Fortgeschrittene funktionale Programmierung

LVA 185.A05, VU 2.0, ECTS 3.0
SS 2021

(Stand: 06.05.2021)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Lecture 7

Part V: Applications

- Chapter 17: Pretty Printing
  - Chap. 17.6: Recommended Reading: Basic, Advanced
- Chapter 18: Functional Reactive Programming
  - Chap. 18.5: Recommended Reading: Basic, Advanced

Part VI: Extensions, Perspectives

- Chapter 19: Extensions to Parallel and 'Real World' Functional Programming
  - Chap. 19.3: Recommended Reading: Basic, Advanced
- Chapter 20: Conclusions and Perspectives
  - Chap. 20.4: Recommended Reading: Basic, Advanced

Outline in more Detail (1)

Part V: Applications

Outline in more Detail (2)

Outline in More Detail (3)

- Chap. 18: Functional Reactive Programming (cont'd)
  - 18.4 In Conclusion
  - 18.5 References, Further Reading

Part VI: Extensions, Perspectives

- Chap. 19: Extensions: Parallel and 'Real World' Functional Programming
  - 19.1 Parallelism in Functional Languages
  - 19.2 Haskell for 'Real World' Programming
  - 19.3 References, Further Reading
- Chap. 20: Conclusions, Perspectives
  - 20.1 Research Venues, Research Topics, and More
  - 20.2 Programming Contest
  - 20.3 In Conclusion
  - 20.4 References, Further Reading

Chapter 17

Pretty Printing

Chapter 17.1

Motivation

Pretty Printing

...is about

- 'beautifully' printing values of tree-like structures as plain text.

A pretty printer is a

- tool (often a library of routines) designed for converting a tree value into plain text

such that the

- tree structure is preserved and reflected by indentation while utilizing a minimum number of lines to display the tree value.

Pretty printing can thus be considered

- dual to parsing.

Pretty Printing

...is just as parsing often used for demonstrating the power and elegance of functional programming, where not just the

- printed result of a pretty printer shall be 'pretty'
- but also the pretty-printer ifself including that its code is short and fast, and its operators enjoy properties which are appealing from a mathematical point of view.

Overall, a 'good' pretty printer must properly balance:

- Ease of use
- Flexibility of layout
- 'Beauty' of output

...while being ifself 'pretty.'

The Prettier Printer

...presented in this chapter has been proposed by Philip Wadler in:

- Philip Wadler. A Prettier Printer. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

which has been designed to improve (cf. Chapter 17.5) on a pretty printer proposed by John Hughes which is widely recognized as a standard:

- John Hughes. The Design of a Pretty-Printer Library. In Johan Jeuring, Erik Meijer (Eds.), Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques. Springer-V., LNCS 925, 53-96, 1995.

Outline and Assumptions

...the implementation of the simple pretty printer and the prettier printer of Philip Wadler assumes some implementation of a type of documents `Doc`.

The

1. simple pretty printer (cf. Chapter 17.2)
   - implements `Doc` as strings.
   - supports for every document only one possible layout, in particular, no attempt is made to compress structure onto a single line.
2. prettier printer (cf. Chapter 17.3)
   - implements `Doc` in terms of suitbable algebraic sum data types.
   - allows multiple layouts of a document and to pick a best one out of them for printing a document.

Chapter 17.2

The Simple Pretty Printer

Chapter 17.2.1

Basic Document Operators

The Simple Pretty Printer

...(as well as the prettier printer later on) relies on six basic document operators:

```
Associative operator for concatenating documents:
 (<>) :: Doc -> Doc -> Doc
The empty document being a right and left unit for (<>):
 nil :: Doc
Converting a string into a document (arguments of
function text shall not contain newline characters):
 text :: String -> Doc
The document representing a line break:
 line :: Doc
Adding indentation to a document:
 nest :: Int -> Doc -> Doc
Layouting a document as a string:
 layout :: Doc -> String
```

String Documents

...choosing for the simple pretty printer strings for implementing documents, i.e.:

- `type Doc = String`

the implementation of the basic operators boils down to:

- `(<>)`: String concatenation `++`.
- `nil`: The empty string `[]`.
- `text`: The identity on strings.
- `line`: The string formed by the newline character `'\n'`.
- `nest i`: indentation, adding `i` spaces (only used after line breaks by means of `line`).
- `layout`: The identity on strings.

Note

...the coupling of `line` and `nest` is an essential difference to the pretty printer of John Hughes, where insertion of spaces is also allowed in front of strings.

This difference is key for succeeding with only one concatenation operator for documents instead of the two in the pretty printer of John Hughes (cf. Chapter 17.5).

Chapter 17.2.2

Normal Forms of String Documents

String Documents

...can always be reduced to a normal form representation alternating applications of function

– text with line breaks nested to a given indentation:

```
text s_0 <> nest i_1 line <> text s_1 <> ...
                    <> nest i_k line <> text s_k
```

where every

– s_j is a string (possibly empty).
– i_j is a natural number (possibly zero).

Example: Normal Form Representation

The document (i.e., a Doc-value):

```
text "bbbbb" <> text "[" <>
nest 2 (
      line <> text "ccc" <> text "," <>
      line <> text "dd"
) <>
line <> text "]" :: Doc
```

which prints as:

```
bbbbb[
  ccc,
  dd
]
```

has the normal form (representation):

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"  :: Doc
```

Normal Form Representations

...of string documents exist because of a variety of laws the basic operators of the simple pretty printer enjoy. In particular:

Lemma 17.2.2.1 (Associativity of Doc. Concatenat.)

(<>) is associative with unit nil.

...as well as the collection of basic operator laws compiled in Lemma 17.2.2.2.

Basic Operators Laws

Lemma 17.2.2.2 (Basic Operator Laws)

1. Operator text is a homomorphism from string to document concatenation:

```
text (s ++ t)    = text s <> text t
text ""          = nil
```

2. Opr. nest is a homomorph. from addition to composition:

```
nest (i+j) x     = nest i (nest j x)
nest 0 x         = x
```

3. Opr. nest distributes through document concatenation:

```
nest i (x <> y) = nest i x <> nest i y
nest i nil       = nil
```

4. Nesting is absorbed by text (differently to the pretty printer of Hughes):

```
nest i (text s) = text s
```

Note

...the laws compiled in Lemma 17.2.2.1 and 17.2.2.2

- come, except of the last one, in pairs with a corresponding law for the unit of the respective operator.
- are sufficient to ensure that every document can be transformed into normal form, where the
  - laws of part 1) and 2) are applied from left to right.
  - last of part 3) and 4) are applied from right to left.

Laws

...relating string documents with their layouts:

Lemma 17.2.2.3 (Layout Operator Laws)

1. Operator layout is a homomorphism from document to string concatenation:

```
layout (x <> y)      = layout x ++ layout y
layout nil           = ""
```

2. Operator layout is the inverse of function text:

```
layout (text s)      = s
```

3. The result of layout applied to a nested line is a newline followed by one space for each level of indentation:

```
layout (nest i line) = '\n' : copy i ' '
```

Chapter 17.2.3

Printing Trees

Using the Simple Pretty Printer

...for prettily printing values of the data type Tree defined by:

```
data Tree = Node String [Tree]
```

For illustration, consider Tree-value t:

```
t = Node "aaa"
      [Node "bbbbb" [Node "ccc" [],Node "dd" []],
       Node "eee" [],
       Node "ffff"
         [Node "gg" [],Node "hhh" [],Node "ii" []]]
```

Two different Layouts of t as Strings

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

where
```
t = Node "aaa"
     [Node "bbbbb" [Node "ccc" [],Node "dd" []],
      Node "eee" [],
      Node "ffff"
       [Node "gg" [],Node "hhh" [],Node "ii" []]]
```

The Layout Strategies

...used for layouting and printing tree `t`:

– Left: Tree siblings start on a new line, properly indented.
– Right: Every subtree starts on a new line, properly indented by two spaces.

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

Implementing the 'Left' Layout Strategy

...by means of a utility function showTree converting a tree into a string document according to the 'left' layout strategy:

```
type Doc  = String
data Tree = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) =
 text s <> nest (length s) (showBracket ts)

showBracket :: [Tree] -> Doc
showBracket [] = nil
showBracket ts =
 text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees :: [Tree] -> Doc
showTrees [t]    = showTree t
showTrees (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

Implementing the 'Right' Layout Strategy

...by means of a utility function showTree′ converting a tree into a string document according to the 'right' layout strategy:

```
type Doc  = String
data Tree = Node String [Tree]

showTree' :: Tree -> Doc
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' :: [Tree] -> Doc
showBracket' [] = nil
showBracket' ts =
 text "[" <> nest 2 (line <> showTrees' ts) <> line
                                          <> text "]"

showTrees' :: [Tree] -> Doc
showTrees' [t]    = showTree t
showTrees' (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

Chapter 17.3

The Prettier Printer

Chapter 17.3.1

Algebraic Documents

Algebraic Documents

...for the prettier printer we consider a document a

– concatenation of items, where each item is a text or a line break indented a given amount.

Documents are thus implemented as an algebraic sum data type:

```
data Doc = Nil
         | String `Text` Doc
         | Int `Line` Doc
```

Note, the data constructors `Nil`, `Text`, and `Line` of `Doc` relate to the basic document operators `nil`, `text`, and `line` of the simple pretty printer as follows:

(1)	`Nil`	$\widehat{=}$	`nil`
(2)	``s `Text` x``	$\widehat{=}$	`text s <> x`
(3)	``i `Line` x``	$\widehat{=}$	`nest i line <> x`

Example: String vs. Algebraic Document Rep.

...the normal form representation of the string document considered in Chapter 17.2.2:

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

...is represented by the algebraic Doc-value:

```
"bbbbb[" `Text` (
2 `Line` ("ccc," `Text` (
2 `Line` ("dd," `Text` (
0 `Line` ("]," `Text` Nil))))))
```

Chapter 17.3.2

Implementing Document Operators on Algebraic Documents

Implementations

...of the basic document operators on algebraic documents can easily be derived from 'equations' (1) - (3) of Chapter 17.3.1:

```
nil                  =  Nil
text s               =  s `Text` Nil
line                 =  0 `Line` Nil

Nil <> y             =  y
(s `Text` x) <> y    =  s `Text` (x <> y)
(i `Line` x) <> y    =  i `Line` (x <> y)

nest i Nil           =  Nil
nest i (s `Text` x)  =  s `Text` nest i x
nest i (j `Line` x)  =  (i+j) `Line` nest i x

layout Nil           =  ""
layout (s `Text` x)  =  s ++ layout x
layout (i `Line` x)  =  `\n` : copy i ` ` ++ layout x
```

Justification

...for the derived definitions can be given using equational reasoning, e.g.:

Proposition 17.3.2.1

```
(s `Text` x) <> y = s `Text` (x <> y)
```

Proof by equational reasoning.

