Circularity and Lambda Abstraction

Olivier Danvy[1], Peter Thiemann[2], and Ian Zerny[1]*

[1] {danvy,zerny}@cs.au.dk
Department of Computer Science, Aarhus University
[2] thiemann@acm.org
Institut für Informatik, Universität Freiburg

Abstract. In this tribute to Doaitse Swierstra, we present the first transformation between lazy circular programs à la Bird and strict circular programs à la Pettorossi. Circular programs à la Bird rely on lazy recursive binding: they involve circular unknowns and make sense equationally. Circular programs à la Pettorossi rely on the inductive construction of functions and their eventual application: they involve no circular unknowns and make sense operationally. Our derivation connects these equational and operational approaches: given a lazy circular program à la Bird, we decouple the circular unknowns from what is done to them, which we lambda-abstract with functions. The circular unknowns then become dead variables, which we eliminate. The result is a strict circular program à la Pettorossi. This transformation is reversible: given a strict circular program à la Pettorossi, we introduce circular unknowns as dead variables, and we apply the functions to them. The result is a lazy circular program à la Bird.
We illustrate the two transformations by mapping an algebraic construct to an isomorphic one with new leaves, reading a binary number as suggested by Knuth, and backpatching.

1 Introduction

You do not have to think operationally:
you can reason equationally
about your programs.
– S. Doaitse Swierstra

I prefer call by value
to call by name
because it is more predictable.
– Mitchell Wand

One of the wonderful things about functional programming is that we can both reason about programs equationally (regarding what they do) and think about them operationally (regarding how they do it). Take circular programs, for example. This technique was invented by Richard Bird in the early 1980's to eliminate multiple traversals of data [1]. It was then phrased operationally by Alberto Pettorossi in the late 1980's [2]. In this hommage to Doaitse Swierstra, we present what we believe to be the first transformation between circular programs à la Bird and circular programs à la Pettorossi. Each of the following sections illustrates this transformation.

* Ian Zerny is a recipient of the Google Europe Fellowship in Programming Technology, and this research is supported in part by this Google Fellowship.

Prerequisites and notation: It seems safe to assume that the reader knows what a circular program is, but we nevertheless explain the concept in Sec. 2. Likewise, in a structurally recursive function that visits an inductive data structure, we readily say that the arguments are "inherited" and the result is "synthesized," in reference to Knuth's attribute grammars [3]. Throughout, our programming language of discourse is Haskell.

2 Minimum list

In his original article [1], Bird illustrated circular programming with a function mapping a binary tree of integers into an isomorphic binary tree where all the integers were replaced by the smallest integer in the given binary tree. Rather than composing two functions – one to compute the smallest integer in the given tree, and one to re-traverse the given tree to construct an isomorphic tree – Bird calculated a 'circular' function that ostensibly traverses the given tree once and yet gets the job done.

In this section, we treat in detail a simplified version of Bird's original function that operates not on binary trees of integers, but on lists of integers. We first present the circular function in the style of Bird (Sec. 2.1), and illustrate its working equationally (Fig. 1). We then present the circular function in the style of Pettorossi (Sec. 2.2), and illustrate its working equationally (Fig. 2). We finally present our transformation to map either function to the other (Sec. 2.3).

2.1 A Bird-style circular program

The circular program à la Bird is a function that uses lazy local recursion to circularly refer to the minimal element of the input list. In the function below, `m` is the circular unknown: it is circularly defined using local recursion and it is unknown in the body of `visit`:

```
minlist_RB :: [Int] -> [Int]
minlist_RB xs = ys
  where
    (m, ys) = visit m xs
    visit :: Int -> [Int] -> (Int, [Int])
    visit m [] =
      (maxBound, [])
    visit m (x : xs) =
      let (m', ys) = visit m xs
      in  (min x m', m : ys)
```

Fig. 1 displays successive unfoldings of this function when it is applied to the list `[3,1,4]`. These unfoldings illustrate equationally the resolution of the circular unknown `m`. The modified part is boxed at each step.

