Introducing Haskell

COS 441 Slides 3

Slide content credits:
Ranjit Jhala (UCSD)
Benjamin Pierce (UPenn)
Course Agenda (Initial Lectures)

• Week 1 (Appel):
  – Syntactic definitions
  – Denotational definitions
  – Proofs by induction

• The coming weeks (Walker):
  – Introduction to Haskell
  – Syntactic definitions in Haskell
  – Denotational definitions in Haskell
  – Proofs in Haskell and about Haskell programs
  – Type classes
  – Applications of denotational semantics:
    • Domain-specific languages for graphics & animation
PL: Some Broad Categories

• Imperative
  – oriented around assignment to variables and simple control flow
  – C, Pascal, Go

• Object-oriented (Class-based)
  – oriented around classes and objects
  – Java, C#

• Logic programming
  – oriented around logical formulae, unification and search
  – Prolog, Twelf

• Functional
  – oriented around functions and immutable data structures
  – SML, O’Caml, F#, Coq, Scheme, Map-Reduce, Erlang, Haskell
Vastly Abbreviated FP Geneology

LCF Theorem Prover (70s)

Edinburgh ML

Miranda (80s)

Haskell (90s - now)

lazy, pure

Standard ML (90s - now)

Caml (80s-now)

OCaml (90s - now)

F# (now)

typed, polymorphic

LISP (50s-now)

Scheme (70s-now)

untyped

Coq (80s - now)

dependently typed
Functional Languages: Who’s using them?

F# in Visual Studio
Functional Languages: Who’s using them?

- Microsoft: F# in Visual Studio
- Facebook: Erlang for concurrency, Haskell for managing PHP
Functional Languages: Who’s using them?

map-reduce in their data centers

Google

Facebook

Erlang for concurrency, Haskell for managing PHP

F# in Visual Studio

Microsoft
Functional Languages: Who’s using them?

- Google: map-reduce in their data centers
- Facebook
- Microsoft: F# in Visual Studio
- Jane Street: Erlang for concurrency, Haskell for managing PHP, O’Caml for reliability, Haskell for specifying equity derivatives
- Barclays
Functional Languages: Who’s using them?

- map-reduce in their data centers
- Scala for correctness, maintainability, flexibility
- F# in Visual Studio
- Erlang for concurrency, Haskell for managing PHP
- O’Caml for reliability
- Haskell for specifying equity derivatives
Functional Languages: Who’s using them?

Google: map-reduce in their data centers

Facebook: Scala for correctness, maintainability, flexibility

Microsoft: F# in Visual Studio

Jane Street: Erlang for concurrency, Haskell for managing PHP

Bluespec: Haskell to synthesize hardware

Barclays: O’Caml for reliability

Haskell: for specifying equity derivatives
Functional Languages: Who’s using them?

- **Google**: map-reduce in their data centers
- **Erlang**: for concurrency
- **Haskell**: for managing PHP
- **Jane Street**: Haskell to synthesize hardware
- **O’Caml**: for reliability
- **Bluespec**: for specifying equity derivatives
- **Facebook**: Scala for correctness, maintainability, flexibility
- **Twitter**: F# in Visual Studio
- **Microsoft**: F# in Visual Studio
- **Barclays**: Haskell for specifying equity derivatives
- **Mathematicians**: Coq proof of 4-color theorem
Functional Languages: Who’s using them?

Google
- F# in Visual Studio
- Scala for correctness, maintainability, flexibility

Facebook
- Erlang for concurrency
- Haskell for managing PHP

Microsoft
- Haskell to synthesize hardware

Jane Street
- O’Caml for reliability

Bluespec
- Coq proof of 4-color theorem

 Barclays
- Haskell for specifying equity derivatives

Twitter
- Map-reduce in their data centers

www.artima.com/scalazine/articles/twitter_on_scala.html
http://gregosuri.com/how-facebook-uses-erlang-for-real-time-chat
http://msdn.microsoft.com/en-us/fsharp/cc742182
http://labs.google.com/papers/mapreduce.html
http://www.haskell.org/haskellwiki/Haskell_in_industry
Haskell vs. ML

