The implementation and optimization of functional languages

Urmas Tamm

Tartu 2013
Intro

- “The implementation of functional programming languages“, Simon L. Peyton Jones, 1987
- “Miranda“
  - fully lazy
  - strict
  - a predecessor of Haskell
Lambda Calculus

- Theoretical framework behind functional programming
- Turing complete
- Simple intermediate language:
  - only a few syntactic constructs
  - no data-types
  - no recursion
  - graph reduction
Evaluation

- Selecting a *reducible expression* or *redex* and reducing it
- Multiple reduction orders:
  - Normal order
  - Applicative order
- All reduction orders always lead to the same normal form (Church-Rosser theorem)
  - Corollary: if lambda-term has a normal form, the normal form is unique.
Translation into Lambda Calculus

• Two ways:
  – Simplify the program until the final step only includes the change of syntax
  – Use a version of enriched lambda calculus and simplify it into ordinary lambda calculus
Enriched Lambda Calculus

- Ordinary Lambda Calculus with extra constructs:
  - `let` and `letrec` expressions
  - pattern matching lambda expressions
  - the infix operator
  - case-expressions

- Translation schemas:
  - TE – returns corresponding lambda expression
  - TD – produces a `letrec` expression
Graph reduction

• Expression is held in the form of its syntax tree
  – constant values
  – built-in functions
  – variable names

• During reduction tree becomes a graph

• Nodes represent small store areas – cells:
  – tag – type of the cell
  – fields, pointers

• Fixed-size cells vs variable-sized cells
Graph reduction

<table>
<thead>
<tr>
<th>Node type</th>
<th>Abstract node</th>
<th>Concrete cell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td>@ (f x)</td>
<td>@ x</td>
</tr>
<tr>
<td><strong>Lambda abstraction</strong></td>
<td>λx (body)</td>
<td>λ &quot;x&quot; (body)</td>
</tr>
<tr>
<td><strong>CONS cell</strong></td>
<td>: (x y)</td>
<td>: x y</td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>34</td>
<td>N 34</td>
</tr>
<tr>
<td><strong>Built-in function</strong></td>
<td>+</td>
<td>P +</td>
</tr>
</tbody>
</table>
Graph reduction

- An example syntax tree
Selecting the next redex

- Evaluator – reduces the graph to the normal form
- *Call by need vs call by value*
- Ingredients of lazy evaluation
  - Call by need
  - Arguments evaluated once or not at all
- Execution speed
- *Weak Head Normal Form (WHNF)*
Selecting the next redex

- Printing out an infinite list as it *is generated*:

```
Print( E )
begin
    E' := Evaluate( E )
    if (IsNumber( E' )) then Output( E' )
    else begin
        Print( Head( E' ) )
        Print( Tail( E' ) )
    end
end
```
Finding the next top level redex

- F may be a:
  - data object
  - built-in function
  - a lambda abstraction
The Spine Stack
Graph reduction of lambda expressions

- *Sharing* – substituting pointers instead of copying values
Graph reduction of lambda expressions

• Constructing a new instance of the lambda body
  – create a new copy of the body
  – make substitutions within the new instance
Graph reduction of lambda expressions

- Lazy graph implementation strategy:
Supercombinators and lambda-lifting

• Instantiating the body of a lambda abstraction:
  – case analysis on the tag of each node
  – test if the node is the formal parameter
  – new instances of sub-expressions containing no free occurrences of the formal parameter

• *Compilation* – a more efficient alternative
  – a fixed sequence of instructions
  – constructed in advance by a compiler

• Free variables in lambda bodies
Supercombinators and lambda-lifting

- Algorithm for generating supercombinators:
  - Choose any lambda abstraction which has no inner lambda abstractions in its body.
  - Take out all its free variables as extra parameters.
  - Give an arbitrary name to the lambda abstraction.
  - Replace the occurrence of the lambda abstraction by the name applied to the free variables.
  - Compile the lambda abstraction and associate the name with the compiled code.
Fully-lazy lambda-lifting

• Template-instantiation procedure risks constructing *multiple instances of the same expression*
• Leave out sub-expressions which contain no free occurrences of the formal parameter
• Maximal free expressions
• Using pointers and sharing results
SK Combinators

• Graph reduction technique based on a fixed set of combinators
• An extremely simple reduction machine
Garbage Collector

- Allocation of new cells during graph reduction
- *Heap* – an unordered collection of cells