2.2 A Pettorossi-style circular program

The circular program à la Pettorossi is a function that uses lambda abstraction to refer to the minimal element of the input list. In the function below, `m` is an abstracted unknown: it is lambda-abstracted in the body of `visit` and it is subsequently instantiated when the lambda-abstraction is applied:

```
minlist_RB_0 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = visit m (3 : 1 : 4 : [])
-- unfold the underlined call to visit
minlist_RB_1 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = visit m (1 : 4 : [])
              in (min 3 n, m : ys)
-- unfold the underlined call to visit
minlist_RB_2 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = visit m (4 : [])
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined call to visit
minlist_RB_3 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = visit m []
                                          in (min 4 n, m : ys)
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined call to visit
minlist_RB_4 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = (maxBound, [])
                                          in (min 4 n, m : ys)
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined let expression
minlist_RB_5 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = (min 4 maxBound, m : [])
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined call to min
minlist_RB_6 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = let (n, ys) = (4, m : [])
                            in (min 1 n, m : ys)
              in (min 3 n, m : ys)
-- unfold the underlined let expression
minlist_RB_7 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = (min 1 4, m : m : [])
              in (min 3 n, m : ys)
-- unfold the underlined call to min
minlist_RB_8 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = let (n, ys) = (1, m : m : [])
              in (min 3 n, m : ys)
-- unfold the underlined let expression
minlist_RB_9 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = (min 3 1, m : m : m : [])
-- unfold the underlined call to min
minlist_RB_10 (3 : 1 : 4 : []) = ys
  where
    (m, ys) = (1, m : m : m : [])
-- unfold the underlined where expression, which is recursive
minlist_RB_11 (3 : 1 : 4 : []) = 1 : 1 : 1 : []
```

Fig. 1. Successive equational unfoldings of `minlist_RB [3,1,4]`

```
minlist_AP_0 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = visit (3 : 1 : 4 : [])
-- unfold the underlined call to visit
minlist_AP_1 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = visit (1 : 4 : [])
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to visit
minlist_AP_2 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = visit (4 : [])
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to visit
minlist_AP_3 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = visit []
                                          in  (min 4 n, \m -> m : ys m)
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to visit
minlist_AP_4 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = let (n, ys) = let (n, ys) = (maxBound, \m -> [])
                                                        in  (min 4 n, \m -> m : ys m)
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined let expression
minlist_AP_5 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = (min 4 maxBound, \m -> m : (\m -> []) m)
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to min and the underlined beta-redex
minlist_AP_6 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = let (n, ys) = (4, \m -> m : [])
                            in  (min 1 n, \m -> m : ys m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined let expression
minlist_AP_7 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = (min 1 4, \m -> m : (\m -> m : []) m)
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined call to min and the underlined beta-redex
minlist_AP_8 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = let (n, ys) = (1, \m -> m : m : [])
              in  (min 3 n, \m -> m : ys m)
-- unfold the underlined let expression
minlist_AP_9 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = (min 3 1, \m -> m : (\m -> m : m : []) m)
-- unfold the underlined call to min and the undernined beta-redex
minlist_AP_10 (3 : 1 : 4 : []) = ys m
  where
    (m, ys) = (1, \m -> m : m : m : [])
-- unfold the underlined where expression, which is not recursive
minlist_AP_11 (3 : 1 : 4 : []) = (\m -> m : m : m : []) 1
-- unfold the underlined beta-redex
minlist_AP_12 (3 : 1 : 4 : []) = 1 : 1 : 1 : []
```

Fig. 2. Successive equational unfoldings of `minlist_AP [3,1,4]`

```
minlist_AP :: [Int] -> [Int]
minlist_AP xs = ys m
  where
    (m, ys) = visit xs
    visit :: [Int] -> (Int, Int -> [Int])
    visit [] =
      (maxBound, \m -> [])
    visit (x : xs) =
      let (m', ys) = visit xs
      in  (min x m', \m -> m : ys m)
```

Fig. 2 displays successive unfoldings of this Pettorossi-style program to the input list `[3,1,4]`. These unfoldings illustrate equationally the resolution of the abstracted unknown `m`. The modified part is boxed at each step.