```
    (s `Text` x) <> y
 =   { Definition of Text, equ. (2) }
    (text s <> x) <> y
 =   { Associativity of <> }
     text s <> (x <> y)
 =   { Definition of Text, equ. (2) }
     s `Text` (x <> y)
```

□

...similarly, correctness of the other equations from the previous slide can be shown.

Chapter 17.3.3

Multiple Layouts of Algebraic Documents

Single vs. Multiple Layouts of Documents

...so far, a document `d` could essentially be considered equivalent to a

- single string defining a unique single layout for `d`.

Next, a document shall be considered equivalent to a

- set of strings, each of them defining a layout for `d`, together thus multiple layouts.

To achieve this, only one new document operator must be added:

```
group :: Doc -> Doc
group x = flatten x <|> x
```

with `flatten` and `(<|>)` to be defined soon.

The Meaning of group

...applied to a document representing a set of layouts, group

- returns the set with one new element added representing the layout, in which everything is compressed on one line.

This is achieved by

- replacing each newline (and the corresponding indentation) with text consisting of a single space.

Note: Variants where

- each newline carries with it the alternate text it should be replaced with

are possible, e.g. some newlines might be replaced by the empty text, others by a single space (but are not considered here).

The relative 'Beauty' of a Layout

...depends much on the preferred maximum line width considered eligible for a layout.

Therefore, the document operator `layout` used so far is replaced by a new operator `pretty`:

```
pretty :: Int -> Doc -> String
```

which picks the 'prettiest' among a set of layouts depending on the `Int`-value of the preferred maximum line width argument.

Example

...replacing showTree of the 'left' layout strategy for trees of Chapter 17.2.3:

```
data Tree = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) =
  text s <> nest (length s) (showBracket ts)
```

by a refined version with an additional call of group:

```
showTree (Node s ts) =
  group (text s <> nest (length s) (showBracket ts))
```

will ensure that

- trees are fit onto one line where possible ($\leq$ *max* width).
- sufficiently many line breaks are inserted in order to avoid exceeding the preferred maximum line width.

Example (cont'd)

...calling, e.g., `pretty 30` will (when completely specified!) yield the output:

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

Defining the new Operators (<|>), flatten

...for completing the implementation of the operators group and pretty.

Union operator, forming the union of two sets of layouts:

```
(<|>) :: Doc -> Doc -> Doc
```

Flattening operator, replacing each line break (and its associated indentation) by a single space:

```
flatten :: Doc -> Doc
```

Note: The operators <|> and flatten will not directly exposed to the user but only via group and the operators fillwords and fill defined in Chapter 17.3.6.

Required Invariant for `(<|>)`

...assuming that a document always represents a non-empty set of layouts, which all flatten to the same layout, the following invariant for the union operator `(<|>)` is required:

- Invariant: In `(x <|> y)` all layouts of `x` and `y` flatten to the same layout.

...this invariant must be ensured when creating a union `(<|>)`.

Distribution Laws

...required for the implementations of `(<|>)` and `flatten`.

Each operator on simple documents extends pointwise through union:

Distributive Laws for `(<|>)`

```
1. (x <|> y) <> z    = (x <> z) <|> (y <> z)
2. x <> (y <|> z)    = (x <> y) <|> (x <> z)
3. nest i (x <|> y) = nest i x <|> nest i y
```

Since flattening gives the same result for each element of a set, the distribution law for `flatten` is simpler:

Distributive Law for `flatten`

```
flatten (x <|> y) = flatten x
```

Interaction Laws

...required for the implementation of `flatten`.

Concerning the interaction of `flatten` with other document operators:

Interaction Laws for `flatten`

```
1. flatten (x <> y)     = flatten x <> flatten y
2. flatten nil          = nil
3. flatten (text s)     = text s
4. flatten line         = text " "
5. flatten (nest i x)   = flatten x
```

Note, laws (4) and (5) are the most interesting ones:

- (4): linebreaks are replaced by a single space.
- (5): indentations are removed.

Recalling the Implementation

...of group in terms of flatten and (<|>):

```
group :: Doc -> Doc
group x = flatten x <|> x
```

Recall, too:

– Documents always represent a non-empty set of layouts whose elements all flatten to the same layout.
– group adds the flattened layout to a set of layouts.

Chapter 17.3.4

Normal Forms of Algebraic Documents

Normal Form Representations

...due to the laws for flattening (`flatten`) and union (`(<|>)`) every document can be reduced to a representation in normal form of the form:

```
x_1 <|> ... <|> x_n
```

where every `x_j` is in the normal form of simple documents (cf. Chapter 17.2.2).

Picking a 'prettiest' Layout

...out of a set of layouts is done by means of an ordering relation on lines depending on the preferred maximum line width, and extended lexically to an ordering between documents.

Out of two lines

- which both do not exceed the maximum width, pick the longer one.
- of which at least one exceeds the maximum width, pick the shorter one.

Note: These rules reqire to pick sometimes a layout where some lines exceed the limit. This is an important difference to the approach of John Hughes, done only, however, if unavoidable.

Adapting the Algebraic Definition of Doc

...the algebraic definition of Doc of Chapter 17.3.1 is extended by a new data vconstructor Union representing the union of two documents:

```
data Doc = Nil
         | String `Text` Doc
         | Int `Line` Doc
         | Doc `Union` Doc    -- Union, the new
                              -- data constructor!
```

Note, these data value constructors relate to the basic document operators as follows:

```
(1)  Nil            ≙  nil
(2)  s `Text` x     ≙  text s <> x
(3)  i `Line` x     ≙  nest i line <> x
(4)  x `Union` y    ≙  x <|> y
```

Required Invariants for `Union`

...assuming again that a document always represents a non-empty set of layouts flattening all to the same layout, two invariants are required for `Union`:

- Invariant 1: In `(x 'Union' y)` all layouts of `x` and `y` flatten to the same layout.
- Invariant 2: Every first line of a document in `x` is at least as long as every first line of a document in `y`.

...these invariants must be ensured when creating a `Union`.

Performance

...of pretty printing is improved by applying the distributive law for `Union` giving

```
(s `Text` (x `Union` y))
```

preference to the equivalent

```
((s `Text` x) `Union` (s `Text` y))
```

Illustrating the Performance Impact (1)

...of distributivity considering the document:

```
group(
   group(
        group(
           group(text "hello" <> line <> text "a")
        <> line <> text "b")
    <> line <> text "c")
<> line <> text "d")
```

...and its possible layouts:

```
hello a b c d   hello a b c   hello a b   hello a   hello
                d             c           b         a
                              d           c         b
                                          d         c
                                                    d
```

Illustrating the Performance Impact (2)

...printing the previous document within a maximum line width of 5, its

- right-most layout must be picked

...ideally, while the other ones are eliminated in one fell swoop.

Intuitively, this is achieved by picking a representation, which brings to the front any common string, e.g.:

```
"hello" `Text` ((" ") `Text` x) `Union` (0 `Line` y))
```

for suitable documents `x` and `y`, where `"hello"` has been factored out of all the layouts in `x` and `y`, and `" "` of all the layouts in `x`.

Since `"hello"` followed by `" "` is of length 6 exceeding the limit 5, the right operand of `Union` can immediately be chosen without further examination of `x`, as desired.

Fixing the Performance Issue

...to realize this, (<>) and nest must be extended to specify how they interact with Union:

```
 (x `Union` y) <> z    = (x <> z) `Union` (y <> z)   (1)
nest k (x `Union` y) = nest k x `Union` nest k y   (2)
```

while the definitions of nil, text, line, (<>), and nest remain unchanged.

Note, (1) and (2) follow from the distributive laws. In particular, they preserve Invariant 2 required by Union.

Algebraic Definitions

...of group and flatten are then easily derived:

```
group Nil              = Nil
group (i `Line` x)     = (" " `Text` flatten x)
                              `Union` (i `Line` x)
group (s `Text` x)     = s `Text` group x
group (x `Union` y)    = group x `Union` y

flatten Nil            = Nil
flatten (i `Line` x)   = " " `Text` flatten x
flatten (s `Text` x)   = s `Text` flatten x
flatten (x `Union` y)  = flatten x
```

Justification (1)

...for the derived definitions can be given using equational reasoning, e.g.:

Proposition 17.3.4.1

```
group (i `Line` x) =
        (" " `Text` flatten x) `Union` (i `Line` x)
```

Proof by equational reasoning.

```
   group (i `Line` x)
=  { Definition of Line, equ. (3) }
   group (nest i line <> x)
=  { Definition of group }
   flatten (nest i line <> x) <|> (nest i line s <> x)
=  { Definition of flatten }
   (text " " <> flatten x) <|> (nest i line <> x)
=  { Definition of Text, Union, Line, equ. (2), (4), (3) }
   (" " `Text` flatten x) `Union` (i `Line` x)
```
□

Justification (2)

Proposition 17.3.4.2

```
group (s `Text` x) = s `Text` group x
```

Proof by equational reasoning.

```
    group (s `Text` x)
 =  {Definition of Text, equ. (2)}
    group (text s <> x)
 =  {Definition of group}
    flatten (text s <> x) <|> (text s <> x)
 =  {Definition of flatten}
    (text s <> flatten x) <|> (text s <> x)
 =  {(<>) distributes through (<|>)}
    text s <> (flatten x <|> x)
 =  {Definition of group}
    text s <> group x
 =  {Definition of Text, equ. (2)}
    s `Text` group x
```

□

Picking the 'best' Layout (1)

...among a set of layouts using functions best and better:

```
best w k Nil             = Nil
best w k (i `Line` x)   = i `Line` best w i x
best w k (s `Text` x)
         = s `Text` best w (k + length s) x
best w k (x `Union` y)
         = better w k (best w k x) (best w k y)

better w k x y
         = if fits (w-k) x then x else y
```

Note:

- best: Converts a 'union'-afflicted document into a 'union'-free document.
- Argument w: Maximum line width.
- Argument k: Already consumed letters (including indentation) on current line.

Picking the ‘best’ Layout (2)

Check, if the first document line stays within the maximum line length w:

```
fits w x | w<0       = False -- cannot fit
fits w Nil           = True  -- fits trivially
fits w (s `Text` x)
  = fits (w - length s) x    -- fits if x fits into
                             -- the remaining space
                             -- after placing s
fits w (i `Line` x) = True  -- yes, it fits
```

Last but not least, the output routine: Pick the best layout and convert it to a string:

```
pretty w x = layout (best w 0 x)
```

Chapter 17.3.5

Improving Performance

Intuitively

...pretty printing a document should be doable in time $\mathcal{O}(s)$, where s is the size of the document, i.e., a count of

- the number of `(<>)`, `nil`, `text`, `nest`, and `group` operations
- plus the length of all string arguments to `text`.

and in space proportional to $\mathcal{O}(w\ max\ d)$, where

- w is the width available for printing
- d is the depth of the document, the depth of calls to `nest` or `group`.

Sources of Inefficiency

...of the prettier printer implementation so far:

1. Document concatenation might pile up to the left:

```
(...((text s_0 <> text s_1) <> ...) <> text s_n
```

   ...assuming each string has length one, this may require time $\mathcal{O}(n^2)$ to process (instead of $\mathcal{O}(n)$ as hoped for).

2. Nesting of documents adds a layer of processing to increment the indentation of the inner document:

```
nest i_o (text s_0 <> nest i_1 (text s_1 <>
                    ... <> nest i_n (text s_n)...))
```

   ...even if we assume document concatenation associates to the right.

   ...assuming again each string has length one, this may also require time $\mathcal{O}(n^2)$ to process (instead of $\mathcal{O}(n)$ as hoped for).

Performance Fixes

...for inefficiency source 1):

- Adding an explicit representation for concatenation, and generalizing each operation to act on a list of concatenated documents.

...for inefficiency source 2):

- Adding an explicit representation for nesting, and maintaining a current indentation that is incremented as nesting operators are processed.

Combining both fixes suggests

- generalizing each operation to work on a list of indentation-document pairs.

Implementing the Fixes

...by switching to a new representation for documents such that there is one data constructor for every operator building a document:

```
data DOC = NIL
         | DOC :<> DOC
         | NEST Int DOC
         | TEXT String
         | LINE
         | DOC :<|> DOC
```

Note: To avoid name clashes with the previous definitions, capital letters are used.

Implementing the Document Operators

...building a document of the new algebraic type is straightforward:

```
nil        = NIL
x <> y     = x :<> y
nest i x = NEST i x
text s     = TEXT s
line       = LINE
```

As before, also the invariants on the equality of flattened layouts and on the relative lengths of first lines are required:

- In `(x :<|> y)` all layouts in `x` and `y` flatten to the same layout.
- No first line in `x` is shorter than any first line in `y`.

Implementing `group` and `flatten`

...for the new algebraic type is straightforward, too:

```
group x              = flatten x :<|> x

flatten NIL          = NIL
flatten (x :<> y)    = flatten x:<> flatten y
flatten (NEST i x) = NEST i (flatten x)
flatten (TEXT s)     = TEXT s
flatten LINE         = TEXT " "
flatten (x :<|> y) = flatten x
```

...the definitions follow immediately from the equations given before.

The Representation Function `rep`

...maps a list of indentation-document pairs into the corresponding document:

```
rep z = fold (<>) nil [nest i x | (i,x) <- z]
```

Finding the 'best' Layout

...the operation best of Chapter 17.3.4 to find the 'best' layout of a document is generalized to act on a list of indentation-document pairs by combining it with the new representation function rep:

```
be w k z = best w k (rep z)      (hypothesis)
```

The new definition is directly derived from the old one:

```
best w k x                   = be w k [(0,x)]

be w k []                    = Nil
be w k ((i,NIL):z)           = be w k z
be w k ((i,x :<> y) : z)     = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z)    = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)      = s `Text` be w (k,+length s) z
be w k ((i,LINE) : z)        = i `Line` be w i z
be w k ((i.x :<|> y) : z) =
  better w k (be w k ((i.x) : z)) (be w k (i,y) : z))
```

Correctness

...of the equations of the previous slide can be shown by equational reasoning, e.g.:

Proposition 17.3.5.1

```
best w k x = be w k [(0,x)]
```

Proof by equational reasoning.

