Functional Programming

Ralf Hinze

Universität Kaiserslautern

April 2019

Part 0

Overview

0.0 Outline

0.1 Aims

- *functional programming* is *programming with values*: *value-oriented programming*
- no 'actions', no side-effects — a radical departure from ordinary (imperative or OO) programming
- surprisingly, it is a powerful (and fun!) paradigm
- ideas are applicable in ordinary programming languages too; aim to introduce you to the ideas, to improve your programming skills
- (I don't expect you all to start using functional languages)
- we use *Haskell*, the standard lazy functional programming language, see `www.haskell.org`

0.2 Motivation

LISP is worth learning [because of] the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot.

Eric S. Raymond, American computer programmer (1957–)
How to Become a Hacker
`www.catb.org/~esr/faqs/hacker-howto.html`

You can never understand one language until you understand at least two.

Ronald Searle, British artist (1920–2011)

0.2 FP and OOP

OOP features originating in FP:

- generics e.g. `Tree<Elem>`
  Haskell: parametric polymorphism Tree elem
- type inference
  enjoy the benefits of static typing without the pain
- lambda expressions e.g. `p -> p.age() >= 18`
  the core of Haskell $\backslash p \to age\ p \geqslant 18$
- immutable classes and value classes
  purity is at the heart of Haskell
- language integrated query languages
  Haskell's list comprehensions
- garbage collection
  you don't want to do without

See Java, C#, F#, Scala etc

0.2 FP in industry

Some big players:

- facebook: Haskell for spam filtering
- Intel: FP for chip design
- Jane Street, Credit Suisse, Standard Chartered Bank: FP for financial contracts etc

Some specialist companies:

- galois: FP for high assurance software
- Well-Typed: Haskell consultants

0.3 Organizational matters

- Website: `https://pl.cs.uni-kl.de/fp19`
- *your goal:* obtain a good grade
- (*my goal:* get you interested in FP)
- *how to achieve your goal:*
  - make good use of me i.e. attend lectures
  - make good use of my teaching assistant: Sebastian Schweizer
  - obtain at least a sufficient grade for 75% of the exercises
    - work and submit in groups of 3–4
    - submission: Tuesday 12:00 noon
    - exercise session: Thursday, 11:45 - 13:15, Room 48-453
  - pass the final exam
- *a gentle request and a suggestion:*
  keep the use of electronic devices to a minimum;
  make notes using pencil and paper

0.4 Contents: part one

0.4 Contents: part two

0.5 Expressions vs statements

- in ordinary programming languages the world is divided into a world of statements and a world of expressions
- statements:
  - x := e, s1 ; s2, **while** e **do** s
  - execution order is important

$$i := i + 1 ; a := a * i \quad \neq \quad a := a * i ; i := i + 1$$

- expressions:
  - $a + b * c$, a and not b
  - evaluation order is unimportant: in

$$(2 * a * y + b) * (2 * a * y + c)$$

    evaluate either parenthesis first (or both simultaneously!)
  - (assumes no side-effects: order matters in $++x + x--$)

0.5 Referential transparency

- useful optimizations:
  - reordering:

```
    x := 0 ; y := e ; if x <> 0 then ... end
=   x := 0 ; if x <> 0 then ... end ; y := e
=   x := 0 ; y := e
```

  - common sub-expression elimination:

```
    z := (2 * a * y + b) * (2 * a * y + c)
=   t := 2 * a * y ; z := (t + b) * (t + c)
```

  - parallel execution: evaluate sub-expressions concurrently
- most optimizations require *referential transparency*
  - all that matters about the expression is its value
  - follows from 'no side effects'
  - …which follows from 'no :='
  - with assignments, side-effect-freeness is hard to check

0.5 Programming with expressions

- expressions are much shorter and simpler than the corresponding statements
- e.g. compare using expression:

```
z := (2 ∗ a ∗ y + b) ∗ (2 ∗ a ∗ y + c)
```

  with not using expressions:

```
ac := 2; ac ∗= a; ac ∗= y; ac += b; t := ac;
ac := 2; ac ∗= a; ac ∗= y; ac += c; ac ∗= t;
z := ac
```

- but in order to discard statements, the expression language must be extended
- functional programming is *programming with an extended expression language*