2.3 From either style to the other

The last steps of Fig. 1 and Fig. 2 differ in two key aspects:

1. In the substitution step from `minlist_RB_10` to `minlist_RB_11`, the where expression is recursive for the Bird-style program, whereas for the Pettorossi-style program, the where expression is non-recursive in the substitution step from `minlist_AP_10` to `minlist_AP_11`.
2. The instantiation of `m` takes place during resolution for the Bird-style program, i.e., `minlist_RB_11` is the final result, whereas for the Pettorossi-style program, the instantiation of `m` takes place subsequently, i.e., `minlist_AP_11` is not the final result.

The key distinction is that `m` is a circular unknown (i.e., a variable that is declared recursively) in the Bird-style program whereas it is not in the Pettorossi-style program.

Using this observation, given a circular program à la Bird, we decouple the circular unknown from what is done to it, which we represent as a function (e.g., initially as the identity function). Consequently, the unknown becomes a dead variable in Bird's program and it is our observation that omitting this dead variable gives exactly a circular program à la Pettorossi – in the present case, the same program as in Sec. 2.2.

Here is the minlist program à la Bird where we have marked (with a trailing underscore) all of the variables that depend on the circular unknown:

```
minlist_mark :: [Int] -> [Int]
minlist_mark xs = ys_
  where
    (m_, ys_) = visit m_ xs
    visit :: Int -> [Int] -> (Int, [Int])
    visit m_ [] =
      (maxBound, [])
    visit m_ (x : xs) =
      let (m', ys_) = visit m_ xs
      in  (min x m', m_ : ys_)
```

We decouple the circular unknown in two steps. Here is the decoupling (as a pair) of the inherited variables that depend on the circular unknown: the first

component of the pair is the circular unknown, and the second component is what is done to the circular unknown:[3]

```
minlist_inher :: [Int] -> [Int]
minlist_inher xs = ys_
  where
    (m, ys_) = visit (m, id) xs
    visit :: (Int, Int -> Int) -> [Int] -> (Int, [Int])
    visit m_ [] =
      (maxBound, [])
    visit (m_ @ (m, f)) (x : xs) =
      let (m', ys_) = visit m_ xs
      in  (min x m', f m : ys_)
```

Here is the abstraction of the synthesized variables that depend on the circular unknown:

```
minlist_synth :: [Int] -> [Int]
minlist_synth xs = ys m
  where
    (m, ys) = visit (m, id) xs
    visit :: (Int, Int -> Int) -> [Int] -> (Int, Int -> [Int])
    visit m_ [] =
      (maxBound, \m -> [])
    visit (m_ @ (m, f)) (x : xs) =
      let (m', ys) = visit m_ xs
      in  (min x m', \m -> f m : ys m)
```

In `visit`, the variable `m` is dead (i.e., it is unused) and the variable `f` solely denotes the identity function (i.e., nothing is done to `m`). Thus, we strike out the first and we symbolically apply the second. The result is precisely the circular program à la Pettorossi from Sec. 2.2. This program is inductively defined and can be transliterated to an eager programming language such as ML.

Overall, each of the steps in the transformation from Bird style to Pettorossi style can be reversed.

In the following sections, we successively consider Bird's original circular program mapping a tree of numbers to an isomorphic tree of the least of these numbers (Sec. 3); Knuth's original attribute grammar for reading a binary number (Sec. 4); and conversely, how to express backpatching as a circular program (Sec. 5).

