What’s really going on?

- Haskell as a re-write system makes sense, but . . .
- . . . how is the rewrite system implemented?
- We know what purity is, but now we need to understand how it is achieved.
- We need to drill down further into the execution model for Haskell

Abstract Syntax Trees

- The Haskell Parser converts Haskell source-text into internal abstract syntax trees (AST).
- These trees are built from boxes of various types and edges (pointers).
- We shall describe an execution model that manipulates these trees directly.

AST Boxes

- Atomic Values and Variables: 3, True, 'c' v
  ![Atomic Values and Variables Diagram]
- Data Constructors: [], :"!
  ![Data Constructors Diagram]
  The “cons” box has 2 pointers to relevant components
- Function Application:
  ![Function Application Diagram]
  The “apply” box also has 2 pointers to relevant components
Consider application `sum (2:1:[])`

- We match against pattern `sum (x:xs)` with binding `x ↦ 2, xs ↦ 1:[]`
  - this is done by matching AST trees recursively
  - the bindings are pointers to relevant AST fragments
- We want to replace application by rhs `x + sum xs`, using the bindings above to get `2 + sum (1:[])`
  - We use rhs as a template,
  - we build a copy, replacing formal arguments using bindings,
  - we replace the application AST by the rhs AST copy.
- The fact we build a copy of the rhs AST is crucial for referential transparency

Note how binary application has a “spine” of 2 @-nodes.

\[
\begin{align*}
a + b &= (((+) a) b) \\
\end{align*}
\]
AST Binding

- The bindings from that successful match:
  binding: \( x \mapsto 2, \, xs \mapsto 1:[] \)

- \( x \mapsto 2 \)
  - \( x \)
  - \( 2 \)

- \( xs \) : \( 1 \) \( [] \)
  - \( : \)
  - \( 1 \)
  - \( [] \)

AST Copying

- The rhs from that successful match: \( x + \text{sum} \, xs \)

- \( x + \text{sum} \, xs \)
  - \( + \)
  - \( x \)
  - \( \text{sum} \)
  - \( xs \)

- The copy built replacing pattern variables by their bindings:
  copy: 2 + \( \text{sum} \, (1:[]) \)

- 2 + \( \text{sum} \, (1:[]) \)
  - \( + \)
  - \( 2 \)
  - \( \text{sum} \)
  - \( (1:[]) \)

AST Shorthand

- The AST Box diagrams take up a lot of space
  Let’s introduce a shorthand version
  - drop single boxes for basic values: 1 [] True v
  - drop triple boxes for application and cons-ing:
    \( f \, a \, x:xs \)
  - so for example, \( x + \text{sum} \, xs \) now looks like:

Haskell AST Execution — another example

- The HOF map is defined as follows:
  \[
  \text{map} \, f \, [] = []
  \text{map} \, f \, (x:xs) = (f \, x) : \text{map} \, f \, xs
  \]

- We have the following rhs ASTs:
  
  \[
  []
  \]

- \( \text{map} \, f \, x \)
  - \( \text{map} \)
  - \( f \)
  - \( x \)
  - \( [] \)
  - \( xs \)
Consider application \texttt{map inc (1:2:3[])} where \texttt{inc x = x+1}

1. We match 2nd case \( f \mapsto \text{inc}, x \mapsto 1, xs \mapsto 2:3:[] \)
   
   We build a \textit{copy} of 2nd rhs, using bindings
   
   \[
   \begin{array}{l}
   \text{(inc 1) : (map inc (2:3:[]))}
   \end{array}
   \]

2. We match 2nd case, \( f \mapsto \text{inc}, x \mapsto 2, xs \mapsto 3:[] \)
   
   We build a \textit{copy} of 2nd rhs, using bindings
   
   \[
   \begin{array}{l}
   \text{(inc 1) : ((inc 2) : (map inc (3:[])))}
   \end{array}
   \]

3. We match 2nd case, \( f \mapsto \text{inc}, x \mapsto 3, xs \mapsto [] \)
   
   We build a \textit{copy} of 2nd rhs, using bindings
   
   \[
   \begin{array}{l}
   \text{(inc 1) : ((inc 2) : ((inc 3) : (map inc [])))}
   \end{array}
   \]

4. We match 1st case, \( f \mapsto \text{inc} \)
   
   We build a \textit{copy} of 2nd rhs, using bindings
   
   \[
   \begin{array}{l}
   \text{(inc 1) : ((inc 2) : ((inc 3) : []))}
   \end{array}
   \]

---

\textbf{The Importance of Copying (I)}

- We clearly need to copy the function rhs, otherwise we
couldn’t re-use that function.
- But in the application \texttt{map inc [1..3]} we not only copied
  the rhs, but that built us a \textit{copy} of the original list.
- Couldn’t a smart implementation realise that the copies
  simply had the leaves changed from \texttt{x} to \texttt{inc x}, and change
  these in place
  (so-called “destructive update”)?

\textbf{Before evaluating map inc [1..3]}

- We have the application as an AST (simplified)

\[
\begin{array}{c}
\text{map} \\
\downarrow \\
\circ \\
\leftarrow \quad [1..3] \\
\end{array}
\]

- \( \bullet \) denotes a pointer to the application
- (We show the original list as one lump)

\textbf{How Haskell does map inc [1..3]}

- We build a copy, and swing our application pointer to indicate
  that copy

\[
\begin{array}{c}
\text{map} \\
\downarrow \\
\circ \\
\leftarrow \quad [inc 1, inc 2, inc 3] \\
\end{array}
\]

- the original list and other arguments are still present
- If there are no further pointers to the original list it becomes
garbage, which is handled behind the scenes.
How we might optimise \(\text{map inc \ [1..3]}\)

- We update the list in place and swing our application pointer to indicate that update.

![Diagram]

- We don't alter the \(\text{map rhs ASTs}\).
- We (the compiler) somehow manage to see that the list structure is unchanged so we do destructive update in place.

The Importance of Copying (II)

- Destructive Update breaks Referential Transparency
- Consider the following program:
  \[
  \text{myfun} \ \text{xs} = (\text{xs, map inc xs})
  \]
  We have paired together references to both the original \(\text{xs}\), and the result of mapping \(\text{inc}\) across it.
- If we use copying, then the two lists returned by \(\text{myfun}\) are different.
- If we use destructive update, then the two lists returned by \(\text{myfun}\) are equal.
  - but this contradicts the following law for non-empty \(\text{xs}\):
    \[
    \text{head} (\text{map inc xs}) = \text{inc} (\text{head xs})
    \]

Copying as a show-stopper (I)

- Imagine that \(\text{bigds}\) is a very large datastructure and \(\text{bigmod}\) is a function with parameters that performs large changes to it.
- Copying means that the following sequence of calls is very expensive to run:
  \[
  \text{let bigds1 = bigmod p1 bigds} \\
  \text{bigds2 = bigmod p2 bigds1} \\
  \ldots \\
  \text{bigdsn' = bigmod pn' bigdsn} \\
  \text{in ...}
  \]
- So pure functional languages are not good for implementing large databases, processing large amounts of data, supporting design of large artefacts (i.e VLSI chips), ...?
Copying and Real-World I/O are inconsistent

- We cannot implement real-world I/O in a pure (referentially transparent) language
- So pure functional languages are just intellectual toys . . .
- Real-world functional languages (e.g. ML, Lisp, Scheme) are impure so they can
  - support real-world I/O
  - allow destructive update for large datastructures
- This slide summarises a view of (pure) functional languages still widely believed today
- This view was justifiable, until the early 1990s (Yes, that long ago !)
  - But the slide title is still correct . . .