JFP **22** (3): 275–299, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000111 First published online 8 May 2012

*Manipulating accumulative functions by swapping call-time and return-time computations**

AKIMASA MORIHATA
Tohoku University, Sendai, Japan
(*e-mail:* morihata@riec.tohoku.ac.jp)

KAZUHIKO KAKEHI
University of Tokyo, Tokyo, Japan

ZHENJIANG HU
National Institute of Informatics, Chiyoda, Tokyo, Japan

MASATO TAKEICHI
National Institution for Academic Degrees and University Evaluation, Kodaira-shi, Tokyo, Japan

Abstract

Functional languages are suitable for transformational developments of programs. However, accumulative functions, or in particular tail-recursive functions, are known to be less suitable for manipulation. In this paper, we propose a program transformation named "IO swapping" that swaps call-time and return-time computations. It moves computations in accumulative parameters to results and thereby enables interesting transformations. We demonstrate effectiveness of IO swapping by several applications: deforestation, higher order removal, program inversion, and manipulation of circular programs.

1 Introduction

It is well recognized that functional languages are suitable for transformational developments of programs. Nontrivial programs can be derived from simple ones by applying semantic-preserving transformations. For example, consider deforestation (Wadler, 1990). Given the following standard *map* and *lrev* functions

$$\begin{aligned} &map\ f\ [] &&= [] \\ &map\ f\ (a : x) &&= f\ a : map\ f\ x \end{aligned}$$

$$\begin{aligned} &lrev\ x = aux_{lrev}\ x\ [] \\ &\quad \textbf{where}\ aux_{lrev}\ []\ h &&= h \\ &\qquad\quad aux_{lrev}\ (a : x)\ h &&= aux_{lrev}\ x\ (a : h), \end{aligned}$$

* A preliminary report of this work was published in Morihata, A., Kakehi, K., Hu, Z. & Takeichi, M. (2006) Swapping arguments and results of recursive functions. In *Proceedings of the 8th International Conference on Mathematics of Program Construction (MPC 2006)*, Kuressaare, Estonia. Lecture Notes in Computer Science, vol. 4014. New York, USA: Springer-Verlag, pp. 379–396.

we may reverse a given list after applying some function to each element.

$$lrev\ (map\ f\ x)$$

This program is clear and simple, but not very efficient because of the intermediate list passed between *map* and *lrev*. We remove it by the classic unfolding–folding method (Burstall & Darlington, 1977), in which we develop programs by repeatedly unfolding function calls to their bodies and folding expressions into corresponding function calls. We introduce a new function that denotes the composition of *map* and *lrev*

$$revMap\ f\ x = lrev\ (map\ f\ x)$$

and try to simplify this equation as follows.

$$\begin{aligned} revMap\ f\ x &= \{\ \text{unfolding } revMap\ \} \\ &\quad lrev\ (map\ f\ x) \\ &= \{\ \text{unfolding } lrev\ \} \\ &\quad aux_{lrev}\ (map\ f\ x)\ [] \end{aligned}$$

The obtained program, $revMap\ f\ x = aux_{lrev}\ (map\ f\ x)\ []$, is not satisfactory. The intermediate list between aux_{lrev} and *map* still exists. As before, we introduce a new function that corresponds to the composition of aux_{lrev} and *map*

$$aux_{revMap}\ f\ x\ h = aux_{lrev}\ (map\ f\ x)\ h$$

Then we work out its recursive definition. Since *map* and aux_{lrev} are defined based on pattern-matching, we perform a case analysis,

$$\begin{aligned} aux_{revMap}\ f\ []\ h &= \{\ \text{unfolding } aux_{revMap}\ \} \\ &\quad aux_{lrev}\ (map\ f\ [])\ h \\ &= \{\ \text{unfolding } map \text{ and } aux_{lrev}\ \} \\ &\quad h \end{aligned}$$

$$\begin{aligned} aux_{revMap}\ f\ (a : x)\ h &= \{\ \text{unfolding } aux_{revMap}\ \} \\ &\quad aux_{lrev}\ (map\ f\ (a : x))\ h \\ &= \{\ \text{unfolding } map \text{ and } aux_{lrev}\ \} \\ &\quad aux_{lrev}\ (map\ f\ x)\ (f\ a : h) \\ &= \{\ \text{folding } aux_{revMap}\ \} \\ &\quad aux_{revMap}\ f\ x\ (f\ a : h) \end{aligned}$$

In summary, we have developed the following program:

$$\begin{aligned} &revMap\ f\ x = aux_{revMap}\ f\ x\ [] \\ &\quad \textbf{where}\ aux_{revMap}\ f\ []\ h = h \\ &\qquad\qquad aux_{revMap}\ f\ (a : x)\ h = aux_{revMap}\ f\ x\ (f\ a : h) \end{aligned}$$

We have succeeded in eliminating the intermediate list.

