Another look at declarative debugging for Haskell

Bernie Pope

VASET, Friday 30 Oct 2015

. . . programmers who write debugging systems wrestle with the problem of providing a proper vantage point.

Reflection and Semantics in Lisp, Brian Cantwell Smith, 1984

What's the problem

- Haskell has many admirable qualities, but it lacks usable debugging tools (despite lots of effort).
- A breakpoint debugger is built into GHCi, but, as expected it is difficult to use:
    - Lazy evaluation
    - Higher-order functions

Lazy evaluation

Consider the map function on lists:

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

The body of the recursive case contains two function applications

```
f x
map f xs
```

Their evaluation fate is determined outside the definition of map.

Lazy evaluation

In lazy languages a **function call** consists of two distinct events:

1. function application (yielding a thunk)
2. reduction (thunk is evaluated to WHNF)

These two events can occur in disparate contexts, dependent on the dynamic properties of the computation.

Lazy evaluation

It can be difficult to relate these two things:

- when (and it what context) reduction happens
- the static description of the program

Higher-order functions

Consider parsers written in the combinator style:

```
newtype Parser a = P (String -> (String, a))

parseExp :: Parser Exp

parseExp = parseInt <|> parseFloat <|> parseParenExp
```

What does `parseExp` do?

Higher-order functions

Conceptually we might think that `parseExp` parses expressions by recognising integers, floats and parenthesised expressions.

However, from a reduction perspective, it does very little except build a function from other functions. The action of parsing happens elsewhere.

What's the problem?

- Haskell encourages - and benefits from - **declarative** reasoning.
- However, traditional breakpoint debuggers impose an operational perspective on computation.
- Therefore it is hard to apply traditional debugging techniques to Haskell because programmers do not (and really cannot) think operationally about Haskell programs.

What I want is Buddha or Hat. They let you evaluate a program, and then drill into subcomputations until you find the base case that is causing your wrong answer.

Because Haskell programs are typically not about imperative, 'steps.' They are about recursive decomposition into subproblems. You need a debugger that lets you inspect that structure.

Posted on reddit /r/haskell, 2014

Declarative debugging

- Computations are represented as trees (or perhaps directed graphs) - **Evaluation Dependency Tree (EDT)**.
- Nodes contain computation steps (such as reductions).
- Edges represent **evaluation dependency**.

Declarative debugging

- An error diagnosis is applied to the tree in search of **buggy nodes**.
- A node is **erroneous** if it contains a computation step which does not agree with the **intended meaning** of the program.
- A node is buggy if:
    - it contains an erroneous computation step
    - it does not depend on any erroneous children nodes

Declarative debugging

- An **oracle** judges the correctness of nodes. The oracle knows the intended meaning of the program.
- A node can be:
    - correct
    - incorrect
    - inadmissible

Intended meaning

- The intended meaning of a program explains what a program is supposed to do.
- It is defined over the (let-bound) variables and data constructors from the program source.

