Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Advanced Functional Programming

Andres Löh

Department of Information and Computing Sciences
Utrecht University

February 11, 2009

4. Debugging and Profiling

4.1 Haskell Program Coverage

Reachable uncovered code

Program code can be classified:

- unreachable code: code that simply is not used by the program, usually library code
- reachable code: code that can in principle be executed by the program

Reachable uncovered code

Program code can be classified:

- unreachable code: code that simply is not used by the program, usually library code
- reachable code: code that can in principle be executed by the program

Reachable code can be classified further:

- covered code: code that is actually executed during a number of program executions (for instance, tests)
- uncovered code: code that is not executed during testing

Reachable uncovered code

Program code can be classified:

- unreachable code: code that simply is not used by the program, usually library code
- reachable code: code that can in principle be executed by the program

Reachable code can be classified further:

- covered code: code that is actually executed during a number of program executions (for instance, tests)
- uncovered code: code that is not executed during testing

Uncovered code is untested code – it could be executed, and it could do anything!

Introducing HPC

- HPC (Haskell Program Coverage) is a tool – integrated into GHC – that can identify uncovered code.
- Using HPC is extremely simple:
  - Compile your program with the flag `-fhpc`.
  - Run your program, possibly multiple times.
  - Run `hpc report` for a short coverage summary.
  - Run `hpc markup` to generate an annotated HTML version of your source code.

What HPC does

- HPC can present your program source code in a color-coded fashion.
- Yellow code is uncovered code.
- Uncovered code is discovered down to the level of subexpressions! (Most tools for imperative language only give you line-based coverage analyis.)
- HPC also analyzes boolean expressions:
  - Boolean expressions that have always been True are displayed in green.
  - Boolean expressions that have always been False are displayed in red.

QuickCheck and HPC

QuickCheck and HPC interact well!

- Use HPC to discover code that is not covered by your tests.
- Define new test properties such that more code is covered.
- Reaching 100% can be really difficult (why?), but strive for as much coverage as you can get.

4.2 Excursion: The Lambda Calculus

The Lambda Calculus

Expressions in the lambda calculus are formed using the following grammar:

$$
\begin{array}{rlll}
e ::= & x & & \text{variables} \\
 & \mid & (e_1\ e_2) & \text{application} \\
 & \mid & \lambda x \to e & \text{lambda-abstraction}
\end{array}
$$

Bound and free variables

Every variable occurrence in a lambda term is either binding, bound or free:

$$\lambda x \to x$$
$$\lambda x \to \lambda y \to x$$
$$\lambda x \to \lambda x \to x$$
$$\lambda f \to (x\ (\lambda x \to f\ x))$$

Alpha-conversion

Binders can be renamed together with their bound variables, as long as no variables are "captured" by another binder:

> $\lambda x \to \lambda y \to x$

can be renamed to

> $\lambda z \to \lambda y \to z$

but not to

> $\lambda y \to \lambda y \to y$

This is called alpha-conversion.

Beta-reduction

A subexpression of the form

> $(\lambda x \to e_1)\ e_2$

is called a (beta-)redex and can be (beta-)reduced to

> $e_1\ [x \mapsto e_2]$

Substitution only subtitutes free occurrences of a variable, and performs alpha-conversion as needed to prevent capture of free variables in e_2.

Examples of beta-reduction

$$
\begin{aligned}
&(\lambda x \to x)\ y && \rightsquigarrow y \\
&(\lambda x \to x)\ (\lambda x \to x) && \rightsquigarrow \lambda x \to x \\
&(\lambda x \to \lambda y \to x)\ y && \rightsquigarrow \lambda z \to y
\end{aligned}
$$

A term without redexes is said to be in normal form.

Lambda Calculus vs. Turing machines

- The lambda calculus can be used to encode all Turing-computable functions, and vice versa.
- This fact justifies the addition of constructs such as numbers and arithmetic operations to the lambda calculus – these extensions do not increase the theoretical expressive power, they just make computation more efficient and more convenient.

