Simple graph reduction with visualisation

Outline

- The miniFP language
- Graph representation of terms
- Tying the knot
- The compiler
- The reduction engine
- Literature

The miniFP language

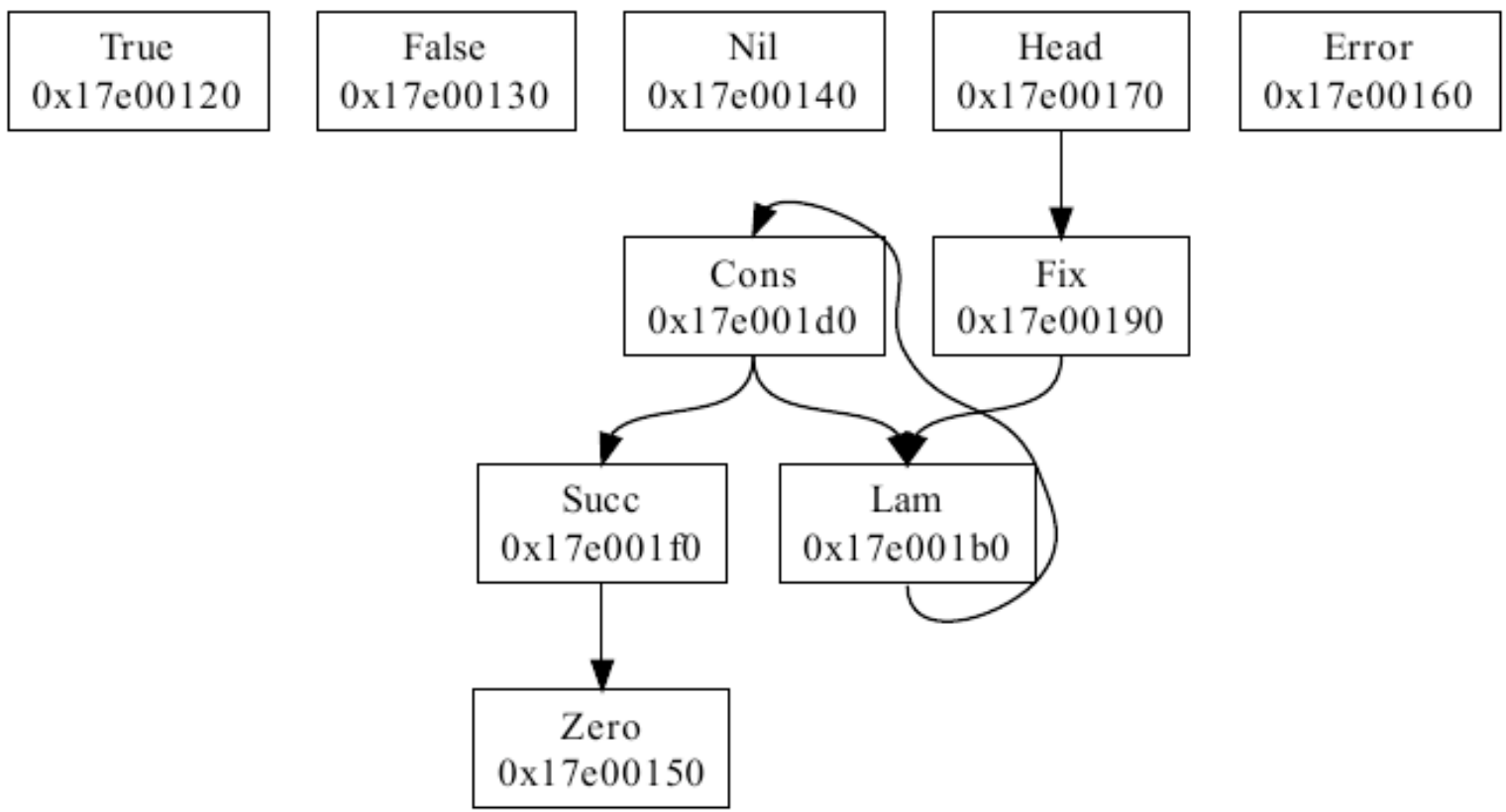
- Small functional language, intended for teaching. Ugly syntax borrowed from Schwartzbach's notes (easy to parse, but annoying to use).
- Hindley/Milner style type inference, with let polymorphism.
- Data types: Bool, Int, List, ->
- No user defined types.
- Explicit fixed-point operator (fix).
- Minimal primitives (head, tail, cons, nil, zero, true, false, pred, succ, isZero, null, if-then-else).

The miniFP language

```
let map = fix (\m -> \f -> \list ->
    if null list
       then nil
       else cons (f (head list)) (m f (tail list))
    fi
    end end end)
in map (\x -> succ x end) [0,1,2] end
```

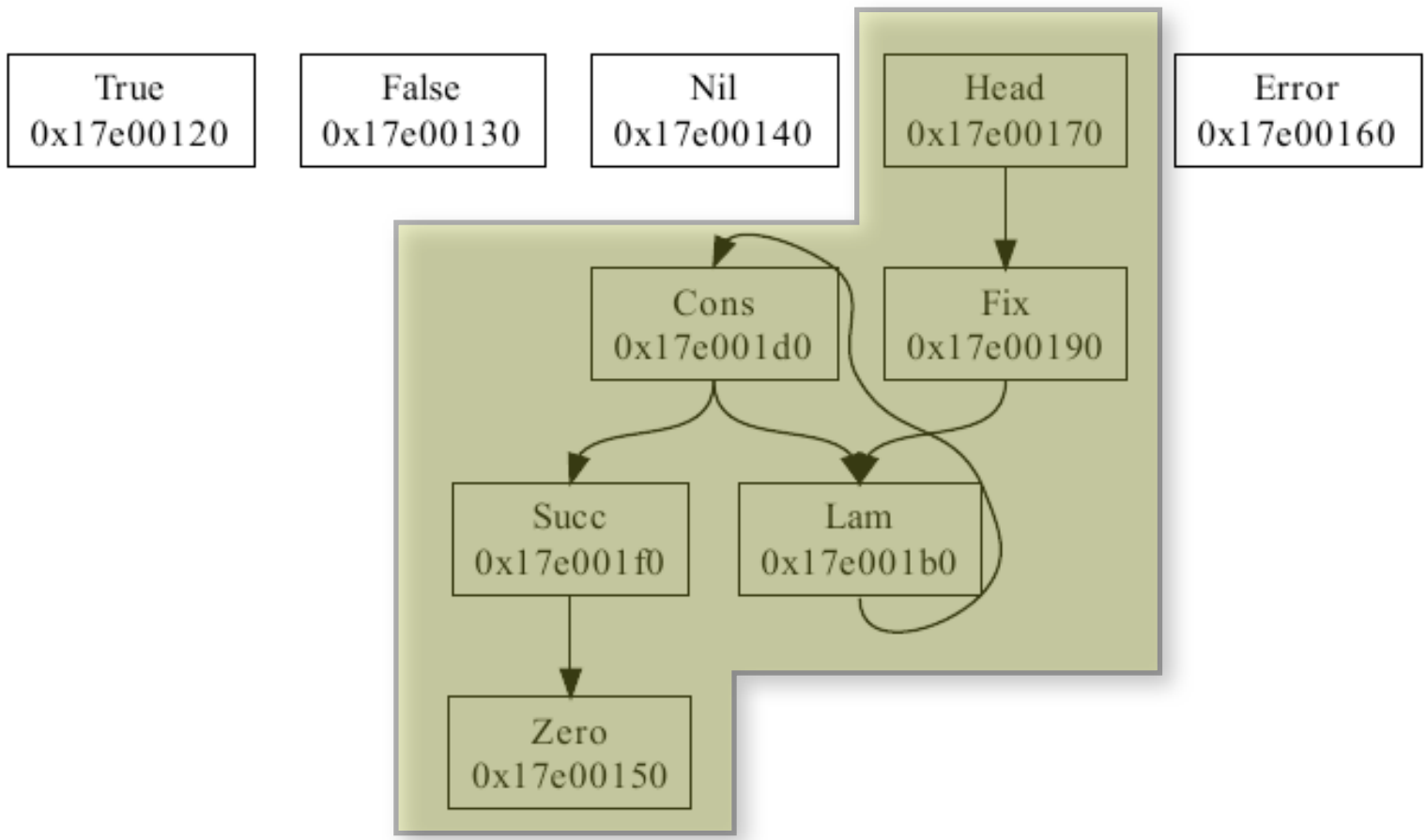
Graph representation of terms

```
head (fix (\rec -> cons 1 rec end))
```



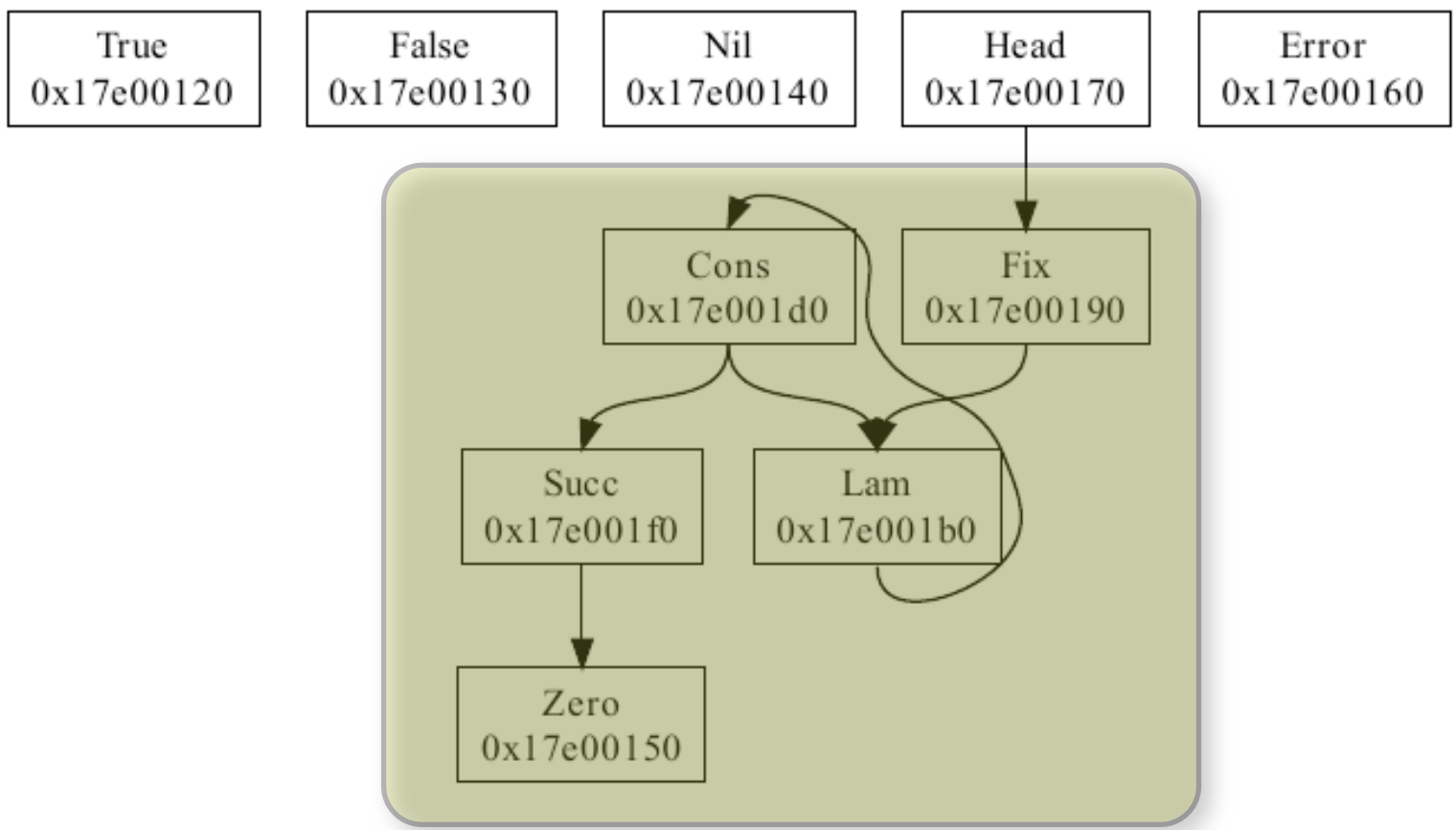
Graph representation of terms

```
head (fix (\rec -> cons 1 rec end))
```