3 Minimum tree

Let us turn to Bird's circular function that given a binary tree of integers, maps it to an isomorphic binary tree where all the integers were replaced by the smallest integer in the given binary tree. The tree data type is declared as follows:

```
data BTree a = Leaf a | Node (BTree a) (BTree a)
```

For example, a tree such as `tin` below should be mapped to the tree `tout`:

[3] In the interest of generality, we fully decouple the inherited variables, here `m_` of `visit`, even though nothing is done to them. In general, the inherited variables could be changed. Such is the case in Knuth's program for reading binary numbers (Sec. 4).

```
tin  = Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
tout = Node (Leaf 1) (Node (Leaf 1) (Leaf 1))
```

Bird's circular program is the starting point of the transformation:

```
mintree_RB :: BTree Int -> BTree Int
mintree_RB t = t'
  where
    (m, t') = visit m t
    visit :: Int -> BTree Int -> (Int, BTree Int)
    visit m (Leaf n)
      = (n, Leaf m)
    visit m (Node l r)
      = let (lm, lt) = visit m l
            (rm, rt) = visit m r
        in (min lm rm, Node lt rt)
```

As in Sec. 2, we decouple the circular unknown (here `m`) from what is done to it. Here is the mintree program where we have marked (with a trailing underscore) all of the variables that depend on the circular unknown:

```
mintree_mark :: BTree Int -> BTree Int
mintree_mark t = t'_
  where
    (m_, t'_) = visit m_ t
    visit :: Int -> BTree Int -> (Int, BTree Int)
    visit m_ (Leaf n)
      = (n, Leaf m_)
    visit m_ (Node l r)
      = let (lm, lt_) = visit m_ l
            (rm, rt_) = visit m_ r
        in (min lm rm, Node lt_ rt_)
```

Here is the decoupling (as a pair: the circular unknown and what is done to it, represented as a function) of the inherited variables that depend on the circular unknown:

```
mintree_inher :: BTree Int -> BTree Int
mintree_inher t = t'_
  where
    (m, t'_) = visit (m, id) t
    visit :: (Int, Int -> Int) -> BTree Int -> (Int, BTree Int)
    visit (m, f) (Leaf n)
      = (n, Leaf (f m))
    visit m_ (Node l r)
      = let (lm, lt_) = visit m_ l
            (rm, rt_) = visit m_ r
        in (min lm rm, Node lt_ rt_)
```

Here is the abstraction of the synthesized variables that depend on the circular unknown:

```
mintree_synth :: BTree Int -> BTree Int
mintree_synth t = t' m
  where
    (m, t') = visit (m, id) t
    visit :: (Int, Int -> Int) -> BTree Int -> (Int, Int -> BTree Int)
    visit (m, f) (Leaf n)
      = (n, \m -> Leaf (f m))
    visit m_ (Node l r)
      = let (lm, lt) = visit m_ l
            (rm, rt) = visit m_ r
        in (min lm rm, \m -> Node (lt m) (rt m))
```

In `visit`, the variable `m` is dead (i.e., it is unused) and the variable `f` solely denotes the identity function (i.e., nothing is done to `m`). Thus, we eliminate both. The result is Pettorossi's one-pass solution to the problem [2]:

```
mintree_AP :: BTree Int -> BTree Int
mintree_AP t = t' m
  where
    (m, t') = visit t
    visit :: BTree Int -> (Int, Int -> BTree Int)
    visit (Leaf n)
      = (n, \m -> Leaf m)
    visit (Node l r)
      = let (lm, lt) = visit l
            (rm, rt) = visit r
        in (min lm rm, \m -> Node (lt m) (rt m))
```

Again, the Pettorossi-style program can be transliterated to an eager programming language. Also, each of the transformation steps is reversible.

4 Reading numbers

Knuth's seminal article on attribute grammars [3] starts with an example of a grammar that gives a precise definition of binary notation for numbers. The grammar generates the language with words of the form *num.mantissa* where both *num* and *mantissa* are bit strings. The attribution of the grammar computes, in an attribute v of the start symbol, the numeric value of the binary notation.