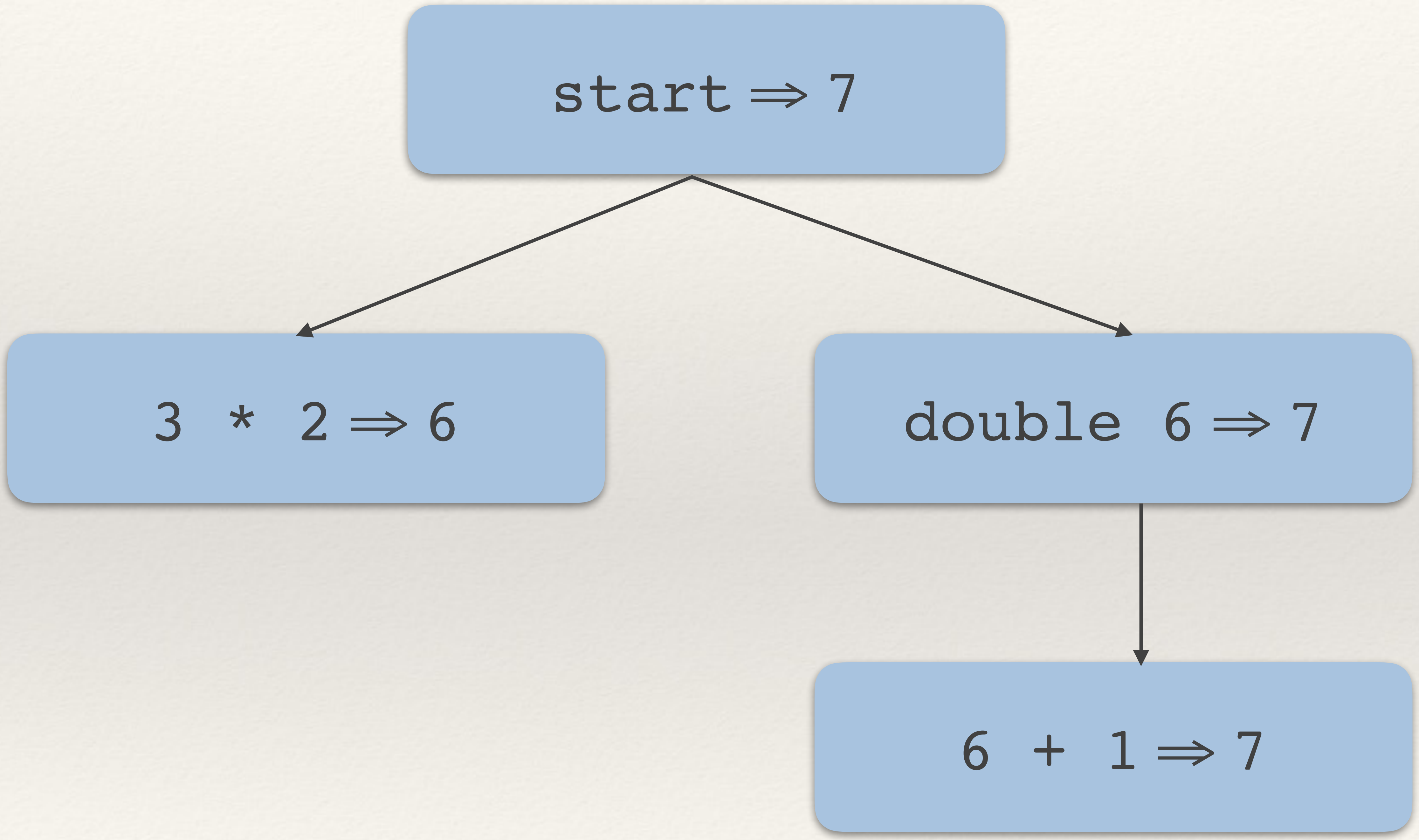
Evaluation dependency tree

Consider this small buggy program:

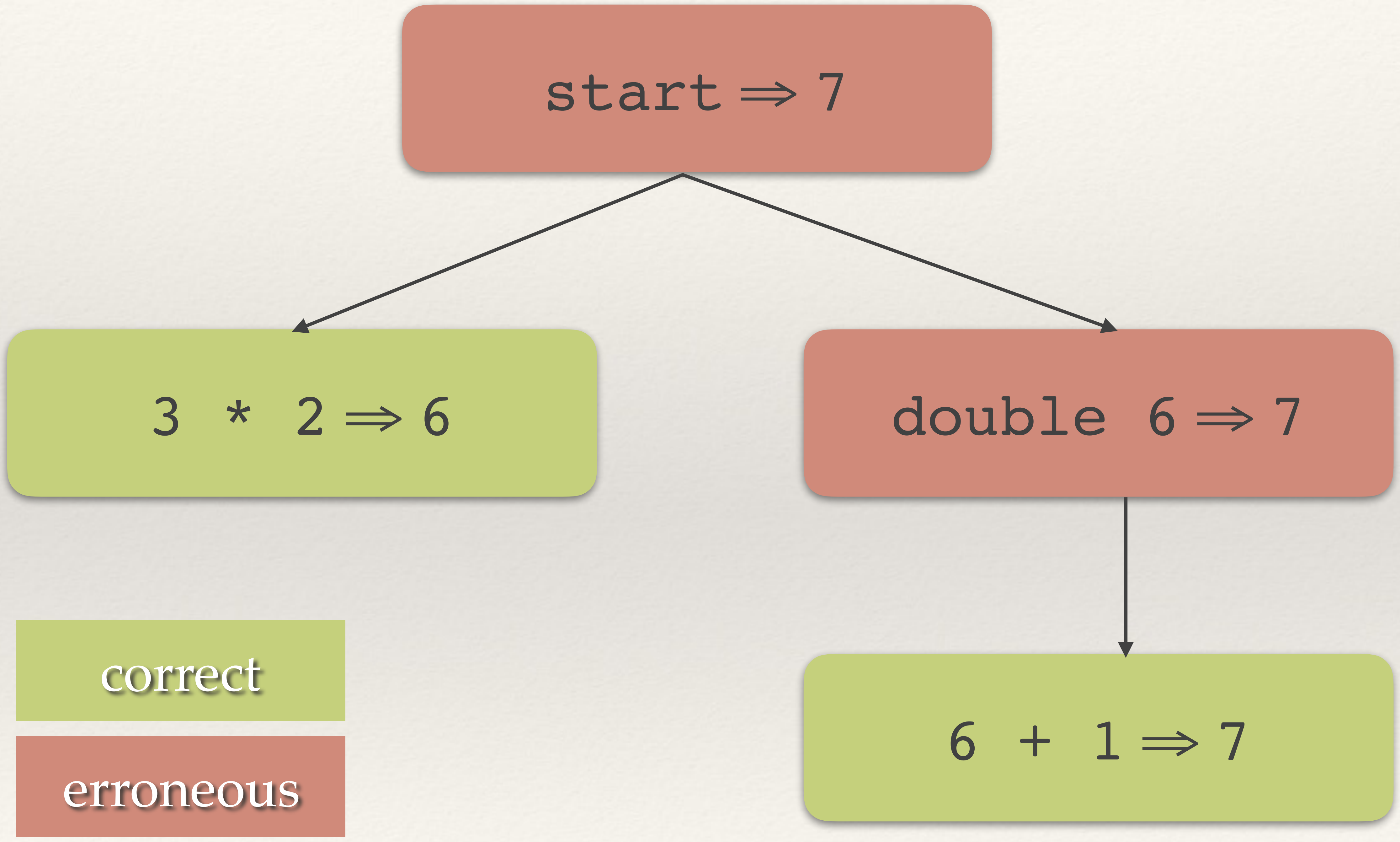
```
  double x = x + 1
  start = double (3 * 2)
```

The evaluation of `start` produces 7, when we expect it to produce 12.

