \quad \textbf{(I-Transformation)}$$
$$\lambda x.\ c \rightarrow K\ \ c \quad \textbf{(K-Transformation)}$$
$$\lambda x.\ e_1 e_2 \rightarrow S(\lambda x.\ e_1)(\lambda x.\ e_2) \quad \textbf{(S-Transformation)}$$

$$\begin{aligned}
& (\lambda x. + x\ x)\ 5 \\
\rightarrow^S\ & S\ (\lambda x. + x)(\lambda x.x)\ 5 \\
\rightarrow^S\ & S\ (S\ (\lambda x.+)\ (\lambda x.x))\ (\lambda x.x)\ 5 \\
\rightarrow^I\ & S\ (S\ (\lambda x.+)\ I)\ (\lambda x.x)\ 5 \\
\rightarrow^I\ & S\ (S\ (\lambda x.+)\ I)\ I\ 5 \\
\rightarrow^K\ & S\ (S\ (K\ +)\ I)\ I\ 5
\end{aligned}$$

How to evaluate *SKI*

$$
\begin{aligned}
S\ f\ g\ x &\Rightarrow f\ x\ (g\ x) \quad \textbf{(S-Reduction)}\\
K\ x\ y &\Rightarrow x \quad \textbf{(K-Reduction)}\\
I\ x &\Rightarrow x \quad \textbf{(I-Reduction)}
\end{aligned}
$$

$$
\begin{aligned}
& S\ (S\ (K\ +)\ I)\ I\ 5\\
\Rightarrow\ & S\ (K\ +)\ I\ 5\ (I\ 5)
\end{aligned}
$$

How to evaluate *SKI*

$$S\ f\ g\ x \Rightarrow f\ x\ (g\ x)\quad \textbf{(S-Reduction)}$$
$$K\ x\ y \Rightarrow x\quad \textbf{(K-Reduction)}$$
$$I\ x \Rightarrow x\quad \textbf{(I-Reduction)}$$

$$\begin{aligned} & S\ (S\ (K\ +)\ I)\ I\ 5 \\ \Rightarrow\ & S\ (K\ +)\ I\ 5\ (I\ 5) \\ \Rightarrow\ & K\ +\ I\ 5\ (I\ 5) \end{aligned}$$

How to evaluate *SKI*

$$S\ f\ g\ x \Rightarrow f\ x\ (g\ x) \quad \textbf{(S-Reduction)}$$
$$K\ x\ y \Rightarrow x \quad \textbf{(K-Reduction)}$$
$$I\ x \Rightarrow x \quad \textbf{(I-Reduction)}$$

$$\begin{aligned} & S\ (S\ (K\ +)\ I)\ I\ 5 \\ \Rightarrow\ & S\ (K\ +)\ I\ 5\ (I\ 5) \\ \Rightarrow\ & K\ +\ I\ 5\ (I\ 5) \\ \Rightarrow\ & +\ (I\ 5)\ (I\ 5) \end{aligned}$$

How to evaluate *SKI*

$$\begin{aligned} S\ f\ g\ x \Rightarrow f\ x\ (g\ x) &\quad \textbf{(S-Reduction)} \\ K\ x\ y \Rightarrow x &\quad \textbf{(K-Reduction)} \\ I\ x \Rightarrow x &\quad \textbf{(I-Reduction)} \end{aligned}$$

$$\begin{aligned} & S\ (S\ (K\ +)\ I)\ I\ 5 \\ \Rightarrow\ & S\ (K\ +)\ I\ 5\ (I\ 5) \\ \Rightarrow\ & K\ +\ I\ 5\ (I\ 5) \\ \Rightarrow\ & +\ (I\ 5)\ (I\ 5) \\ \Rightarrow\ & +\ 5\ (I\ 5) \\ \Rightarrow\ & +\ 5\ 5 \\ \Rightarrow\ & 10 \end{aligned}$$

Analysis

- `S` is the only 'complex' reduction
- Very small steps
- Introduce other (shortcut) combinators ⇒ higher compilation effort, less reduction steps.
- Native code-generation possible (we will see how later on)

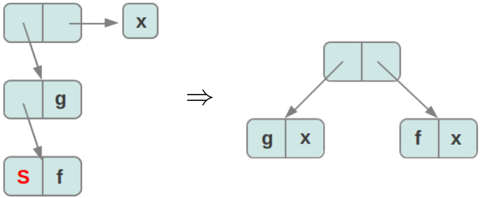


Figure: Reduction of $S\ f\ g\ x \Rightarrow f\ (g\ x)$. Each Application-Node is represented as cell with two elements.