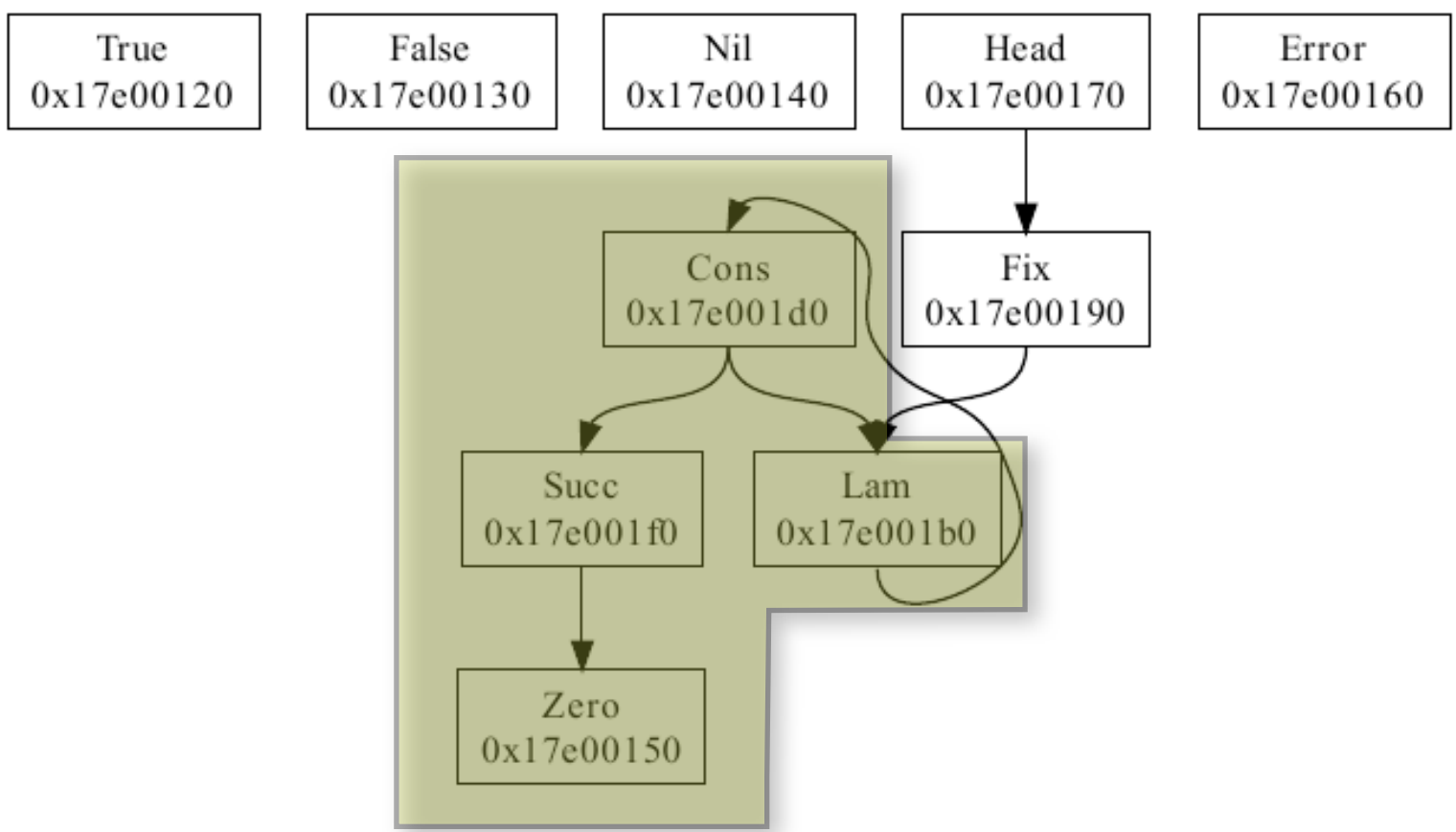
Graph representation of terms

```
head (fix (\rec -> cons 1 rec end))
```



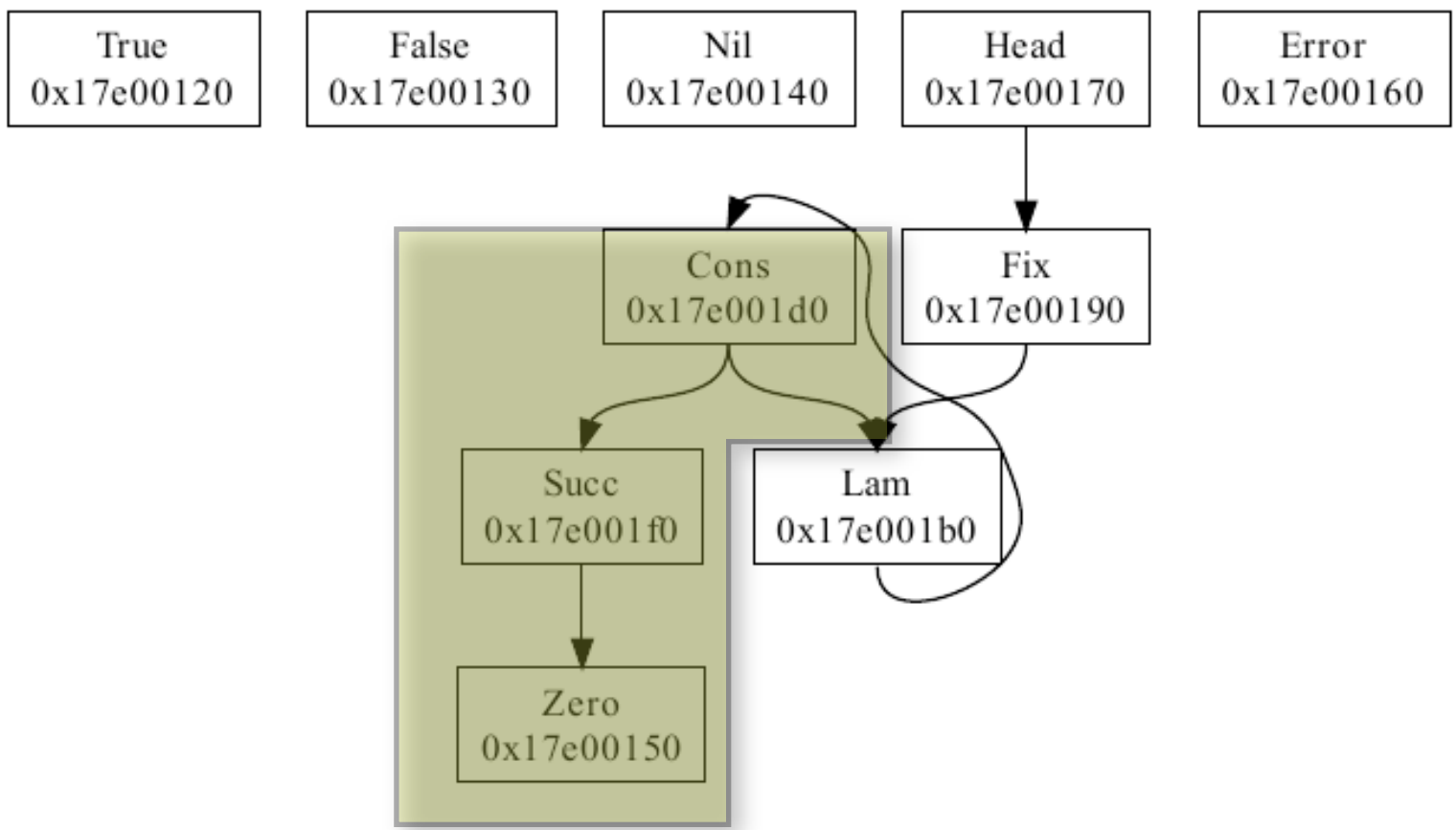
Graph representation of terms

```
head (fix (\rec -> cons 1 rec end))
```



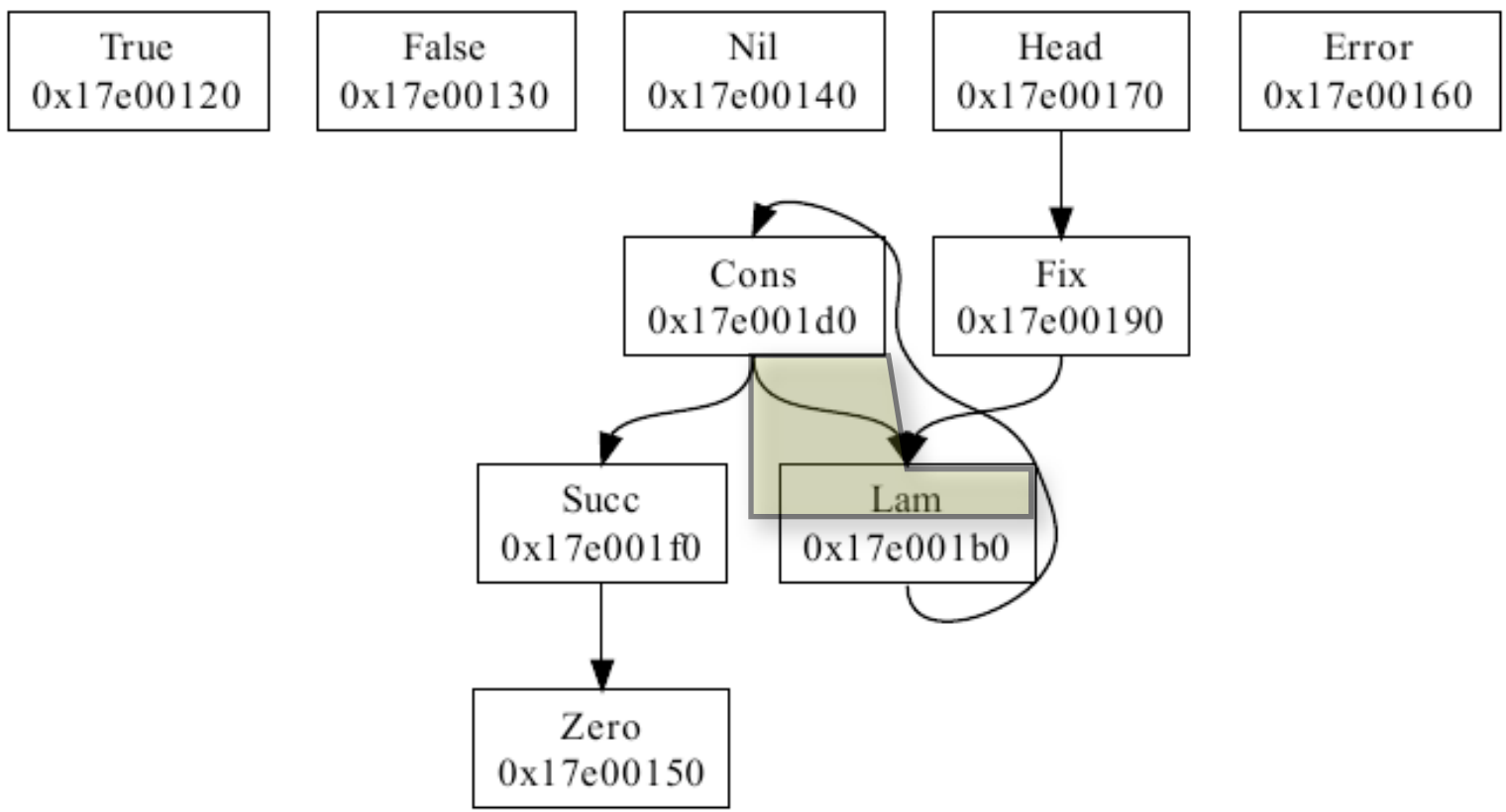
Graph representation of terms

```
head (fix (\rec -> cons 1 rec end))
```