• My research, many of my courses have used ML
  – SML or O’Caml

• What do ML and Haskell have in common?
  – functions as first-class data
  – rich, sound type systems & type inference
  – rich data types and algebraic pattern matching
  – immutable data is the default

• ML has:
  – A powerful module system
  – SML has a complete, formal definition

• Haskell has:
  – Type classes, Pure functions, Monads
  – Lazy evaluation

• I vastly prefer programming in ML or Haskell vs. C or Java
INTRODUCING HASKELL
A Haskell program is much like a set of mathematical equations – that’s why we’ll use it to implement math.

All computation occurs via substitution of one expression for another equal expression, like in ordinary mathematics:

\[ 3 \times (4 + 5) \]
A Haskell program is much like a set of mathematical equations – that’s why we’ll use it to implement math.

All computation occurs via substitution of one expression for another equal expression, like in ordinary mathematics:

\[
3 \times (4 + 5) = 3 \times 9 \quad \text{(by add } 4 + 5 = 9)\]
Computation by Calculation

• A Haskell program is much like a set of mathematical equations – that’s why we’ll use it to implement math

• All computation occurs via substitution of one expression for another equal expression, like in ordinary mathematics:

\[
3 \times (4 + 5) \\
= 3 \times 9 \quad \text{(by add } 4 + 5 = 9) \\
= 27 \quad \text{(by mult } 3 \times 9 = 27) 
\]
• A Haskell program is much like a set of mathematical equations – that’s why we’ll use it to implement math

• All computation occurs via substitution of one expression for another equal expression, like in ordinary mathematics:

\[
3 \times (4 + 5) \\
= 3 \times 9 \quad \text{(by add } 4 + 5 = 9) \\
= 27 \quad \text{(by mult } 3 \times 9 = 27) \\
\]

• This seems pretty obvious but the remarkable thing is that it holds \textit{all the time} in Haskell, unlike in C:

```c
int x = 0;
...
y = x + x;
```
A Haskell program is much like a set of mathematical equations – that’s why we’ll use it to implement math.

All computation occurs via substitution of one expression for another equal expression, like in ordinary mathematics:

\[
3 \times (4 + 5) \\
= 3 \times 9 \quad \text{(by add \ 4 + 5 = 9)} \\
= 27 \quad \text{(by mult \ 3 \times 9 = 27)}
\]

This seems pretty obvious but the remarkable thing is that it holds \textit{all the time} in Haskell, unlike in C:

```c
int x = 0; // int x = 0;
... // ... 
? // 
... // 

y = x + x; // y = 0 + 0;
```
Computation by Calculation

• A Haskell program is much like a set of mathematical equations – that’s why we’ll use it to implement math.

• All computation occurs via substitution of one expression for another equal expression, like in ordinary mathematics:

\[
3 \times (4 + 5) \\
= 3 \times 9 \quad \text{(by add } 4 + 5 = 9) \\
= 27 \quad \text{(by mult } 3 \times 9 = 27) \\
\]

• This seems pretty obvious but the remarkable thing is that it holds *all the time* in Haskell, unlike in C:

```haskell
int x = 0;  \quad \text{int x = 0; } \quad \text{int x = 0;}
\quad \quad \quad \text{int x = 0;}
\quad \quad \quad \text{int x = 0;}
... \quad \quad \quad \text{...} \quad \quad \quad \text{\cancel{x = 1;}}
\quad \quad \quad \text{\cancel{\cancel{\text{\cancel{x = 1;}}}}} \\
\text{y = x + x; } \quad \text{y = 0 + 0;} \quad \text{y = x + x; }
```
Computation with Abstraction

