Haskell Expressions as Trees

- Expression `x:xs`
- Expression `if null xs then 0 else 1 + len xs`

Haskell Expressions as Acyclic Graphs

- Expression `if null xs then 0 else 1 + len xs`

Haskell Expressions as Cyclic Graphs

- Expression `ones where ones = 1:ones`
- `ones` is an infinite list of 1s
- We can represent this infinitely large value by a finite-sized but cyclic graph.
- Haskell’s implementation of laziness supports the finite representation of infinite values.
let-graph correspondence

- There is a correspondence between let-expressions and graphs.
- Consider the following let-expression:
  ```
  let x = \text{exp}_1
  \quad y = \text{exp}_2
  \quad z = \text{exp}_3
  \quad \text{in } \text{exp}_0
  ```

  The variables $x$, $y$ and $z$ are in scope in all $\text{exp}_i$.
- Each lhs variable labels a graph node, and any occurrence in $\text{exp}_i$ points to that node.

  ![Graph Diagram]

  Here $\leftarrow\rightarrow$ indicates possible linkages.

let-graph correspondence: ones

- We can write $\text{ones}$ using let
  ```
  let ones = 1:\text{ones}
  \text{in } \text{ones}
  ```

- The graph is
  ![Graph Diagram]

Pattern Matching Revisited

- Patterns are matched in order until a match occurs, or all fail to match (runtime error)
- We don’t re-evaluate arguments from scratch for later patterns
- An early general match (e.g. $x:xs$) masks a later more specific one (e.g. $1:xs$)
- This “first-come first-served” approach handles overlapping patterns gracefully
- Haskell can warn about overlapping and (some) missing patterns.
The Need for Laziness

- We shall present an example were laziness is very helpful
- Selecting sub-expressions
- We have a simple expression datatype:
  ```haskell
data Expr = Num Int
  | Plus Expr Expr
  | Times Expr Expr
```
- We want to display it in a nice text format:
  ```text```
  \[(3 + (5 * 2))\]
  instead of
  ```haskell```
  Plus (Num 3) (Times (Num 5) (Num 2))
- We want to be able to select sub-expressions (somehow)
- We want to “use” sub-expressions in some way:
  - display (in/out of context)
  - replace

Selecting sub-expressions

- We need to identify the selected sub-expression — the “focus”.
- We want an operation `setf` that selects the top-level of an expression.
- We want an operation `fdown i` that selects the \(i\)th sub-child of the current focus.
- We want an operation `fup` that selects the parent of the current focus.
- We want an operation `ftop` that jumps to the outermost expression level.
- We want an operation `fshow` that displays the focus in isolation.
- We want an operation `cshow` that displays the focus in the context of its surrounding expression.
- We want an operation `frep` that replaces the focus expression by another.

Example

```
+  <- focus
\[ 3 \]
\[ \times \]  <- focus
\[ 5 \]
\[ \downarrow \]
\[ 2 \]
```

```
sclf (\((3+(5*2))\)) cshow (\[(3+(5*2))\])
fdown 2 fshow (5*2) cshow (3+[(5*2)])
```
Focus Datatype and \texttt{setf}

First solution attempt: keep a list (a.k.a. “path”) that records the \texttt{fdown} parameter needed to get to the focus.

\begin{verbatim}
  type P_Focus = (Expr,[Int])
\end{verbatim}

We then set focus by adjoining an empty path.

\begin{verbatim}
p_setf e = (e,[])
\end{verbatim}

We shall append \texttt{P} and \texttt{p} as appropriate for this “path-based” solution.

The \texttt{case} expression

- We have just used the Haskell \texttt{case}-expression for the first time.

\begin{verbatim}
  case e of
    pat\textsubscript{1} \rightarrow res\textsubscript{1}
    pat\textsubscript{2} \rightarrow res\textsubscript{2}
    ...
    pat\textsubscript{n} \rightarrow res\textsubscript{n}
\end{verbatim}

- We match expression \texttt{e} against the patterns (\texttt{pat\textsubscript{i}}) in order.

- A successful match against \texttt{pat\textsubscript{i}} returns \texttt{res\textsubscript{i}}, with substitution made using match bindings.

- Analagous to pattern matching in function definitions.