Multiple redexes

Many terms have multiple redexes.

Let id abbreviate $\lambda x \to x$. How many redexes are in the following term?

$$\text{id}\ (\text{id}\ (\lambda z \to \text{id}\ z))$$

Multiple redexes

Many terms have multiple redexes.

Let id abbreviate $\lambda x \to x$. How many redexes are in the following term?

$$\mathrm{id}\ (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z))$$

$$\begin{array}{l} \mathrm{id}\ (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z)) \\ \quad (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z)) \\ \qquad\qquad \mathrm{id}\ z \end{array}$$

Multiple redexes

Many terms have multiple redexes.

Let id abbreviate $\lambda x \to x$. How many redexes are in the following term?

$$\text{id}\ (\text{id}\ (\lambda z \to \text{id}\ z))$$

$$(\lambda x \to \lambda y \to x * x)\ (1 + 2)\ (3 + 4)$$

$$\begin{array}{l} \text{id}\ (\text{id}\ (\lambda z \to \text{id}\ z)) \\ \quad (\text{id}\ (\lambda z \to \text{id}\ z)) \\ \qquad\qquad \text{id}\ z \end{array}$$

Multiple redexes

Many terms have multiple redexes.

Let id abbreviate $\lambda x \to x$. How many redexes are in the following term?

$$\mathrm{id}\ (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z))$$

$$(\lambda x \to \lambda y \to x * x)\ (1 + 2)\ (3 + 4)$$

$$\begin{array}{l} \mathrm{id}\ (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z)) \\ \quad (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z)) \\ \qquad\qquad \mathrm{id}\ z \end{array}$$

$$\begin{array}{lll} (\lambda x \to \lambda y \to x * x)\ (1 + 2) & & \\ & (1 + 2) & \\ & & (3 + 4) \end{array}$$

Operations such as $+$ and $*$ that require their arguments to be (partially) evaluated are called strict in their arguments.

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Call by name

Functions are reduced before their arguments. Used by some macro languages (TEX, for instance).

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Call by name

Functions are reduced before their arguments. Used by some macro languages (TeX, for instance).

Call by need / lazy evaluation

Optimized version of "Call by name": function arguments are only reduced when needed, but shared if used multiple times.

> λf g x → combine (f x) (g x)

Church-Rosser

Theorem (Church-Rosser)

If a lambda term e *can be reduced to* e_1 *and* e_2*, there is a term* e_3 *such that both* e_1 *and* e_2 *can be reduced to* e_3*.*

Church-Rosser

Theorem (Church-Rosser)

If a lambda term e *can be reduced to* e_1 *and* e_2*, there is a term* e_3 *such that both* e_1 *and* e_2 *can be reduced to* e_3*.*

Corollary

Each lambda term has at most one normal form.

Church-Rosser

Theorem (Church-Rosser)

If a lambda term e *can be reduced to* e_1 *and* e_2*, there is a term* e_3 *such that both* e_1 *and* e_2 *can be reduced to* e_3.

Corollary

Each lambda term has at most one normal form.

Theorem

If a term has a normal form, then lazy evaluation reaches this normal form.

Haskell vs. the Lambda Calculus

- Haskell is just an enriched form of the lambda calculus.
- Almost all of Haskell's language constructs can be translated into the lambda calculus.
- For instance, numbers, algebraic datatypes, **case** and **let** can all be encoded in the lambda calculus.
  Example:

  > **let** $x = e_1$ **in** e_2

  (if x is not free in e_1) is just syntactic sugar for

  > $(\lambda x \rightarrow e_2)\ e_1$

- The type system serves as a way to restrict the admissible lambda terms.

