Compiler construction
in4020 – lecture 13

Koen Langendoen
Delft University of Technology
The Netherlands

Functional programming

- languages: LISP, Scheme, ML, Miranda, Haskell
- features:
  - high abstraction level: what vs how, where, when
  - equational reasoning
  - functions as first class citizens

compiler must work harder!!

Overview of a typical functional compiler

high-level language
(Haskell)
desugaring
type inference
functional core
optimizations
code generation

C + runtime system

Function application

- concise notation

<table>
<thead>
<tr>
<th>Haskell</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f \ 11 \ 13 )</td>
<td>( f(11, 13) )</td>
</tr>
</tbody>
</table>

- precedence over all other operators

<table>
<thead>
<tr>
<th>Haskell</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f \ n+1 )</td>
<td>( f(n) + 1 )</td>
</tr>
</tbody>
</table>

Syntactic sugar

- offside rule: end-of-equation marking
- list notation: \([ \) \([1,2,3 \) \((1 : (2 : (3 : []) \))\]
- pattern matching: case analysis of arguments
- list comprehension: mathematical sets

Polymorphic typing

- an expression is polymorphic if it ‘has many types’
- examples
  - empty list: \([]\)
  - list handling functions

  \[
  \begin{align*}
  \text{length} & : [a] \rightarrow \text{Int} \\
  \text{length} [\] & = 0 \\
  \text{length} (x:xs) & = 1 + \text{length} \ xs
  \end{align*}
  \]
Polymorphic type inference

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

\[\text{map} :: t_1 \rightarrow t_2 \rightarrow t_3\]

\[\text{map} \ [\ ] = [\ ]\]

\[\text{map} :: t_1 \rightarrow [a] \rightarrow [b]\]

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

\[x :: a\]

\[f :: a \rightarrow b\]

\[\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]\]

Exercise (5 min.)

- infer the polymorphic type of the following higher-order function:

\[
\text{filter} \ f \ [\ ] = [\ ] \\
\text{filter} \ f \ (x:xs) = \text{if} \not{f} \ x \ \text{then} \ \text{filter} \ f \ xs \\
\text{else} \ x : (\text{filter} \ f \ xs)
\]

Answers

Referential transparency

\(f \ x\) always denotes the same value

- advantage: high-level optimization is easy
  
  \[\text{let} \ a = f \ x \ \text{in} \ g \ a \ a\]

- disadvantage: no efficient in-place update
  
  \[\text{add} \_ \_ \_ \ [\ ] = [\ ]\]

\[\text{add} \_ \_ \_ \ (x:xs) = x+1 : \text{add} \_ \_ \_ \ xs\]

Higher-order functions

- functions are first-class citizens
- higher-order functions accept functions as parameters and/or return a function as result
- functions may be created “on the fly”

\[D \ f = f' \ \text{where} \ f'(x) = \lim_{h\downarrow 0} \frac{f(x+h) - f(x)}{h}\]

\[\text{diff} \ f = f_\_ \ \text{where} \]

\[f_\_ \ x = (f \ (x+h) - f \ x) / h\]

\[\text{h} = 0.0001\]

Currying: specialize functions

\[\text{deriv} \ x = (f \ (x+h) - f \ x) / h\]

\[\text{h} = 0.0001\]

\[Q:\ \text{diff} \ \text{(unary function)} \equiv \text{deriv} \ \text{(binary function)}?\]

\[A:\ \text{yes!} \ \forall f, \ \forall x \ (\text{diff} \ f) \ x = \text{deriv} \ f \ x\]

binary function \(\equiv\) a unary function returning a unary function

\(f \ e_1 \ldots e_n \equiv (f \ e_1) \ldots e_n\)

(\text{deriv square}) is a curried function
**Lazy evaluation**

An expression will only be evaluated when its value is needed to progress the computation

- additional expressive power (infinite lists)
  
  $$\text{deriv } f x = \lim \left\{ \left( f (x+h) - f x \right)/h \mid h \leftarrow \text{downto } 0 \right\}$$

  where
  
  $$\text{downto } x = [x + 1/2^n \mid n \leftarrow \{1\ldots\}]$$

  $$\text{lim } (a:b:lst) = \text{if } \text{abs } (a/b - 1) < \epsilon\text{ then } b \text{ else } \text{lim } (b:lst)$$

- overhead for delaying/resuming computations

**Graph reduction**

- implement h.o.f + lazy evaluation
- key: function application
  
  $$\text{f e}_1 \ldots \text{e}_n \equiv (\text{f } \text{e}_1) \text{e}_2) \ldots \text{e}_n$$