Another approach: Supercombinators [Hughes, 1982]

- Combinators have no free variables
- Eliminate free variables systematically via subsequent transformations
- Notation of *Full-Laziness*
- Pass free varables as extra arguments

Supercombinators

A supercombinator S of arity n is a lambda expression of the form:

$$\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$$

where E is not a lambda abstraction such that:

- S has no free variables
- any lambda abstraction in E is a supercombinator
- $n \geq 0$, that is, there need to be no λ's at all.

Lambdalifting by example

```
(Lam x . Lam y . x + y) 4 5
```

$\lambda y \mathrel{.} x + y$ is no combinator. Lift x.

```
(Lam x . (Lam w y . w + y) x) 4 5
```

Next we give the supercombinator a name, say $\$Y$

```
Let $Y w y = w + y
In (Lam x . $Y x) 4 5
```

The remaining expression is itself a combinator—and again we give it a name.

```
Let $Y w y = w + y
In
  Let $X x = $Y x
  In $X 4 5
```

The G-Machine [Johnsson, 1984]

- Lambda-Lifting sensible for laziness
- Generate code for each combinator
- Combinator *applications* must perform proper substituion
- Introduce abstract machine which performs graph reduction
- Perform optimizations on abstract machine code
- Finally: emit machine code

Tagging

Value kinds

In the G-Machine cells may be of four kinds:

1. application
2. cons-Cel
3. primitive type like `int`
4. *n*-ary application of a supercombinator

- Call sites in general unknown. Children of *application* nodes may be applications again, or primitive values etc.
- Reduction code depends on kind of the cell
- Inspect tag and dispatch reduction code

The Spineless Tagless Machine (STG) [Jones, 1992]

Improvements over G-Machine

- Uniform representation of suspensions and values
- Safes indirections respective value dispatch
- If evaluated, each closure patches its code pointer to return the cached value.
- Supports unboxed values
- Better support for pattern matching etc.

Recent developments (1)

Dynamic Pointer Tagging [Marlow et al., 2007]

- STG has many indirect jumps
- Each closure is entered even if evaluated (over and over again!)
- `eval` often called on values. Indirect jump is expensive
- Closures always *word-size* aligned. 2bits on 32bit unused!
- Encode status of closure in *LSB* bits and perform conditional jumps - no need to enter closures for values
- Performance, up to $+14\%$, ($+2\%$ code size)

Recent Developments (2)

Optimistic Evaluation [Ennals, 2004]

- Constant overhead for lazy evaluation (and space leaks!)
- Idea: evaluate thunk speculatively - if evaluation seems to diverge: abort execution and continue lazily.
- Lives in separate branch of GHC
- GHC is a static compiler, not optimal for ‘dynamic‘ optimizations
- Performance, up to $+30\%$ performance

Conclusions

- Some good reasons for lazy evaluation
- Basic approaches and Implementation techniques
- Recent developments in this field
- More details in paper

Questions?

Bibliography

Ennals, R. J. (2004).
Adaptive evaluation of non-strict programs.
Technical report.

Hughes, J. (1989).
Why functional programming matters.
Comput. J., 32(2):98–107.

Hughes, R. J. M. (1982).
Super-combinators a new implementation method for applicative languages.
In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, LFP '82, pages 1–10, New York, NY, USA. ACM.

Johnsson, T. (1984).
Efficient compilation of lazy evaluation.
In *SIGPLAN NOTICES*, pages 58–69.