We would like to develop many programs in such a transformational way. However, it is known that in general deforestation for accumulative functions is difficult (Chin, 1994; Kühnemann, 1999). In particular, naive unfolding–folding fails to eliminate intermediate structures constructed in accumulative parameters.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

To see the difficulty, consider the following slightly different program,

$$map\ f\ (lrev\ x)$$

We expect that $map\ f\ (lrev\ x) = revMap\ f\ x$ holds. To confirm this intuition, let us develop a recursive definition of $revMap'\ f\ x = map\ f\ (lrev\ x)$ and see whether $revMap' = revMap$ holds,

$$\begin{aligned} revMap'\ f\ x &= \{\ \text{unfolding } revMap'\ \} \\ &\quad map\ f\ (lrev\ x) \\ &= \{\ \text{unfolding } lrev\ \} \\ &\quad map\ f\ (aux_{lrev}\ x\ []) \end{aligned}$$

We have encountered another composition that consists of map and aux_{lrev}, and therefore we introduce another auxiliary function:

$$aux_{revMap'}\ f\ x\ h = map\ f\ (aux_{lrev}\ x\ h).$$

We go on with the derivation,

$$\begin{aligned} aux_{revMap'}\ f\ []\ h &= \{\ \text{unfolding } aux_{revMap'}\ \} \\ &\quad map\ f\ (aux_{lrev}\ []\ h) \\ &= \{\ \text{unfolding } aux_{lrev}\ \} \\ &\quad map\ f\ h \\ aux_{revMap'}\ f\ (a : x)\ h &= \{\ \text{unfolding } aux_{revMap'}\ \} \\ &\quad map\ f\ (aux_{lrev}\ (a : x)\ h) \\ &= \{\ \text{unfolding } aux_{lrev}\ \} \\ &\quad map\ f\ (aux_{lrev}\ x\ (a : h)) \\ &= \{\ \text{folding } aux_{revMap'}\ \} \\ &\quad aux_{revMap'}\ f\ x\ (a : h) \end{aligned}$$

We have developed the following program:

$$\begin{aligned} &revMap'\ f\ x = aux_{revMap'}\ f\ x\ [] \\ &\quad \textbf{where}\ aux_{revMap'}\ f\ []\ h = map\ f\ h \\ &\qquad aux_{revMap'}\ f\ (a : x)\ h = aux_{revMap'}\ f\ x\ (a : h) \end{aligned}$$

The derived $revMap'$ is different from $revMap$. In particular, it still uses an intermediate list that is accumulated by $aux_{revMap'}$ and then consumed by $map\ f$.

As seen, deforestation for accumulative functions is nontrivial. Several solutions have been proposed (Kühnemann, 1998; Correnson *et al.*, 1999; Nishimura, 2004; Voigtländer, 2004; Voigtländer & Kühnemann, 2004; Katsumata & Nishimura, 2008). Here we introduce another method. It is based on a new program transformation named *IO swapping*. From $lrev$, IO swapping yields another but equivalent function, $lrev^{\curvearrowleft}$.

$$\begin{aligned} &lrev^{\curvearrowleft}\ x = \textbf{let}\ (r, []) = aux^{\curvearrowleft}_{lrev}\ x\ x\ \textbf{in}\ r \\ &\quad \textbf{where}\ aux^{\curvearrowleft}_{lrev}\ []\ x = ([], x) \\ &\qquad aux^{\curvearrowleft}_{lrev}\ (\_ : y)\ x = \textbf{let}\ (r, a : x') = aux^{\curvearrowleft}_{lrev}\ y\ x \\ &\qquad\qquad\qquad\qquad \textbf{in}\ (a : r, x') \end{aligned}$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

The auxiliary function, $aux^{\curvearrowleft}$, is in the *there-and-back-again* style (Danvy & Goldberg, 2005): It traverses the input list at call time, and then it traverses the input list again at return time. In the definition of *lrev*, the result is accumulated at call time. In the definition of $lrev^{\curvearrowleft}$, the result is accumulated at return time.