Non-termination

- In lambda calculus, it is possible to encode recursion and therefore nonterminating (diverging) terms.
- A nonterminating value is usually written $\perp$.
- In Haskell, $\perp$ is available as $\mathsf{undefined} :: \mathsf{a}$.
- Error values are also often denoted $\perp$, and $\mathsf{error} :: \mathsf{String} \to \mathsf{a}$ can be seen as an optimization of a diverging computation that prints an error message and breaks off program execution rather than actually diverging.
- In the presence of $\perp$, a function f is called strict if $\mathsf{f}\ \perp \equiv \perp$.
  In other words, a strict function diverges if applied to a diverging argument.

Examples of lazy evaluation

1. $(\lambda x \rightarrow x)\ \mathsf{True} \quad \rightsquigarrow^{*}$
2. $(\lambda x \rightarrow x)\ \bot \quad \rightsquigarrow^{*}$
3. $(\lambda x \rightarrow ())\ \bot \quad \rightsquigarrow^{*}$
4. $(\lambda x \rightarrow \bot)\ () \quad \rightsquigarrow^{*}$
5. $(\lambda x\ f \rightarrow f\ x)\ \bot \quad \rightsquigarrow^{*}$
6. $\bot_1\ \bot_2 \quad \rightsquigarrow^{*}$
7. $\mathsf{length}\ (\mathsf{map}\ \bot\ [1,2]) \quad \rightsquigarrow^{*}$

Examples of lazy evaluation

1. $(\lambda x \to x)\ \mathsf{True} \leadsto^* \mathsf{True}$
2. $(\lambda x \to x)\ \bot \leadsto^* \bot$
3. $(\lambda x \to ())\ \bot \leadsto^* ()$
4. $(\lambda x \to \bot)\ () \leadsto^* \bot$
5. $(\lambda x\ f \to f\ x)\ \bot \leadsto^* \lambda f \to f\ \bot$
6. $\bot_1\ \bot_2 \leadsto^* \bot_1$
7. $\mathsf{length}\ (\mathsf{map}\ \bot\ [1,2]) \leadsto^* 2$

Forcing evaluation

- Haskell has the following primitive function

```
seq :: a → b → b    -- primitive
```

Forcing evaluation

- Haskell has the following primitive function

```
seq :: a → b → b    -- primitive
```

- The call

```
seq x y
```

  evaluates x before returning y.

Forcing evaluation

- Haskell has the following primitive function

  ```
  seq :: a → b → b    -- primitive
  ```

- The call

  ```
  seq x y
  ```

  evaluates x before returning y.

- The function seq can be used to define strict function application:

  ```
  ($!) :: (a → b) → a → b
  f $! x = x `seq` f x
  ```

  Recall sharing!

Examples of forcing

1. $(\lambda x \to ())\ \$!\ \perp$ $\quad\rightsquigarrow^*$
2. $\mathsf{seq}\ (\perp_1, \perp_2)\ ()$ $\quad\rightsquigarrow^*$
3. $\mathsf{snd}\ \$!\ (\perp_1, \perp_2)$ $\quad\rightsquigarrow^*$
4. $(\lambda x \to ())\ \$!\ (\lambda x \to \perp)$ $\quad\rightsquigarrow^*$
5. $\perp_1\ \$!\ \perp_2$ $\quad\rightsquigarrow^*$
6. $\mathsf{length}\ \$!\ \mathsf{map}\ \perp\ [1, 2]$ $\quad\rightsquigarrow^*$
7. $\mathsf{seq}\ (\perp_1 + \perp_2)\ ()$ $\quad\rightsquigarrow^*$
8. $\mathsf{seq}\ (\mathsf{foldr}\ \perp_1\ \perp_2)\ ()$ $\quad\rightsquigarrow^*$
9. $\mathsf{seq}\ (1 : \perp)\ ()$ $\quad\rightsquigarrow^*$

