A Key Principle

- Haskell execution replaces sub-expressions, by ones defined to be equal (but hopefully simpler).
- This is an example of a general principle that is very desirable in functional languages — *Referential Transparency*.
- A language is *Referentially Transparent* if
  - replacing an expression by another equal expression does not change the meaning/value of the program as a whole.
  - e.g Given program `2 * sum (3:2:1:[]) + x`, then the following are all equivalent programs:

```
2 * (3 + sum (2:1:[])) + x
2 * (3 + 2 + 1 + 0) + x
2 * 6 + x
12 + x
```

Referential Transparency (Examples)

- Referentially Transparent:
  - A function whose output depends only on its inputs.
  - Expressions built from standard arithmetic operators.
  - None of the above have any "side-effects".
- Referentially Opaque:
  - A function whose value depends on some global variable or state-component.
  - A procedure/function that modifies global state.
  - The assignment statement.
  - A function that performs I/O,
    it depends on the global state of "real world", and modifies it.
  - Most of the above are examples of "side-effects"

Why Referential Transparency matters

- Reasoning about program behaviour is easier
  "substituting equals for equals"
- Code optimization is much simpler
- Scope for code optimization is much greater
- A programming language where every construct is referentially transparent, *w.r.t. to the "obvious" semantics*, is called "*pure*"
  - Haskell (and Clean) are pure functional languages
  - ML, Scheme, LISP are generally considered impure functional languages (they have explicit assignment and I/O side-effects), but this is w.r.t. a simple functional semantics for such languages.

What Referential Transparency isn't

- Referential Transparency does *not* mean:
  - the language is functional
  - the language has no side-effects
- Referential Transparency is a property relating a language and its semantics
  - most languages can be given a semantics that makes them referentially transparent.
  - The issue is one of degree: such a semantics may be very complex.
  - Pure functional languages are referentially transparent w.r.t. a relatively simple and obvious semantics.
  - An imperative language with a *full* semantics is also referentially transparent.

a relatively simple and obvious semantics???

According to Amr Sabry, a purely functional language is one that:

1. is a *conservative extension* of the simply typed lambda-calculus,
2. has well-defined *call-by-value*, *call-by-need*, and *call-by-name* evaluation functions,
3. and all three evaluation functions are *weakly equivalent*.

All will be a little clearer when we see lambda-calculus after Study Week!

P.S. He thinks the notion of "referential transparency" is broken — he has a point!

Further Reading

- From `haskell.org`: `http://www.haskell.org/haskellwiki/Referential_transparency`
- Linked to by the above: `http://www.cas.mcmaster.ca/~kahl/reftrans.html`
- See `http://stackoverflow.com/questions/210835/what-is-referential-transparency` for an interesting discussion.

What's really going on?

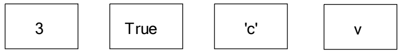
- Haskell as a re-write system makes sense, but ...
- ... how is the rewrite system implemented ?
- We know what purity is, but now we need to understand how it is achieved.
- We need to drill down further into the execution model for Haskell

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.

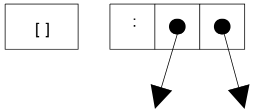
AST Boxes

- Atomic Values and Variables: `3`, `True`, `'c'` `v`



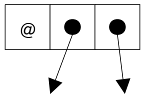
—a variable box holds its name, not its value!

- Data Constructors: `[]`, `:`



The "cons" box has 2 pointers to relevant components

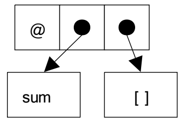
- Function Application:



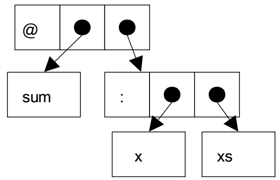
The "apply" box also has 2 pointers to relevant components

AST Examples (I)

- `sum []`

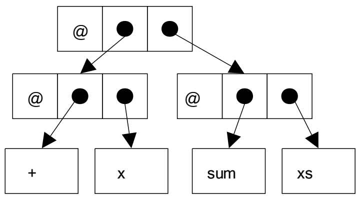


- `sum (x:xs)`



AST Examples (II)