0.5 Comparison with 'ordinary' programming

- insertion sort
- quicksort
- binary search trees

0.5 Insertion sort: Modula-2

```
PROCEDURE InsertionSort(VAR a:ArrayT);
VAR i, j: CARDINAL;
    t: ElementT;
BEGIN
  FOR i := 2 TO Size DO
    (* a[1..i-1] already sorted *)
    t := a[i];
    j := i;
    WHILE (j > 1) AND (a[j−1] > t) DO
      a[j] := a[j−1]; j := j−1
    END;
    a[j] := t
  END
END InsertSort;
```

0.5 Insertion sort: Haskell

$$
\begin{aligned}
&\mathit{insertionSort}\ [\,] &&= [\,]\\
&\mathit{insertionSort}\ (x : xs) &&= \mathit{insert}\ x\ (\mathit{insertionSort}\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insert}\ a\ [\,] &&= [a]\\
&\mathit{insert}\ a\ (b : xs) &&\\
&\quad \mid a \leqslant b &&= a : b : xs\\
&\quad \mid \mathit{otherwise} &&= b : \mathit{insert}\ a\ xs
\end{aligned}
$$

0.5 Quicksort: C

```
void quicksort(int a[], int l, int r)
{
  if (r > l)
    {
      int i = l; int j = r;
      int p = a[(l + r) / 2];
      for (;;) {
        while (a[i] < p) i++;
        while (a[j] > p) j--;
        if (i > j) break;
        swap(&a[i++], &a[j--]);
      };
      quicksort(a, l, j);
      quicksort(a, i, r);
    }
}
```

0.5 Quicksort: Haskell

$$
\begin{aligned}
&\mathit{quickSort}\ [\,] &&= [\,] \\
&\mathit{quickSort}\ (x : xs) &&= \mathit{quickSort}\ \mathit{littles} \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{quickSort}\ \mathit{bigs} \\
&\quad \mathbf{where}\ \mathit{littles} &&= [\, a \mid a \leftarrow xs, a < x \,] \\
&\qquad\qquad \mathit{bigs} &&= [\, a \mid a \leftarrow xs, x \leqslant a \,]
\end{aligned}
$$

0.5 Binary search trees: Java

```
public class BinarySearchTree<Elem>
{
  private Tree<Elem> root;
  public BinarySearchTree () {
    root = new Empty();
  }
  public void inorder() {
    root.inorder();
  }
  public void insert (Elem e) {
    root = root.insert(e);
  }
}

public interface Tree<Elem> {
  void inorder();
  Tree insert (Elem e);
}
```

```
class Empty<Elem extends Comparable<Elem>>
    implements Tree<Elem> {
  public void inorder() {}
  public Tree insert (Elem k) {
    return new Node
      (new Empty(), k, new Empty());
  }
}

class Node<Elem extends Comparable<Elem>>
   implements Tree<Elem> {
  private Elem a;
  private Tree l, r;
  public Node (Tree l, Elem a, Tree r) {
      this.l = l; this.a = a; this.r = r;
  }
  public void inorder() {
      l.inorder();
      System.out.println(a);
      r.inorder();
  }
  public Tree insert (Elem k) {
      if (k.compareTo(a) <= 0)
          l = l.insert(k);
      else
          r = r.insert(k);
      return this;
  }
}
```

0.5 Binary search trees: Haskell

$$\textbf{data}\ \mathit{Tree\ elem} = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree\ elem})\ \mathit{elem}\ (\mathit{Tree\ elem})$$

$$\begin{aligned}
&\mathit{inorder\ Empty} &&= [\,] \\
&\mathit{inorder}\ (\mathit{Node}\ l\ a\ r) &&= \mathit{inorder}\ l \mathbin{+\!\!+} [\,a\,] \mathbin{+\!\!+} \mathit{inorder}\ r
\end{aligned}$$

$$\begin{aligned}
&\mathit{insert}\ k\ \mathit{Empty} = \mathit{Node\ Empty}\ k\ \mathit{Empty} \\
&\mathit{insert}\ k\ (\mathit{Node}\ l\ a\ r) \\
&\quad \mid k \leqslant a \qquad\ \ = \mathit{Node}\ (\mathit{insert}\ k\ l)\ a\ r \\
&\quad \mid \mathit{otherwise} = \mathit{Node}\ l\ a\ (\mathit{insert}\ k\ r)
\end{aligned}$$

0.6 Literature

- Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*, No Starch Press, 2011.
- Richard Bird, *Thinking Functionally with Haskell*, Cambridge University Press, 2015.
- Paul Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
- Graham Hutton, *Programming in Haskell (2nd Edition)*, Cambridge University Press, 2016.
- Bryan O'Sullivan, John Goerzen, Don Stewart, *Real World Haskell*, O'Reilly Media, 2008.
- Simon Thompson, *Haskell: The Craft of Functional Programming (3rd Edition)*, Addison-Wesley Professional, 2011.