Graph representation of terms

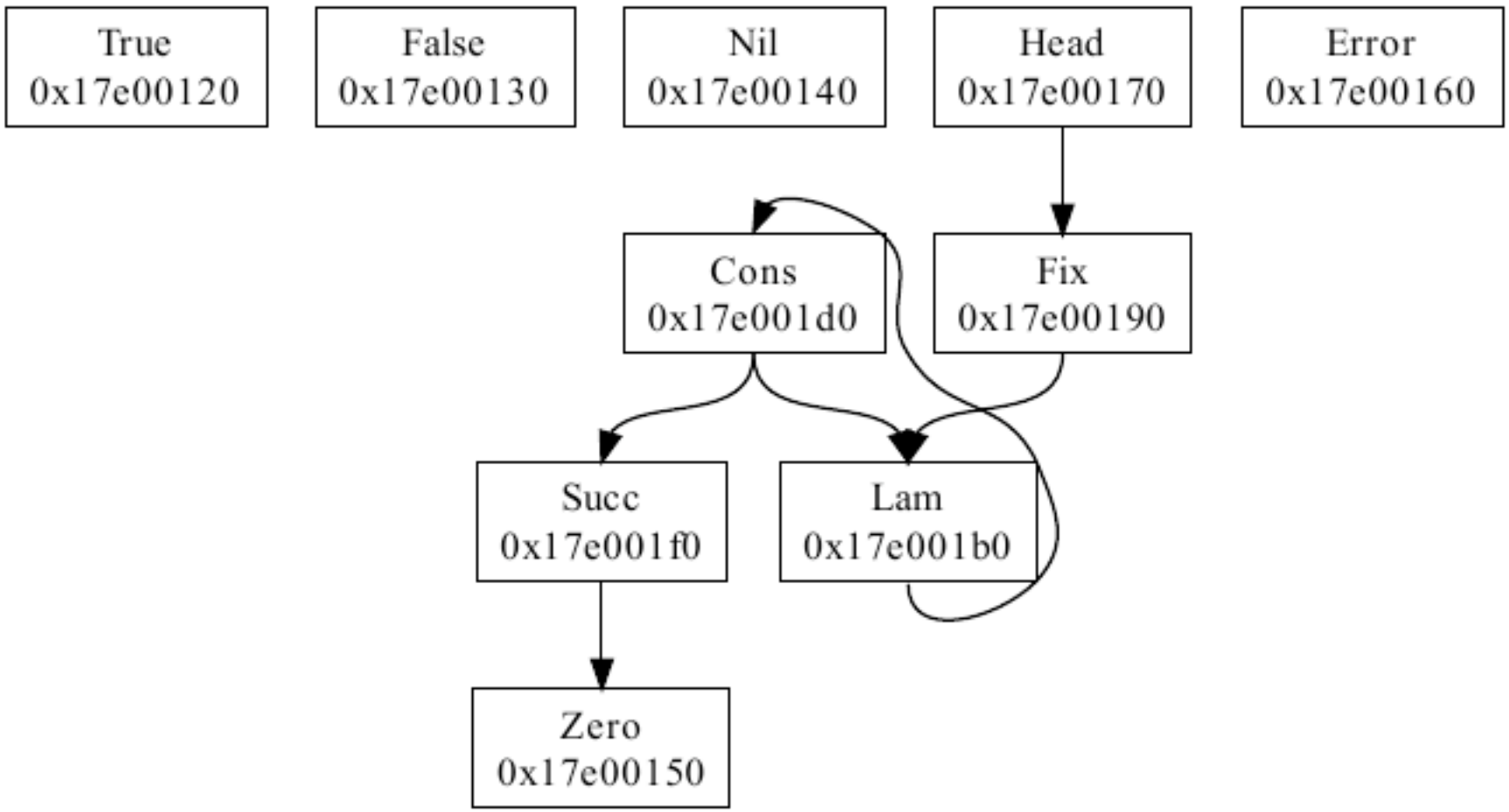
```
head (fix (\rec -> cons 1 rec end))
```



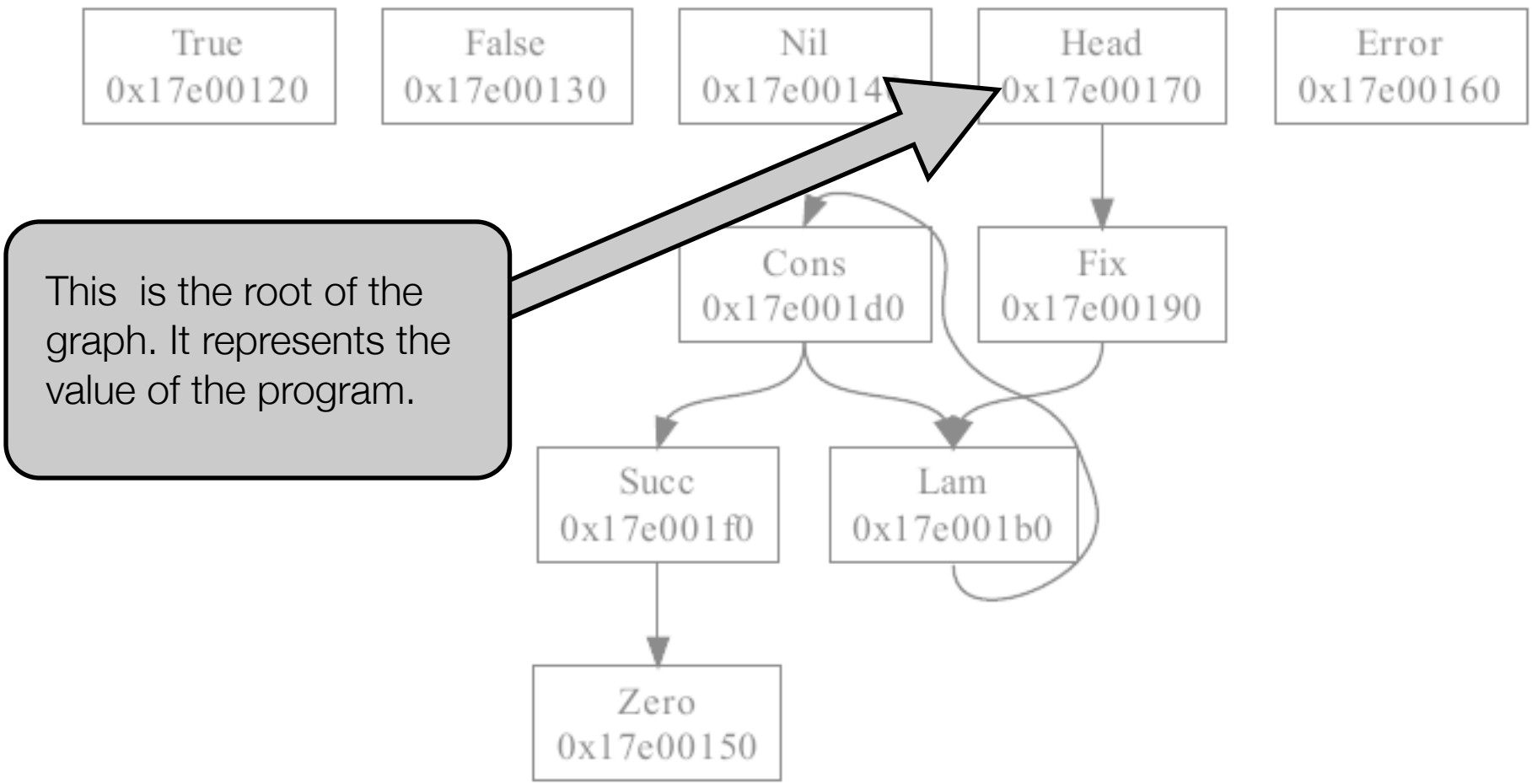
Tying the knot

- Step through the reduction of: `head (fix (\rec -> cons 1 rec end))`
- Observe the reduction of the fixed point carefully.
- The “infinite” list of ones becomes a cyclic graph.
- Normal order reduction ensures that the program terminates.
- Diagrams generated by GraphViz. http://graphviz.org/
- The compiled program writes “dot graph” files at each important step in reduction.