Now compare $lrev^{\curvearrowleft}$ to *lrev* carefully. In *lrev*, each list element is shifted to the accumulative parameter, *h*, on each *call* of aux_{lrev}. In $lrev^{\curvearrowleft}$, each element is shifted to *r*, the first component of the result, on each *return* of $aux^{\curvearrowleft}_{lrev}$. Here lies the essence of IO swapping: It swaps call-time and return-time computations.

Function $lrev^{\curvearrowleft}$ leads to a successful deforestation because it constructs the list not in an accumulative parameter but in a return value. We confirm it by developing $revMap^{\curvearrowleft}\ f\ x = map\ f\ (lrev^{\curvearrowleft}\ x)$. The main part is to work out the following function,

$$aux^{\curvearrowleft}_{revMap}\ f\ y\ x = \mathbf{let}\ (r, x') = aux^{\curvearrowleft}\ y\ x\ \mathbf{in}\ (map\ f\ r, x'),$$

and it is achieved as follows:

$$\begin{aligned}
aux^{\curvearrowleft}_{revMap}\ f\ []\ x &= \{\ \text{unfolding } aux^{\curvearrowleft}_{revMap}\ \} \\
&\quad \mathbf{let}\ (r, x') = aux^{\curvearrowleft}_{lrev}\ []\ x\ \mathbf{in}\ (map\ f\ r, x') \\
&= \{\ \text{unfolding } aux^{\curvearrowleft}_{lrev}\ \} \\
&\quad (map\ f\ [], x) \\
&= \{\ \text{unfolding } map\ f\ \} \\
&\quad ([], x) \\
aux^{\curvearrowleft}_{revMap}\ f\ (_ : y)\ x &= \{\ \text{unfolding } aux^{\curvearrowleft}_{revMap}\ \} \\
&\quad \mathbf{let}\ (r, x') = aux^{\curvearrowleft}_{lrev}\ (_ : y)\ x\ \mathbf{in}\ (map\ f\ r, x') \\
&= \{\ \text{unfolding } aux^{\curvearrowleft}_{lrev}\ \} \\
&\quad \mathbf{let}\ (r', a : x') = aux^{\curvearrowleft}_{lrev}\ y\ x\ \mathbf{in}\ (map\ f\ (a : r'), x') \\
&= \{\ \text{unfolding } map\ f\ \} \\
&\quad \mathbf{let}\ (r', a : x') = aux^{\curvearrowleft}_{lrev}\ y\ x\ \mathbf{in}\ (f\ a : map\ f\ r', x') \\
&= \{\ \text{folding } aux^{\curvearrowleft}_{revMap}\ \} \\
&\quad \mathbf{let}\ (r'', a : x') = aux^{\curvearrowleft}_{revMap}\ y\ x\ \mathbf{in}\ (f\ a : r'', x')
\end{aligned}$$

We have obtained the following program:

$$\begin{aligned}
revMap^{\curvearrowleft}\ f\ x &= \mathbf{let}\ (r, []) = aux^{\curvearrowleft}_{revMap}\ f\ x\ x\ \mathbf{in}\ r \\
&\quad \mathbf{where}\ aux^{\curvearrowleft}_{revMap}\ f\ []\ x = ([], x) \\
&\qquad aux^{\curvearrowleft}_{revMap}\ f\ (_ : y)\ x = \mathbf{let}\ (r, a : x') = aux^{\curvearrowleft}_{revMap}\ f\ y\ x \\
&\qquad\qquad \mathbf{in}\ (f\ a : r, x')
\end{aligned}$$

The $revMap^{\curvearrowleft}$ function does not contain construction of intermediate lists; moreover, by applying IO swapping backward to it, we indeed obtain *revMap*.

We have considered deforestation of accumulative functions. In fact, for several kinds of manipulations besides deforestation, it has been pointed out that accumulative functions are difficult to manipulate. Nontrivial methods have been proposed for dealing with them, including those for automatic theorem proving (Boyer & Moore, 1975; Boyer *et al.*, 1976; Giesl, 2000; Giesl *et al.*, 2007), those for higher order removal (Nishimura, 2003; Katsumata & Nishimura, 2008), and those for program inversion (Glück & Kawabe, 2005; Mogensen, 2006; Matsuda *et al.*, 2010).