The interest in Knuth's second attribution of the grammar arises from a non-trivial attribute dependency that requires a two-pass traversal for evaluating all attributes. Thus, the "obvious" translation of the attribute grammar into a functional program results in a circular program of a slightly more general form as in the preceding examples.

But to start from the beginning, a slightly rephrased and simplified version of this example is sufficient to demonstrate the transformation. The simplified grammar only generates bit strings and the attribution considers the generated bit string as the binary notation for a number and computes it. The underlying grammar has terminals `O` and `I` representing the low bit and the high bit and three non-terminals *Bit*, *Bits*, and S, the start symbol. All non-terminals have a synthesized attribute v; *Bit* and *Bits* have an inherited attribute p; and *Bits* has an additional synthesized attribute l. The intention is that the attribute v computes the value of the respective bit or bit string relative to its starting position p — a *Bit* at position p counts 2^p. The attribute l computes the length of a bit string.

Fig. 3 contains the productions of the grammar and their attribution. The attribution rules use the notation suggested by Johnsson for referring to the attribute occurrences, with $\uparrow$ indicating synthesized attributes and $\downarrow$ indicating inherited ones [4]. His translation of an attribute grammar into a lazy program interprets a non-terminal as a function from its inherited attributes to its synthesized attributes. Applying this technique to the attribute grammar in Fig. 3

$$
\begin{array}{lll}
S & \to \mathit{Bits} & S \uparrow v = \mathit{Bits} \uparrow v \\
 & & \mathit{Bits} \downarrow p = \mathit{Bits} \uparrow l - 1 \\
\mathit{Bits} & \to \varepsilon & \mathit{Bits} \uparrow v = 0 \\
 & & \mathit{Bits} \uparrow l = 0 \\
\mathit{Bits} & \to \mathit{Bit}\ \mathit{Bits}_1 & \mathit{Bits} \uparrow v = \mathit{Bit} \uparrow v + \mathit{Bits}_1 \uparrow v \\
 & & \mathit{Bit} \downarrow p = \mathit{Bits} \downarrow p \\
 & & \mathit{Bits}_1 \downarrow p = \mathit{Bits} \downarrow p - 1 \\
 & & \mathit{Bits} \uparrow l = \mathit{Bits}_1 \uparrow l + 1 \\
\mathit{Bit} & \to \texttt{O} & \mathit{Bit} \uparrow v = 0 \\
\mathit{Bit} & \to \texttt{I} & \mathit{Bit} \uparrow v = 2\texttt{\^{}}(\mathit{Bit} \downarrow p)
\end{array}
$$

Fig. 3. Attribute grammar for interpreting numbers in binary notation

```
data Bit = O | I

digitval :: Int -> Bit -> Int
digitval p O = 0
digitval p I = 2 ^ p

dec n = n - 1
inc n = n + 1
```

Fig. 4. Auxiliary definitions for interpreting numbers in binary notation

leads to the circular program `lexnum_RB` shown in Fig. 5 which uses the definitions in Fig. 4.

In Fig. 4, the function `digitval` is the interpretation of the *Bit* non-terminal. It has one (inherited) argument and one (synthesized) result. Its `Bit`-typed argument serves to distinguish the two production rules for *Bit*. The functions `dec` and `inc` stand for the decrement and increment operations in the attribution.
In Fig. 5, the function `lexnum_RB` is the transliteration of the attributions of the non-terminals S and *Bits*. There is no choice in the first equation of the `where` block because S has one production.

The function `visit` is the interpretation of the *Bits* non-terminal. Its first argument holds the inherited attribute p and its second determines the production. It computes a pair comprising the synthesized attributes.