Examples of forcing

1. $(\lambda x \to ())\ \$!\ \bot \quad \rightsquigarrow^* \quad \bot$
2. $\mathsf{seq}\ (\bot_1, \bot_2)\ () \quad \rightsquigarrow^* \quad ()$
3. $\mathsf{snd}\ \$!\ (\bot_1, \bot_2) \quad \rightsquigarrow^* \quad \bot_2$
4. $(\lambda x \to ())\ \$!\ (\lambda x \to \bot) \quad \rightsquigarrow^* \quad ()$
5. $\bot_1\ \$!\ \bot_2 \quad \rightsquigarrow^* \quad \bot_2$
6. $\mathsf{length}\ \$!\ \mathsf{map}\ \bot\ [1, 2] \quad \rightsquigarrow^* \quad 2$
7. $\mathsf{seq}\ (\bot_1 + \bot_2)\ () \quad \rightsquigarrow^* \quad \bot_1$
8. $\mathsf{seq}\ (\mathsf{foldr}\ \bot_1\ \bot_2)\ () \quad \rightsquigarrow^* \quad ()$
9. $\mathsf{seq}\ (1 : \bot)\ () \quad \rightsquigarrow^* \quad ()$

Forcing only evaluates to (weak) head normal form (i.e., a lambda abstraction, literal or constructor application).

4.3 Debugging

The GHCi debugger

- GHCi (since version 6.8) has an integrated debugger that works similar to a debugger for imperative languages.
- You can add breakpoints to interpreted, but not compiled code.
- You can step through your program execution.
- At each break point, you can inspect values that are in context.
- Requires a bit of experience due to lazy evaluation.
- Even trickier to implement because it can be useful to print values of which the type is not statically known.
- The GHCi debugger can also be used to debug infinite loops.
- More documentation: Section 3.5 of the GHC User's Guide.

Tracing isort in the debugger

```
$ ghci Sort.hs
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( Sort.hs, interpreted )
Ok, modules loaded: Main.
*Main> :break isort
Breakpoint 0 activated at Sort.hs:(14,0)-(15,35)
*Main> :set stop :list
*Main> isort [2,3,1]
[...]
Stopped at Sort.hs:(14,0)-(15,35)
_result :: [a] = _
13  isort :: Ord a => [a] -> [a]
14  isort []       =  []
15  isort (x:xs)   =  insert x (isort xs)
16
[Sort.hs:(14,0)-(15,35)] *Main> :step
```

Tracing isort in the debugger (contd.)

Keep in mind that

- Haskell uses lazy evaluation, and
- pattern matching drives evaluation (causes strictness)

```
  isort [2, 3, 1]
≡   { reducing the outermost redex }
  insert 2 (isort [3, 1])
≡   { insert 2 is strict in its argument }
  insert 2 (insert 3 (isort [1]))
≡   { insert 3 is strict in its argument }
  insert 2 (insert 3 (insert 1 (isort [])))
≡   { insert 1 is strict in its argument }
  insert 2 (insert 3 (insert 1 []))
≡   { insert 3 is strict in its argument }
  ...
```

Tracing isort in the debugger (contd.)

$\ldots$

$\equiv$

insert 2 (insert 3 [1])

$\equiv$ { insert 2 is strict in its argument }

insert 2 (1 : insert 3 [ ])

$\equiv$ { argument of insert 2 is now in WHNF }

1 : insert 2 (insert 3 [ ])

$\equiv$ { 1 is already determined as first element of the result }

1 : insert 2 (insert 3 [ ])

$\equiv$ { evaluation continues; insert 2 is strict in its argument }

1 : 2 : insert 3 [ ]

$\equiv$ { 2 is determined as second element of the result }

1 : 2 : insert 3 [ ]

$\equiv$ { definition of insert }

1 : 2 : [3]

Next lecture

- Next lecture(s) will be on “Functional data structures”.
- Read until next week:
  - “Rewriting Haskell Strings”
  - “Finger Trees: A Simple General-purpose Data Structure”

  Both papers are linked from the Wiki.
- Tomorrow: first deadline for the weekly assignments.