```
  best w k x
=  {0 is unit for nest}
  best w k (nest 0 x)
=  {nil is unit for <>}
  best w k (nest 0 x <> nil)
=  {Definition of rep, hypothesis}
  be w k [(0,x)]
```

□

Last but not least

...while the argument to best is represented using

- DOC

its result is represented using the formerly introduced type

- Doc

Hence, pretty can be defined as in Chapter 17.3.4:

```
pretty w x = layout (best w 0 x)
```

The functions layout, better, and fits, finally, remain unchanged.

Chapter 17.3.6

Utility Functions

Utility Functions (1)

...for recurringly occurring tasks, e.g.:

- Separating two documents by inserting a space:

```
x <+> y           = x <> text " " <> y
```

- Separating two documents by inserting a line break:

```
x </> y           = x <> line <> y
```

- Folding a document:

```
folddoc f []      = nil
folddoc f [x]     = x
folddoc f (x:xs) = f x (folddoc f xs)
```

- Advanced document folding:

```
spread            = folddoc (<+>)
stack             = folddoc (</>)
```

Utility Functions (2)

...as abbreviations of frequently occurring tasks, e.g.:

- An opening bracket, followed by an indented portion, followed by a closing bracket, abbreviated by `bracket`:

```
bracket l x r = group (text l <>
                       nest 2 (line <> x) <>
                       line <> text r)
```

- The 'right' layout strategy for trees of Chapter 17.2.3, abbreviated by `showBracket'`:

```
showBracket' ts = bracket "[" (showTrees' ts) "]"
```

- Taking a string, returning a document, where every line is filled with as many words as will fit (note: `words` is from the Haskell Standard Library), abbreviated by `fillwords`:

```
x <+/> y  = x <> (text " " :<|> line) <> y
fillwords = folddoc (<+/>) . map text . words
```

Utility Functions (3)

...abbreviations (cont'd):

- A variant of `fillwords` collapsing a list of documents to a single document by putting a space between two documents when this leads to a reaonsable layout, and a newline otherwise, abbreviated by `fill`:

```
fill []         = nil
fill [x]        = x
fill (x:y:zs) =
  (flatten x <+> fill (flatten y : zs)) :<|>
                               (x </> fill (y : zs)
```

Note: `fill` is copied from pretty printer library of Simon Peyton Jones, which extends the one of John Hughes.

Chapter 17.3.7

Printing XML-like Documents

Printing XML Documents

...enjoying a simplified XML syntax with elements, attributes, and text defined by:

```
data XML = Elt String [Att] [XML]
         | Txt String

data Att = Att String String
```

Utility Functions (1)

...for printing XML documents:

- Showing documents:

```
showXML x = folddoc (<>) (showXMLs x)
```

- Showing elements:

```
showXMLs (Elt n a []) =
 [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c) =
 [text "<" <> showTag n a <> text ">" <>
  showFill showXMLs c <>
  text "</" <> text n <> text ">"]
```

- Showing text:

```
showXMLs (Txt s) = map text (words s)
```

- Showing attributes:

```
showAtts (Att n v) =
 [text n <> text "=" <> text (quoted v)]
```

Utility Functions (2)

...for printing XML documents (cont'd):

- Adding quotes:

```
quoted s = "\"" ++ s ++ "\""
```

- Showing tags:

```
showTag n a    = text n <> showFill showAtts a
```

- Filling lines:

```
showFill f [] = nil
showFill f xs =
  bracket "" (fill (concat (map f xs))) ""
```

Example: 1st Layout of an XML Document

...for a maximum line width of 30 characters:

```
<p
  color="red" font="Times"
  size="10"
>
  Here is some
  <em> emphasized </em> text.
  Here is a
  <a
    href="http://www.eg.com/"
  > link </a>
  elsewhere.
</p>
```

Example: 2nd Layout of an XML Document

...for a maximum line width of 60 characters:

```
<p color="red" font="Times" size="10" >
  Here is some <em> emphasized </em> text. Here is a
  <a href="http://www.eg.com/" > link </a> elsewhere.
</p>
```

Example: 3rd Layout of an XML Document

...dropping the two occurrences of `flatten` in `fill` (cf. Chapter 17.3.6) leads to the following output:

```
<p color="red" font="Times" size="10" >
  Here is some <em>
    emphasized
  </em> text. Here is a <a
    href="http://www.eg.com/"
  > link </a> elsewhere.
</p>
```

...in the above layout start and close tags of the emphasis and anchor elements are crammed together with other text, rather than getting lines to themselves; it thus looks less 'beautiful.'

Chapter 17.4

The Prettier Printer Code Library

A Summary

...of the code of the

- performance-improved fully-fledged prettier printer.
- tree example.
- XML-documents example.

according to:

- Philip Wadler. A Prettier Printer. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 2003.

Chapter 17.4.1

The Prettier Printer

The Prettier Printer (1)

Defining operator priorities

```
infixr 5:<|>
infixr 6:<>
infixr 6 <>
```

Defining algebraic document types

```
data DOC = NIL
         | DOC :<> DOC
         | NEST Int DOC
         | TEXT String
         | LINE
         | DOC :<|> DOC

data Doc = Nil
         | String `Text` Doc
         | Int `Line` Doc
```

The Prettier Printer (2)

Defining basic operators algebraically

```
nil        = NIL
x <> y     = x :<> y
nest i x = NEST i x
text s     = TEXT s
line       = LINE
```

Layouting normal form documents

```
layout Nil          = ""
layout (s `Text` x) = s ++ layout x
layout (i `Line` x) = '\n': copy i ' ' ++ layout x

copy i x            = [x | _ <- [1..i]]
```

The Prettier Printer (3)

Generating multiple layouts

```
group x = flatten x :<|> x
```

Flattening layouts

```
flatten NIL          = NIL
flatten (x :<> y)    = flatten x:<> flatten y
flatten (NEST i x)   = NEST i (flatten x)
flatten (TEXT s)     = TEXT s
flatten LINE         = TEXT " "
flatten (x :<|> y)   = flatten x
```

The Prettier Printer (4)

Ordering and comparing layouts

```
best w k x = be w k [(0,x)]

be w k [] = Nil
be w k ((i,NIL):z) = be w k z
be w k ((i,x :<> y) : z) = be w k ((i,x) : (i,y): z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z) = s `Text` be w (k+length s) z
be w k ((i,LINE) : z) = i `Line` be w i z
be w k ((i.x :<|> y) : z) =
  better w k (be w k ((i.x) : z)) (be w k (i,y) : z))

better w k x y = if fits (w-k) x then x else y

fits w x | w<0       = False
fits w Nil           = True
fits w (s `Text` x) = fits (w - length s) x
fits w (i `Line` x) = True
```

The Prettier Printer (5)

Printing documents prettily

```
pretty w x = layout (best w 0 x)
```

Defining utility functions

```
x <+> y          = x <> text " " <> y
x </> y          = x <> line <> y
x <+/> y         = x <> (text " " :<|> line) <> y

folddoc f []     = nil
folddoc f [x]    = x
folddoc f (x:xs) = f x (folddoc f xs)

spread           = folddoc (<+>)
stack            = folddoc (</>)

bracket l x r    =
 group (text l <> nest 2 (line <> x) <>
                                      line <> text r)
```

The Prettier Printer (6)

Defining utility functions (cont'd)

```
fillwords      = folddoc (<+/>) . map text . words

fill []        = nil
fill [x]       = x
fill (x:y:zs) =
 (flatten x <+> fill (flatten y : zs))
                          :<|> (x </> fill (y : zs)
```

Chapter 17.4.2

The Tree Example

The Tree Example (1)

Defining trees

```
data Tree = Node String [Tree]
```

Defining utility functions

```
showTree (Node s ts) =
 group (text s <> nest (length s) (showBracket ts))

showBracket [] = nil
showBracket ts =
 text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees [t]     = showTree t
showTrees (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

The Tree Example (2)

Defining utility functions (cont'd)

```
showTree′ (Node s ts) = text s <> showBracket′ ts

showBracket′ [] = nil
showBracket′ ts = bracket "[" (showTrees′ ts) "]"

showTrees′ [t]    = showTree t
showTrees′ (t:ts) =
 showTree t <> text "," <> line <> showTrees ts
```

The Tree Example (3)

Defining a tree value for illustration

```
tree = Node "aaa"[ Node "bbbb"[ Node "ccc"[],
                                Node "dd"[]
                              ],
                   Node "eee"[],
                   Node "ffff"[ Node "gg"[],
                                Node "hhh"[],
                                Node "ii"[]
                              ]
                 ]
```

Defining two testing environments

```
testtree w  = putStr(pretty w (showTree tree))
testtree′ w = putStr(pretty w (showTree′ tree))
```

Chapter 17.4.3

The XML Example

The XML Example (1)

Defining the XML-like document format

```
data XML = Elt String [Att] [XML]
         | Txt String

data Att = Att String String
```

Defining utility functions

```
showXML x = folddoc (<>) (showXMLs x)

showXMLs (Elt n a []) =
 [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c) =
 [text "<" <> showTag n a <> text ">" <>
   showFill showXMLs c <>
    text "</" <> text n <> text ">"]
showXMLs (Txt s) = map text (words s)
```

The XML Example (2)

Defining utility functions (cont'd)

```
showAtts (Att n v) =
 [text n <> text "=" <> text (quoted v)]

quoted s = "\"" ++ s ++ "\""

showTag n a = text n <> showFill showAtts a

showFill f [] = nil
showFill f xs =
 bracket "" (fill (concat (map f xs))) ""
```

The XML Example (3)

Defining an XML-document value for illustration

```
xml =
 Elt "p"[Att "color" "red",
         Att "font" "Times",
         Att "size" "10"
        ] [Txt "Here is some",
           Elt "em" [] [Txt "emphasized"],
           Txt "text.",
           Txt "Here is a",
           Elt "a" [Att "href" "http://www.eg.com/"]
                   [Txt "link" ],
           Txt "elsewhere."
          ]
```

Defining a testing environment

```
testXML w = putStr (pretty w (showXML xml))
```

Chapter 17.5

Summary

Summary

...the pretty printer library proposed by John Hughes is widely recognized as a standard:

- John Hughes. The Design of a Pretty-Printer Library. In Johan Jeuring, Erik Meijer (Eds.), Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques. Springer-V., LNCS 925, 53-96, 1995.

...a variant of it is implemented in the Glasgow Haskell Compiler:

- Simon Peyton Jones. Haskell pretty-printer library. 1997. `www.haskell.org/libraries/#prettyprinting`

Why ‘prettier’ than ‘pretty’?

...the pretty printer of John Hughes

- uses two operators for the horizontal and vertical concatenation of documents
  - one without a unit (vertical)
  - one with a right-unit but no left-unit (horizontal).

...the prettier printer of Philip Wadler can be considered an improvement of the pretty printer of John Hughes because it

- uses only one operator for document concatenation which
  - is associative.
  - has a left-unit and a right-unit.
- consists of about 30% less code.
- is about 30% faster.

In Closing

...a hint to an early work on an imperative pretty printer by:

- Derek Oppen. Pretty-printing. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.

and a functional realization of it by:

- Olaf Chitil. Pretty Printing with Lazy Dequeues. In Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.

Chapter 17.6

References, Further Reading

Chapter 17: Basic Reading

- Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 223-243, 2003.
- John Hughes. *The Design of a Pretty-Printer Library*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 53-96, 1995.
- Tillmann Rendel, Klaus Ostermann. *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing*. In Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell 2010), 1-12, 2010.
- Simon Peyton Jones. *Haskell pretty-printer library*. 1997. `www.haskell.org/libraries/#prettyprinting`

Chapter 17: Selected Advanced Reading

- Derek Oppen. *Pretty-printing*. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.
- Olaf Chitil. *Pretty Printing with Lazy Dequeues*. In Proceedings of the ACM SIGPLAN 2001 Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.
- Manuel M.T. Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 13.1.2, Ausdrücke formatieren; Kapitel 13.2.1, Formatieren und Auswerten in erweiterter Version)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 5, Writing a Library: Working with JSON Data – Pretty Printing a String, Fleshing Out the Pretty-Printing Library)

Chapter 18

Functional Reactive Programming

Chapter 18.1

Motivation

Hybrid Systems

...are systems composed of

- continuous
- discrete

components.

