Today's Topic

- Pretty Printing
  Like parsing a typical demo-application
- Parallelism in Functional Programming Languages
  A hot research topic
- The Story of Haskell
  Behind the scenes of Haskell (and Functional Programming)

Part I: Pretty Printing

Pretty Printing
...like lexical and syntactical analysis another typical application for demonstrating the elegance of functional programming.

What's it all about?

A *pretty printer* is...

- a tool (often a library of routines) to convert a tree into text

Essential goals...

- a minimum number of lines while preserving and illustrating the structure of the tree by indentation

"Good" Pretty-Printer

...are distinguished by properly balancing

- Simplicity of usage
- Flexibility of the format
- "Niceness" of output

Reference

The following presentation is based on...

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

Distinguishing Feature

...of the "Prettier Printer" proposed by Philip Wadler:

- There is only a single way to concatenate documents, which is
  - associative
  - with left-unit and right-unit

Why "prettier" than "pretty"?

Wadler considers his "Prettier Printer" an improvement of the pretty printer library proposed by John Hughes, which is widely recognized as a standard.

- *The design of a pretty-printer library*. In Johan Jeuring, Erik Meijers (Hrsg.), *Advanced Functional Programming*, LNCS 925, Springer, 1995.

Hughes' library enjoys the following characteristics:

- Two ways to concatenate documents (horizontal and vertical), one of which
  - without unit (horizontal)
  - with right-unit (only) (vertical)
- ca. 40% more code, ca. 40% slower as Wadler's proposal

A Simple Pretty Printer: The Basis

Characteristic: For each document there is only one possible layout (e.g., no attempt is made to compress structure onto a single line).

The *basic operators* needed are:

```
(<>)   :: Doc -> Doc -> Doc   -- ass. concatenation
nil    :: Doc                 -- Right- and left-unit for (<>)
text   :: String -> Doc       -- Conversion function
line   :: Doc                 -- Line break
nest   :: Int -> Doc -> Doc   -- Adding indentation
layout :: Doc -> String       -- Output
```

Convention:

- Arguments of `text` are free of *newline* characters

A Simple Implementation

Implement...

- `doc` as strings (i.e. as `String`)

with...

- `(<>)` ...concatenation of strings
- `nil` ...empty string
- `text` ...identity on strings
- `line` ...new line
- `nest i` ...i blanks indentation (after each line break by means of `line`)
- `layout` ...identity on strings

Example

...converting trees into documents (here: `Strings`) and their output as text (here: `Strings`).

Consider the following type of trees:

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

A concrete value `B` of type `Tree`...

```
Node "aaa" [Node "bbbbb" [Node "cc" [], Node "dd" []],
            Node "eee" [],
            Node "ffff" [Node "gg" [],
                         Node "hhh" [],
                         Node "ii" []
                        ]
           ]
```


And its desired output

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


Implementation

The below implementation achieves this...

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


Another possibly wanted output of B

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


An implementation producing the latter output

```
data Tree             = Node String [Tree]

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

showBracket' :: [Tree] -> Doc
showBracket' []       = nil
showBracket' ts       = bracket "[" (showTrees' ts) "]"

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


A Normal Form of Documents

Normal form...

- text alternating with line breaks nested to a given indentation

```
text s0 <> nest i1 line <> text s1 <> ...
                             <> nest ik line <> text sk
```

Note:

- Documents can always be reduced to *normal form*

Normal Forms: An Example 1(3)

The document...

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

Normal Forms: An Example 2(3)

...has the normal form:

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


Normal Forms: An Example 3(3)

...and prints as follows:

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


Why does it work

...because of the properties (laws) the functions enjoy.

More on this next...

Properties of the Functions – Laws 1(2)

We have:

```
text (s ++ t)   = text s <> text t     (text is homomorphism from
text ""         = nil                   string concatenation to
                                        document concatenation)

nest (i+j) x    = nest i (nest j x)    (nest is homomorphism from
nest 0 x        = x                     addition to composition)

nest i (x <> y) = nest i x <> nest i y (nest distributes through
nest i nil      = nil                   document concatenation)

nest i (text s) = text s         (Nesting is absorbed by text)
```


Properties of the Functions – Laws 2(2)

Meaning

- The above laws are sufficient to establish that documents can always be transformed into normal form (first four laws: application left to right; last three laws: application right to left)

Further Properties – Laws

...on the relationship of documents and their layouts

```
layout (x <> y)      = layout x ++ layout y (layout is homomorphism
layout nil           = ""                    from document
                                             concatenation to
                                             string concatenation)

layout (text s)      = s                    (layout is the inverse
                                             of text)

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


The Implementation of `Doc`

Intuition

...representing documents as a concatenation of items, where each item is a text or a line break indented to a given amount.

