Introducing Haskell

COS 441 Slides 3

Slide content credits:
Ranjit Jhala (UCSD)
Benjamin Pierce (UPenn)

Course Agenda (Initial Lectures)

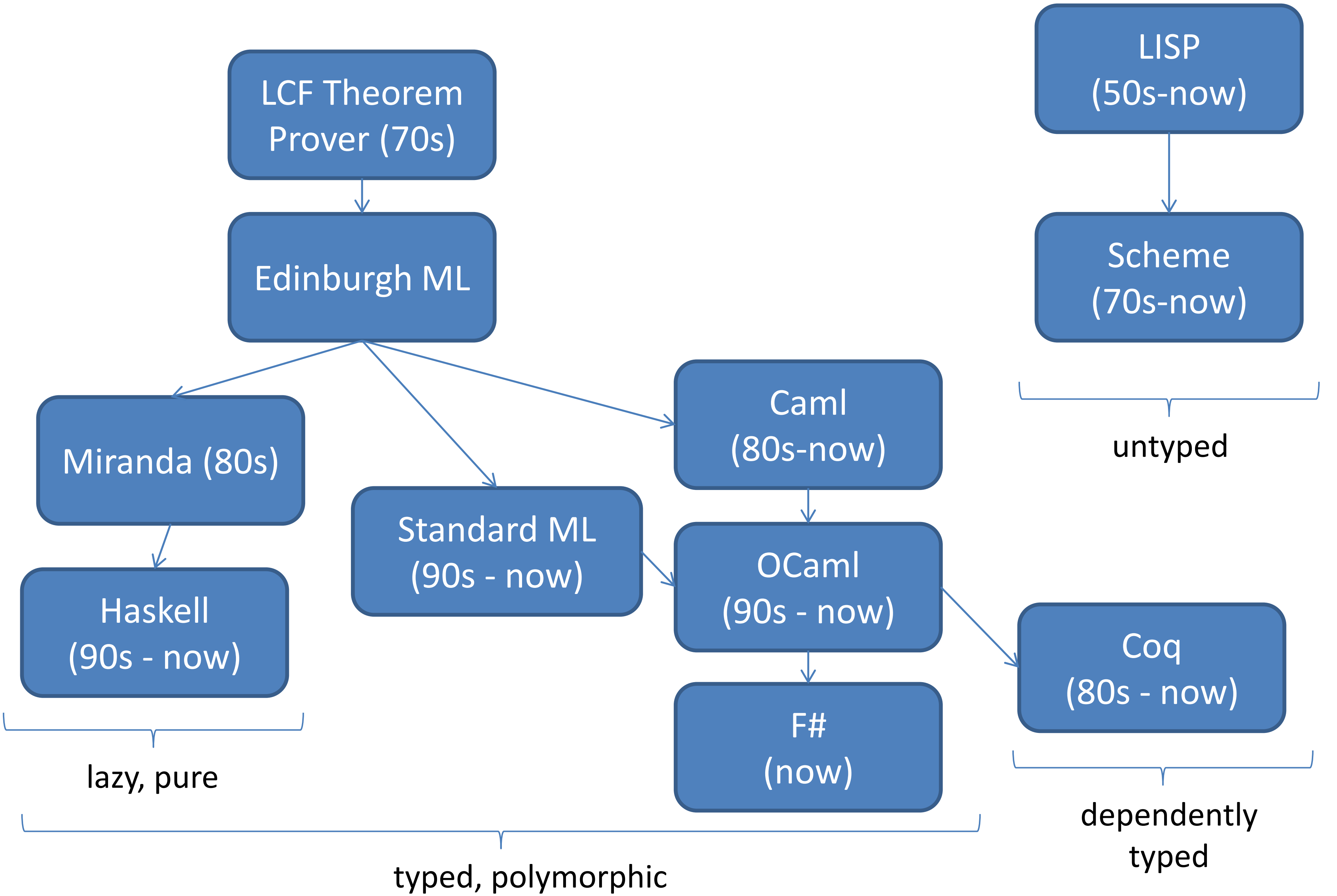
- Week 1 (Appel):
  - Syntactic definitions
  - Denotational definitions
  - Proofs by induction

- 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



Functional Languages: Who's using them?

F# in Visual Studio

Functional Languages: Who's using them?

Microsoft Be what's next.

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

Microsoft Be what's next.

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

Microsoft Be what's next.