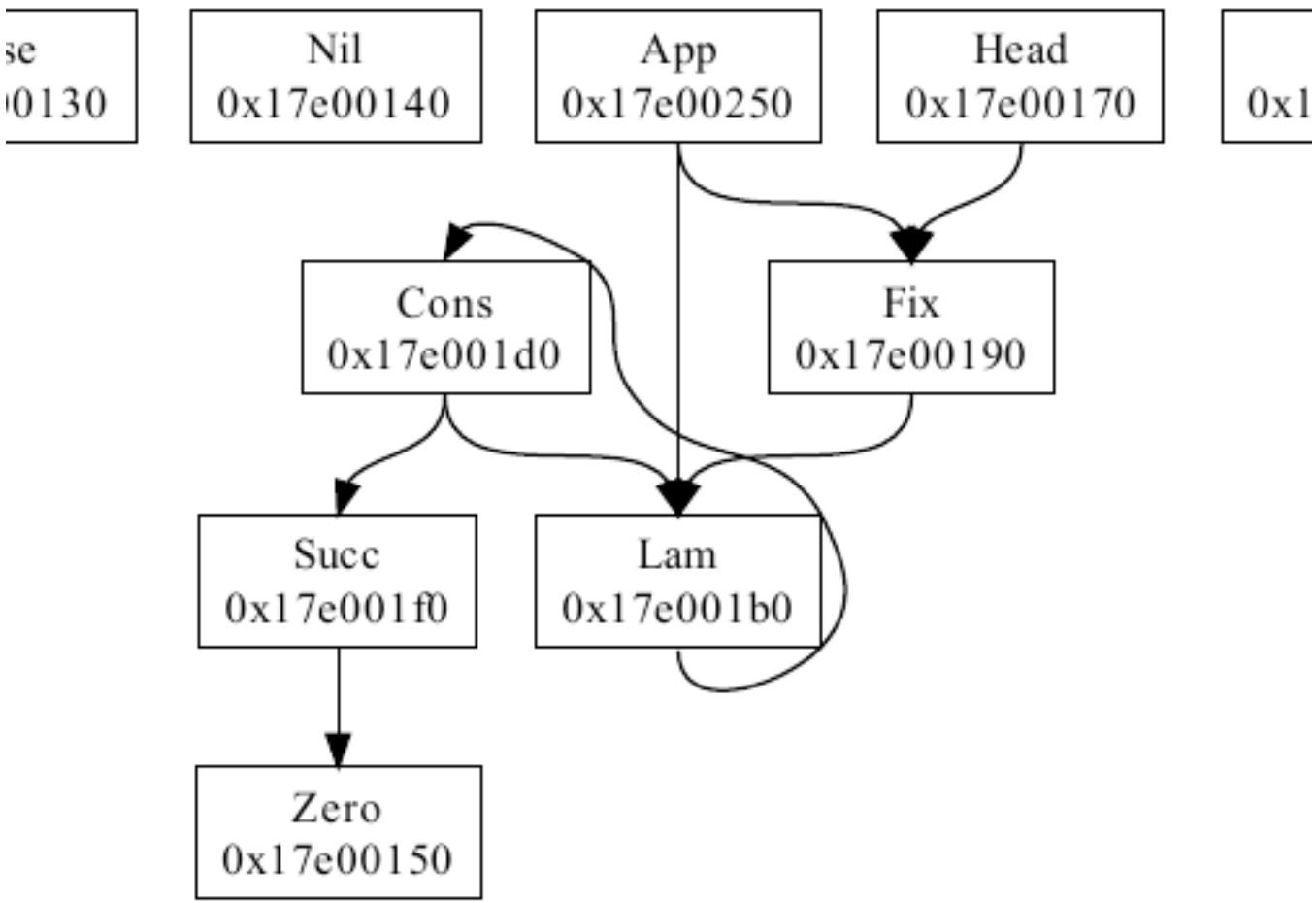
Step 1. The initial graph



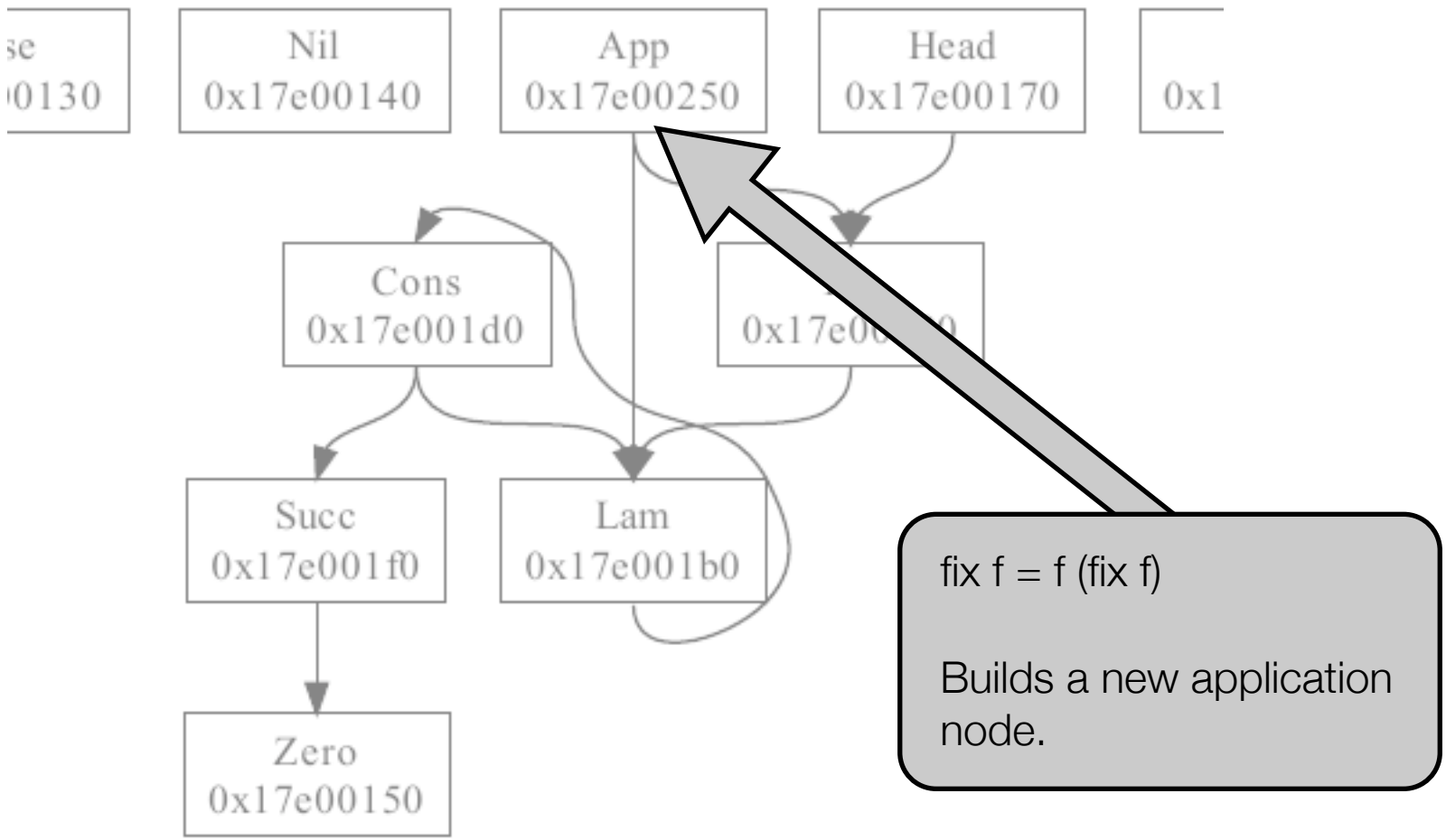
Step 1. The initial graph



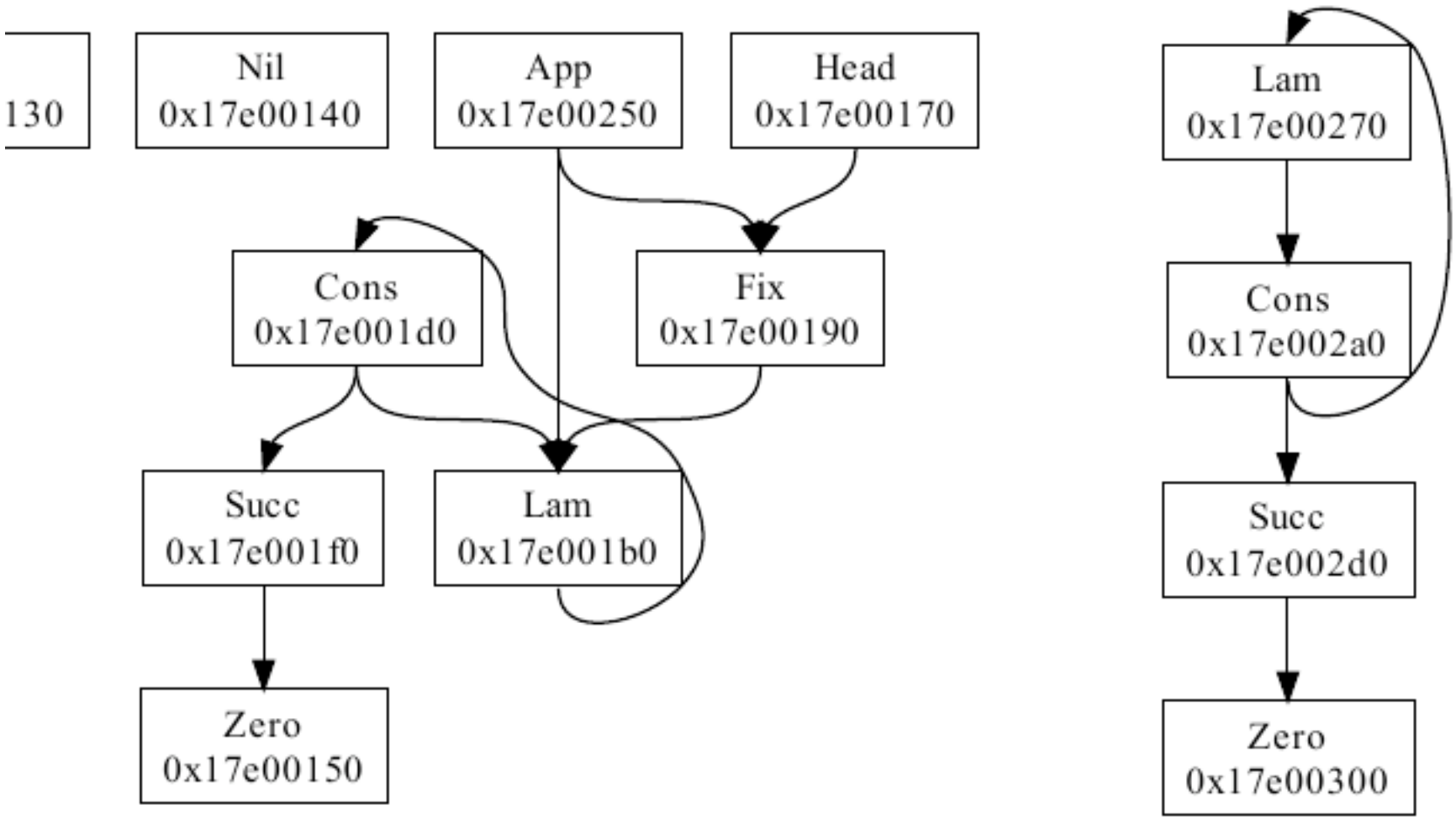
Step 2. unfold the fixed point



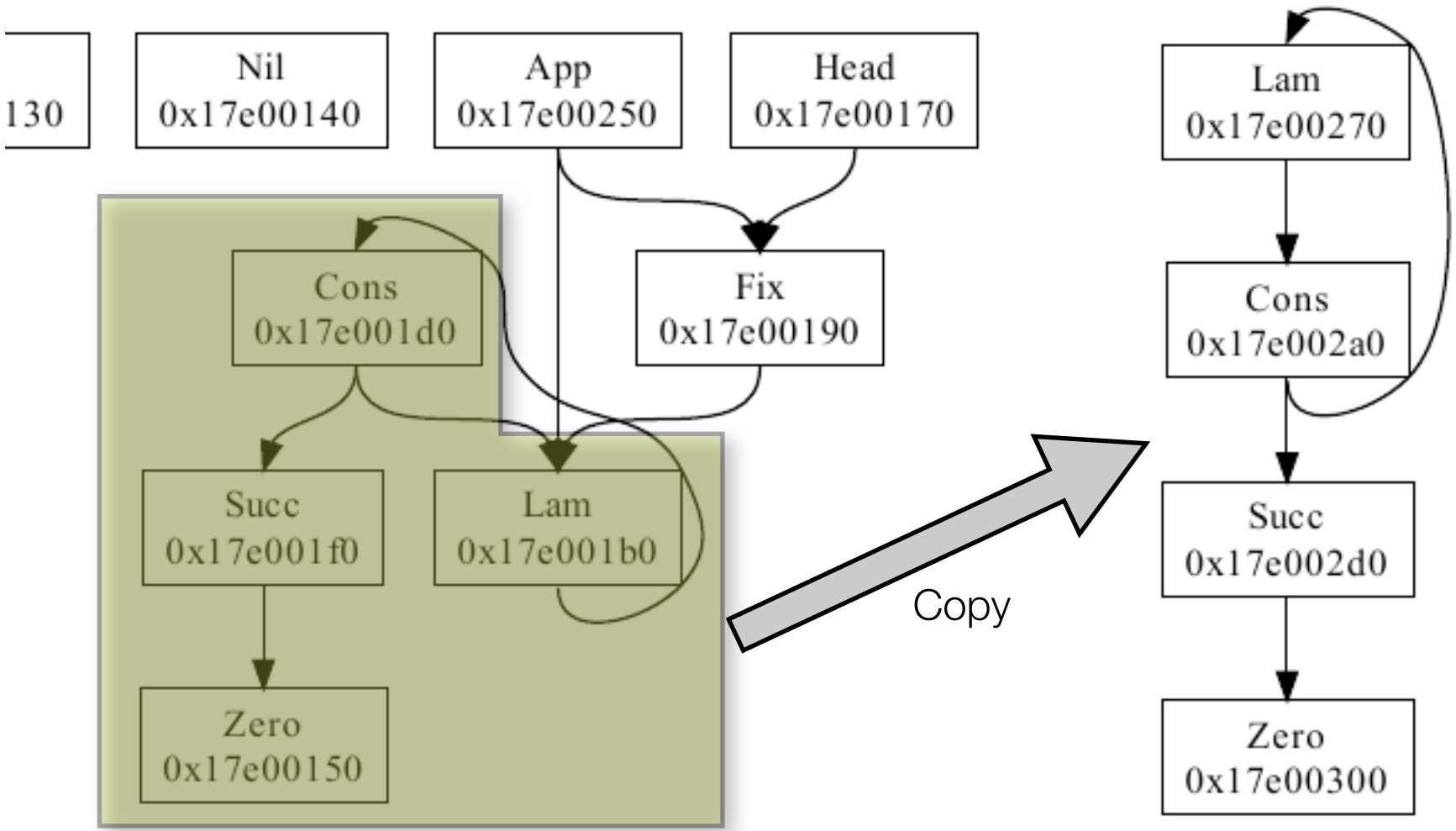
Step 2. unfold the fixed point