...as a sum type (the algebra of documents):

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

...and the relationship of the constructors to document operators:

```
Nil        = nil
s 'Text' x = text s <> x
i 'Line' x = nest i line <> x
```


Example

The normal form (considered previously already)...

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

...has the representation:

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

Derived Implementations 1(2)

...of the document operators:

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

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


Derived Implementations 2(2)

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

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


On the Correctness

...of the derived implementations:

- Derivation of `(s 'Text' x) <> y = s 'Text' (x <> y)`

```
  (s 'Text' x) <> y
=   { Definition of Text }
  (text s <> x) <> y
=   { Associativity of <> }
  text s <> (x <> y)
=   { Definition of Text }
  s 'Text' (x <> y)
```

- Remaining equations: Similar reasoning

Documents with Multiple Layouts

- *Up to now*... documents are equivalent to a string
- *Now*... documents are equivalent to a set of strings

where each string correponds to a layout.

All what is needed: A new function

```
group :: Doc -> Doc
```

Informally:
...returns an additional element, which is provided in a new line

Preferred Layouts

- `layout` is replaced by `pretty`

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

- `pretty`'s integer-argument specifies the preferred maximum line length of the output (and hence the nicest layout out of the set alternatives at hand)

Example

Using...

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

...the call of `pretty 30` yields the output:

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

This ensures

- Output in one line where possible (i.e. length ≤ 30)
- Insertion of sufficiently many line breaks in order to avoid exceeding the given maximum line length

Implementation of the new Functions

The following supporting functions are required:

```
-- Union of two sets of layouts
(<|>)   :: Doc -> Doc -> Doc
-- Replacement of each line break (including subsequent
-- indentation) by a single space
flatten :: Doc -> Doc
```

- *Observation* ...documents always represent a non-empty set of layouts
- *Requirements*
  - ...in `(x <|> y)` all layouts of `x` and `y` enjoy the same flat layout
  - ...each first line in `x` is no shorter than each first line in `y`

Properties (Laws) of `(<|>)`

```
(x <|> y) <> z   = (x <> z) <|> (y <> z)
x <> (y <|> z)   = (x <> y) <|> (x <> z)
nest i (x <|> y) = nest i x <|> nest i y
```

Properties (Laws) of `flatten`

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

flatten (x <> y)  = flatten x <> flatten y
flatten nil       = nil
flatten (text s)  = text s
flatten line      = text " "          -- most interesting case
flatten (nest i x) = flatten x
```


Implementation of `group`

...by means of `flatten` and `(<>)`

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


Normal Form

Using the following settings each document can be reduced to a *normal form* of the form

```
x1 <|> ... <|> xn
```

where each `xi` is in the normal form of simple documents (which was introduced previously).

Selecting of a "best" Layout

...by defining an ordering relation on lines in dependence of the given maximum line length

Out of two lines...

- which do not exceed the maximum length, select the longer one
- of which at least one exceeds the maximum length, select the shorter one

The Adapted Implementation of `Doc`

```
data Doc = -- The first 3 alternatives as before
           Nil
         | String 'Text' Doc
         | Int 'Line' Doc
           -- We add a construct representing the
           -- union of two documents
         | Doc 'Union' Doc
```


Relationship of Constructors and Document Operators

```
Nil          = nil
s 'Text' x   = text s <> x
i 'Line' x   = nest i line <> x
x 'Union' y  = x <|> y
```


Example 1(8)

The document...

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


Example 2(8)

...has the 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
```

Example 3(8)

Task: ...print the above document under the constraint that the maximum line length is 5
⇝the right-most layout of the previous slide is requested

Initial considerations:

- ...Factoring out `"hello"` of all layouts of `x` and `y`

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

- ...Defining the interplay of `(<>)` and `nest` with `Union`

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


Example 4(8)

Implementing `group` and `flatten`

```
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
```


Example 5(8)

Considerations on correctness...

Derivation of `group (i 'Line' x)` (see line two)

```
  group (i 'Line' x)
=   { Definition of Line }
  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 }
  (" " 'Text' flatten x) 'Union' (i 'Line' x)
```


Example 6(8)

Correctness considerations...

