Contents

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

Atomic Values and Variables:

- `3`
- `True`
- `'c'`

Data Constructions:

- `[]`
- `:`

Application Example

- Consider application `sum (2:1:[])`
- The LHS of `sum` is `sum (x:xs)` and bind `x` -> `2`, `xs` -> `1:[]`
  - This is done by matching the *syntax trees* recursively
  - The bindings are pointers to relevant AST fragments
- We want to replace the LHS by the RHS: `x + sum xs`, using the bindings above to get `2 + sum (1:[])`
  - We use RHS as a *template*
  - We build a *copy*, replacing arguments with their bound values
  - We replace the function application with the *copy*
- The fact we build a *copy* of the RHS AST is cricial for referential transparency

Importance of Copying

- We clearly need to copy the function RHS, otherwise we couldn't re-use that function, because we'd have modified the definition
- Destructive Update breaks Referential Transparency (with respect to the "natural" semantics of Haskell: programs are functions)

Copying and Real-World I/O are Inconsistent

- We cannot implement real-world I/O in a functionally pure language (referential transparent w.r.t. function semantics)
- So pure functional languages are just inellectual toys…
- Real-world function languages (e.g. ML, Lisp, Scheme) are impure so they can
  - Support real-world I/O
  - Allow destructive update for large datastructures