- `x + sum xs`



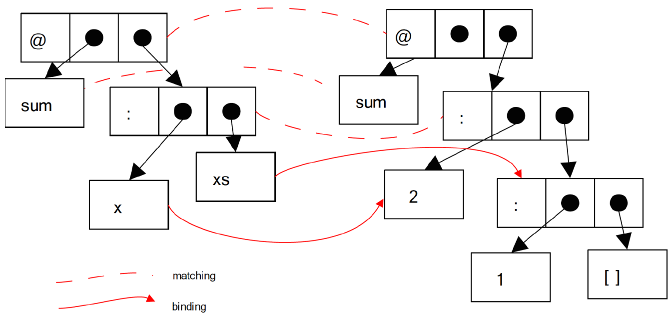
- Note how binary application has a "spine" of 2 @-nodes.
- Remember that Haskell functions are *partially evaluated* so `a + b = (((+) a) b)`

Application Example

- Consider application `sum (2:1:[])`
- The LHS of `sum` is: `sum (x:xs)`
  we match this with `sum (2:1:[])`
  and bind $\mathtt{x} \mapsto 2, \mathtt{xs} \mapsto 1 : []$
  - 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 crucial for referential transparency

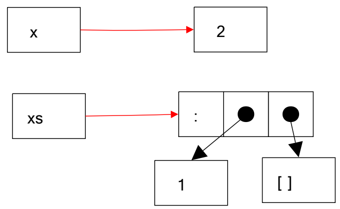
AST Matching

- A successful match using ASTs
  pattern: `sum (x:xs)`
  candidate application: `sum (2:1:[])`



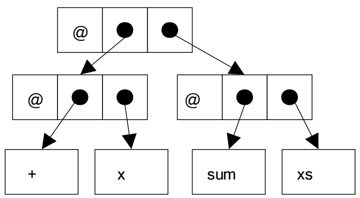
AST Binding

- The bindings from that successful match:
  binding: $\mathtt{x} \mapsto 2, \mathtt{xs} \mapsto 1 : []$

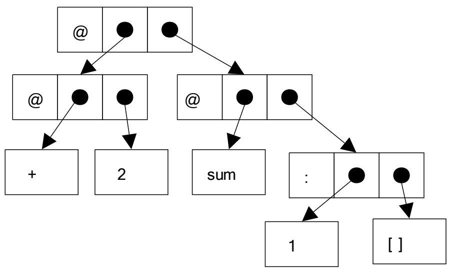


AST Copying

- The RHS from that successful match: `x + sum xs`

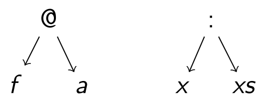


- The *copy* built replacing pattern variables by their bindings:
  copy: `2 + sum (1:[])`

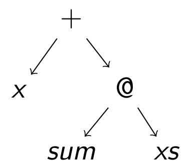


AST Shorthand

- 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:



- So for example, `x + sum xs` now looks like:

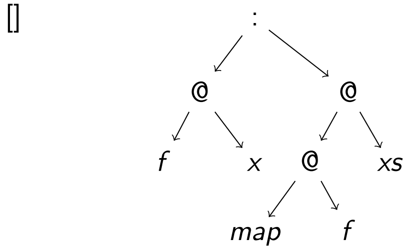


Haskell AST Execution — another example

- The function `map` is defined as follows:

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

- We have the following RHS ASTs:



Map AST Example

Consider application `map inc (1:2:3[])` where `inc x = x+1`

1. We match 2nd case `f` $\mapsto$ `inc`, `x` $\mapsto$ 1, `xs` $\mapsto$ `2:3:[]`
   We build a *copy* of 2nd RHS, using bindings
   `(inc 1) : (map inc (2:3:[]))`

2. We match 2nd case, `f` $\mapsto$ `inc`, `x` $\mapsto$ 2, `xs` $\mapsto$ `3:[]`
   We build a *copy* of 2nd RHS, using bindings
   `(inc 1) : ((inc 2) : (map inc (3:[])))`