Mobile Robots

...are special hybrid systems (or cyber-physical systems) from both a physical and logical perspective:

- Physically
  - Continuous components: Voltage-controlled motors, batteries, range finders,...
  - Discrete components: Microprocessors, bumper switches, digital communication,...
- Logically
  - Continuous notions: Wheel speed, orientation, distance from a wall,...
  - Discrete notions: Running into another object, receiving a message, achieving a goal,...

In this chapter

...designing and implementing two

- imperative-style languages for controlling robots

Beyond the concrete application, this provides two examples of

- domain specific language (DSL)

and an application of the type constructor classes

- Monad
- Arrow
- Functor

Note, the languages aim at simulating robots in order to allow running simulations at home without having to buy (possibly expensive) robots first.

Reading

...for Chapter 18.2 (using monads):

- Paul Hudak. The Haskell School of Expression – Learning Functional Programming through Multimedia. Cambridge University Press, 2000. (Chapter 19, An Imperative Robot Language)

...for Chapter 18.3 (using arrows):

- Paul Hudak, Antony Courtney, Herik Nilsson, John Peterson. Arrows, Robots, and Functional Reactive Programming. Summer School on Advanced Functional Programming 2002, Springer-V., LNCS 2638, 159-187, 2003.

Note: Chapter 18.2 and 18.3 are independent and do not build upon each other.

Chapter 18.2

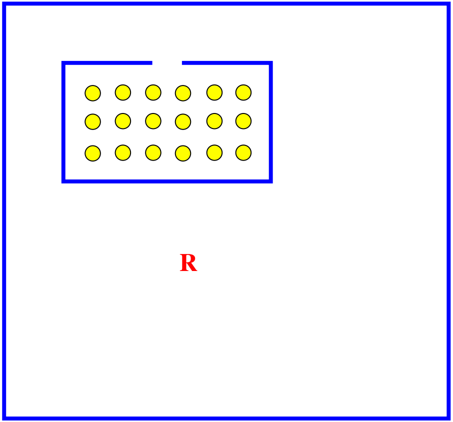
An Imperative Robot Language

Chapter 18.2.1

The Robot's World

The Robot's World

...a two-dimensional grid surrounded by walls, with rooms having doors, and gold coins as treasures!



In more detail

...the robot's world is

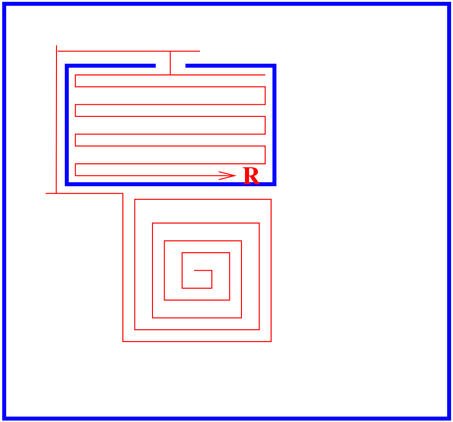
- a finite two-dimensional grid of square form
  - equipped with walls
  - possibly forming rooms, possibly having doors
  - with gold coins placed on some grid points

The preceding example shows

- a robot's world with one room, an open door, full of gold: Eldorado!
- a robot sitting in the centre of the world ready for exploring it!

The Robot's Mission

...exploring the world, collecting treasures, leaving footprints!



In more detail

...the robot's mission is

- to explore its world, to collect the treasures in it, and to leave footprints of its exploration, i.e., to
  - strolling and searching through its world, e.g., following the path way of an outward-oriented spiral.
  - picking up the gold coins it finds on its way and saving them in its pocket.
  - dropping gold coins at some (other) grid points.
  - marking its way with differently colored pens.

Objective

...developing an imperative-like robot language allowing to write programs, which advise a robot how to explore and shape its world!

E.g., programs such as:

(1) drawSquare =

```
drawSquare =
    do penDown
       move
       turnRight
       move
       turnRight
       move
       turnRight
       move
```

(2) moveToWall =

```
moveToWall =
  while (isnt blocked)
   do move
```

(3) getCoinsToWall =

```
getCoinsToWall =
  while (isnt blocked) $
   do move
      checkAndPickCoin
```

In more detail

...assuming that Robot is a monad:

```
newtype Robot a = Rob...
instance Monad Robot where...

drawSquare =
 do penDown     (penDown :: Robot () / pen ready to write)
    move        (move :: Robot () / moving one space for-
    turnRight                          ward)
    move
    turnRight   (turnRight: Robot () / turn 90 degrees
    move                                   clock-wise)
    turnRight
    move
```

Note, for the robot monad, operation (>>) is relevant!

The Implementation Environment

...required modules:

```
module Robot where
 import Array
 import List
 import Monad
 import SOEGraphics
 import Win32Misc (timeGetTime)
 import qualified GraphicsWindows as GW (getEvent)
```

Note:

- Graphics, SOEGraphics are two commonly used graphics libraries being Windows compatible.
- Double-check the SOE homepage at haskell.org/soe regarding the availability of the modules SOEGraphics and GraphicsWindows.

Chapter 18.2.2

Modelling the Robot's World

Modelling the World

...the robots live and act in a 2-dimensional grid.

Positions are given by their x and y coordinates:

```
type Position = (Int,Int)
```

Directions a robot can face or head to:

```
data Direction = North | East | South | West
                      deriving (Eq, Show, Enum)
```

World, a two-dimensional grid as `Array`-type:

```
type Grid = Array Position [Direction]
```

Chapter 18.2.3

Modelling Robots

Modelling Robots

...by their internal states, which are characterized by 6 values:

1. Robot position
2. Robot orientation
3. Pen status (up or down)
4. Pen color
5. Treasure map
6. Number of coins in the robot's pocket

Note, the grid does not change and is thus not part of a robot (state).

Modelling Internal Robot States

...as an algebraic product type:

```
data RobotState = RState { position :: Position
                         , facing :: Direction
                         , pen :: Bool
                         , color :: Color
                         , treasure :: [Position]
                         , pocket :: Int
                         } deriving Show
```

where the number of coins at a position is given by the number of its occurrences in `treasure`, and `Color` defines the set of possible pen colors:

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
            deriving (Eq, Ord, Bounded, Enum,
                      Ix, Show, Read)
```

Note

...the definition of RobotState takes advantage of Haskell's field-label (or record) syntax: The field labels (position, facing, pen, color, treasure, pocket) offer

- access to state components by names instead of position without requiring specific selector functions.

This advantage would have been lost defining robot states equivalently but without field-label syntax as in:

```
data RobotState = RState
                   Position
                   Direction
                   Bool
                   Color
                   [Position]
                   Int deriving Show
```

Illustrating Field-label Syntax Usage (1)

...generating, modifying, and accessing values of robot-state components.

Example 1: Generating field values

The definition

```
s1 = RState { position = (0,0)
            , facing   = East
            , pen      = True
            , color    = Green
            , treasure = [(2,3),(7,9),(12,42)]
            , pocket   = 2
            } :: RobotState
```

is equivalent to:

```
s2 = RState (0,0) East True Green
            [(2,3),(7,9),(12,42)] 2 :: RobotState
```

Illustrating Field-label Syntax Usage (2)

Example 2: Modifying field values

```
s3 = s2 { position = (22,43), pen = False }
 ->> RState { position = (22,43)
            , facing = East
            , pen = False
            , color = Green
            , treasure = [(2,3),(7,9),(12,42)]
            , pocket = 2
            } :: RobotState
```

Example 3: Accessing field values

```
position s1  ->>  (0,0)
treasure s3  ->>  [(2,3),(7,9),(12,42)]
color    s3  ->>  Green
```

Example 4: Using field names in patterns

```
jump (RState { position = (x,y) }) = (x+2,y+1)
```

Benefits and Advantages

...of using field-label syntax:

- It is more 'informative' (due to field names).
- The order of fields gets irrelevant, e.g., the definition of:

```
s4 = RState { position = (0,0)
            , pocket   = 2
            , pen      = True
            , color    = Green
            , treasure = [(2,3),(7,9),(12,42)]
            , facing   = East
            } :: RobotState
```

  is equivalent to the robot state defined by s1.

Chapter 18.2.4

Modelling Robot Commands as State Monad

Modelling Robot Commands

...by Robot, a 1-ary type constructor, defined by:

```
newtype Robot a =
 Rob (RobotState -> Grid -> Window
                              -> IO (RobotState,a))
```

allows making Robot an instance of type class Monad (matching the pattern of a state monad by concepually considering the Grid argument part of the state):

```
instance Monad Robot where
 Rob sf0 >>= f = Rob $ \s0 g w ->
                          do (s1,a1) <- sf0 s0 g w
                             let Rob sf1 = f a1
                             (s2,a2) <- sf1 s1 g w
                             return (s2,a2)
 return a       = Rob (\s _ _ -> return (s,a))
```

Note

- $ can be replaced by parentheses:

```
instance Monad Robot where
 Rob sf0 >>= f = Rob (\s0 g w ->
                       do (s1,a1) <- sf0 s0 g w
                          let Rob sf1 = f a1
                          (s2,a2) <- sf1 s1 g w
                          return (s2,a2))
 return a      = Rob (\s _ _ -> return (s,a))
```

- the Grid argument in

```
newtype Robot a =
 Rob (RobotState -> Grid -> Window
                             -> IO (RobotState,a))
```

  can conceptually be considered a 'read- only' part of a robot state; the Window argument allows specifying the window, in which the graphics is displayed.

Chapter 18.2.5

The Imperative Robot Language

IRL: The Imperative Robot Language

Key insight:

- Taking state as input
- Possibly querying the state in some way
- Returning a possibly modified state

...reveals the imperative nature of IRL commands.

Utility Functions

...not intended (except of at) for direct usage by an IRL programmer.

- Direction commands:

```
right, left :: Direction -> Direction
right d = toEnum (succ (mod (fromEnum d) 4))
left d  = toEnum (pred (mod (fromEnum d) 4))

at :: Grid -> Position -> [Direction]
at = (!)
```

- Supporting functions for updating and querying states:

```
updateState :: (RobotState -> RobotState)
                                         -> Robot ()
updateState u = Rob (\s _ _ -> return (u s, ()))

queryState :: (RobotState -> a) -> Robot a
queryState q = Rob (\s _ _ -> return (s, q s))
```

Recalling the Definition of Type Class Enum

...of the Standard Prelude:

```
class Enum a where
 succ, pred     :: a -> a
 toEnum         :: Int -> a
 fromEnum       :: a -> Int
 enumFrom       :: a -> [a]              -- [n..]
 enumFromThen   :: a -> a -> [a]         -- [n,n'..]
 enumFromTo     :: a -> a -> [a]         -- [n..m]
 enumFromThenTo :: a -> a -> a -> [a]    -- [n,n'..m]

 succ                 = toEnum . (+1) . fromEnum
 pred                 = toEnum . (subtract 1) . fromEnum
 enumFrom x           = map toEnum [fromEnum x..]
 enumFromThen x y     = map toEnum [fromEnum x, fromEnum y..]
 enumFromTo x y       = map toEnum [fromEnum x..fromEnum y]
 enumFromThenTo x y z = map toEnum [fromEnum x,
                                    fromEnum y..fromEnum z]

 toEnum, fromEnum =    ...implementation is type-dependent
```

Recalling the Usage of Type Class Enum

The following 'equalities' hold:

`enumFrom n`	$\widehat{=}$	`[n..]`
`enumFromThen n n'`	$\widehat{=}$	`[n,n'..]`
`enumFromTo n m`	$\widehat{=}$	`[n..m]`
`enumFromThenTo n n' m`	$\widehat{=}$	`[n,n'..m]`

Example:

```
data Color = Red | Orange | Yellow | Green
             | Blue | Indigo | Violet deriving Enum

[Red..Green]      ->>  [Red, Orange, Yellow, Green]
[Red, Yellow..]   ->>  [Red, Yellow, Blue, Violet]
fromEnum Blue     ->>  4
toEnum 3          ->>  Green
```

IRL Commands for Robot Orientation

...by updating the internal robot state.

- Turn right:

```
turnLeft :: Robot ()
turnLeft =
 updateState (\s -> s {facing = left (facing s)})
```

- Turn left:

```
turnRight :: Robot ()
turnRight =
 updateState (\s -> s {facing = right (facing s)})
```

- Turn to:

```
turnTo :: Direction -> Robot ()
turnTo d = updateState (\s -> s {facing = d})
```

- Facing what direction?