As in Sec. 2 and Sec. 3, we decouple the circular unknown (here `l`) from what is done to it. Here is the lexnum program where we have marked (with a trailing underscore) all of the variables that depend on the circular unknown:

```
lexnum_mark :: [Bit] -> Int
lexnum_mark bs = v_
  where
    (l_, v_) = visit (dec l_) bs
    visit :: Int -> [Bit] -> (Int, Int)
    visit p_ [] =
      (0, 0)
    visit p_ (b:bs) =
      let (l, v_) = visit (dec p_) bs
      in (inc l, v_ + digitval p_ b)
```

```
lexnum_RB :: [Bit] -> Int
lexnum_RB bs = v
  where
    (l, v) = visit (dec l) bs
    visit :: Int -> [Bit] -> (Int, Int)
    visit p [] =
      (0, 0)
    visit p (b:bs) =
      let (l, v) = visit (dec p) bs
      in  (inc l, v + digitval p b)
```

Fig. 5. Bird-style circular program for interpreting numbers in binary notation

Here is the decoupling (as a pair: the circular unknown and what is done to it, represented as a function) of the inherited variables that depend on the circular unknown:

```
lexnum_inher :: [Bit] -> Int
lexnum_inher bs = v_
  where
    (l, v_) = visit (l, dec) bs
    visit :: (Int, Int -> Int) -> [Bit] -> (Int, Int)
    visit (p, f) [] =
      (0, 0)
    visit (p, f) (b:bs) =
      let (l, v_) = visit (p, dec . f) bs
      in  (inc l, v_ + digitval (f p) b)
```

In contrast to Sec. 2 and Sec. 3, something is actually being done to the circular unknown at call time (i.e., it is decremented).

Here is the abstraction of the synthesized variables that depend on the circular unknown:

```
lexnum_synth :: [Bit] -> Int
lexnum_synth bs = v l
  where
    (l, v) = visit (l, dec) bs
    visit :: (Int, Int -> Int) -> [Bit] -> (Int, Int -> Int)
    visit (p, f) [] =
      (0, \p -> 0)
    visit (p, f) (b:bs) =
      let (l, v) = visit (p, dec . f) bs
      in  (inc l, \p -> v p + digitval (f p) b)
```

In `visit`, the variable `p` is dead (i.e., it is unused) so we eliminate it. The result is a Pettorossi-style program:

```
lexnum_AP :: [Bit] -> Int
lexnum_AP bs = v l
  where
    (l, v) = visit dec bs
    visit :: (Int -> Int) -> [Bit] -> (Int, Int -> Int)
    visit f [] =
      (0, \p -> 0)
    visit f (b:bs) =
      let (l, v) = visit (dec . f) bs
      in  (inc l, \p -> v p + digitval (f p) b)
```

Again, this Pettorossi-style program can be transliterated to an eager programming language. Also, each of the transformation steps is reversible.

```
type Lab = Int
type Addr = Int
type Env = Map Lab Addr
data Source =
  SSUB | SPSH Int | SJMP Lab | SCJP Lab | SLAB Lab
data Target =
  TSUB | TPSH Int | TJMP Addr | TCJP Addr
```

Fig. 6. Type definitions for backpatching

```
backpatch_AP :: [Source] -> [Target]
backpatch_AP ss = f env
  where
    (env, f) = collect 0 ss
    collect :: Addr -> [Source] -> (Env, Env -> [Target])
    collect a [] =
      (empty, \env -> [])
    collect a (si : sis) =
      let (env, f) = collect (a + tsize si) sis in
      case si of
        SSUB   -> (env, \env -> TSUB : f env)
        SPSH i -> (env, \env -> TPSH i : f env)
        SJMP l -> (env, \env -> TJMP (env ! l) : f env)
        SCJP l -> (env, \env -> TCJP (env ! l) : f env)
        SLAB l -> (insert l a env, f)
```

Fig. 7. Pettorossi-style program for backpatching

5 Backpatching

Backpatching is a traditional compilation technique [5]. It applies in a compiler that generates code using symbolic labels for jump targets in the first place. The main argument for doing so is to simplify the code generator and in particular subsequent transformation steps that may insert or remove instructions, or even rearrange entire code blocks.