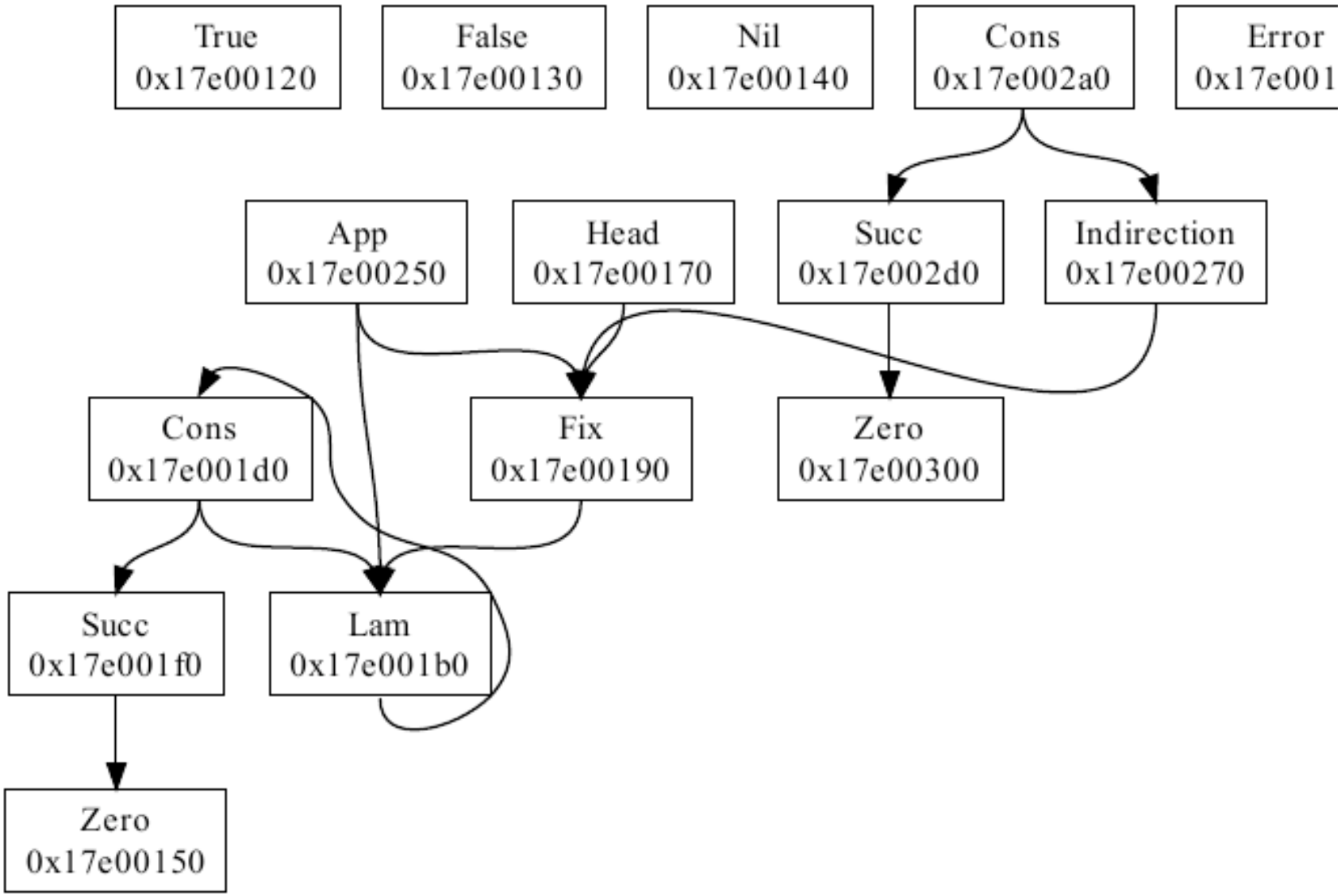
Step 3. copy the left child of the application



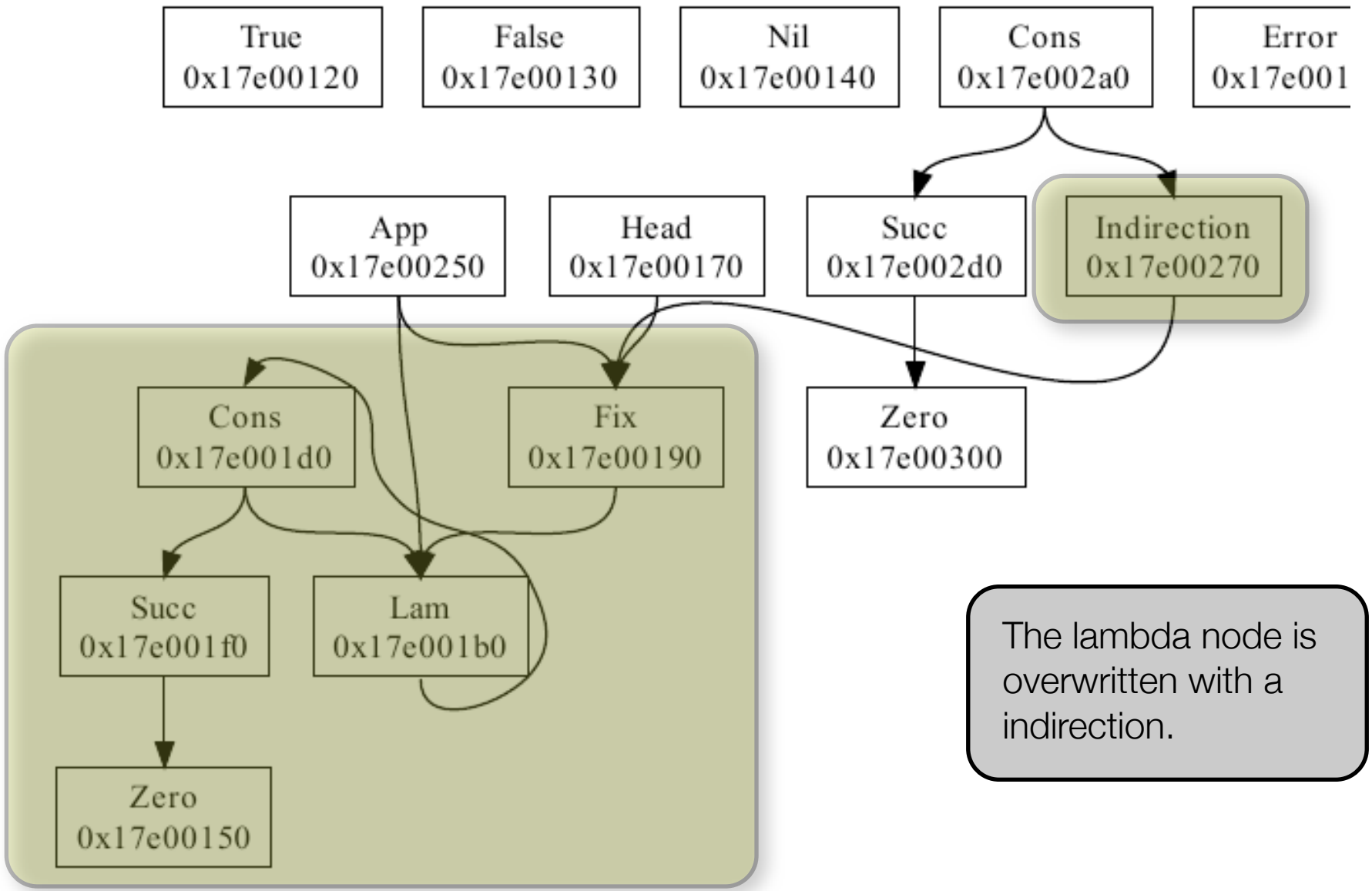
Step 3. copy the left child of the application



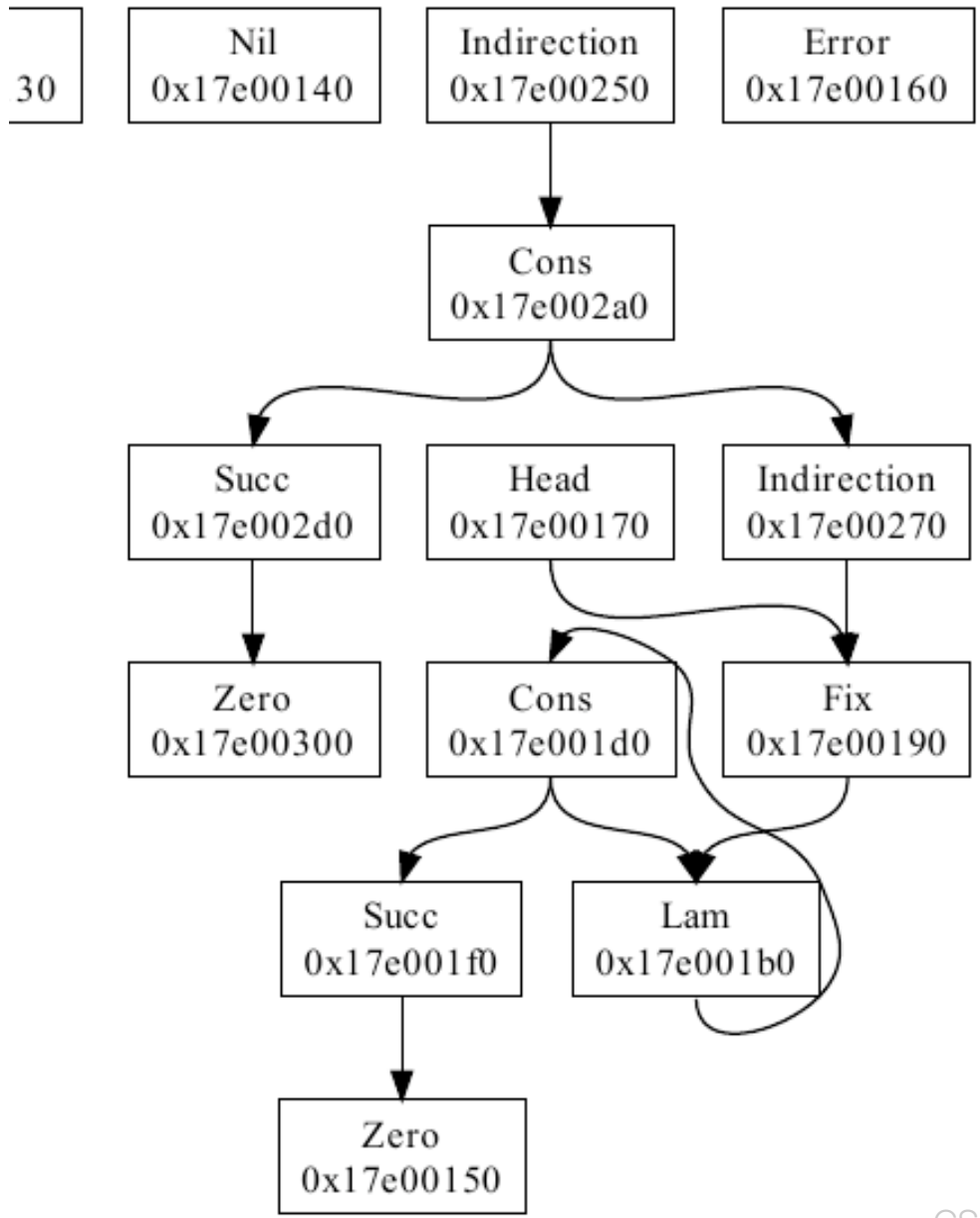
Step 4. substitute the parameter with the argument