Part 1

Programming with expressions and values

1.0 Outline

1.1 Calculators

- functional programming is like using a pocket calculator
- user enters in expression, the system evaluates the expression, and prints result
- interactive ‘read-eval-print’ loop

```
⟫⟫ product [1..40]
815915283247897734345611269596115894272000000000
⟫⟫ sort "hello, world\n"
"\n ,dehllloorw"
```

- powerful mechanism for defining new functions
- we can calculate not only with numbers, but also with lists, trees, grammars, pictures, music ...

1.1 Scripts and sessions

- we will use *GHCi*, an interactive version of the *Glasgow Haskell Compiler*, a popular implementation of *Haskell*
- a program is a collection of modules
- a module is a collection of definitions: a *script*
- running a program consists of loading script and evaluating expressions: a *session*
- a standalone program includes a ‘main’ expression
- scripts may or may not be *literate* (emphasis on comments)

1.1 An illiterate script (.hs suffix)

```
-- compute the square of an integer
square :: Integer -> Integer
square x = x * x

-- smaller of two arguments
smaller :: (Integer, Integer) -> Integer
smaller (x, y) = if x <= y then x else y
```

1.1 A literate script (.lhs suffix)

```
The following function squares an integer.

> square :: Integer -> Integer
> square x = x * x

This one takes a pair of integers as an argument,
and returns the smaller of the two as a result.
For example,

  smaller (3, 4) = 3

> smaller :: (Integer, Integer) -> Integer
> smaller (x, y)  =  if x <= y then x else y
```

1.1 Layout

- elegant and unobtrusive syntax
- structure obtained by layout, not punctuation
- all definitions in same scope must start in the same column
- indentation from start of definition implies continuation

```
smaller :: (Integer, Integer) → Integer
smaller (x, y)
  = if
      x ⩽ y
    then
      x
    else
      y
```

- leave blank lines around code in literate script!
- use spaces, not tabs!

1.1 A session

$\rangle\rangle\rangle\rangle$ 42
42

$\rangle\rangle\rangle\rangle$ $6 * 7$
42

$\rangle\rangle\rangle\rangle$ $\mathit{square}\ 7 - \mathit{smaller}\ (3, 4) - \mathit{square}\ (\mathit{smaller}\ (2, 3))$
42

$\rangle\rangle\rangle\rangle$ $\mathit{square}\ 1234567890$
1524157875019052100

1.2 Notation: evaluation of expressions

- we use the following layout for evaluations

$$
\begin{array}{ll}
 & \mathit{expr1} \\
\Longrightarrow & \quad \{\ \text{why?}\ \} \\
 & \mathit{expr2} \\
\Longrightarrow & \quad \{\ \text{why?}\ \} \\
 & \mathit{expr3}
\end{array}
$$

1.2 Evaluation

- interpreter evaluates expression by reducing to simplest possible form
- reduction is rewriting using meaning-preserving simplifications: *replacing equals by equals*

$$
\begin{array}{ll}
 & \mathit{square}\ (3 + 4) \\
\Longrightarrow & \quad \{\ \text{definition of `}+\text{'}\ \} \\
 & \mathit{square}\ 7 \\
\Longrightarrow & \quad \{\ \text{definition of } \mathit{square}\ \} \\
 & 7 * 7 \\
\Longrightarrow & \quad \{\ \text{definition of `}*\text{'}\ \} \\
 & 49
\end{array}
$$

- expression 49 cannot be reduced any further: *normal form*
- *applicative order* evaluation: reduce arguments before expanding function definition (call by value, eager evaluation)

1.2 Alternative evaluation orders

- other evaluation orders are possible:

$$
\begin{aligned}
& \mathit{square}\ (3 + 4) \\
\Longrightarrow \quad & \{ \text{ definition of } \mathit{square} \} \\
& (3 + 4) * (3 + 4) \\
\Longrightarrow \quad & \{ \text{ definition of '+' } \} \\
& 7 * (3 + 4) \\
\Longrightarrow \quad & \{ \text{ definition of '+' } \} \\
& 7 * 7 \\
\Longrightarrow \quad & \{ \text{ definition of '}*\text{' } \} \\
& 49
\end{aligned}
$$

- final result is the same: if two evaluation orders terminate, both yield the same result (*confluence*)
- *normal order* evaluation: expand function definition before reducing arguments (call by need, lazy evaluation)