```
direction :: Robot Direction
direction =  queryState facing
```

IRL Command for Blockade Checking

- Motion blocked in direction currently facing?

```
blocked :: Robot Bool
blocked =
 Rob $ \s g _ ->
  return (s, facing s `notElem` (g `at` position s))
```

with `notElem` from the Standard Prelude.

IRL Commands for Motion

- Moving forward one space if not blocked:

```
move :: Robot ()
move =
 cond1 (isnt blocked)
  (Rob $ \s _ w -> do
    let newPos = movePos (position s) (facing s)
    graphicsMove w s newPos
    return (s {position = newPos}, ())
   )
```

- Moving forward one space in direction of:

```
movePos :: Position -> Direction -> Position
movePos (x,y) d = case d of North -> (x,y+1)
                            South -> (x,y-1)
                            East  -> (x+1,y)
                            West  -> (x-1,y)
```

IRL Commands for Pen Usage

- Choose pen color for writing:

```
setPenColor :: Color -> Robot ()
setPenColor c = updateState (\s -> s {color = c})
```

- Pen down to start writing:

```
penDown :: Robot ()
penDown = updateState (\s -> s {pen = True})
```

- Pen up to stop writing:

```
penUp :: Robot ()
penUp = updateState (\s -> s {pen = False})
```

IRL Commands for Coin Handling (1)

- At position with coin according to treasure map?

```
onCoin :: Robot Bool
onCoin = queryState (\s ->
                position s `elem` treasure s)
```

- Pick coin:

```
pickCoin :: Robot ()
pickCoin =
 cond1 onCoin
  (Robot $ \s _ w ->
     do eraseCoin w (position s)
        return (s {treasure =
                    position s `delete` treasure s,
                   pocket = pocket s+1}, ())
  )
```

IRL Commands for Coin Handling (2)

- How many coins currently in pocket?

```
coins :: Robot Int
coins = queryState pocket
```

- Drop coin, if there is at least one in the pocket:

```
dropCoin :: Robot ()
dropCoin =
 cond1 (coins >* return 0)
  (Robot $ \s _ w ->
     do drawCoin w (position s)
        return (s {treasure =
                     position s : treasure s,
                  pocket = pocket s-1}, ())
  )
```

Utility Functions for Logic and Control (1)

- Conditionally performing commands:

```
cond :: Robot Bool -> Robot a
                -> Robot a -> Robot a
cond p c a = do pred <- p
                if pred then c else a

cond1 p c = cond p c (return ())
```

- Performing commands while some condition is met:

```
while :: Robot Bool -> Robot () -> Robot ()
while p b = cond1 p (b >> while p b)
```

- Connecting commands 'disjunctively:'

```
(||*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 ||* b2 = do p <- b1
               if p then return True
                    else b2
```

Utility Functions for Logic and Control (2)

- Connecting commands ‘conjunctively:’

```
(&&*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 &&* b2 = do p <- b1
               if p then b2
                    else return False
```

- Lifting negation to commands:

```
isnt :: Robot Bool -> Robot Bool
isnt =  liftM not
```

- Lifting comparisons to commands:

```
(>*) :: Robot Int -> Robot Int -> Robot Bool
(>*) =  liftM2 (>)

(<*) :: Robot Int -> Robot Int -> Robot Bool
(<*) =  liftM2 (<)
```

Recalling the Definitions of the Lift Operators

...the higher-order lift operations `liftM` and `liftM2` are defined in the library `Monad` (as well as `liftM3`, `liftM4`, and `liftM5`):

```
liftM :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f =  \a -> do a′ <- a
                    return (f a′)

liftM2 :: (Monad m) => (a -> b -> c)
                             -> (m a -> m b -> m c)
liftM2 f =  \a b -> do a′ <- a
                       b′ <- b
                       return (f a′ b′)
```

Note

The implementations of

- `isnt`, `(>*)`, and `(<*)` are based on `liftM` and `liftM2`, thereby avoiding the usage of special lift functions.
- `(||*)` and `(&&*)` are not based on `liftM2`, thereby avoiding (unnecessary) strictness in their second arguments.

Illustrating the Usage of cond and cond1

...moving the robot one space forward if it is not blocked; moving it one space to the right if it is.

An implementation using

- cond:

```
evade :: Robot ()
evade = cond blocked
             (do turnRight
                 move)
             move
```

- cond1:

```
evade′ :: Robot ()
evade′ = do cond1 blocked turnRight
            move
```

Moving in a Spiral

...an example of an advanced IRL program:

```
spiral :: Robot ()
spiral = penDonw >> loop 1
 where loop n =
        let twice = do turnRight
                       moven n
                       turnRight
                       moven n
        in con blocked
            (twice >> turnRight >> moven n)
            (twice >> loop (n+1))

moven :: Int -> Robot ()
moven n = mapM . (const move) [1..]
```

Chapter 18.2.6

Defining a Robot's World

The Robot's World: Preliminary Definitions

The robots' world is a grid of type `Array`:

```
type Grid = Array Position [Direction]
```

Grid points can be accesssed using:

```
at :: Grid -> Position -> [Direction]
at =  (!)
```

Defining the Initial World g0 (1)

The size of the initial grid world g0 is given by:

```
size :: Int
size =  20
```

with the grid world's

- centre at: `(0,0)`
- corners at: `(-size,size)` `(size,size)` `((-size),(-size))` `(size,(-size))`

Defining the Initial World g0 (2)

..inner, border, and corner points of world g0 are characterized by the directions of motion they allow:

- Inner points of g0 allow moving toward:
```
interior = [North, South, East, West]
```
- Border points at the north, east, south, and west border allow moving toward:
```
nb  = [South, East, West]       (nb: north border)
eb  = [North, South, West]
sb  = [North, East, West]
wb  = [North, South, East]      (wb: west border)
```
- Corner points at the northwest, northeast, southeast, and southwest corner allow moving toward:
```
nwc = [South, East]      (nwc: northwest corner)
nec = [South, West]
sec = [North, West]
swc = [North, East]      (swc: southwest corner)
```

Defining the Initial World g0 (3)

...all grid points, i.e., inner and border grid points can thus be enumerated using list comprehension, which allows to define the initial world grid g0 as follows:

```
g0 :: Grid
g0 = array ((-size, -size), (size, size))
      ([((i, size), nb)   | i <- r ] ++
      ([((i, -size), sb)  | i <- r ] ++
      ([((size, i), eb)   | i <- r ] ++
      ([((-size, i), wb)  | i <- r ] ++
      ([((size, i), eb)   | i <- r ] ++
      ([((i,j), interior) | i <- r, j <- r ] ++
      ([((size, size), nec), ((size, -size), sec),
        ((-size, size), nwc),
        ((-size, -size), swc)])
    where r = [1-size..size-1]
```

Building World g1 from World g0

...by erecting a west/east-oriented wall leading from (-5,10) to (5,10):

```
g1 :: Grid
g1 =  g0 // mkHorWall (-5) 5 10
```

where (//) is the Array library function (cf. Chapter 7.2):

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

Recalling the (//) Function

...of the Array library:

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

and illustrating its usage: To this end, let:

```
colors :: Array Int Color
colors = array (0,7)
          [(0,Black),(1,Blue),(2,Green),(3,Cyan),
           (4,Red),(5,Magenta),(6,Yellow),
           (7,White)]
```

then:

```
colors // [(0,White),(7,Black)]
 ->> array (0,7) [(0,White),(1,Blue),(2,Green),(3,Cyan),
                  (4,Red),(5,Magenta),(6,Yellow),
                  (7,Black)] :: Array Int Color
```

swaps the 'black' und 'white' entries in colors.

Note

Type Color is defined as in the

- Graphics library:

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
            deriving (Eq, Ord, Bounded, Enum,
                      Ix, Show, Read)
```

Equivalently but more concisely we could have defined

- colors by:

```
colors :: Array Int Color
colors = array (0,7) (zip [0..7] [Black..White])
```

Utility Functions for Building Walls

Building walls horizontally (west/east-oriented, leading from (x1,y) to (x2,y)):

```
mkHorWall :: Int -> Int -> Int -> [(Position,[Direction])]
mkHorWall x1 x2 y =
 [((x,y), nb) | x <- [x1..x2]] ++
    [((x,y+1), sb) | x <- [x1..x2]]
```

Building walls vertically (north/south-oriented, leading from (x,y1) to (x,y2)):

```
mkVerWall :: Int -> Int -> Int -> [(Position,[Direction])]
mkVerWall y1 y2 x =
 [((x,y), eb) | y <- [y1..y2]] ++
    [((x+1,y), wb) | y <- [y1..y2]]
```

Utility Functions for Building Rooms

...naively, rooms could be built using `mkHorWall` and `mkVerWall` straightforwardly:

```
mkBox :: Position -> Position
                  -> [(Position, [Direction])]
mkBox (x1, y1) (x2, y2) =
 mkHorWall (x1+1) x2 y1 ++ mkHorWall (x1+1) x2 y2 ++
 mkVerWall (y1+1) y2 x1 ++ mkVerWall (y1+1) y2 x2
```

This, however, creates two field entries for each of the four inner corners causing their values undefined after the call is finished (cf. Chapter 7.2).

This problem can elegantly be overcome by using the `Array` library operation `accum` (cf. Chapter 7.2) in combination with `mkBox`.

Recalling the accum Function

...of the `Array` library:

```
accum :: (Ix a) => (b -> c -> b)
                -> Array a b -> [(a,c)] -> Array a b
```

As discussed in Chapter 7.2, `accum`

- is quite similar to `(//)`.
- in case of replicated entries the function of the first argument is applied for resolving conflicts.
- the `intersect` function of the `List` library is appropriate for this in the case of our example, e.g.:

  ```
  [South, East, West] `intersect`
      [North, South, West] ->> [South, West]
  ```

  represents the northeast corner.

Building World g2 from World g0

...by building a room with its lower left and upper right corner at positions (-10,5) and (-5,10), respectively:

```
g2 :: Grid
g2 = accum intersect g0 (mkBox (-15,8) (2,17))
```

using `accum`, `intersect`, and `mkBox`.

Building World g3 from World g2

...by adding a door (to the middle of the top wall of the room)

```
g3 :: Grid
g3 = accum union g2 [((-7,17), interior),
                     ((-7,18), interior)]
```

using accum, union, and interior.

Chapter 18.2.7

Robot Graphics: Animation in Action

Objective of Animation

...drawing the world the robot lives in and then showing the robot running around (at some predetermined rate) accomplishing its mission:

- Drawing lines if the pen is down.
- Picking up coins.
- Dropping coins, letting them thereby appear in possibly other locations.

This requires to incrementally update the drawn and displayed graphics, which will be achieved by means of the operations of the `Graphics` library.

Updating the Graphics Incrementally

...key for incrementally updating the displayed world the Graphics library operation drawInWindowNow:

```
drawInWindowNow :: Window -> Color
                      -> Point -> Point -> IO ()
```

which draws the updated graphics immediately after any changes, and can be used, e.g., for drawing lines:

```
drawLine :: Window -> Color
                      -> Point -> Point -> IO ()
drawLine w c p1 p2 =
 drawInWindowNow w (withColor c (line p1 p2))
```

Note

...in order to work properly, the incremental update of the world must be organized such that the

- absence of interferences of graphics actions

is ensured.

This is achieved by assuming:

1. Grid points are 10 pixels apart.
2. Walls are drawn halfway between grid points.
3. The robot pen draws lines directly from one grid point to the next.
4. Coins are drawn as yellow circles just above and to to the left of each grid point.
5. Coins are erased by drawing black circles over the yellow ones which are already there.

Defining Top-level Constants

...for dealing with the preceding assumptions.