• Good programmers use abstraction:
  – we recognize repeated patterns and capture them succinctly in one place instead of many
  – for example:

\[
3 \times (4 + 5) \quad 9 \times (1 + 7) \quad 200 \times (1 - 8)
\]
• Good programmers use abstraction:
  – we recognize repeated patterns and capture them succinctly in one place instead of many
  – for example:

\[
3 \times (4 + 5) \quad 9 \times (1 + 7) \quad 200 \times (1 - 8)
\]

– captured by:

\[
easy x \; y \; z = x \times (y + z)
\]
Good programmers use abstraction:

- we recognize repeated patterns and capture them succinctly in one place instead of many
- for example:

\[
\begin{align*}
3 \times (4 + 5) & \quad 9 \times (1 + 7) & \quad 200 \times (1 - 8) \\
\end{align*}
\]

- captured by:

\[
easy x y z = x \times (y + z)
\]

- and specific instances written:

\[
easy 3 4 5 \quad easy 9 1 7 \quad easy 200 1 (-8)
\]
Good programmers use abstraction:

- we recognize repeated patterns and capture them succinctly in one place instead of many
- for example:

\[
\begin{align*}
3 \times (4 + 5) & \quad 9 \times (1 + 7) & \quad 200 \times (1 - 8) \\
\end{align*}
\]

- captured by:

\[
easy x y z = x \times (y + z)
\]

- and specific instances written:

\[
\begin{align*}
easy 3 4 5 & \quad easy 9 1 7 & \quad easy 200 1 (-8) \\
\end{align*}
\]

This is functional abstraction: the process of capturing repeated idioms and representing them as functions
Computation by Calculation with Abstraction

Computation by calculation with function abstraction is done by unfolding function definitions (just like we unfolded mathematical definitions):

\[
\text{easy } 3 \ 4 \ 5 \\
= \ 3 \ * \ (4 + 5) \quad \text{(by unfold/by definition)} \\
= \ 3 \ * \ 9 \quad \text{(by add)} \\
= \ 27 \quad \text{(by multiply)}
\]

definition:
\[
easy \ x \ y \ z = x \ * \ (y + z)
\]
We can also reason with symbolic values:

\[
\text{easy } a \ b \ c \\
= a \times (b + c) \quad \text{(by unfold)}
\]
\[
= a \times (c + b) \quad \text{(by commutativity of add)}
\]
\[
= \text{easy } a \ c \ b \quad \text{(by fold)}
\]

With these concepts:

- computation by calculation
- abstraction
- symbolic values

... we are well on our way to reasoning about Haskell definitions just like we reasoned about mathematical definitions, though Haskell gives us an implementation!
HASKELL BASICS:
EXPRESSIONS, VALUES, TYPES
Expressions, Values, Types

• The phrases on which we calculate are called expressions.
• When no more unfolding of user-defined functions or application of primitives like + is possible, the resulting expression is called a value.
• A type is a collection of expressions with common attributes. Every expression (and thus every value) belongs to a type.
• We write exp :: T to say that expression exp has type T.
Basic Types

- **Integers**
  
  \[3 + 4 \times 5\] :: Integer

- **Floats**
  
  \[3 + 4.5 \times 5.5\] :: Float

- **Characters**
  
  \['a'\] :: Char
Functions

• The type of a function taking arguments A and B and returning a result of type C is written A -> B -> C

  (+) :: Integer -> Integer -> Integer
  easy :: Integer -> Integer -> integer

• Note that (+) is syntax for treating an infix operator as a regular one. Conversely, we can take a non-infix operator and make it infix:

  plus x y = x + y

  easier x y z = x * (y ‘plus’ z)
A SHORT DEMO
Summary

• Haskell is
  – a functional language emphasizing immutable data
  – where every expression has a type:
    • Char, Int, Int -> Char

• Reasoning about Haskell programs involves
  – substitution of “equals for equals,” unlike in Java or C
  – proofs about Haskell programs often:
    • unfold function abstractions
    • push symbolic names around like we do in mathematical proofs
    • reason locally using properties of operations (eg: + commutes)
    • fold function abstractions back up

• Homework: Install Haskell. Read LYAHFGG Intro, Chapter 1