\texttt{fdown}

To go down we get the focus (\texttt{p_getfocus}) and check that we can descend. If so, we use \texttt{p_godown} to note our descent.

\begin{verbatim}
p_fdown i (e,is) = let f = p_getfocus e is in
  case f of
    (Num _) -> (e,is)
    (Plus e1 e2) -> p_godown i (e,is)
    (Times e1 e2) -> p_godown i (e,is)
\end{verbatim}

\texttt{getfocus}

Getting the focus expression is simply a recursive descent through \texttt{Plus} and \texttt{Times} nodes using the numbers in the path to identify the first or second branch.

\begin{verbatim}
p_getfocus (Plus e _) (1:is) = p_getfocus e is
p_getfocus (Times e _) (1:is) = p_getfocus e is
p_getfocus (Plus _ e) (2:is) = p_getfocus e is
p_getfocus (Times _ e) (2:is) = p_getfocus e is
p_getfocus e _ = e
\end{verbatim}

If the path and expression do not correspond we return the expression as is.
This function simply records a descent by appending the branch number to the end of the path list.

\[
\begin{align*}
p_{\text{godown}} 1 \ (e, is) &= (e, is + [1]) \\
p_{\text{godown}} 2 \ (e, is) &= (e, is + [2]) \\
p_{\text{godown}} \ _\ (e, is) &= (e, is)
\end{align*}
\]

Going up simply removes the last element of the path, if present \text{(init)} is defined in the Prelude).

\[
\begin{align*}
p_{\text{fup}} \ (e, []) &= (e, []) \\
p_{\text{fup}} \ (e, xs) &= (e, \text{init} \ xs)
\end{align*}
\]

Going to the top simply sets an empty path.

\[
p_{\text{ftop}} \ (e, \_) &= (e, [])
\]

Showing the focus in isolation is easy: get it and display it

\[
p_{\text{fshow}} \ (e, is) = \text{disp} \ (p_{\text{getfocus}} \ e \ is)
\]

Showing in context, however, is a lot of work!

\[
\begin{align*}
p_{\text{cshow}} \ (e, []) &= " \ [ "++\text{disp} \ e++" \ ] " \\
p_{\text{cshow}} \ ((\text{Plus} \ e1 \ e2),(i:is)) &= \\
| \ i == 1 &= \ "( "++p_{\text{cshow}} \ (e1, is) ++ "+" ++ \text{disp} \ e2 ++ ")" \\
| \ i == 2 &= \ "( "++\text{disp} \ e1 ++ "+" ++ p_{\text{cshow}} \ (e2, is) ++ ")"
\end{align*}
\]

We effectively have to write a customised version of \text{disp}!

\text{(This will not scale!)}

Replacing the focus is also very tricky

\[
\begin{align*}
p_{\text{frep}} \ (e, [],) \ e' &= (e', [],) \\
p_{\text{frep}} \ ((\text{Plus} \ e1 \ e2),(i:is)) \ e' &= \\
| \ i == 1 &= \ (\text{Plus} \ e1' \ e2,(i:is)) \\
| \ i == 2 &= \ (\text{Plus} \ e1 \ e2',(i:is))
\end{align*}
\]

where

\[
\begin{align*}
(e1',\_), &= p_{\text{frep}} \ (e1, is) \ e' \\
(e2',\_), &= p_{\text{frep}} \ (e2, is) \ e'
\end{align*}
\]

\[
\begin{align*}
p_{\text{frep}} \ ((\text{Times} \ e1 \ e2),(i:is)) \ e' &= \\
| \ i == 1 &= \ (\text{Times} \ e1' \ e2,(i:is)) \\
| \ i == 2 &= \ (\text{Times} \ e1 \ e2',(i:is))
\end{align*}
\]

where

\[
\begin{align*}
(e1',\_), &= p_{\text{frep}} \ (e1, is) \ e' \\
(e2',\_), &= p_{\text{frep}} \ (e2, is) \ e'
\end{align*}
\]

Again, this will not scale!
The Real Problem

- We don’t want it to work for \textbf{Expr} as just shown.
- We want it to work for the following:

\begin{verbatim}
| Setc QVars Pred Expr | Seqc QVars Pred Expr | Map ([Expr,Expr]) | Cond Pred Expr Expr | Build String [Expr]
| The QVars Expr Pred | Eval String | Eqvar String | Eabs QVars Expr
| Esub Expr ESubst | Eerror String
data Pred = TRUE | FALSE | Obs Type Expr | Defd Expr | TypeOf Expr Type
| Not Pred | And [Pred] | Or [Pred] | Imp Pred Pred | Eq Pred Pred
| Forall QVars Pred Pred | Exists QVars Pred Pred | Exists1 QVars Pred Pred
| Univ Pred | Sub Pred ESubst | Pair String | If Pred Pred Pred
| NDC Pred Pred | RfdBy Pred Pred | LPredPred String Int Pred Pred
| LVarExpr String Int String Expr | LExprPred String Pred
| LPredExprPred String Int Pred Expr Pred | Peabs String Pred
| Ppabs String Pred | Papp Pred Pred
\end{verbatim}

(Plus other stuff we haven’t shown !)