F# in Visual Studio

facebook

Erlang for concurrency, Haskell for managing PHP

JANE STREET

O'Caml for reliability

BARCLAYS

Haskell for specifying equity derivatives

Functional Languages: Who's using them?

map-reduce in their data centers

Scala for
correctness, maintainability, flexibility

Erlang for
concurrency,
Haskell for
managing PHP

F# in Visual Studio

O'Caml
for reliability

Haskell
for specifying
equity derivatives

Functional Languages: Who's using them?

map-reduce in their data centers

Scala for
correctness, maintainability, flexibility

Erlang for
concurrency,
Haskell for
managing PHP

F# in Visual Studio

O'Caml
for reliability

Haskell to
synthesize hardware

Haskell
for specifying
equity derivatives

Functional Languages: Who's using them?

map-reduce in their data centers

Scala for
correctness, maintainability, flexibility

Erlang for
concurrency,
Haskell for
managing PHP

F# in Visual Studio

O'Caml
for reliability

Haskell to
synthesize hardware

Haskell
for specifying
equity derivatives

mathematicians

Coq proof of
4-color theorem

Functional Languages: Who's using them?

map-reduce in their data centers

Scala for
correctness, maintainability, flexibility

Erlang for
concurrency,
Haskell for
managing PHP

F# in Visual Studio

Haskell to
synthesize hardware

O'Caml
for reliability

Haskell
for specifying
equity derivatives

mathematicians

Coq proof of
4-color theorem

www.artima.com/scalazine/articles/twitter_on_scala.html
http://gregosuri.com/how-facebook-uses-erlang-for-real-time-chat
http://www.janestcapital.com/technology/ocaml.php
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

    not my favourite as a default
- I *vastly* prefer programming in ML or Haskell vs. C or Java

INTRODUCING HASKELL

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:

$$3 * (4 + 5)$$

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:

$$
\begin{aligned}
& 3 * (4 + 5) \\
& \quad = 3 * 9 \qquad\qquad \text{(by add } 4 + 5 = 9\text{)}
\end{aligned}
$$

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 * (4 + 5)
  = 3 * 9        (by add  4 + 5 = 9)
  = 27           (by mult 3 * 9 = 27)
```

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 * (4 + 5)
  = 3 * 9          (by add  4 + 5 = 9)
  = 27             (by mult 3 * 9 = 27)
```

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

```
int x = 0;

…

y = x + x;
```

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 * (4 + 5)
 = 3 * 9          (by add  4 + 5 = 9)
 = 27             (by mult 3 * 9 = 27)
```

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

$$\begin{array}{lcl} \texttt{int x = 0;} & & \texttt{int x = 0;} \\ \ldots & \xrightarrow{?} & \ldots \\ \texttt{y = x + x;} & & \texttt{y = 0 + 0;} \end{array}$$

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:

$$
\begin{aligned}
&3 * (4 + 5) \\
&\quad = 3 * 9 && \text{(by add } 4 + 5 = 9\text{)} \\
&\quad = 27 && \text{(by mult } 3 * 9 = 27\text{)}
\end{aligned}
$$

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

int x = 0;		int x = 0;		int x = 0;
…	?	…		x = 1;
y = x + x;		y = 0 + 0;		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 * (4 + 5)   9 * (1 + 7)   200 * (1 – 8)

Computation with Abstraction

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

    3 * (4 + 5)   9 * (1 + 7)   200 * (1 – 8)

  - captured by:

    easy x y z = x * (y + z)

Computation with Abstraction

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

    3 * (4 + 5)   9 * (1 + 7)   200 * (1 – 8)

  - captured by:

    easy x y z = x * (y + z)

  - and specific instances written:

    easy 3 4 5   easy 9 1 7   easy 200 1 (-8)

Computation with Abstraction

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

    3 * (4 + 5)   9 * (1 + 7)   200 * (1 – 8)

  - captured by:

    easy x y z = x * (y + z)

  - and specific instances written:

    easy 3 4 5   easy 9 1 7   easy 200 1 (-8)

- 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):

easy 3 4 5	
= 3 * (4 + 5)	(by unfold/by definition)
= 3 * 9	(by add)
= 27	(by multiply)

definition:
easy x y z = x * (y + z)

Computation by Calculation with Abstraction

- We can also reason with symbolic values:

```
easy a b c
=  a * (b + c)       (by unfold)
=  a * (c + b)       (by commutativity of add)
=  easy a c b        (by fold)
```

definition:
easy x y z = x * (y + z)

- 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 * 5 :: Integer

- Floats

  3 + 4.5 * 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