Pretty Printing with Delimited Continuations

Olaf Chitil

University of Kent, UK

IFL 2005
What is Pretty Printing?

```
if True
  then if True then True else True
else
  if False
    then False
    else False
```

Pretty printing library interface

```
text :: String -> Doc
line :: Doc
(<> ) :: Doc -> Doc -> Doc
nest :: Int -> Doc -> Doc
group :: Doc -> Doc
pretty :: Int -> Doc -> String
```
What is Pretty Printing?

User code

toDoc :: Exp -> Doc
toDoc (If e1 e2 e3) =
  group (nest 3 (group (nest 3 (text "if" <> line <> toDoc e1)) <> line <>
              group (nest 3 (text "then" <> line <> toDoc e2)) <> line <>
              group (nest 3 (text "else" <> line <> toDoc e3))))
What is Pretty Printing?

```plaintext
if True
    then if True then True else True
else
    if False
        then False
        else False
```

Pretty printing library interface

```plaintext
text :: String -> Doc
text :: String -> Doc
line :: Doc
line :: Doc
(<>): Doc -> Doc -> Doc
(<>): Doc -> Doc -> Doc
(nest :: Int -> Doc -> Doc)
(nest :: Int -> Doc -> Doc)
group :: Doc -> Doc
group :: Doc -> Doc
pretty :: Int -> Doc -> String
```
A document may be formatted *horizontally* or *vertically*.

```haskell
type Horizontal = Bool
```

A document has many different layouts.

```haskell
type Doc = Horizontal -> [String]
```

Layouts for each document:

- `(text t) _ = [t]`
- `line True = [' ' ' ']`
- `line False = ['\n']`
- `(d1 <> d2) h = [l1 ++ l2 | l1 <- d1 h, l2 <- d2 h]`
- `(group d) True = d True`
- `(group d) False = d False ++ d True`

Prettiest: compare line by line; within width-limit longer line better.

```haskell
pretty w d = minimumBy (compareLayout w) (d False)
```
Specification: Further Properties

- **time:**
  - linear in document size
  - independent of document width

- (optimally) bounded
  
  pretty 4 (group (text "Hi" <> line <> text "you" <> undefined))

  yields

  Program error: {undefined}

  Instead want

  Hi
  you

  Program error: {undefined}

- **space (lazy input/output):** linear in width-limit
Outline of a Linear Algorithm

Two passes:
1. Use position in document to determine width of each group.
2. Use remaining space on line to determine for each group if horizontal.

Linear but unbounded.
Correctness Problem

pretty 6 (group (text "Hi" <> line <> text "you") <> text "!"))

algorithm yields

Hi you!

but specification says

Hi you!
Correctness Problem

pretty 6 (group (text "Hi" <> line <> text "you") <> text "!")

algorithm yields

Hi you!

but specification says

Hi
you!

Only group-closed documents:

A line between end of each group and next text.
Represent Document as Token List

Represent

group (text "Hi" <> line <> text "you") <> text "!"
as

[Open, Text "Hi", Line, Text "you", Close, Text "!"]

Group-closed document via rewriting:

\[
\begin{align*}
\text{Close (Text } s & \text{ ts)} \Rightarrow \text{ Text } s \ (\text{Close ts}) \\
\text{Open (Text } s & \text{ ts)} \Rightarrow \text{ Text } s \ (\text{Open ts}) \\
\text{Open (Close ts)} & \Rightarrow \text{ ts}
\end{align*}
\]

Effect of representation change:

- Rewritten document describes same set of layouts.
- Algorithm always selects correct layout.
A Linear Unbounded Algorithm

- single pass
- no laziness
A Linear Unbounded Algorithm

- single pass
- no laziness

output
already produced
A Linear Unbounded Algorithm

- single pass
- no laziness

\[ \text{output already produced} \]
\[ \text{group output function} \]

\[
\text{type Out} = \text{Remaining} \rightarrow \text{String} \\
\text{type OutGroup} = \text{Horizontal} \rightarrow \text{Out} \rightarrow \text{Out}
\]

\[
\text{inter :: Tokens} \rightarrow \text{Width} \rightarrow \text{Position} \rightarrow \langle (\text{Position, OutGroup}) \rangle \rightarrow \text{Out}
\]

Olaf Chitil (University of Kent, UK) Pretty Printing with Delimited Continuations IFL 2005 8 / 11
A Linear Unbounded Algorithm