1.2 Values

- in FP, as in maths, the sole purpose of an expression is to denote a value
- other characteristics (time to evaluate, number of characters, etc) are irrelevant
- values may be of various kinds: numbers, truth values, characters, tuples, lists, functions, etc
- important to distinguish *abstract value* (the number 42) from concrete representation (the characters '4' and '2', the string "XLII", the bit-sequence 0000000000101010)
- evaluator prints *canonical representation* of value
- some values have no canonical representation (e.g. functions), some have only infinite ones (e.g. π)

1.3 Functions

- naturally, FP is a matter of functions
- script defines *functions* (*square*, *smaller*)
- (script actually defines *values*; indeed, in FP functions are values)
- function transforms (one or more) arguments into result
- *deterministic*: same arguments always give same result
- may be *partial*: result may sometimes be undefined
- e.g. cosine, square root; distance between two cities; compiler; text formatter; process controller

1.3 Function types

- *type declaration* in script specifies type of function
- e.g. $square :: Integer \to Integer$
- in general, $f :: A \to B$ indicates that function f takes arguments of type A and returns results of type B
- *the interface of a function is manifest*
- *apply* function to argument: $f\ x$
- sometimes parentheses are necessary: $square\ (3 + 4)$ (function application is an operator, binding more tightly than the operator $+$)

1.3 Lambda expressions

- notation for anonymous functions (inventing names is hard)
- e.g. $\backslash x \to x * x$ as another way of writing *square*
- x is the formal parameter; $x * x$ is the body of the function
- ASCII '\' is nearest equivalent to Greek λ
- from Church's λ-calculus theory of computability (1941)

1.3 Lambda expressions

- notation for anonymous functions (inventing names is hard)
- e.g. $\backslash x \rightarrow x * x$ as another way of writing *square*
- x is the formal parameter; $x * x$ is the body of the function
- ASCII '\' is nearest equivalent to Greek λ
- from Church's λ-calculus theory of computability (1941)
- evaluation rule for λ-expressions (β-rule)

$$(\backslash x \rightarrow \mathit{body})\ \mathit{arg} \Longrightarrow \mathit{body}\ \{x := \mathit{arg}\}$$

- function applied to argument reduces to function body, where every occurrence of the formal parameter is replaced by the actual parameter e.g.

$$(\backslash x \rightarrow x + x)\ 47 \Longrightarrow x + x\ \{x := 47\} \Longrightarrow 47 + 47 \Longrightarrow 94$$

1.3 Operators

- functions with alphabetic names are *prefix*: $f\ 3\ 4$
- functions with symbolic names are *infix*: $3 + 4$
- make an alphabetic name infix by enclosing in back-quotes: 17 ‘*mod*‘ 10
- make symbolic operator prefix by enclosing it in parentheses: $(+)\ 3\ 4$
- extend notion to include one argument too: *sectioning*
- e.g. $(1/)$ is the reciprocal function, (>0) is the positivity test

1.3 Associativity

- why operators at all? why not prefix notation?
- there is a problem of ambiguity:

$$x \otimes y \otimes z$$

  what does this mean: $(x \otimes y) \otimes z$ or $x \otimes (y \otimes z)$?
- sometimes it doesn't matter, e.g. addition

$$(x + y) + z = x + (y + z)$$

  the operator $+$ is associative
- *recommendation*: use infix notation *only* for associative operators
- the operator $+$ has also a unit element

$$x + 0 = x = 0 + x$$

- 0 and $+$ form a monoid (more later)

1.3 Association

- some operators are not associative ($-$, $/$, $\hat{}$)
- to disambiguate without parentheses, operators may *associate* to the left or to the right
- e.g. subtraction associates to the left: $5 - 4 - 2 = -1$
- function application associates to the left: $f\ a\ b$ means $(f\ a)\ b$
- function type operator associates to the right:
  $Integer \to Integer \to Integer$ means
  $Integer \to (Integer \to Integer)$
- not to be confused with *associativity*, when adjacent occurrences of same operator are unambiguous anyway

1.3 Precedence

- association does not help when operators are mixed

  $$x \oplus y \otimes z$$

  what does this mean: $(x \oplus y) \otimes z$ or $x \oplus (y \otimes z)$?
- to disambiguate without parentheses, there is a notion of *precedence* (binding power)
- e.g. $*$ has higher precedence (binds more tightly) than $+$

  infixl 7 $*$
  infixl 6 $+$

- function application can be seen as an operator, and has the highest precedence, so $\mathit{square}\ 3 + 4 = 13$