Half the distance between grid points:

```
d :: Int
d = 5
```

Color of walls and coins:

```
wc, cc :: Color
wc = Blue
cc = Yellow
```

Window size:

```
xWin, yWin :: Int
xWin = 600
yWin = 500
```

Defining Utility Functions (1)

Drawing grids:

```
drawGrid :: Window -> Grid -> IO ()
drawGrid w wld =
 let (low@(xMin,yMin),hi@(xMax,yMax)) = bounds wld
     (x1,y1)                          = trans low
     (x2,y2)                          = trans hi
 in
  do drawLine w wc (x1-d,y1+d) (x1-d,y2-d)
     drawLine w wc (x1-d,y1+d) (x1+d,y2+d)
     sequence_ [drawPos w (trans (x,y)) (wld `at` (x,y))
               | x <- [xMin..xMax], y <- [yMin..yMax]]
```

Defining Utility Functions (2)

Used by drawGrid:

```
drawPos :: Window -> Point -> [Direction] -> IO ()
drawPos x (x,y) ds =
 do if North `notElem` ds
       then drawLine w wc (x-d,y-d) (x+d,y-d)
       else return ()
    if East `notElem` ds
       then  drawLine w wc (x+d,y-d) (x+d,y+d)
       else return ()
```

Used by drawGrid, from the Array library:

```
bounds :: Ix a => Array a b -> (a,a)
 -- yields the bounds of its array argument
```

Defining Utility Functions (3)

Dropping and drawing coins:

```
drawCoins :: Window -> RobotState -> IO ()
drawCoins w s = mapM_ (drawCoin w) (treasure s)

drawCoin :: Window -> Position -> IO ()
drawCoin w p =
 let (x,y) = trans p
 in drawInWindowNow w
    (withColor cc (ellipse (x-5,y-1) (x-1,y-5)))
```

Erasing coins:

```
eraseCoin :: Window -> Position -> IO ()
eraseCoin w p =
 let (x,y) = trans p
 in drawInWindowNow w
     (withColor Black (ellipse (x-5,y-1) (x-1,y-5)))
```

Defining Utility Functions (4)

Drawing robot moves:

```
graphicsMove :: Window -> RobotState
                              -> Position -> IO ()
graphicsMove w s newPos =
 do if pen s
       then drawLine w (color s) (trans (position s))
                                 (trans newPos)
       else return ()
    getWindowTick w

trans       :: Position -> Point
trans (x,y) = (div xWin 2+2*d*x, div yWin 2-2*d*y)
```

Causing a short delay after each robot move

```
getWindowTick :: Window -> IO ()
```

Running IRL Programs: The Top-level Prg. (1)

...putting it all together.

Running an IRL program:

```
runRobot :: Robot () -> RobotState -> Grid -> IO ()
runRobot (Robot sf) s g =
 runGraphics $
 do w <- openWindowEx "Robot World" (Just (0,0))
          (Just (xWin, yWin)) drawGraphic (Just 10)
    drawGrid w g
    drawCoins w s
    spaceWait w
    sf s g w
    spaceClose w
```

Running IRL Programs: The Top-level Prg. (2)

Intuitively, runRobot

- opens a window
- draws a grid
- draws the coins
- waits for the user to hit the spacebar
- continues running the program with starting state s and grid g
- closes the window when execution is complete and the spacebar is pressed again.

where spaceWait provides the user with progress control by awaiting the user's pressing the spacebar:

```
spaceWait   :: Window -> IO ()
spaceWait w = do k <- getKey w
                 if k == ' ' then return ()
                             else spaceWait w
```

Animation in Action (1)

...the grids g0 through g3 can now be used to run IRL programs with.

1) Fixing s0 as a suitable starting state:

```
s0 :: RobotState
s0 = RobotState { position = (0,0)
                , pen = False
                , color = Red
                , facing = North
                , treasure = tr
                , pocket = 0
                }
```

2) Placing 'treasure' (all coins are placed inside the room in grid g3):

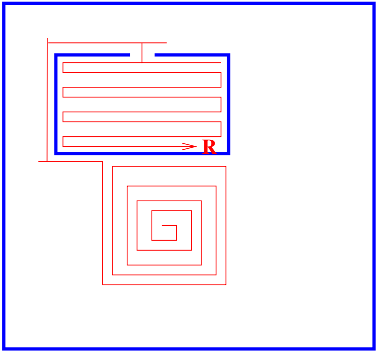
```
tr :: [Position]
tr = [(x,y) | x <- [-13,-11..1], y <- [9,11..15]]
```

Animation in Action (2)

3) Running the ‘spiral’ program with s0, g0:

```
main = runRobot spiral s0 g0
```

...leads to the ‘spiral’ example shown for illustration at the beginning of this chapter:



Chapter 18.3

Robots on Wheels

Outline

...we consider and define a simulation of

- mobile robots (called Simbots)

using functional reactive programming.

The implementation will make use of the type class

- `Arrow`

which is another example of a type constructor class generalizing the concept of a monad.

Chapter 18.3.1

The Setting

The Configuration of Mobile Robots (1)

...is assumed to be as follows:

"Robots are differential drive robots having two wheels that are each driven by an independent motor. The relative velocity of these two wheels governs the turning rate of the robot. If the velocities are identical, the robot will go straight.

A robot has several kinds of sensors. Among these, (1) a bumper switch to detect when the robot gets 'stuck' because of being blocked by something, (2) a range finder to determine the nearest object in any given direction (in the following it is assumed that there are four independent range finders that only look forward, backward, left and right; the range finder will thus only be queried at these four angles), (4) an animate object tracker that gives the current position of all other robots and possibly those of some free-moving balls that are within a certain distance from the robot.

The Configuration of Mobile Robots (2)

This object tracker can be thought of as *modelling either a visual subsystem that can 'see' these objects, or a communication subsystem through which the robots and balls share each other's coordinates. Some further capabilities will be introduced as need occurs.*

Last but not least, each robot has a unique ID."

The Application Scenario: Robot Soccer

...the overall task:

"Write a program to play 'robocup soccer' as follows:

Use wall segments to create two goals at either end of the field.

Decide on a number of players on each team and write generic controllers, such as one for a goalkeeper, one for attack, and one for defense.

Create an initial world where the ball is at the center mark, and each of the players is positioned strategically while being on-side (with the defensive players also outside of the center circle. Each team may use the same controller, or different ones."

Code for 'Robots on Wheels'

...can be down-loaded at the Yampa homepage at

http://www.haskell.org/yampa

In the following we highlight essential code snippets.

Chapter 18.3.2

Modelling the Robots' World

Signal Functions, Signals, and Simbots

Signal functions are

- signal transformers, i.e., functions mapping signals to signals,
- of type `SF`, a 2-ary type constructor defined in Yampa, which is an instance of type constructor class `Arrow`.

Yampa provides

- a number of primitive signal functions and a set of special composition operators (or combinators) for constructing (more) complex signal functions from simpler ones.

Signals are no

- first-class values in Yampa but can only be manipulated by means of signal functions to avoid time- and space-leaks (abstract data type).

Simbot is a short hand for simulated robot.

Modelling Time, Signals, and Signal Functions

SF is an instance of class Arrow:

```
type Time       = Double
type Signal a˜  = Time -> a
type SF a b     = Signal a -> Signal b
```

Intuitively: SF-values are signal transformers resp. signal functions (thus the type name SF).

Modelling Simbots

```
type RobotType = String
type RobotId   = Int

type SimbotController =
      SimbotProperties -> SF SimbotInput SimbotOutput

Class HasRobotProperties i where
 rpType     :: i -> RobotType    -- Type of robot
 rpId       :: i -> RobotId      -- Identity of robot
 rpDiameter :: i -> Length       -- Distance between wheels
 rpAccMax   :: i -> Acceleration -- Max translational acc
 rpWSMax    :: i -> Speed        -- Max wheel speed
```

Modelling the World

```haskell
type WorldTemplate = [ObjectTemplate]

data ObjectTemplate =
    OTBlock    otPos  :: Position2  -- Square obstacle
  | OTVWall    otPos  :: Position2  -- Vertical wall
  | OTHWall    otPos  :: Position2  -- Horizontal wall
  | OTBall     otPos  :: Position2  -- Ball
  | OTSimbotA  otRId  :: RobotId,   -- Simbot A robot
               otPos  :: Position2,
               otHdng :: Heading
  | OTSimbotB  otRId  :: RobotId,   -- Simbot B robot
               otPos  :: Position2,
               otHdng :: Heading
```

Chapter 18.3.3

Classes of Robots

Types of Robots

...usually, there are different types of robots

- differing in their features (2 wheels, 3 wheels, camera, sonar, speaker, blinker, etc.)

The type of a robot is fixed by its

- input and output types

which are encoded in input and output classes together with the functions operating on the class elements.

Input Classes (1)

...and functions operating on their elements:

```
data BatteryStatus = BSHigh | BSLow | BSCritical
                     deriving (Eq, Show)

class HasRobotStatus i where
 -- Current battery status
 rsBattStat :: i -> BatteryStatus
 -- Currently stuck or not stuck
 rsIsStuck  :: i -> Bool

-- Derived event sources:
rsBattStatChanged  :: HasRobotStatus i =>
                          SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i => SF i (Event ())
rsBattStatCritical :: HasRobotStatus i => SF i (Event ())
rsStuck            :: HasRobotStatus i => SF i (Event ())
```

Input Classes (2)

```haskell
class HasOdometry where
 -- Current position
 odometryPosition :: i -> Position2
 -- Current heading
 odometryHeading  :: i -> Heading

class HasRangeFinder i where
 rfRange    :: i -> Angle -> Distance
 rfMaxRange :: i -> Distance

-- Derived range finders:
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance
```

Input Classes (3)

```
class HasAnimateObjectTracker i where
  aotOtherRobots :: i -> [(RobotType, Angle, Distance)]
  aotBalls       :: i -> [(Angle, Distance)]

class HasTextualConsoleInput i where
  tciKey :: i -> Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i =>
                   Maybe Char -> SF i (Event Char)
tciKeyDown    :: HasTextualConsoleInput i =>
                   SF i (Event Char)
```

Output Classes

...and functions operating on their elements:

```
class MergeableRecord o => HasDiffDrive o where
 -- Brake both wheels
 ddBrake :: MR o
 -- Set wheel velocities
 ddVelDiff :: Velocity -> Velocity -> MR o
 -- Set velocities and rotation
 ddVelTR :: Velocity -> RotVel -> MR o

class MergeableRecord o =>
  HasTextConsoleOutput o where
    tcoPrintMessage :: Event String -> MR o
```

Chapter 18.3.4

Robot Simulation in Action

Typical Structure of a Robot Control Program

```
module MyRobotShow where

  import AFrob
  import AFrobRobotSim

  main :: IO ()
  main = runSim (Just world) rcA rcB

  world :: WorldTemplate
  world = ...

  -- controller for simbot A
  rcA :: SimbotController
  rcA = ...

  -- controller for simbot B
  rcB :: SimbotController
  rcB = ...
```

Robot Simulation in Action

Running a robot simulation:

```
runSim :: Maybe WorldTemplate
                   -> SimbotController
                        -> SimbotController -> IO ()
```

Simbot controllers:

```
rcA :: SimbotController
rcA rProps =
  case rrpId rProps of
    1 -> rcA1 rProps
    2 -> rcA2 rProps
    3 -> rcA3 rProps
rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
```

Chapter 18.3.5

Examples

Robot Actions: Control Programs (1)

A stationary robot:

```
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)
```

A blind robot moving at constant speed:

```
rcBlind1 _ =
 constant (mrFinalize $ ddVelDiff 10 10)
```

A blind robot moving at half the maximum speed:

```
rcBlind2 rps =
 let max = rpWSMax rps
 in constant (mrFinalize $
                  ddVelDiff (max/2) (max/2))
```

Robot Actions: Control Programs (2)

A robot rotating at a pre-given speed:

```
rcTurn :: Velocity -> SimbotController
rcTurn vel rps =
  let vMax = rpWSMax rps
      rMax = 2 * (vMax - vel) / rpDiameter rps
  in constant (mrFinalize $ ddVelTR vel rMax)
```

Chapter 18.4

In Conclusion

The Origins

...of functional reactive programming (FRP) can be traced back to functional reactive animation (FRAn):

- Conal Elliot, Paul Hudak. Functional Reactive Animation. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.
- Conal Elliot. Functional Implementations of Continuous Modeled Animation. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.

Seminal Works

...on functional reactive programming (FRP):

- Zhanyong Wan, Paul Hudak. Functional Reactive Programming from First Principles. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Languages Design and Implementation (PLDI 2000), ACM Press, 2000.
- John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. Yale FRP User's Manual. Department of Computer Science, Yale University, January 2001. http://www.haskell.org/frp/manual.html
- Henrik Nilsson, Antony Courtney, John Peterson. Functional Reactive Programming, Continued. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.

Applications of FRP (1)

...on Functional Reactive Robotics (FRob):

- Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.
- John Peterson, Gregory Hager, Paul Hudak. A Language for Declarative Robotic Programming. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.