Derivation of `group (s 'Text' x)` (see line three)

```
  group (s 'Text' x)
=   { Definition Text }
  group (text s <> x)
=   { Definition group}
  flatten (text s <> x) <|> (text s <> x)
=   { Definition flatten }
  (text s <> flatten x) <|> (text s <> x)
=   { <> distributiert ueber <|> }
  text s <> (flatten x <|> x)
=   { Definition group }
  text s <> group x
=   { Definition Text }
  s 'Text' group x
```


Example 7(8)

Selecting the "best" layout...

```
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
```

Remark:

- `best` ...converts a "union"-afflicted document into a "union"-free document
- Argument `w` ...maximum line length
- Argument `k` ...already consumed letters (including indentation) on current line

Example 8(8)

Check, if the first document line stays within the maximum line length...

```
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
```

Last but not least, the output routine (layout remains unchanged)...

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


A more efficient variant

...by means of a new implementation of documents

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

Remark:

- In distinction to the previous document type we here use capital letters

Implementing the Document Operators

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

Implementing group and flatten

As before, we require:

- ...in (x :<|> y) all layouts of x and y have the same flat layout
- ...each first line in x is no shorter than each first line in y

```
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
```


Representation Function

...generating the document from an indentation-afflicted document

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


Selecting the "best" Layout

Generalizing the function "best"...

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

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

where...

```
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))
```


In Preparation of further Applications 1(3)

...first some useful supporting functions

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

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

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


In Preparation of further Applications 2(3)

...further supporting functions

```
-- Often recurring output pattern
bracket l x r           = group (text l <>
                                 nest 2 (line <> x) <>
                                 line <> text r)

-- Abbreviation of the alternative tree layout function
showBracket' ts         = bracket "[" (showTrees' ts) "]"

-- Filling up lines (using words out of the Haskell Standard Lib.)
x <+/> y                = x <> (text " " :<|> line) <> y
fillwords               = folddoc (<+/>) . map text . words
```


In Preparation of further Applications 3(3)

fill, a variant of fillwords
$\leadsto$ ...collapses a list of documents to a single document

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


Application 1(2)

Printing XML-documents (simplified syntax)...

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

data Att              = Att String String

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)

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


Application 2(2)

Continuation...

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

showTag n a   = text n <> showFill showAtts a

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

Example 1

…for a given maximum line length of 30 letters

```
<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 2

…for a given maximum line length of 60 letters

```
<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 3

…after dropping of `flatten` in `fill`

```
<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>
```


Overview of the Code 1(11)

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

```
-- The pretty printer
infixr 5:<|>
infixr 6:<>
infixr 6 <>

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

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


Overview of the Code 2(11)

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

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
```


Overview of the Code 3(11)

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

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

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
```

Overview of the Code 4(11)

```
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

pretty w x                 = layout (best w 0 x)

-- Utility functions
x <+> y                    = x <> text " " <> y
x </> y                    = x <> line <> y

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


Overview of the Code 5(11)

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

bracket l x r    = group (text l <>
                          nest 2 (line <> x) <>
                          line <> text r)
x <+/> y         = x <> (text " " :<|> line) <> y

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

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

Overview of the Code 6(11)

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

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
```


Overview of the Code 7(11)

```
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
```


Overview of the Code 8(11)

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

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


Overview of the Code 9(11)

```
-- XML Example

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

data Att              = Att String String

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)
```


Overview of the Code 10(11)

```
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))) ""
```


Overview of the Code 11(11)

```
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."
                  ]

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


Further Readings 1(2)

On an imperative Pretty Printer

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

...and its functional realization

- Olaf Chitil. *Pretty printing with lazy dequeues*. In ACM SIGPLAN Haskell Workshop, 183-201, Florence, Italy, 2001. Universiteit Utrecht UU-CS-2001-23.

Further Readings 2(2)

Overview on the evolution of a Pretty Printer Library and origin of the development of the *Prettier Printers* proposed by Phil Wadler.

- John Hughes. *The design of a pretty-printer library*. In Johan Jeuring, Erik Meijers (Eds.), *Advanced Functional Programming*, LNCS 925, Springer, 1995.

...a variant implemented in the Glasgow Haskell Compiler

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

Part II: Parallelism in Functional Programming Languages

Parallelism

- Implicit
- Explicit
- Skeletons

Reference

The following presentation is based on...

- Chapter 21
  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*, Springer, 2006. (In German).