Once the transformations are finished with the code, the symbolic labels have to be transformed to addresses. A typical approach is to traverse the code and build an environment that maps symbolic labels to addresses. A second pass uses this environment to resolve all jump addresses.

Backpatching is a one-pass implementation technique for this two-pass transformation. In this pass, label definitions are entered in the environment as they occur. For a label use, there are two possibilities, either the label is already defined (a backward reference) in which case the target address can be inserted directly, or the label is not yet defined (a forward reference), in which case this use-before-definition is registered in the environment. In general, there may be multiple uses before a definition is found, so the environment entry for a label may contain a list of unresolved targets. When defining a label that already has some uses, the unresolved targets are visited and overwritten with the address, hence the name backpatching.

This traditional algorithm is imperative. However, a purely functional implementation of backpatching can be given by abstracting the generation of the

program with absolute addresses from the environment as illustrated by the code in Fig. 7. It relies on the datatype definitions given in Fig. 6. They define a type `Source` of source instructions for a stack machine, which are subtraction, push a constant, jump, conditional jump, and label definition. Both jump instructions refer to symbolic labels of type `Label`. The type `Target` of target instructions comprises subtraction, push, jump, and conditional jump, where the latter two refer to addresses. As an auxiliary definition, an environment of type `Env` is a mapping from labels to addresses. Furthermore, there is a function `tsize :: Source -> Int` that computes the size of the generated target code for each source instruction.

The function `collect` in Fig. 7 traverses the list of source instructions and keeps track of the current target address in its first argument `a`. It returns a pair of an environment and a function that expects an environment and returns the target code. The transformation removes the label instruction `SLAB l` so that its target size is `0`.

This function is written in Pettorossi style: it is defined inductively and can be expressed directly in an eager programming language. We use it as the starting point to demonstrate the reverse transformation from Pettorossi style to Bird style, taking the reverse sequence of steps.

The first step is to introduce the abstracted value as a circular unknown of visit, here with the formal parameter `env1` of visit:

```
backpatch_circ :: [Source] -> [Target]
backpatch_circ ss = f env
  where
    (env, f) = collect 0 ss env
    collect :: Addr -> [Source] -> Env -> (Env, Env -> [Target])
    collect a [] env1 =
      (empty, \env -> [])
    collect a (si : sis) env1 =
      let (env, f) = collect (a + tsize si) sis env1 in
      case si of
        SSUB   -> (env, \env -> TSUB : f env)
        SPSH i -> (env, \env -> TPSH i : f env)
        SJMP l -> (env, \env -> TJMP (env ! l) : f env)
        SCJP l -> (env, \env -> TCJP (env ! l) : f env)
        SLAB l -> (insert l a env, f)
```

Next, we specialize the synthesized abstractions with respect to `env1`. Each abstraction passes `env1` unchanged and so `f env1` becomes the lazily constructed result, `ts`:

```
backpatch_synth :: [Source] -> [Target]
backpatch_synth ss = ts
  where
    (env, ts) = collect 0 ss env
    collect :: Addr -> [Source] -> Env -> (Env, [Target])
    collect a [] env1 =
      (empty, [])
    collect a (si : sis) env1 =
      let (env, ts) = collect (a + tsize si) sis env1 in
      case si of
        SSUB   -> (env, TSUB : ts)
        SPSH i -> (env, TPSH i : ts)
        SJMP l -> (env, TJMP (env1 ! l) : ts)
        SCJP l -> (env, TCJP (env1 ! l) : ts)
        SLAB l -> (insert l a env, ts)
```

Since there are no inherited abstractions we are done and the result is a circular backpatching program à la Bird.