Applications of FRP (2)

...on Functional Animation Languages (FAL):

- Paul Hudak. The Haskell School of Expression – Learning Functional Programming through Multimedia. Cambridge University Press, 2000. (Chapter 15, A Module of Reactive Animations)

...on Functional Vision Systems (FVision):

- Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. Prototyping Real-Time Vision Systems: An Experiment in DSL Design. In Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), 484-493, 1999.

...on Functional Reactive User Interfaces (FRUIt):

- Antony Courtney, Conal Elliot. Genuinely Functional User Interfaces. In Proceedings of the 2001 Haskell Workshop, September 2001.

Applications of FRP (3)

...towards Real-Time FRP (RT-FRP):

- Zhanyong Wan, Walid Taha, Paul Hudak. Real-Time FRP. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
- Zhanyong Wan. Functional Reactive Programming for Real-Time Embedded Systems. PhD thesis. Department of Computer Science, Yale University, December 2002.

...towards Event-Driven FRP (ED-FRP):

- Zhanyong Wan, Walid Taha, Paul Hudak. Event-Driven FRP. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.

Chapter 18.5

References, Further Reading

Chapter 18: Basic Reading

- Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 19, An Imperative Robot Language)
- Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.), *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.
- Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. *Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages*. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.

Chapter 18: Selected Advanced Reading (1)

- Zhanyong Wan, Paul Hudak. *Functional Reactive Programming from First Principles*. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), 242-252, 2000.
- Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.
- Zhanyong Wan, Walid Taha, Paul Hudak. *Real-Time FRP*. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.

Chapter 18: Selected Advanced Reading (2)

- Zhanyong Wan, Walid Taha, Paul Hudak. *Event-Driven FRP*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.
- John Peterson, Gregory D. Hager, Paul Hudak. *A Language for Declarative Robotic Programming*. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.
- John Peterson, Paul Hudak, Conal Elliot. *Lambda in Motion: Controlling Robots with Haskell*. In Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), Springer-V., LNCS 1551, 91-105, 1999.

Chapter 18: Selected Advanced Reading (3)

- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 16, Developing reactive functional programs)
- Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD Thesis, Department of Computer Science, Yale University, December 2002.
- Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis. Chalmers University of Technology, 1999.

Part VI

Extensions, Perspectives

Chapter 19

Extensions: Parallel and 'Real World' Functional Programming

Chapter 19.1

Parallelism in Functional Languages

Motivation, Background

...recall:

- Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4):86-90, 2009.

  ...adopting a functional programming style could make your programs more robust, more compact, and **more easily parallelizable.**

Reading for this chapter:

- Peter Pepper, Petra Hofstedt. Funktionale Programmierung, Springer-V., 2006. (In German). (Kapitel 21, Massiv Parallele Programme)

Parallelism in Programming Languages

Predominant in imperative languages:

- Libraries (PVM, MPI) $\rightsquigarrow$ Message Passing Model (C++, C, Fortran)
- Data-parallel Languages (e.g., High Performance Fortran)

Predominant in functional languages:

- Implicit (expression) parallelism
- Explicit parallelism
- Algorithmic skeletons

Implicit Parallelism

...also known as expression parallelism.

Idea: If `f(e1,...,en)` is a functional expression, then

- arguments (and functions) can be evaluated in parallel.

Most important

- advantage: Parallelism for free! No effort for the programmer at all.
- disadvantage: Results often unsatisfying; e.g. granularity, load distribution, etc., is not taken into account.

Overall, expression parallelism is

- easy to detect (for the compiler) but hard to fully exploit.

Explicit Parallelism

Idea: Introducing and using

- meta-statements (e.g., for controlling the data and load distribution, communication).

Most important

- advantage: Often very good results thanks to explicit hands-on control of the programmer.
- disadvantage: High programming effort and loss of functional elegance.

Algorithmic Skeletons

...a compromise between

- explicit imperative parallel programming
- implicit functional expression parallelism

In the following

...we consider a setting with

- massively parallel systems
- algorithmic skeletons

Massively Parallel Systems

...are typically characterized by a

- large number of processors with
  - local memory
  - communication by message exchange
- MIMD-Parallel Processor Architecture (Multiple Instruction/Multiple Data)

Here we focus and restrict ourselves to

- SPMD-Programming Style (Single Program/Multiple Data)

Algorithmic Skeletons

- represent typical patterns for parallelization (Farm, Map, Reduce, Branch&Bound, Divide&Conquer,...).
- are easy to instantiate for the programmer.
- allow parallel programming at a high level of abstraction.

Implementing Algorithmic Skeletons

...in functional languages

- by special higher-order functions.
- with parallel implementation.
- embedded in sequential languages.
- using message passing via skeleton hierarchies.

Advantages:

- Hiding of parallel implementation details in the skeleton.
- Elegance and (parallel) efficiency for special application patterns.

Example: Parallel Map on Distributed List

Consider the higher-order function map on lists:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)
```

Observation:

– Applying f to a list element does not depend on other list elements.

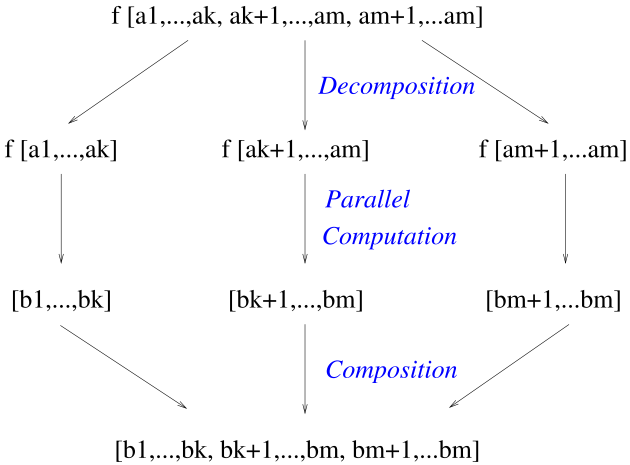
Parallelization idea:

– Divide the list into sublists followed by parallel application of map to the sublists:

$\rightsquigarrow$ parallelization pattern Farm.

Parallel Map on Distributed Lists

Illustration:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 445.

Implementing

...the parallel map function requires

- special data structures, which take into account the aspect of distribution (ordinary lists are inefficient for this purpose).

Skeletons on distributed data structures are so-called

- data-parallel skeletons.

Note the difference between:

- Data-parallelism: Supposes an a priori distribution of data on different processors.
- Task-parallelism: Processes and data to be distributed are not known a priori but dynamically generated.

Implementing a Parallel Application

...using algorithmic skeletons requires:

- Recognizing problem-inherent parallelism.
- Selecting an adequate data distribution (granularity).
- Selecting a suitable skeleton from a library.
- Instantiating the skeleton problem-specifically.

Remark:

- Some languages (e.g., Eden) support the implementation of skeletons (in addition to those which might be provided by a library).

Data Distribution on Processors

...is crucial for

- the structure of the complete algorithm.
- efficiency.

The hardness of the distribution problems depends on

- Independence of all data elements (like in the map-example): Distribution is easy.
- Independence of subsets of data elements.
- Complex dependences of data elements: Adequate distribution is challenging.

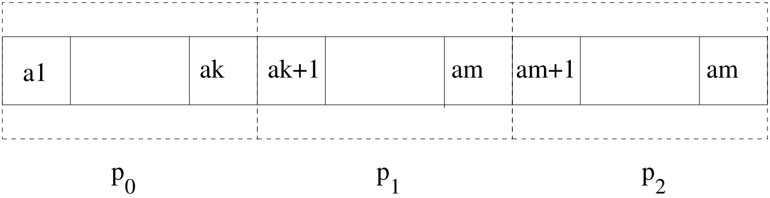
Auxiliary means: So-called covers for

- describing the decomposition and communication pattern of a data structure (investigated by various researchers).

Example (1)

...illustrating a simple list cover.

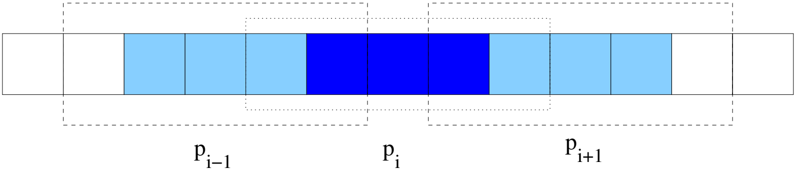
Distributing a list on three processors p_0, p_1, and p_2:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

Example (2)

...illustrating a list cover with overlapping elements.



Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer, 2006, S. 446.

General Structure of a Cover