- execution (interpretation)
  
  - build graph for main expression
  
  - find reducible expression (redex = func + args)
  
  - instantiate body (build graph for rhs)

**Reduction order**

- a graph may contain multiple redexes
- lazy evaluation: choose top-most @-node

  - built-in operators (+, -, *, etc) may have strict arguments that must be evaluated => recursive invocation

**Example**

let

$$\text{let } \text{twice f x} = \text{f (f x)}$$

$$\text{square } n = n^2$$

in

$$\text{twice square } 3$$

**Implementation**

Graph reduction

- find next redex
- instantiate rhs
- update root

- unwind application spine ($f(a_1 \ldots a_n)$)
- call f, pass arguments in array (stack)
- update root with result

typedef struct node *Pnode;
extern Pnode eval( Pnode root);

```c
Pnode mul(Pnode arg[]) {
    Pnode a = eval(arg[0]);
    Pnode b = eval(arg[1]);
    return Num(a->nd.num * b->nd.num);
}
```

```c
typedef struct node *Pnode;
extern Pnode eval( Pnode root);
```
**Code generation**

```plaintext
average a b = (a+b) / 2

Pnode average(Pnode arg[]) {
    Pnode a = arg[0];
    Pnode b = arg[1];
    return Appl(Appl(fun_div, Appl(Appl(fun_add,a),b)), Num(2));
}
```

**Short-circuiting application spines**

```plaintext
average a b = (a+b) / 2

Pnode average(Pnode arg[]) {
    Pnode a = arg[0];
    Pnode b = arg[1];
    return div(Appl(Appl(fun_add,a),b), Num(2));
}
```

* call leftmost outermost function directly

**Strict arguments**

```plaintext
average a b = (a+b) / 2

Pnode average(Pnode arg[]) {
    Pnode a = arg[0];
    Pnode b = arg[1];
    return div(add(a,b), Num(2));
}
```

* evaluate expressions supplied to strict built-in functions immediately

**Strictness analysis**

user-defined functions:

- propagate sets of strict arguments up the AST

```plaintext
foo x y = if x>0 then x*y else 0
```

**Strictness propagation**

<table>
<thead>
<tr>
<th>language construct</th>
<th>propagated set</th>
</tr>
</thead>
<tbody>
<tr>
<td>L @ R</td>
<td>L U R</td>
</tr>
<tr>
<td>if C then T else E</td>
<td>C U (T \ E)</td>
</tr>
<tr>
<td>fun_m @ A_1 @ ... @ A_n</td>
<td>[ \bigcup_{i=n}^{m} \text{strict}(fun,i) A_i \text{ if } n \geq m ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Recursive functions**

```plaintext
foo x y = if x>0 then y else foo (x+1) y
```

* problem: conservative estimation of strictness
Recursive functions

\[ \text{foo } x \ y = \begin{cases} y & \text{if } x > 0 \\ \text{foo } (x+1) \ y & \text{else} \end{cases} \]

\[ \text{solution: } \begin{cases} y & \text{if } x > 0 \\ \text{foo } (x+1) \ y & \text{else} \end{cases} \]

Answer (6 min.)

- infer the strict arguments of the following recursive function:

\[ g \ x \ y \ 0 = x \]
\[ g \ x \ y \ z = g \ y \ x \ (z-1) \]

- how many iterations are needed?

Summary

<table>
<thead>
<tr>
<th>Haskell feature</th>
<th>compiler phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>offside rule</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>list notation</td>
<td>parser</td>
</tr>
<tr>
<td>list comprehension</td>
<td>semantic analyzer</td>
</tr>
<tr>
<td>pattern matching</td>
<td></td>
</tr>
<tr>
<td>polymorphic typing</td>
<td></td>
</tr>
<tr>
<td>referential transparency</td>
<td></td>
</tr>
<tr>
<td>higher-order functions</td>
<td></td>
</tr>
<tr>
<td>lazy evaluation</td>
<td></td>
</tr>
<tr>
<td>run-time system</td>
<td></td>
</tr>
<tr>
<td>(graph reducer)</td>
<td></td>
</tr>
</tbody>
</table>

Answers

<table>
<thead>
<tr>
<th>step</th>
<th>assumption</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TODO

- assignment 2:
  - make Asterix OO
  - deadline June 4 08:59
- study book
  - chapter 1 – 7, except 4.2.6
- make appointment by e-mail for oral exam
  - 30 min per group
  - koen@ubicom.tudelft.nl