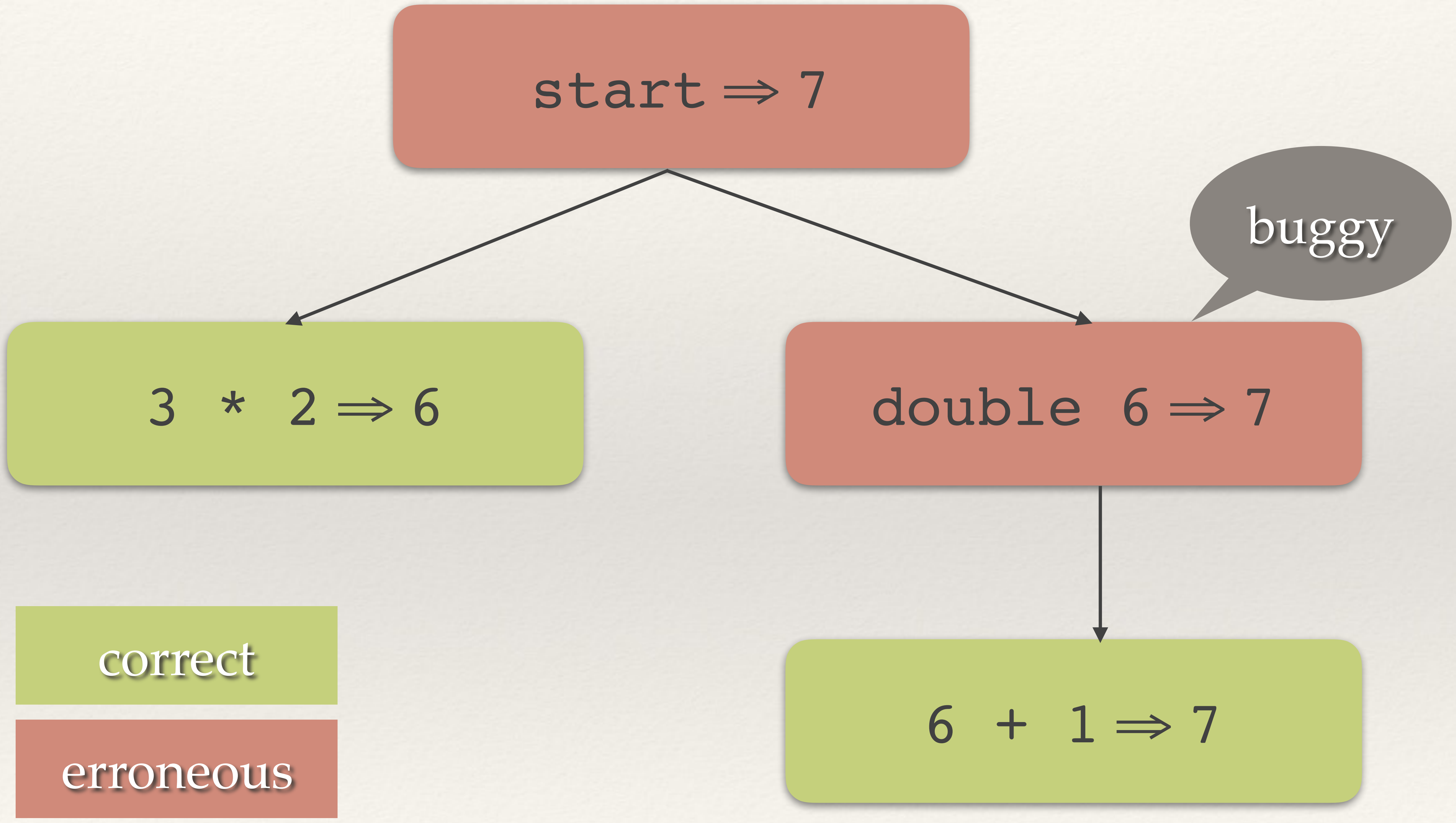
Evaluation dependency tree



Evaluation dependency tree



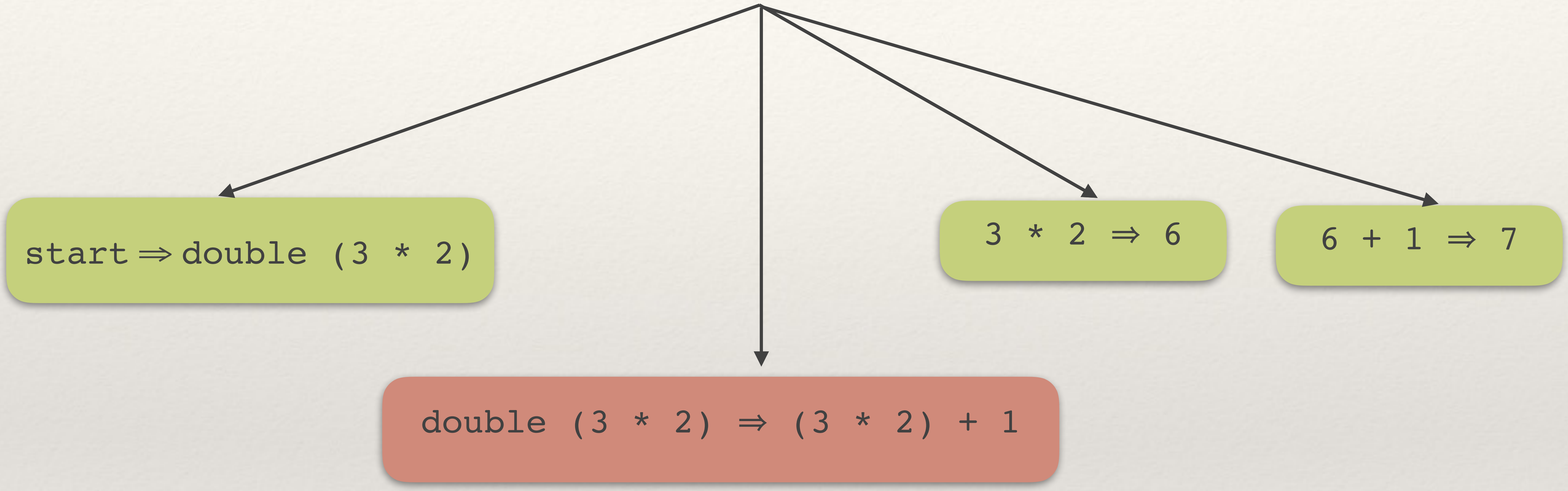
Evaluation dependency tree



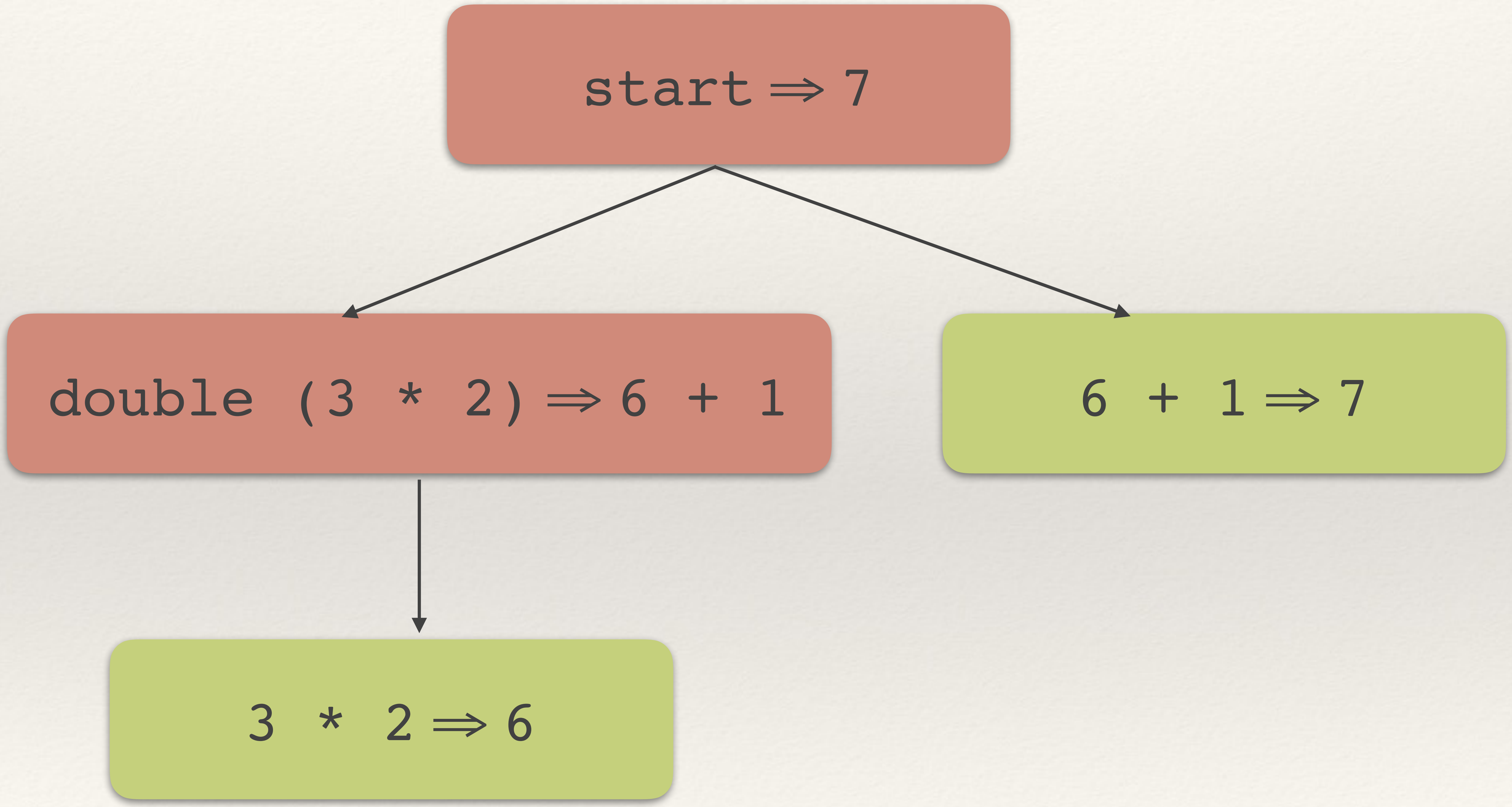
Evaluation dependency tree

- Nodes in the EDT in the previous example contain **big-step** reductions, making it a big-step EDT.
- Big-step EDTs are traditional for declarative debugging.
- However, the declarative debugging algorithm does not require big-step trees.

Small-step EDT



Mixed-step EDT



Which EDT is best?

- Big-step EDTs are often the easiest to understand because arguments and results are shown in their **most evaluated form**.
- But this may not always be true.

Bigger-step EDTs

- For higher-order functions it sometimes make sense to consider all the reductions of a function together:
- Compare:

```
map (plus 1) [1, 2] ⇒ [2, 3]

map {1 ⇒ 2, 2 ⇒ 3} [1, 2] ⇒ [2, 3]
```

Bigger-step EDTs

- For higher-order functions it sometimes make sense to consider all the reductions of a function together:
- Compare:

```
map (plus 1) [1, 2] ⇒ [2, 3]

map {1 ⇒ 2, 2 ⇒ 3} [1, 2] ⇒ [2, 3]
```

These require different shaped EDTs

Bigger-step EDTs

- For higher-order functions it sometimes make sense to consider all the reductions of a function together:
- Compare:

```
map (plus 1) [1, 2] ⇒ [2, 3]
map {1 ⇒ 2, 2 ⇒ 3} [1, 2] ⇒ [2, 3]
```

This is not a term

Some Observations

- The formalisation of declarative debugging for Haskell could do some refinement - currently tied to big-step EDTs.
- Evaluation dependency is not well defined in terms of Haskell's semantics (also Haskell does not have a official, precise semantics).
- We ought to be able to develop a theory in which all step sizes of EDTs are correct and inter-convertible.
- The theory needs to be grounded in a semantics for Haskell.
- The small-step EDT is essentially a reduction trace of the computation.

buddha was radically better than the GHCi debugger

Posted on haskell-irc channel in 2010

What's the problem?

- If declarative debugging is so great why doesn't a usable debugger exist already?

What's the problem?

- Tools such as Hat and Buddha are **post mortem**.
- Debugging only happens after the computation has finished.
- This requires the whole computation history to be recorded and saved.
- For non-toy examples, this can be HUGE.

What’s the problem?

- You can write it to secondary storage (like HAT), but that doesn’t buy you much in the long run, and slows things down considerably.
- The Mercury declarative debugger recomputed subtrees during debugging, but that is difficult/ineffective with lazy evaluation, and requires tabling of I/O.

Possible solution

- It is desirable to interleave computation and debugging.
- We can compute a small-step trace of the computation in a bounded-size memory buffer.
- When the buffer is full, debugging is initiated.
- A partial EDT can be reconstructed from the small-step trace.

Possible solution

- The current debugging session may reach a fringe node of the partial EDT.
- The computation can be resumed to fill-in more of the tree.
- The partial EDT must be continually pruned to keep space usage under control.

Conjecture

- A more thorough formalisation of the EDT will:
    - explain the correctness of declarative debugging for Haskell
    - provide more flexible EDT structure (we can re-shape the tree during debugging)
    - provide the foundation for interleaving program execution and debugging

A possible way forward

- Start with the STG machine
  - is used by GHC
  - has an operational semantics
- Extend the semantics to provide a program trace
- Mini-STG might be a good starting point:

  https://wiki.haskell.org/Ministg