```
Cover = {
  Type S a         -- Whole object
       C b         -- Cover
       U c         -- Local sub-objects
split :: S a -> C (U a) -- Decomposing the
                        -- original object
glue :: C (U a) -> S a  -- Composing the
                        -- original object
}
```

where it must hold: `glue . split = id`

Note: The above code snippet is not (valid) Haskell.

Implementing Covers

...requires support for

- the specification of covers.
- the programming of algorithmic skeletons on covers.
- the provision of often used skeletons in libraries.

which is currently a

- hot research topic

in functional programming.

Chapter 19.2

Haskell for 'Real World' Programming

'Real World' Haskell (1)

...Haskell these days provides considerable, mature, and stable support for:

- Systems Programming
- (Network) Client and Server Programming
- Data Base and Web Programming
- Multicore Programming
- Foreign Language Interfaces
- Graphical User Interfaces
- File I/O and filesystem programming
- Automated Testing, Error Handling, and Debugging
- Performance Analysis and Tuning
- ...

‘Real World’ Haskell (2)

This support comes mostly in terms of

- sophisticated libraries

and makes Haskell a reasonable choice for addressing and solving

- real world problems

as the choice of a language depends much on the ability and support a programming language provides for linking and connecting to the ‘outer world:’ the language’s

- eco-system.

See e.g.:

Bryan O’Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O’Reilly, 2008.

Chapter 19.3

References, Further Reading

Chapter 19.1: Basic Reading (1)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 21, Massiv Parallele Programme)
- Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilley, 2013.
- Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.
- Fethi A. Rabhi. *Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach*. In J. R. Davy, P. M. Dew (Eds.), Abstract Machine Models for Highly Parallel Computers, Oxford University Press, 118-139, 1995.

Chapter 19.1: Basic Reading (2)

- Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming 8(1):23-60, 1998.
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 11, Parallel Evaluation)
- Simon Peyton Jones, Satnam Sing. *A Tutorial on Parallel and Concurrent Programming in Haskell*. *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 267-305, 2008.

Chapter 19.1: Selected Advanced Reading (1)

- Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*. In Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 295-308, 1996.
- Robert F. Pointon, Philip W. Trinder, Hans-Wolfgang Loidl. *The Design and Implementation of Glasgow Distributed Haskell*. In Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), LNCS 2011, Springer-V., 53-70, 2000.
- Manuel M.T. Chakravarty, Roman Leshchinsky, Simon Peyton Jones, Gabriele Keller, Simon Marlow. *Data Parallel Haskell: A Status Report*. In Proceedings on the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), ACM, New York, 10-18, 2007.

Chapter 19.1: Selected Advanced Reading (2)

- Peng Li, Simon Marlow, Simon Peyton Jones, Andrew Tolmach. *Lightweight Concurrency Primitives for GHC*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007), 107-118, 2007.
- Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Pointon. *Parallel and Distributed Haskells*. Journal of Functional Programming 12(4&5):469-510, 2002.
- Martin Braun, Oleg Lobachev, Philip W. Trinder: *Arrows for Parallel Computation*. CoRR, http://arxiv.org/abs/1801.02216, 2018.

Chapter 19.1: Selected Advanced Reading (3)

- Joe Armstrong, Robert Virding, Claes Wikstrom, Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.
- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 14, Writing parallel functional programs)
- Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. Higher-Order and Symbolic Computation 16(3):203-251, 2003.

Chapter 19.2: Basic Reading (1)

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 17, Interfacing with C: The FFI; Chapter 19, Error Handling; Chapter 20, Systems Programming in Haskell; Chapter 21, Using Data Bases; Chapter 22, Extended Example: Web Client Programming; Chapter 23, GUI Programming with gtk2hs; Chapter 24, Concurrent and Multicore Programming; Chapter 27, Sockets and Syslog; Chapter 25, Profiling and Optimization; Chapter 28, Software Transactional Memory)
- Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 19, Agenten und Prozesse; Kapitel 20, Graphische Schnittstellen (GUIs))

Chapter 19.2: Basic Reading (2)

- "Haskell community." *Hackage: A Repository for Open Source Haskell Libraries*. hackage.haskell.org
- "Haskell community." *Haskell wiki*. haskell.org/haskellwiki/Applications_and_libraries
- "Haskell community." *Haskell in Industry and Open Source*. www.haskell.org/haskellwiki/Haskell_in_industry
- Hoogle, Hayoo. Useful search engines. www.haskell.org/hoogle, holumbus.fh-wedel.de/hayoo/hayoo.html

Chapter 19.2: Selected Advanced Reading(1)

- Magnus Carlsson, Thomas Hallgren. *Fudgets – A Graphical User Interface in a Lazy Functional Language*. In Proceedings of the 6th ACM International Conference on Functional Programming Languages and Computer Architecture (FPCA'93), 321-330, 1993.
- Thomas Hallgren, Magnus Carlsson. *Programming with Fudgets*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 137-182, 1995.
- Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.

Chapter 19.2: Selected Advanced Reading(1)

- Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. Journal of Functional Programming 7(2):163-182, 1997.
- Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

Chapter 20

Conclusions, Perspectives

Chapter 20.1

Research Venues, Research Topics, and More

Research Venues, Research Topics, and More

...for functional programming and functional programming languages:

- Research/publication/dissemination venues
  - Conference and Workshop Series
  - Archival Journals
  - Summer Schools
- Research Topics
- Functional Programming in the Real World

Relevant Conference and Workshop Series

For functional programming:

- Annual ACM SIGPLAN International Conference on Functional Programming (ICFP) Series, since 1996.
- Annual Symposium on Functional and Logic Programming (FLPS) Series, since 2000.
- Annual ACM SIGPLAN Haskell Workshop Series, since 2002.
- HAL Workshop Series, since 2006.

For programming in general:

- Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages and Systems (POPL), since 1973.
- Annual ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI), since 1988 (resp. 1973).

Relevant Archival Journals

For functional programming:

- Journal of Functional Programming, since 1991.

For programming in general:

- ACM Transactions on Programming Languages and Systems (TOPLAS), since 1979.
- ACM Computing Surveys, since 1969.

Summer Schools

Focused on functional programming:

- Summer School Series on Advanced Functional Programming. Springer-V., LNCS series.

Hot Research Topics – Haskell Symposium (1)

...in theory and practice of functional programming considering the 2012 Call for Papers of the Haskell Symposium:

"The purpose of the Haskell Symposium is to discuss experiences with Haskell and future developments for the language.

Topics of interest include, but are not limited to:

- Language Design, with a focus on possible extensions and modifications of Haskell as well as critical discussions of the status quo;
- Theory, such as formal treatments of the semantics of the present language or future extensions, type systems, and foundations for program analysis and transformation;
- Implementations, including program analysis and transformation, static and dynamic compilation for sequential, parallel, and distributed architectures, memory management as well as foreign function and component interfaces;

Hot Research Topics – Haskell Symposium (2)

- Tools, in the form of profilers, tracers, debuggers, pre-processors, testing tools, and suchlike;
- Applications, using Haskell for scientific and symbolic computing, database, multimedia, telecom and web applications, and so forth;
- Functional Pearls, being elegant, instructive examples of using Haskell;
- Experience Reports, general practice and experience with Haskell, e.g., in an education or industry context."

More on Haskell 2012, Copenhagen, DK, 13 Sep 2012:
http://www.haskell.org/haskell-symposium/2012/

Hot Research Topics – ICFP (1)

...in theory and practice of functional programming considering the 2012 Call for Papers of ICFP:

"ICFP 2012 seeks original papers on the art and science of functional programming. Submissions are invited on all topics from principles to practice, from foundations to features, and from abstraction to application. The scope includes all languages that encourage functional programming, including both purely applicative and imperative languages, as well as languages with objects, concurrency, or parallelism.

Topics of interest include (but are not limited to):

- Language Design: concurrency and distribution; modules; components and composition; metaprogramming; interoperability; type systems; relations to imperative, object-oriented, or logic programming

Hot Research Topics – ICFP (2)

- Implementation: abstract machines; virtual machines; interpretation; compilation; compile-time and run-time optimization; memory management; multi-threading; exploiting parallel hardware; interfaces to foreign functions, services, components, or low-level machine resources
- Software-Development Techniques: algorithms and data structures; design patterns; specification; verification; validation; proof assistants; debugging; testing; tracing; profiling
- Foundations: formal semantics; lambda calculus; rewriting; type theory; monads; continuations; control; state; effects; program verification; dependent types
- Analysis and Transformation: control-flow; data-flow; abstract interpretation; partial evaluation; program calculation

Hot Research Topics – ICFP (3)

- Applications and Domain-Specific Languages: symbolic computing; formal-methods tools; artificial intelligence; systems programming; distributed-systems and web programming; hardware design; databases; XML processing; scientific and numerical computing; graphical user interfaces; multimedia programming; scripting; system administration; security
- Education: teaching introductory programming; parallel programming; mathematical proof; algebra
- Functional Pearls: elegant, instructive, and fun essays on functional programming
- Experience Reports: short papers that provide evidence that functional programming really works or describe obstacles that have kept it from working"

Chapter 20.2

Programming Contest

Programming Contest Series: Background (1)

...considering the 2012 contest edition for illustration.

The ICFP Programming Contest 2012 is the 15th instance of the annual programming contest series sponsored by The ACM SIGPLAN International Conference on Functional Programming. This year, the contest starts at 12:00 July 13 Friday UTC and ends at 12:00 July 16 Monday UTC. There will be a lightning division, ending at 12:00 July 14 Saturday UTC.

The task description will be published at `icfpcontest2012.wordpress.com/task` when the contest starts. Solutions to the task must be submitted online before the contest ends. Details of the submission procedure will be announced along with the contest task.

This is an open contest. Anybody may participate except for the contest organisers and members of the same group as the contest chairs. No advance registration or entry fee is required.

Programming Contest Series: Background (2)

Any programming language(s) may be used as long as the submitted program can be run by the judges on a standard Linux environment with no network connection. Details of the judges' environment will be announced later.

There will be cash prizes for the first and second place teams, the team winning the lightning divison, and a discretionary judges' prize. There may also be travel support for the winning teams to attend the conference. (The prizes and travel support are subject to the budget plan of ICFP 2012 pending approval by ACM.)...

More on ICFP 2012, Copenhagen, DK, 10-12 Sep 2012:
`http://icfpconference.org/icfp2012/cfp.html`

The 23rd Programming Contest at ICFP 2020

In 2020, the programming contest started on

- Friday 17 July 2020 10:00am UTC. The 24hr lightning division will end at Saturday 18 July 2020 10:00am UTC and the 72hr full contest will end at Monday 20 June 2020 10:00am UTC; full information is available online:

  `https://icfpcontest2020.github.io`

- News are available at the following sites:
  - Programming contest series at the ICFP conf. series: `https://www.icfpconference.org/contest.html`
  - 23nd Programming contest edition in 2020: `https://icfpcontest2020.github.io/`
  - 2020 Host conference:
    ICFP 2020, Online Conference, 2020:
    `https://icfp20.sigplan.org/`

Contest Announcement at ICFP 2021

...coming soon!

Key dates can be expected to be similar as in 2020.

ICFP 2021, Online Conference,
Sun 22 - Fri 27 August 2021:
https://icfp21.sigplan.org/home

...stay tuned for conference and contest news at:

- Programming contest series at the ICFP conf. series:
  https://www.icfpconference.org/contest.html
- 24th Programming contest edition in 2021:
  https://icfp21.sigplan.org/track/icfp-2021-icfp-progr
- 2021 Host conference:
  ICFP 2021, Online Conf., Sun 22 - Fri 27 August 2021:
  https://icfp21.sigplan.org/

Chapter 20.3

In Conclusion

Functional Programming

...certainly arrived in the real world:

- Curt J. Simpson. Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
- Jerzy Karczmarczuk. Scientific Computation and Functional Programming. Computing in Science and Engineering 1(3):64-72, 1999.
- Bryan O'Sullivan, John Goerzen, Don Stewart. Real World Haskell. O'Reilly, 2008.
- Haskell in Industry and Open Source: `www.haskell.org/haskellwiki/Haskell_in_industry`

A Plea for Functional Programming

...even though the titles of:

- Philip Wadler. Why no one uses Functional Languages. ACM SIGPLAN Notices 33(8):23-27, 1998.
- Philip Wadler. An angry half-dozen. ACM SIGPLAN Notices 33(2):25-30, 1998.

might suggest the opposite, Philip Wadler's lamentation is only an apparent one and much more an impassioned

- plea for functional programming

in the real world summarizing a number of very general obstacles preventing good or even superior ideas also in the field of programming to make their way into mainstream practices easily and fast.

More Pleas for Functional Programming

...in and for the real world:

- Konrad Hinsen. The Promises of Functional Programming. Computing in Science and Engineering 11(4): 86-90, 2009.
- Konstantin Läufer, Geoge K. Thiruvathukal. The Promises of Typed, Pure, and Lazy Functional Programming: Part II. Computing in Science and Engineering 11(5): 68-75, 2009.
- Yaron Minsky. OCaml for the Masses. Communications of the ACM, 54(11):53-58, 2011.
- Neal Ford. Functional Thinking: Why Functional Programming is on the Rise. IBM developerWorks, 10 pages, 2013.

and quite recently:

- Neil Savage. Using Functions for Easier Programming. Communications of the ACM 61(5):29-30, 2018.

Recall Edsger W. Dijkstra's Prediction

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.[*)]

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

*) Quote from: Introducing a course on calculi. Announcement of a lecture course at the University of Texas at Austin, 1995.

In the Words of Simon Peyton Jones

When the limestone of
imperative programming has worn away,
the granite of functional programming
will be revealed underneath.

Simon Peyton Jones

In the Words of John Carmack

Sometimes, the elegant implementation is a function.
Not a method. Not a class. Not a framework.
Just a function.

John Carmack

Chapter 20.4

References, Further Reading

Chapter 20: Basic Reading (1)

- Neal Ford. *Functional Thinking: Why Functional Programming is on the Rise*. IBM developerWorks, 10 pages, 2013. https://www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf
- John Hughes. *Why Functional Programming Matters*. Computer Journal 32(2):98-107, 1989.
- John Hughes. *Why Functional Programming Matters*. Invited Keynote, Bangalore, 2016. https://www.youtube.com/watch?v=XrNdvWqxBvA.
- Konrad Hinsen. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.

Chapter 20: Basic Reading (2)

- Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.
- David A. Turner. *Total Functional Programming*. Journal of Universal Computer Science 10(7):751-768, 2004.
- Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
- Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
- Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.

Chapter 20: Basic Reading (3)

- Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2):25-30, 1998.
- Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8):23-27, 1998.

Chapter 20: Selected Advanced Reading (1)

- John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8):613-641, 1978.
- Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 25, Profiling and Optimization)
- Marcos Viera, S. Doaitse Swierstra, Wouter S. Swierstra. *Attribute Grammars fly First Class: How do we do Aspect Oriented Programming in Haskell*. In Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009), 245-256, 2009.

Chapter 20: Selected Advanced Reading (2)

- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.
- Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC.
- Greg Michaelson. *Programming Paradigms, Turing Completeness and Computational Thinking*. The Art, Science, and Engineering of Programming 4(3), Article 4, 21 pages, 2020.
- Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.

Concluding Note

...for additional information and details refer to

- full course notes

available in TUWEL and at the homepage of the course at:

http://www.complang.tuwien.ac.at/knoop/ffp185A05_ss2021.html

Assignment for 14-25 June 2021

...preparing the project demos for the period 14-25 June 2021 plus independent study of Part V, Chapters 17 and 18, Part VI, Chapters 19 and 20 and of Central and Control Questions VII and VIII for self-assessment and final questions for being asked and discussed alongside the demos.

Lecture, Flipped Classroom	Topic Lecture	Topic Flip. Classr.
Thu, 03/04/2021, 4.15-6.00 pm	P. I, Ch. 1 P. II, Ch. 2	n.a. / Prel. Mtg.
...	...	...
Thu, 04/22/2021, 4.15-6.00 pm	P. III, Ch. 5, 6	P. IV, Ch. 12, 13
Thu, 04/29/2021, 4.15-6.00 pm	P. V, Ch. 15, 16	P. III, Ch. 5, 6
Thu, 05/20/2021, 4.15-6.00 pm	**P. V, Ch. 17, 18** **P. VI, Ch. 19, 20**	**P. V, Ch. 15, 16**
06/14/2021 – 06/25/2021 **(2 to 3 afternoon sessions)**	**Project Demos**	**All Parts,** **All Chapters**