Step 4. substitute the parameter with the argument

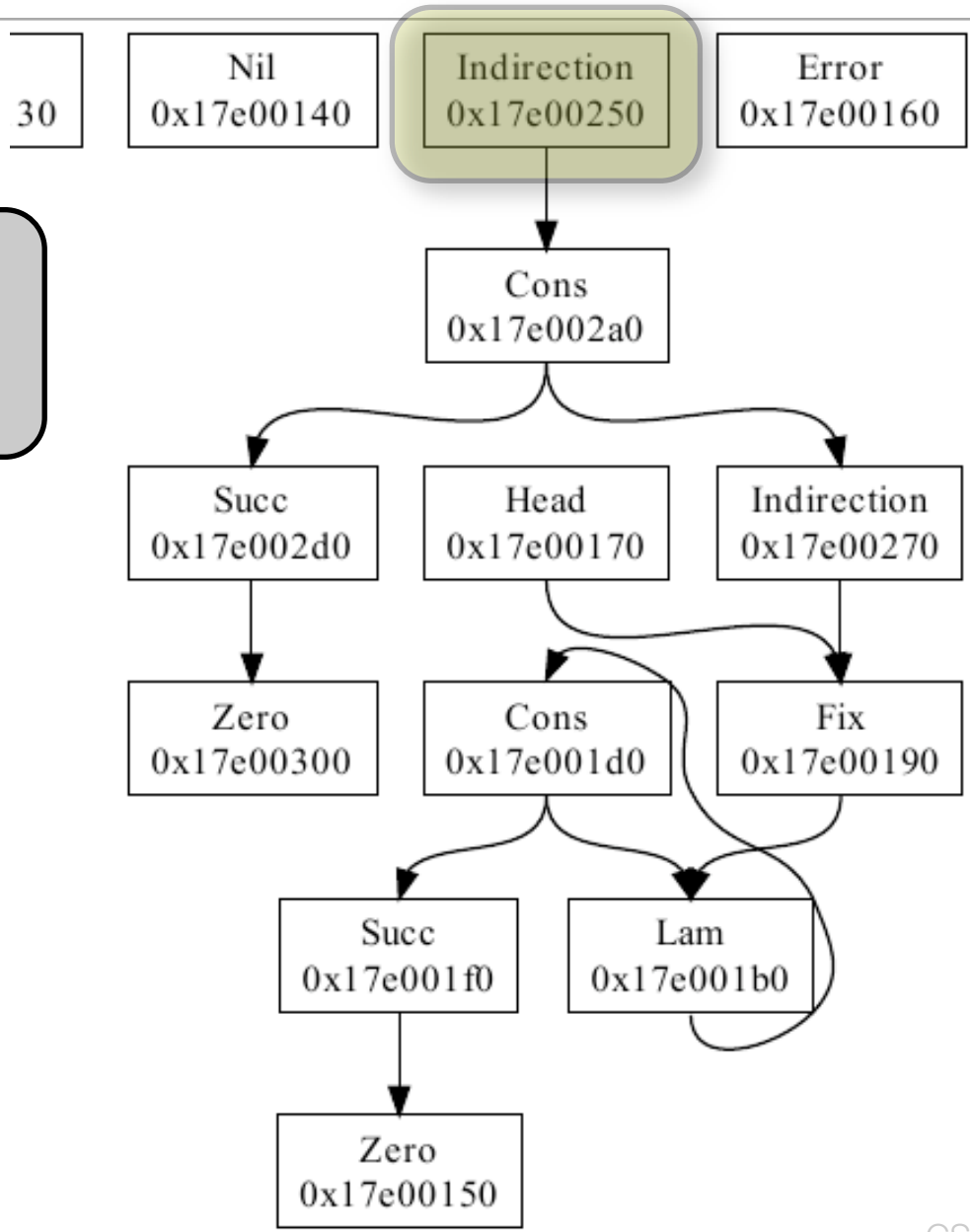


Step 5. overwrite the application node with its result

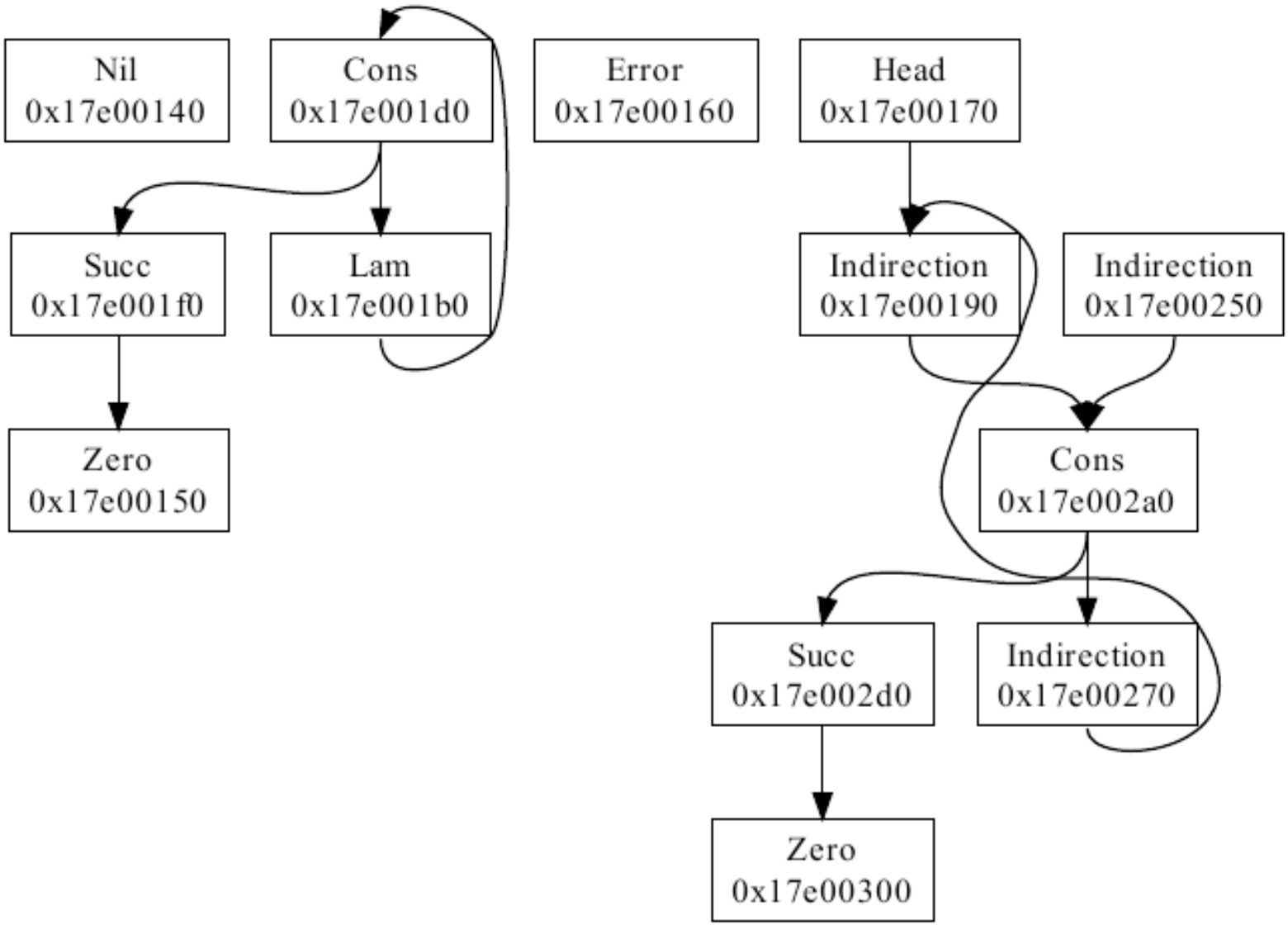


Step 5. overwrite the application node with its result

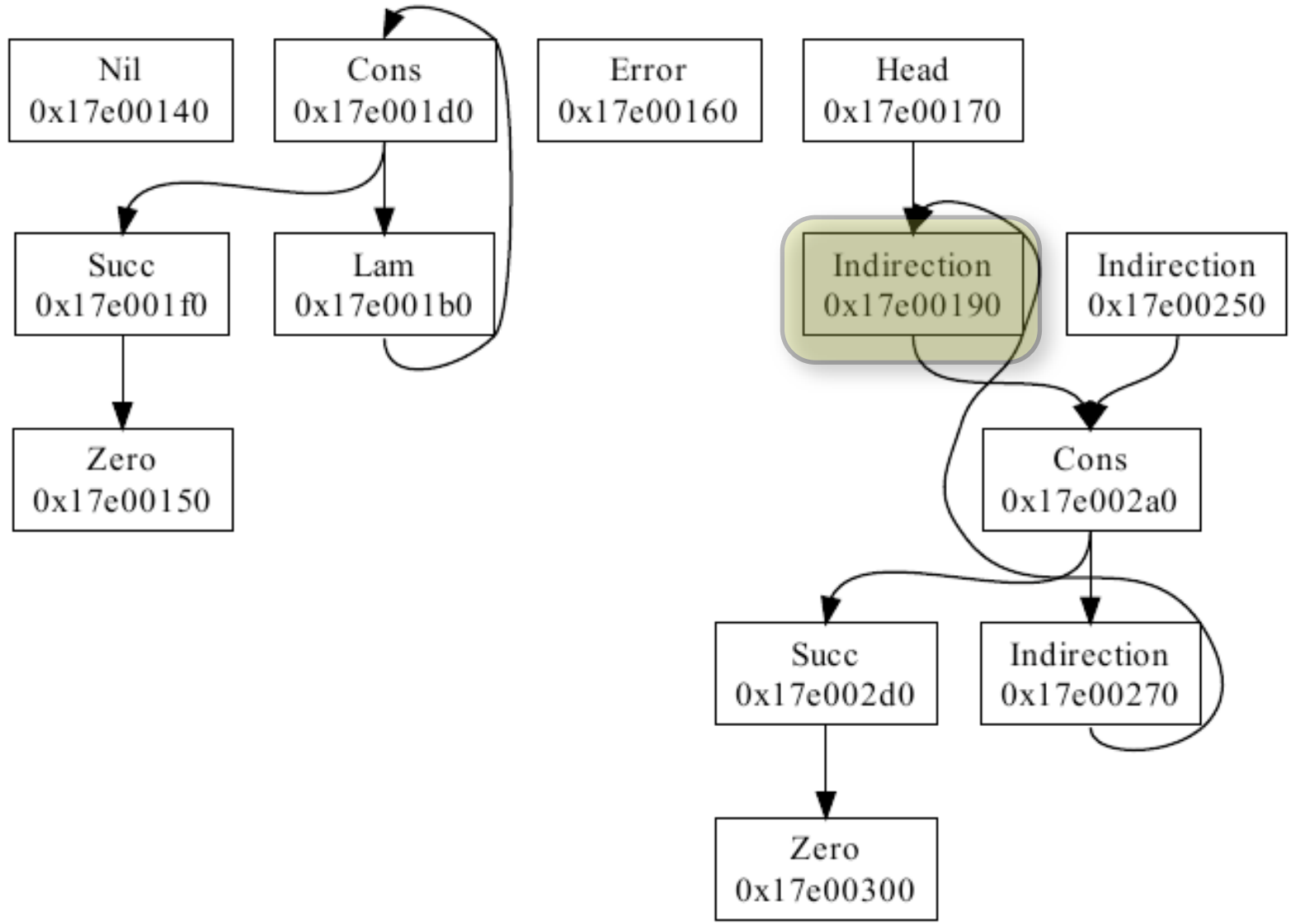
The application node is overwritten with a indirection.