Related and relevant in this context...

- Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*, The MIT Press, 1989.
- Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon L. Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming, 8(1):23-60, 1998.
- Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Pointon. *Parallel and Distributed Haskells*. Journal of Functional Programming, 12(4&5):469-510, 2002.

Parallelism in Imperative Languages

In particular...

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

Parallelism in Functional Languages

In particular...

- Implicit/Expression parallelism
- Explicit
- Algorithmic skeletons

Implicit Parallelism

...resp. *expression parallelism*

Consider the functional expression of the form `f(e1,...,en)`:

Note:

- Arguments (and functions) can be evaluated in parallel.
- Advantages: Parallelism *for free*! No effort for the programmer.
- Disadvantages: Results often unsatisfying. E.g. granularity, load distribution, etc. not taken into account.

Thus:

- Easy to detect parallelism (i.e., for the compiler), but hard to fully exploit.

Explicit Parallelism

By...

- Introducing meta-statements (e.g. to control the data and load distribution, communication)
- Advantages: Possibly superior results by explicit hands-on control of the programmer
- Disadvantages: High programming effort

Algorithmic Skeletons

Compromise between...

- *explicit imperative* parallel programming
- *implicit functional* parallel programming

In the following

- Massively parallel systems
- Algorithmic skeletons

Massively Parallel Systems

...characterized by

- large number of processors with
  - local memory
  - communication by message exchange
- MIMD-Parallel Processor Architecture (*Multiple Instruction/Multiple Data*)
- Here: SPMD-Programming Style (*Single Program/Multiple Data*)

Algorithmic Skeletons

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

Realization of Algorithmic Skeletons

...in functional languages

- by special higher-order functions
- with parallel implementation
- embedded in sequential languages

Thus

- 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

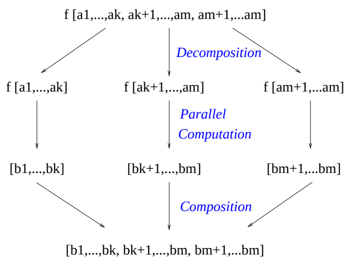
- Application of `f` to a list element does not depend on other list elements

Apparent

- Dividing the list into sublists followed by *parallel* application of map to the sublists (parallelization pattern *Farm*)

Parallel Map on Distributed Lists

For illustration...



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

On the Implementation

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

- so-called *data-parallel skeletons*

Difference

- *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, hence dynamically generated

Programming of a Parallel Application

...using algorithmic skeletons

- Recognizing problem-inherent parallelism
- Selecting an adequate data distribution (granularity)
- Selecting a suitable skeleton from a library
- Problem-specific instantiation of the skeleton(s)

Remark:

- Some languages (e.g. Eden) support also the implementation of skeletons

Data Distribution on Processors

...is

- crucial for
  - structure of the complete algorithm
  - efficiency

Hardness dependent 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

An auxiliary means

- So-called *covers* (investigated by various authors)

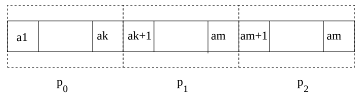
Covers

...describe

- Decomposition and communication pattern of a data structure

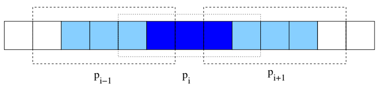
Example: Simple List Cover

Distributing a list on 3 processors p_1, p_2, and p_3:



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

Example: List Cover with Overlapping Elements



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

General Cover Structure

```
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
}
```

It is required:

```
glue . split = id
```

Note: No (valid) Haskell

Realization in a Programming Language

...implementing covers requires support for

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

...is

- current hot research topic in functional programming

Further Reading

- Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. Higher-Order and Symbolic Computation, 16(3):203-251, 2003.

Part III: The Story of Haskell

16 Years of Haskell: A Retrospective on the occasion of its 15th Anniversary

by

Simon Peyton Jones

Wearing the Hair Shirt: A Retrospective on Haskell

`http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/`

Haskell at HOPL III

Most recently...

- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the Third ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), (San Diego, California, June 09 - 10, 2007), 12-1 - 12-55.

Check out the ACM Digital Library (`www.acm.org/dl`) for this article!

Last but not least

Final (oral) examination...

- In principle, any time (except of the period from July 3rd to July 25th. Just make an appointment by email (`knoop@complang.tuwien.ac.at`) or phone (58801-18510).
- Topics: Assignments plus lecture materials.