1.3 Composition

- glue functions together with *function composition*
- defined as follows:

$$(g \circ f)\ x = g\ (f\ x)$$

- equivalent definitions: $g \circ f = \backslash x \rightarrow g\ (f\ x)$ and $(\circ)\ g\ f\ x = g\ (f\ x)$
- e.g. function $\mathit{square} \circ \mathit{double}$ takes 3 to 36
- associative, so parentheses not needed in $f \circ g \circ h$

1.4 Declaration vs expression style

- Haskell is a committee language
- Haskell supports two different programming styles
- *declaration style*: using equations, patterns and expressions

$$quad :: Integer \to Integer$$
$$quad\ x = square\ x * square\ x$$

- *expression style*: emphasizing the use of expressions

$$quad :: Integer \to Integer$$
$$quad = \backslash x \to square\ x * square\ x$$

- expression style is often more flexible
- experienced programmers use both simultaneously

1.4 Evaluation of expressions: definition style

- e.g. given (declaration style)

$$
\begin{aligned}
&\mathit{spread}\ f\ g\ x = (f\ x)\ (g\ x) \\
&\mathit{kill}\ a\ \ \ \ \ \ \ \ x = a
\end{aligned}
$$

- we calculate

$$
\begin{aligned}
& \mathit{spread}\ \mathit{kill}\ \mathit{kill}\ 4711 \\
\Longrightarrow\ & \quad \{\ \text{definition of } \mathit{spread}\ \} \\
& (\mathit{kill}\ 4711)\ (\mathit{kill}\ 4711) \\
\Longrightarrow\ & \quad \{\ \text{definition of } \mathit{kill}\ \} \\
& 4711
\end{aligned}
$$

- definitions are applied from left to right

1.4 Evaluation of expressions: expression style

- e.g. given (expression style)

$$\begin{aligned} spread &= \backslash f \rightarrow \backslash g \rightarrow \backslash x \rightarrow (f\ x)\ (g\ x) \\ kill &= \backslash a \rightarrow \backslash x \rightarrow a \end{aligned}$$

- we calculate

$$\begin{aligned} & spread\ kill\ kill\ 4711 \\ \Longrightarrow & \quad \{ \text{definition of } spread \} \\ & (\backslash f \rightarrow \backslash g \rightarrow \backslash x \rightarrow (f\ x)\ (g\ x))\ kill\ kill\ 4711 \\ \Longrightarrow & \quad \{ \beta\text{-rule} \} \\ & (\backslash g \rightarrow \backslash x \rightarrow (kill\ x)\ (g\ x))\ kill\ 4711 \\ \Longrightarrow & \quad \{ \beta\text{-rule} \} \\ & (\backslash x \rightarrow (kill\ x)\ (kill\ x))\ 4711 \\ \Longrightarrow & \quad \{ \beta\text{-rule} \} \\ & (kill\ 4711)\ (kill\ 4711) \\ & \quad \vdots \end{aligned}$$

1.4 Definitions

- we've seen some simple definitions of functions so far
- can also define other kinds of values:

```
name :: String
name = "Ralf"
```

- all definitions so far have had an identifier (and perhaps formal parameters) on the left, and an expression on the right
- other forms possible: conditional, pattern-matching, and local definitions
- also recursive definitions (later sections)

1.4 Conditional definitions

- earlier definition of *smaller* used a *conditional expression*:

$$\begin{aligned}&smaller :: (Integer, Integer) \to Integer\\&smaller\ (x, y) = \mathbf{if}\ x \leqslant y\ \mathbf{then}\ x\ \mathbf{else}\ y\end{aligned}$$

- could also use *guarded equations*:

$$\begin{aligned}&smaller :: (Integer, Integer) \to Integer\\&smaller\ (x, y)\\&\quad \mid x \leqslant y \quad\quad\; = x\\&\quad \mid otherwise = y\end{aligned}$$

- each *clause* has a *guard* and an *expression* separated by =
- last guard can be *otherwise* (synonym for *True*)
- especially convenient with three or more clauses
- *declaration style*: guard; *expression style*: **if** ... **then** ... **else**...