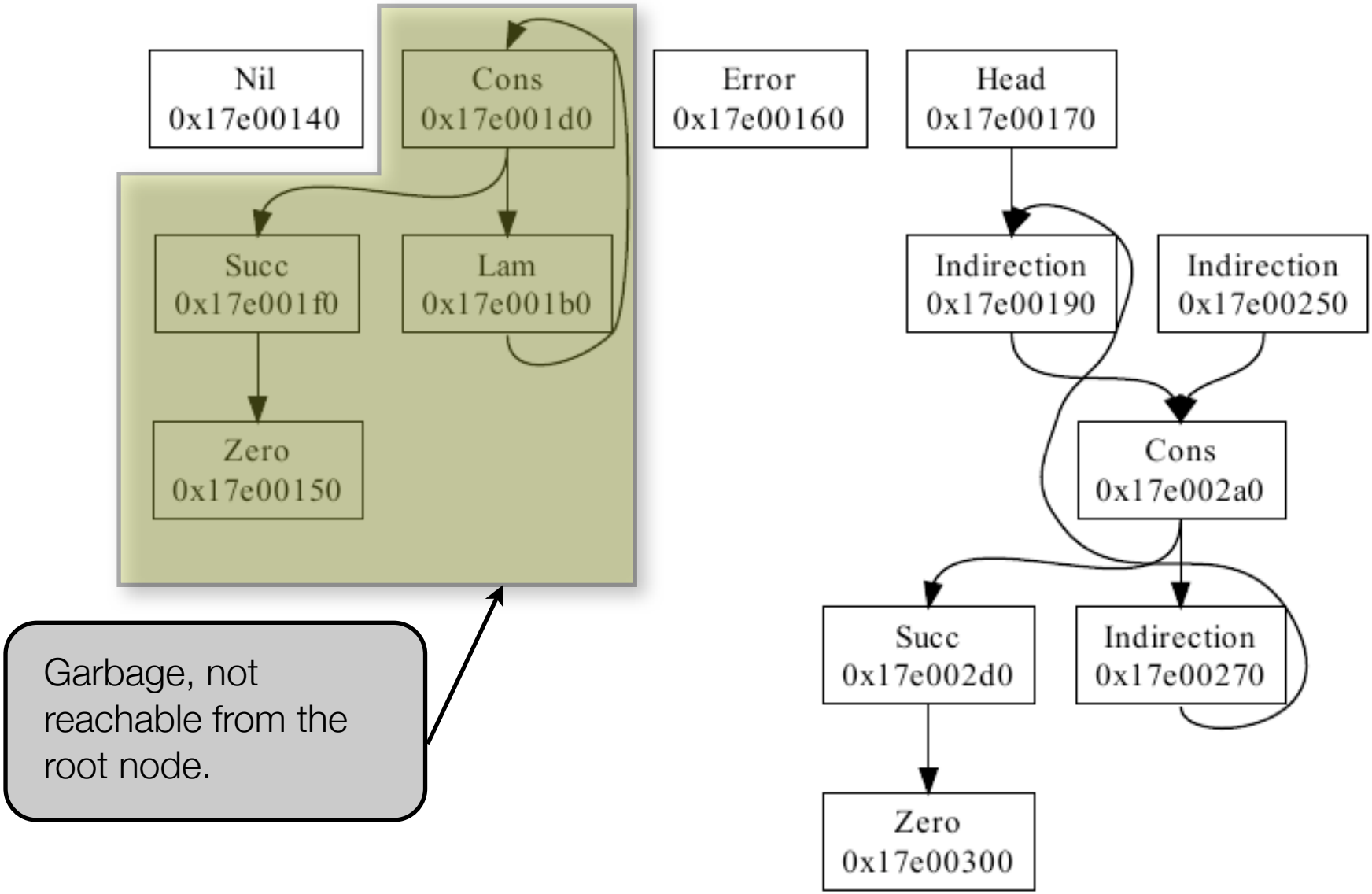
Step 6. overwrite the fix node with its result



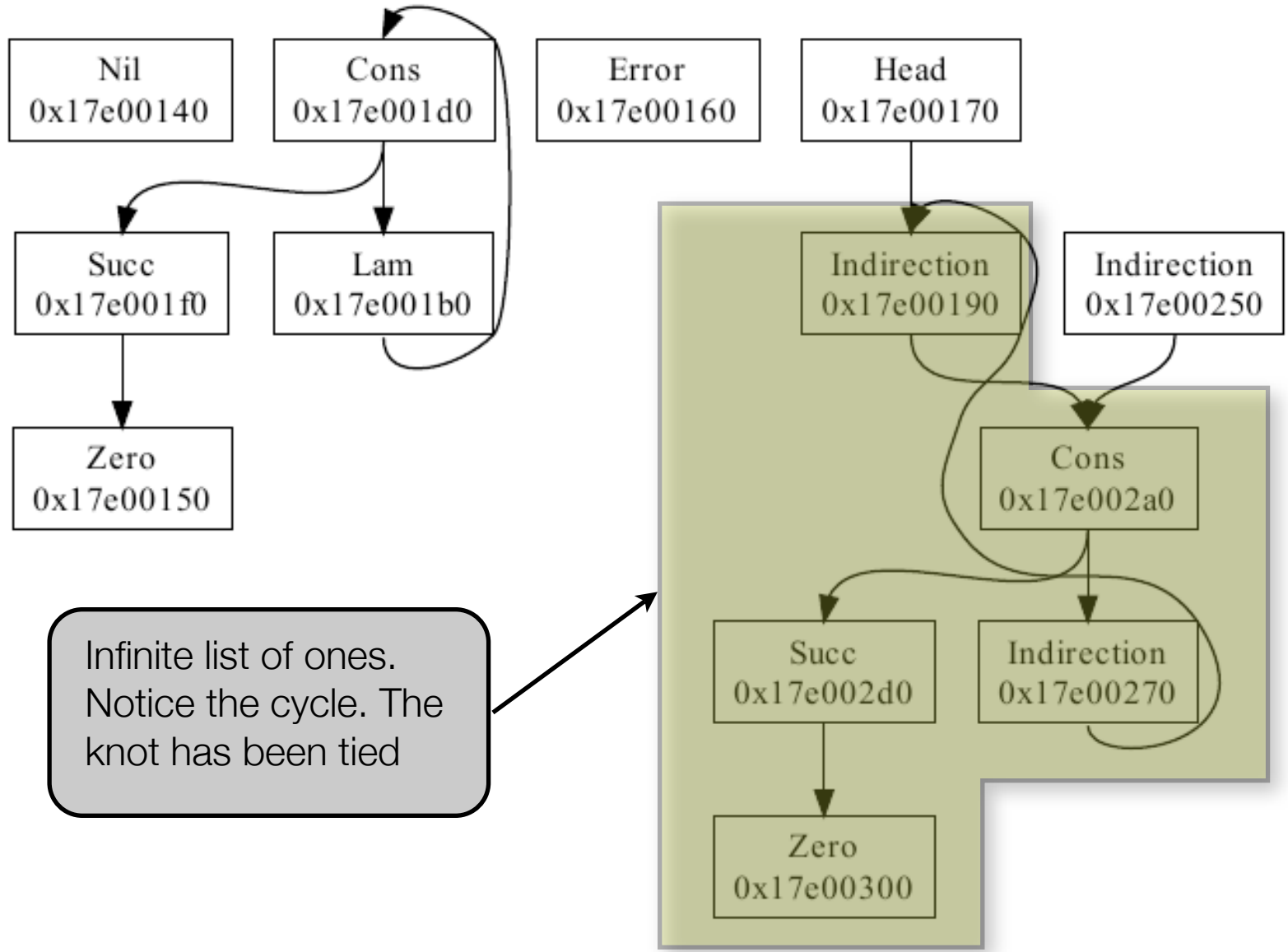
Step 6. overwrite the fix node with its result