- single pass
- no laziness

\[\text{Output} = \text{Remaining} \rightarrow \text{String}\]
\[\text{OutGroup} = \text{Horizontal} \rightarrow \text{Out} \rightarrow \text{Out}\]

\[\text{inter} :: \text{Tokens} \rightarrow \text{Width} \rightarrow \text{Position} \rightarrow \langle \text{Position, OutGroup} \rangle \rightarrow \text{Out}\]
A Linear Unbounded Algorithm

- single pass
- no laziness

\[ \text{output} \]
\[ \text{group} \]
\[ \text{group} \]
\[ \text{group} \]
\[ \text{group} \]
\[ \text{output} \]
\[ \text{already} \]
\[ \text{output} \]
\[ \text{output} \]
\[ \text{output} \]
\[ \text{produced} \]
\[ \text{function} \]
\[ \text{function} \]
\[ \text{function} \]

\[
\text{type Out = Remaining} \rightarrow \text{String} \\
\text{type OutGroup = Horizontal} \rightarrow \text{Out} \rightarrow \text{Out} \\
\text{inter :: Tokens} \rightarrow \text{Width} \rightarrow \text{Position} \rightarrow \langle (\text{Position}, \text{OutGroup}) \rangle \rightarrow \text{Out}
\]
A Linear Unbounded Algorithm

- single pass
- no laziness

```
inter :: Tokens -> Width -> Position -> ((Position, OutGroup)) -> Out
```

```
type Out = Remaining -> String

type OutGroup = Horizontal -> Out -> Out
```

Olaf Chitil (University of Kent, UK)
Pretty Printing with Delimited Continuations
IFL 2005
A Linear Unbounded Algorithm

- single pass
- no laziness

\text{output already produced function}

\begin{align*}
\text{type Out} & = \text{Remaining} \rightarrow \text{String} \\
\text{type OutGroup} & = \text{Horizontal} \rightarrow \text{Out} \rightarrow \text{Out} \\
\text{inter} & : : \text{Tokens} \rightarrow \text{Width} \rightarrow \text{Position} \rightarrow \langle \langle \text{Position, OutGroup} \rangle \rangle \rightarrow \text{Out}
\end{align*}
A Linear Unbounded Algorithm

- single pass
- no laziness

\[
\text{output} \quad \text{group} \quad \text{group} \quad \text{group}
\]

\[
\text{already output} \quad \text{output} \quad \text{output} \quad \text{output}
\]

\[
\text{produced function} \quad \text{function} \quad \text{function}
\]

type Out = Remaining -> String
type OutGroup = Horizontal -> Out -> Out

\[
\text{inter :: Tokens} \rightarrow \text{Width} \rightarrow \text{Position} \rightarrow \langle ((\text{Position}, \text{OutGroup}) \rightarrow \text{Out}}
\]
The Linear Bounded Algorithm

- continuously check whether outermost surrounding group fits

\[
\text{current position}
\]

\[
\text{output group output function}
\]

\[
\text{already output output function}
\]

\[
\text{produced function function function}
\]

type Out = Remaining -> String

type OutGroup = Horizontal -> Out -> Out

\[
\text{inter :: Tokens -> Width -> Position -> \langle (Position, OutGroup) \rangle -> Out}
\]
The Linear Bounded Algorithm

- continuously check whether outermost surrounding group fits

output already produced

\[ \text{type Out} = \text{Remaining} \rightarrow \text{String} \]
\[ \text{type OutGroup} = \text{Horizontal} \rightarrow \text{Out} \rightarrow \text{Out} \]

\[ \text{inter} :: \text{Tokens} \rightarrow \text{Width} \rightarrow \text{Position} \rightarrow \langle (\text{Position}, \text{OutGroup}) \rangle \rightarrow \text{Out} \]
Replace

\text{inter} :: \text{Tokens} \to \text{Width} \to \text{Position} \to \langle \text{OutGroup} \rangle \to \text{Out}

by

\text{noGroup} :: \text{Tokens} \to \text{Width} \to \text{Position} \to \text{Out}
\text{oneGroup} :: \text{Tokens} \to \text{Width} \to \text{Position} \to \text{Position} \to \text{OutGroup} \to \text{Out}
\text{multiGroup} :: \text{Tokens} \to \text{Width} \to \text{Position} \to \text{Position} \to \text{OutGroup} \\
\to \langle \text{Position}, \text{OutGroup} \rangle \to \text{OutGroup} \to \text{Out}
Summary

- Delimited continuations express explicitly switching between consuming input and producing output.
- Dequeue is buffer between input consumed and output produced.
- Specialisation improves performance but duplicates code.
- Laziness gives space linear in width, but irrelevant for correctness and linearity.
- Higher-order functions essential; defunctionalised algorithm incomprehensible.
- How to prove equivalence of specification and implementation?