1.4 Pattern matching

- define function by several equations
- arguments on lhs not just variables, but *patterns*
- patterns may be *variables* or *constants* (or *constructors*, later)
- e.g.

$$
\begin{aligned}
&day :: Integer \to String \\
&day\ 1 = \texttt{"Saturday"} \\
&day\ 2 = \texttt{"Sunday"} \\
&day\ \_ = \texttt{"Weekday"}
\end{aligned}
$$

- also *wild-card pattern* _
- evaluate by reducing argument to normal form, then applying first matching equation
- result is undefined if argument has no normal form, or no equation matches

1.4 Local definitions

- repeated sub-expression can be captured in a *local definition*

$$
\begin{aligned}
&sqroots :: (Float, Float, Float) \to (Float, Float)\\
&sqroots\ (a, b, c) = ((-b - sd) \mathbin{/} (2 * a), (-b + sd) \mathbin{/} (2 * a))\\
&\quad \mathbf{where}\ sd = sqrt\ (b * b - 4 * a * c)
\end{aligned}
$$

- scope of **where** clause extends over whole right-hand side
- multiple local definitions can be made:

$$
\begin{aligned}
&demo :: Integer \to Integer \to Integer\\
&demo\ x\ y = (a + 1) * (b + 2)\\
&\quad \mathbf{where}\ a = x - y\\
&\qquad\qquad b = x + y
\end{aligned}
$$

  (nested scope, so layout rule applies here too: all definitions must start in same column)
- in conjunction with guarded equations, the scope of a **where** clause covers all guard clauses

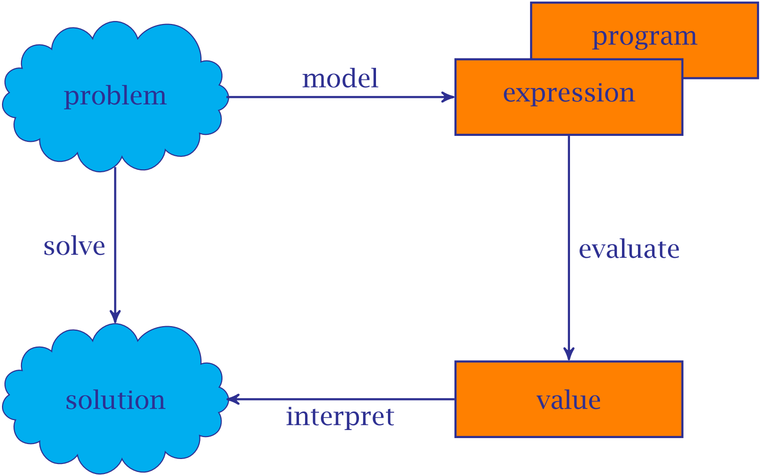
1.4 let-expressions

- a **where** clause is syntactically attached to an equation
- also: definitions local to an expression

$$
\begin{aligned}
&\mathit{demo} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}\\
&\mathit{demo}\ x\ y = \mathbf{let}\ a = x - y\\
&\qquad\qquad\qquad\quad\ b = x + y\\
&\qquad\qquad\quad \mathbf{in}\ (a + 1) * (b + 2)
\end{aligned}
$$

- *declaration style*: **where**; *expression style*: **let ... in...**
- **let**-expressions are more flexible than **where** clauses

1.5 The art of functional programming



1.5 The art of functional programming

- a problem is given by an expression
- a solution is a value
- a solution is obtained by evaluating an expression to a value
- a program introduces vocabulary to express problems and specifies rules for evaluating expressions
- the art of functional programming: finding rules
- Haskell has a very simple computational model
- … as in primary school: replacing equals by equals
- we can calculate not only with numbers, but also with lists, trees, grammars, pictures, music …

Part 2

Types and polymorphism

2.0 Outline

2.1 Strong typing

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- type checking guarantees that type errors cannot occur

2.1 Strong typing

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- type checking guarantees that type errors cannot occur
- Haskell is *statically typed*: type checking occurs before run-time (after syntax checking)
- experience shows well-typed programs are likely to be correct

2.1 Strong typing

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- type checking guarantees that type errors cannot occur
- Haskell is *statically typed*: type checking occurs before run-time (after syntax checking)
- experience shows well-typed programs are likely to be correct
- Haskell can *infer types*: determine the most general type of each expression
- wise to specify (some) types anyway, for documentation and redundancy
- *slogan:* types don't just contain data, types explain data

2.2 Simple types

- Booleans
- characters
- strings
- numbers

2.2 Booleans

- type *Bool* (note: type names capitalized)
- two constants, *True* and *False* (note: constructor names capitalized)
- e.g. definition by pattern-matching

$$
\begin{aligned}
&not :: Bool \to Bool \\
&not\ False = True \\
&not\ True = False
\end{aligned}
$$