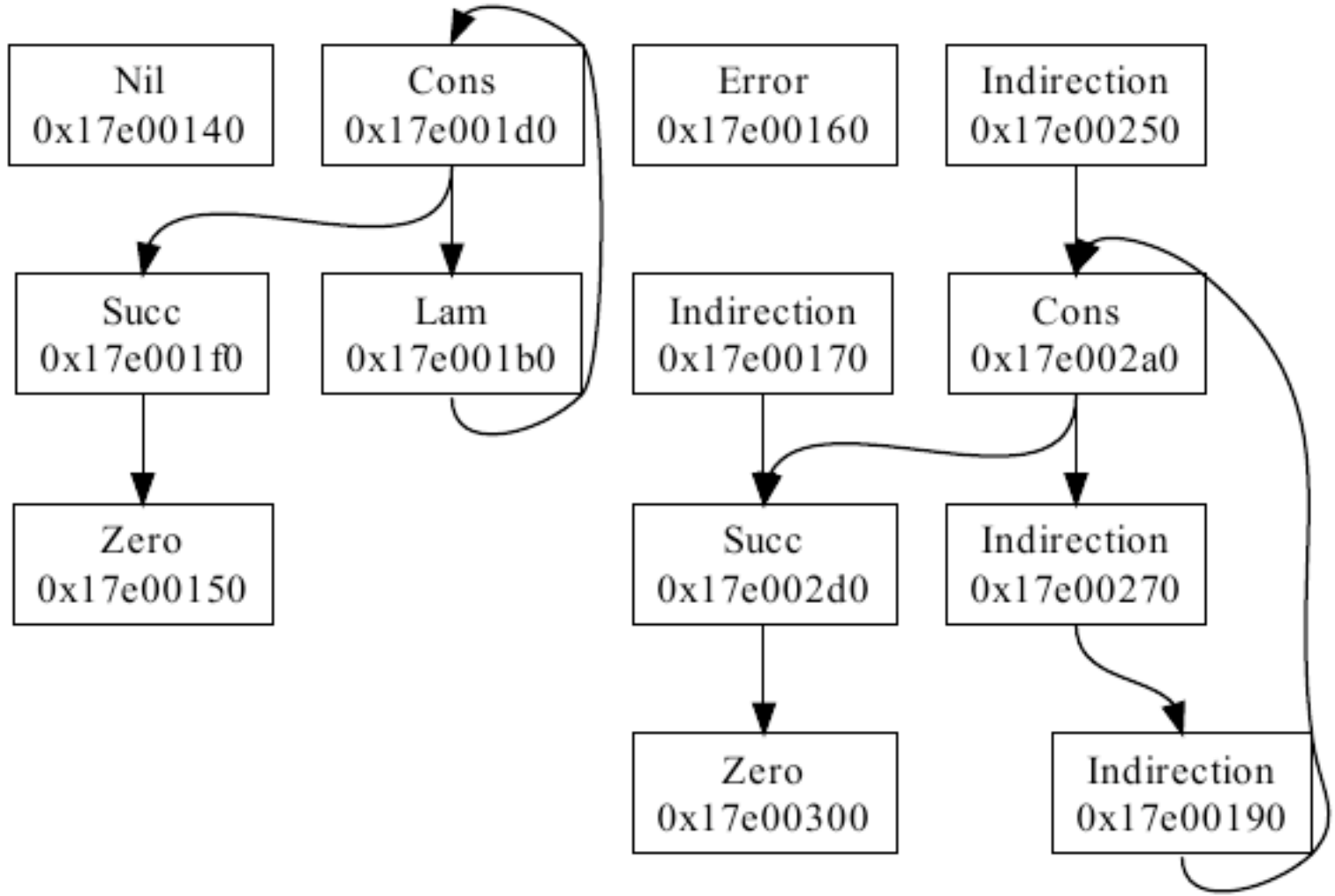
Step 6. overwrite the fix node with its result



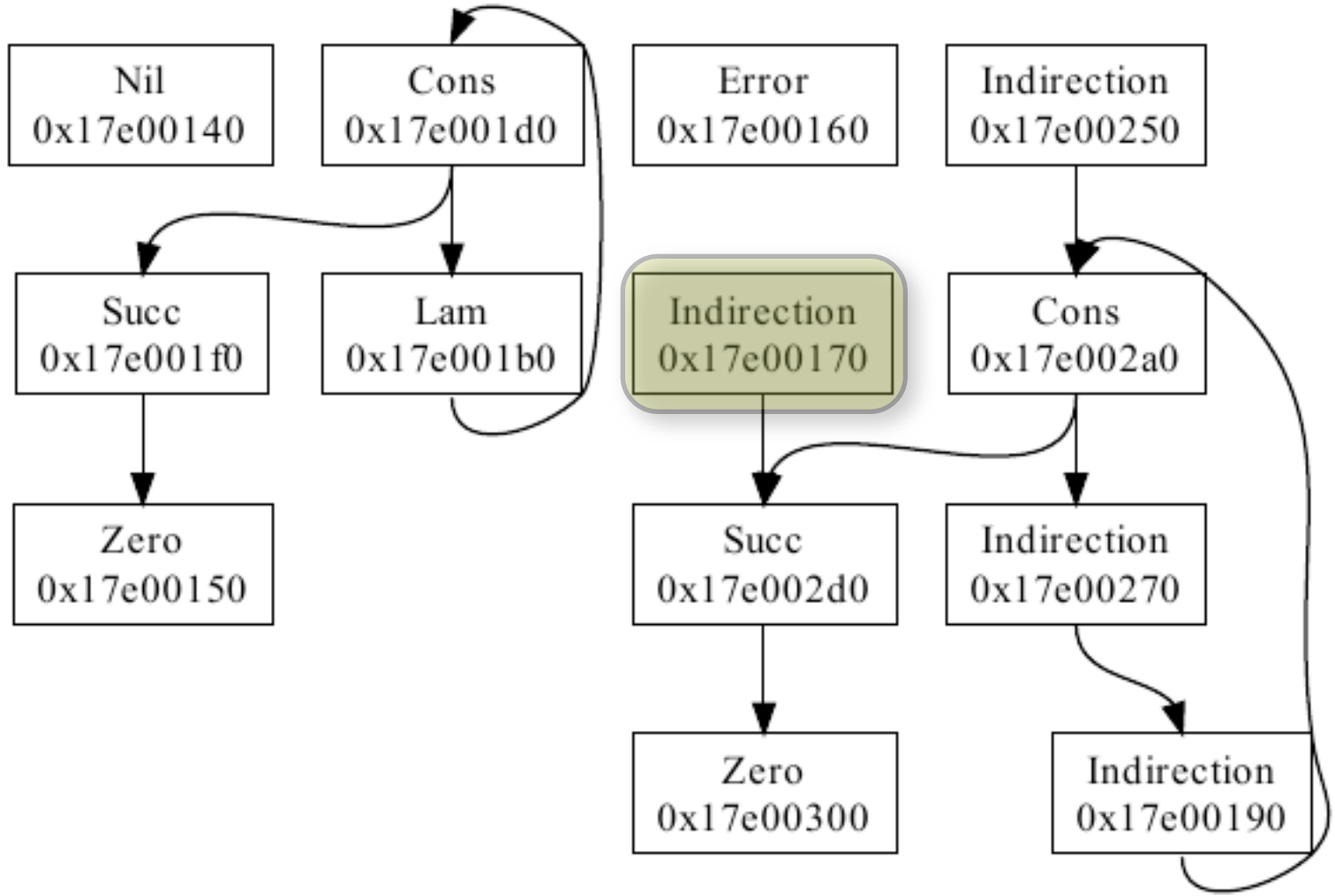
Step 6. overwrite the fix node with its result



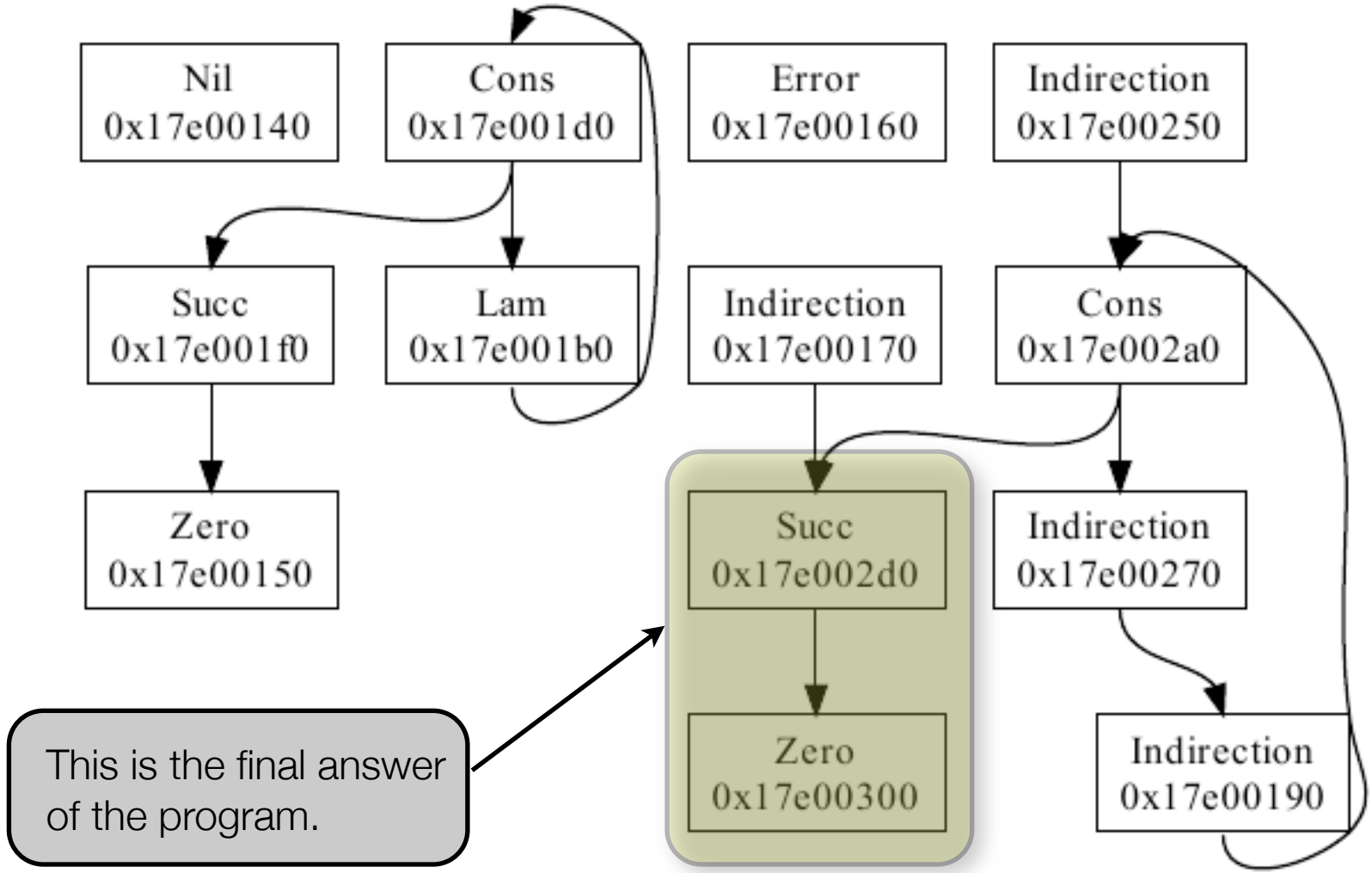
Step 7. overwrite head node with its result



Step 7. overwrite head node with its result



Step 7. overwrite head node with its result



The compiler

- Translates miniFP programs to C code.
- The output C code is a function which builds the initial graph representation of the program as a data structure in the heap.
- The output C code is linked with a library which contains a reduction engine.
- The reduction engine rewrites the graph until it reaches WHNF (or it loops).
- The final value is pretty printed on standard output.

Compiled output

```
#include "build.h"
GraphPtr build (void)
{
   GraphPtr v_0 = HEAD;
   GraphPtr v_1 = FIX;
   GraphPtr v_2 = LAM;
   GraphPtr v_3 = CONS;
   GraphPtr v_4 = SUCC;
   EDGE(v_4,0,zero);
   EDGE(v_3,0,v_4);
   EDGE(v_3,1,v_2);
   EDGE(v_2,0,v_3);
   EDGE(v_1,0,v_2);
   EDGE(v_0,0,v_1);
   return v_0;
}
```

Reduction engine

```
GraphPtr reduce (GraphPtr g)
{
   GraphPtr result = NULL;

   if (is_whnf (g))
   {
      return g;
   }
   else
   {
      switch (g->tag)
      {
          /* ... reduce graph and build result ... */
      }
      indirect (g, result);
      return result;
   }
}
```

Selected Literature

- *Theorie des ensembles*, Bourbaki (1954) (according to Shivers and Wand).
- *Semantics and Pragmatics of the Lambda Calculus*, Wadsworth (1971).
- *The implementation of functional languages*, (Chapter 12) Peyton Jones (1987).
- *An algorithm for optimal lambda-calculus reduction*, Lamping (1990).
- *Lambda Calculi plus Letrec*, Ariola and Blom (1997).
- *Bottom up beta substitution*, Shivers and Wand (2004).
- *Lambda animator*, Mike Thyer (2007). http://thyer.name/lambda-animator/