3. We match 2nd case, `f` $\mapsto$ `inc`, `x` $\mapsto$ 3, `xs` $\mapsto$ `[]`
   We build a *copy* of 2nd RHS, using bindings
   `(inc 1) : ((inc 2) : ((inc 3) : (map inc [])))`

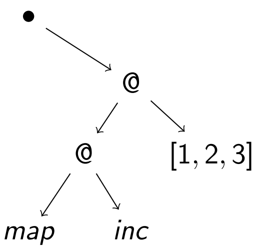
4. We match 1st case, `f` $\mapsto$ `inc`
   We build a *copy* of 2nd RHS, using bindings
   `(inc 1) : ((inc 2) : ((inc 3) : []))`

The Importance of Copying (I)

- We clearly need to copy the function RHS, otherwise we couldn't re-use that function, because we'd have modified the definition.
- But in the application `map inc [1..3]` we not only copied the RHS, but that built us a *copy* of the original argument list.
- Couldn't a smart implementation realise that the copies simply had the leaves changed from `x` to `inc x`, and change these in place
  (so-called "destructive update")?

Before evaluating `map inc [1..3]`

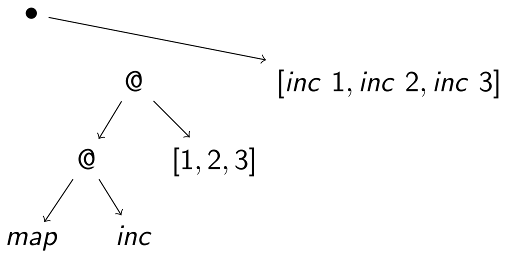
- We have the application as an AST (simplified)



- • denotes a pointer to the expression `map inc [1,2,3]` from whatever contains that expression.
- (We show the original list as one lump)

How Haskell does `map inc [1..3]`

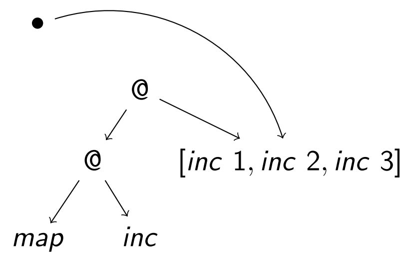
- We build a copy, and swing our pointer to indicate that copy



- 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(?) `map inc [1..3]`

- We might suggest that we update the list in place and swing our application pointer to indicate that update:



- we don't alter the `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 (w.r.t. the "natural" semantics of Haskell: programs are functions.)
- Consider the following program:

```
myfun xs = (xs,map inc xs)
```

  We have paired together references to both the original `xs`, and the result of mapping `inc` across it.
- If we use copying, then the two lists returned by `myfun` are different
- If we use destructive update, then the two lists returned by `myfun` are equal.
  - but this means that `xs` and `map inc xs` are the same, which is clearly wrong.

Copying as a show-stopper (I)

- Imagine that `bigds` is a very large datastructure and `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:

```
let bigds1 = bigmod p1 bigds
    bigds2 = bigmod p2 bigds1
    ...
    bigdsn = bigmod pn bigdsn
in ...
```

- So pure functional languages are not good for implementing large databases, processing large amounts of data, supporting design of large artifacts (i.e VLSI chips), ...?

Copying as a show-stopper (II)

- Assume `fwrite f d` writes data `d` to file named `f` and returns a status value
  and `fread f` returns data read from file named `f`
- We cannot use copying to implement the following behaviour:

```
let d1 = fread "in1.dat"
    s2 = fwrite "out1.dat" (myfun d1)
    d3 = fread "out1.dat"
    s4 = fwrite "out1.dat" (another fun d3)
in ...
```

  (think multipass compiler...)
- Why not? Because `fwrite` modifies the file-system of the computer on which it runs — and we *cannot* copy that !
- The side-effects in I/O are inherent, and we cannot "implement them away".

Copying and Real-World I/O are inconsistent

- We cannot implement real-world I/O in a functionally pure language (referentially transparent w.r.t. function semantics)
- 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 ...