Towards a better solution

We want a solution that \textit{scales}:
- We don’t want to have \texttt{cshow} as a fully re-written/modified version of \texttt{disp}.
- We don’t want \texttt{frep} to have to have a clause for every \texttt{Expr} variant
- If we add a new variant to \texttt{Expr} the only functions we should need to change are:
  - \texttt{disp}: just by adding clause for new variant
  - \texttt{fdown}: just by adding clause for new variant
  - \texttt{frep}: just by adding clause for new variant
- The solution actually used depends on laziness in an essential way

Towards a Lazy Solution (1)

- First, we introduce an new \texttt{Expr} variant to “mark” the focus:
  \begin{verbatim}
data Expr = ... | Focus Expr
\end{verbatim}

- The term \texttt{Efocus e} is now represented in the graph as:

\begin{verbatim}
focus node e focus expression
\end{verbatim}

- Implementing \texttt{disp} is easy — we just add one line:
  \begin{verbatim}
disp (Focus e) = "[" ++ disp e ++ ""]"
\end{verbatim}

- However, \texttt{getfocus} is now a tree-search (hunt everywhere for the \texttt{Focus} variant).

Towards a Lazy Solution (1a)

Using \textbf{Focus} we can now represent a focus:

\begin{verbatim}
focus + 2
3
5
\end{verbatim}

by using the explicit focus marker:

\begin{verbatim}
+ 2
3
5
\end{verbatim}
Towards a Lazy Solution (2)

- We want to avoid searching for the focus, so we keep the focus and top-level together in a pair.
  
  ```haskell```
  type Focus1 = (Expr, Expr)
  ```

- So `(focus, top)` represents both the focus expression, and the top-level:

```
( . )
   ↙
   +
   2
```

- Here `( . )` is a “pair”-node, similar to `:` or `@`, with pointers to left (1st) and right (2nd) components.

Towards a Lazy Solution (2a)

- `getfocus` is now trivial.
- Both `fshow` and `cshow` are now cheap:
  - For `fshow` just take first pair component and use `disp`
  - For `cshow` just take second pair component and use `disp`
- But `fup` is now expensive, involving a tree-search.

Towards a Lazy Solution (3)

- We optimize `fup` by providing a list of backlinks from the focus up through its parents.
- Each element of the backlink list contains:
  - the parent expression
  - a number identifying the relevant child
- `type BackLink = (Expr, Int)`
- `type Focus2 = (Expr, [BackLink])`

```
( . )
   ↙
   [(•, 1)]
   3
   ?
   5
```

- But where should the link from `+` point?

Towards a Lazy Solution (3a)

- We need to back up a bit
- We want the focus to be part of the `Expr` datatype
  ```haskell```
  type BackLink = (Expr, Int)
  data Expr = ... | Focus Expr [BackLink]
  ```
- So our focus node now has the form: `[(.)]`
- Our picture looks like:

```
[(.)]
   ↙
   [(•, 1)]
   3
   ?
   5
```

- Now, where should the pointer go?
Towards a Lazy Solution (3b)

- We could simply point at the sub-expression:

```
  \[ (. , 1) \]
  \[ \]
  \[ 3 \]
  \[ 5 \]
  \[ + \]
  \[ 2 \]
```

- But then `cshow` becomes impossible in general!
  - Descending from `+` how do we know we have reached the focus?

Towards a Lazy Solution (3c)

- We need to point at the focus node:

```
  \[ (. , 1) \]
  \[ \]
  \[ 3 \]
  \[ 5 \]
  \[ + \]
  \[ 2 \]
```

- But now the graph is cyclic
  - We need laziness, and must use `let` to build this.

Too complicated?

- Why not separate marking the focus from pairing it with the backlinks?
  - Type `TopFocus` = `(Expr, [BackLink])`
    ```
    data Expr = ... | Focus Expr
    ```

- Problem — in Saoithín we have two mutually recursive datatypes, `Pred` and `Expr` and we want focus to jump between the two.
  - This is not possible with this acyclic “`TopFocus`” approach.