6 Related work

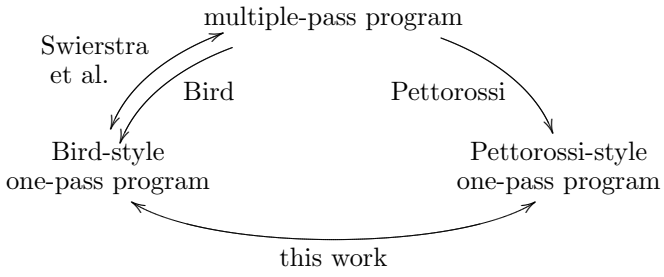
Kuiper and Swierstra [6] noted the connection between attribute grammars and functional programs around the same time as Johnsson [4]. They note that rewrite rules employing tuples and derivations of circular programs can be conveniently expressed using attribute grammars. They define two mappings from attribute grammars to functional programs. One of the mappings can give rise to multiple traversals of a data structure whereas the other yields circular programs that traverse the structure at most once, but require lazy evaluation.

Fernandes and Saraiva [7] transform circular programs into efficient, strict and deforested, multiple-traversal programs by using attribute grammars-based techniques, in particular ordered attribute grammars [8]. This approach draws on ideas from earlier work by Saraiva, Swierstra, and Kuiper [9]. Both works rely on intricate analysis techniques for attribute grammars. Their transformations yield strict, but potentially multi-pass programs.

Fernandes and coworkers [10] suggest a strictification transformation for circular programs. Their transformation is based on a dependency analysis to discover the circularity. They naively split the circular call, which returns a tuple, into several ones with each computing only one component. Specialization of these calls yields independent, non-circular definitions. The resulting programs are suitable for strict evaluation, but they are not in Pettorossi-style in that they might require multiple passes over the input data.

7 Conclusion

In the course of the 1980's, Bird and Pettorossi investigated how to calculate programs that traverse their input only once [1, 2]:



In his joint work on circular attribute grammars, Swiestra has shown how to transform Bird-style programs into multiple-pass programs and vice versa [6, 9].

In this tribute to Doaitse Swierstra, we have shown how to connect Bird-style and Pettorossi-style programs. A Bird-style program inductively extends a

circular unknown until this extended unknown can be solved. We decouple the circular unknowns in a Bird-style program from what is inductively done to them, which we represent with functions. The circular unknowns then become dead variables. Symbolically applying the functions to the circular unknowns gives back a Bird-style program, and eliminating the dead variables gives a Pettorossi-style program. A Pettorossi-style program is therefore one that computes over differences, and so could be considered as the derivative of a Bird-style program.

Acknowledgments: We are grateful to Julia Lawall for comments on a preliminary version of this article, and to Doaitse Swierstra for many years of academic camaraderie. We wish him many happy returns!

References

1. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
2. Pettorossi, A.: Derivation of programs which traverse their input data only once. In Cioni, G., Salwicki, A., eds.: Advanced Programming Methodologies. Academic Press, Limited, San Diego, CA, USA (1989) 165–184
3. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2) (1968) 127–145
4. Johnsson, T.: Attribute grammars as a functional programming paradigm. In Kahn, G., ed.: Functional Programming Languages and Computer Architecture. Number 274 in Lecture Notes in Computer Science, Portland, Oregon, Springer-Verlag (September 1987) 154–173
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Second edn. Pearson Education, Inc. Addison-Wesley, London, United Kingdom (2006)
6. Kuiper, M.F., Swierstra, S.D.: Using attribute grammars to derive efficient functional programs. Technical Report RUU-CS-86-16, University of Utrecht (August 1986)
7. Fernandes, J.P., Saraiva, J.: Tools and libraries to model and manipulate circular programs. In Ramalingam, G., Visser, E., eds.: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007), Nice, France, ACM Press (January 2007) 102–111
8. Kastens, U.: Ordered attributed grammars. Acta Informatica **13** (1980) 229–256
9. Saraiva, J., Swierstra, S.D., Kuiper, M.F.: Strictification of computations on trees. In: 3th Latin-American Conference on Functional Programming, CLaPF'99. (1999)
10. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In Khoo, S.C., Siek, J., eds.: Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011), Austin, Texas, ACM Press (January 2011) 131–140