- and &&, or || (both non-strict in second argument, DIY short-circuit operators: $a \neq 0\ \&\&\ b\ /\ a > 1$)

$$
\begin{aligned}
&(\&\&) :: Bool \to Bool \to Bool \\
&False\ \&\&\ \_ = False \\
&True\ \&\&\ x = x
\end{aligned}
$$

- comparisons $==$, $\neq$, orderings $<$, $\leqslant$ etc

2.2 Boole design pattern

- every type comes with a pattern of definition
- *task:* define a function $f :: Bool \to S$
- *step 1:* solve the problem for *False*

$$f\ False = \ \ldots$$

- *step 2:* solve the problem for *True*

$$\begin{aligned} f\ False &= \ \ldots \\ f\ True &= \ \ldots \end{aligned}$$

- (*exercise:* define your own conditional)

2.2 Characters

- type *Char*
- constants in single quotes: `'a'`, `'?'`
- special characters escaped: `'\''`, `'\n'`, `'\\'`
- ASCII coding: $ord :: Char \to Int$, $chr :: Int \to Char$ (defined in library module *Data.Char*)
- comparison and ordering, as before

2.2 Strings

- type *String*
- (actually defined in terms of *Char*, see later)
- constants in double quotes: `"Hello"`
- comparison and (lexicographic) ordering
- concatenation $++$
- display function $show :: Integer \to String$ (actually more general than this; see later)

2.2 Numbers

- fixed-precision (32-bit) integers *Int*
- arbitrary-precision integers *Integer*
- single- and double-precision floats *Float*, *Double*
- others too: rationals, complex numbers, ...
- comparisons and ordering
- $+$, $-$, $*$, $\hat{}$
- *abs*, *negate*
- /, *div*, *mod*, *quot*, *rem*
- etc
- operations are overloaded (more later)

2.2 Enumerations

- mechanism for declaring new types

  data *Day* = *Mon* | *Tue* | *Wed* | *Thu* | *Fri* | *Sat* | *Sun*

- e.g. *Bool* is not built in (although **if** ... **then** ... **else** syntax is):

  data *Bool* = *False* | *True*

- types may even be parameterized and recursive! (more later)

2.3 Functions

- naturally, FP is a matter of functions
- function types: e.g. $\mathit{Char} \to \mathit{Int}$
- $X \to Y \to Z$ is shorthand for $X \to (Y \to Z)$
- values in a similar syntax: $\backslash c \to \mathit{ord}\ c - \mathit{ord}$ `'0'`
- i.e. lambda expressions
- recall: c is the formal parameter; $\mathit{ord}\ c - \mathit{ord}$ `'0'` is the body of the function
- $\backslash x\ y \to z$ is shorthand for $\backslash x \to \backslash y \to z = \backslash x \to (\backslash y \to z)$
- function application: $f\ x$ ("space operator")
- $f\ x\ y$ is shorthand for $(f\ x)\ y$

2.4 Tuples

- pairing types: e.g. $(Char, Integer)$
- values in the same syntax: ('a', 440)
- selectors *fst*, *snd*
- definition by pattern matching:

$$fst\ (x, \_) = x$$

- nested tuples: $(Integer, (Char, Bool))$
- triples, etc: $(Integer, Char, Bool)$
- nullary tuple (); *the* value in the same syntax: ()
- *fixed-length* sequences of values of (possibly) *different* types
- comparisons and (lexicographic) ordering

2.5 Parametric polymorphism

- what is the type of *fst*?
- applicable at different types: *fst* (1, 2), *fst* ('a', *True*), ...
- what about strong typing?
- *fst* is *polymorphic* — it works for *any* type of pairs:

$$fst :: (a, b) \to a$$

- *a*, *b* here are *type variables* (uncapitalized)

2.5 A little game

- here is a little game: I give you a type, you give me a function of that type
  - $Int \to Int$
  - $a \to a$
  - $(Int, Int) \to Int$
  - $(a, a) \to a$
  - $(a, b) \to a$
  - $Int \to (Int \to Int)$
  - $(Int \to Int) \to Int$
  - $a \to (a \to a)$
  - $(a \to a) \to a$
- polymorphic functions: flexible to use, “hard” to define
- polymorphism is a property of an algorithm: same code for all types

2.5 Type-driven program development

- types are a vital part of any program
- types are not an afterthought
- first specify the type of a function
- its definition is then driven by the type

$$f :: T \to U$$

- f consumes a T value: suggests
  - case analysis if T is a datatype (more later)
  - use of application if T is a function type
- f produces a U value: suggests
  - use of constructors if U is a datatype (more later)
  - use of lambda expressions if U is a function type

2.5 Example: code inference

- define a total function of type $(((Int \to a) \to a) \to b) \to b$

2.5 Example: code inference

- define a total function of type $(((Int \to a) \to a) \to b) \to b$
- a function is introduced using a lambda expression

$$\backslash f \to \square$$

- f has type $((Int \to a) \to a) \to b$; its body $\square$ has type b

2.5 Example: code inference

- define a total function of type $(((Int \to a) \to a) \to b) \to b$
- a function is introduced using a lambda expression

$$\backslash f \to \square$$

- f has type $((Int \to a) \to a) \to b$; its body $\square$ has type b
- we need to apply the function f

$$\backslash f \to f\ \square$$

- the argument $\square$ of f has type $(Int \to a) \to a$

2.5 Example: code inference

- define a total function of type $(((Int \to a) \to a) \to b) \to b$
- a function is introduced using a lambda expression

$$\backslash f \to \square$$

- f has type $((Int \to a) \to a) \to b$; its body $\square$ has type b
- we need to apply the function f

$$\backslash f \to f\ \square$$

- the argument $\square$ of f has type $(Int \to a) \to a$
- a function is introduced using a lambda expression

$$\backslash f \to f\ (\backslash g \to \square)$$

- g has type $Int \to a$; its body $\square$ has type a

2.5 Example: code inference

- define a total function of type $(((Int \to a) \to a) \to b) \to b$
- a function is introduced using a lambda expression

$$\backslash f \to \square$$

- f has type $((Int \to a) \to a) \to b$; its body $\square$ has type b
- we need to apply the function f

$$\backslash f \to f\ \square$$

- the argument $\square$ of f has type $(Int \to a) \to a$
- a function is introduced using a lambda expression

$$\backslash f \to f\ (\backslash g \to \square)$$

- g has type $Int \to a$; its body $\square$ has type a
- we need to apply the function g

$$\backslash f \to f\ (\backslash g \to g\ \square)$$

- the argument $\square$ of g has type Int

$$\backslash f \to f\ (\backslash g \to g\ 0)$$

2.5 Parametric polymorphism

- h is *polymorphic* — it works for *any* type a and *any* type b

$$h :: (((Int \to a) \to a) \to b) \to b$$
$$h = \backslash f \to f\,(\backslash g \to g\,0)$$

- parametric polymorphism: same code for all types
- values of type a and b are not "inspected" — they are treated as black boxes
- they are only passed around (or ignored, or duplicated)
- algorithm is insensitive to parts of the data

2.6 Type synonyms

- alternative names for types
- brevity, clarity, documentation
- e.g.

$$\textbf{type}\ Card = (Rank, Suit)$$

- cannot be recursive
- just a ‘macro’: no new type

2.7 Type classes

- what about numeric operations?
- $(+) :: Integer \to Integer \to Integer$
- $(+) :: Float \to Float \to Float$
- cannot have $(+) :: a \to a \to a$ (too general)
- the solution is *type classes* (sets of types)
- e.g. the type class *Num* is a set of numeric types; includes *Integer*, *Float*, etc
- now $(+) :: (Num\ a) \Rightarrow (a \to a \to a)$
- *ad-hoc polymorphism* (different code for different types), as opposed to *parametric polymorphism* (same code for all types)

2.7 Some standard type classes

- *Eq*: $==$, $\neq$
- *Ord*: $<$ etc, *min* etc
- *Enum*: *succ*, ..
- *Bounded*: *minBound*, *maxBound*
- *Show*: $show :: a \to String$
- *Read*: $read :: String \to a$
- *Num*: $+$, $*$ etc
- *Real* (ordered numeric types)
- *Integral*: *div* etc
- *Fractional*: $/$ etc
- *Floating*: *exp* etc
- more later

2.7 Derived type classes

- new datatypes are not automatically instances of type classes
- possible to install as instances:

```
data Gender = Female | Male
instance Eq Gender where
  Female == Female = True
  Female == Male   = False
  Male   == Female = False
  Male   == Male   = True
```

- (default definition of $\neq$ obtained for free from $==$, more later)
- tedious for simple cases, which can be derived automatically:

```
data Gender = Female | Male
  deriving (Eq, Ord, Enum, Bounded, Show, Read)
```

2.8 Summary

- type-driven program development
- type safety and flexibility are in tension
- polymorphism partially releases the tension
- *ad-hoc polymorphism:* different code for different types
- *parametric polymorphism:* same code for all types