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

$$
\begin{array}{lll}
[] ++ y & = & y \\
(a : x) ++ y & = & a : (x ++ y)
\end{array}
\qquad
\begin{array}{lll}
\mathit{reverse}\ [] & = & [] \\
\mathit{reverse}\ (a : x) & = & \mathit{reverse}\ x ++ [a]
\end{array}
$$

$$
\begin{array}{lll}
\mathit{take}\ 0\ x & = & [] \\
\mathit{take}\ n\ [] & = & [] \\
\mathit{take}\ (n+1)\ (a : x) & = & a : \mathit{take}\ n\ x
\end{array}
\qquad
\begin{array}{lll}
\mathit{drop}\ 0\ x & = & x \\
\mathit{drop}\ n\ [] & = & [] \\
\mathit{drop}\ (n+1)\ (a : x) & = & \mathit{drop}\ n\ x
\end{array}
$$

$$
\begin{array}{lll}
\mathit{foldr}\ f\ e\ [] & = & e \\
\mathit{foldr}\ f\ e\ (a : x) & = & f\ a\ (\mathit{foldr}\ f\ e\ x)
\end{array}
\qquad
\begin{array}{lll}
\mathit{foldl}\ f\ e\ [] & = & e \\
\mathit{foldl}\ f\ e\ (a : x) & = & \mathit{foldl}\ f\ (f\ e\ a)\ x
\end{array}
$$

$$
\begin{array}{lll}
\mathit{unfoldr}\ \psi\ v & = & \textbf{case}\ \psi\ v\ \textbf{of}\ \mathit{Nothing} \rightarrow [] \\
 & & \qquad\qquad \mathit{Just}\ (a, v') \rightarrow a : \mathit{unfoldr}\ \psi\ v'
\end{array}
$$

Fig. 1. Definitions of some standard functions.

We show that IO swapping, which we introduce in Section 3, could reduce the difficulty of manipulating accumulative programs. We demonstrate effectiveness of IO swapping through several examples: deforestation (Sections 4 and 7), higher order removal (Section 5), and program inversion (Section 6). In addition, we show manipulation of tree-operating functions in Section 7, and discuss a relationship to circular programs (Bird, 1984) in Section 8. It is worth noting that existing methods are usually stronger than IO swapping. They can deal with most examples presented here; moreover, they may be simpler and may be able to deal with examples that cannot be dome by IO swapping. A distinctive feature of IO swapping is that it can be used for several kinds of manipulations.

2 Preliminaries

2.1 Basic definitions

We use Haskell (Peyton Jones, 2003) for describing programs. We do not consider undefined values and nonterminating computations. Some standard functions we use are summarized in Figure 1.

We call computations in arguments "call-time computations" and those in return values "return-time computations." Since we consider lazy semantics, these terminologies may sound slightly odd – all computations are to be performed by need. We will revisit this issue after introducing an IO swapping rule, Theorem 5.

2.2 Shortcut deforestation

Deforestation (Wadler, 1990) (also called fusion) is a method of eliminating intermediate structures that are used for combining functions. Here we introduce *shortcut deforestation* devised by Gill (1996) and Gill *et al.* (1993).

Theorem 1 (*foldr*/*build* (Gill *et al.*, 1993))
The following equation holds:

$$\mathit{foldr}\ f\ e\ (\mathit{build}\ g) = g\ f\ e,$$

where $\mathit{build} :: (\forall \beta.(A \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta) \rightarrow [A]$ is defined by $\mathit{build}\ g = g\ (:)\ []$.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

Since we know that *foldr f e* replaces list constructors (:) and [] by parameters f and e, respectively, we can avoid producing the intermediate list once the constructors appearing in the intermediate list are specified. Function *build* certainly captures the constructors, as its polymorphic type guarantees.

There is the dual of the *foldr*/*build* rule, called the *destroy*/*unfoldr* rule. It is firstly formulated by Takano and Meijer (1995) and later rephrased by Svenningsson (2002).

Theorem 2 (*destroy*/*unfoldr* (Svenningsson, 2002))
The following equation holds:

$$destroy\ g\ (unfoldr\ \psi\ e) = g\ \psi\ e$$

where $destroy :: (\forall\alpha.(\alpha \to Maybe\ (B, \alpha)) \to \alpha \to C) \to [B] \to C$ is defined below.

$$\begin{aligned} destroy\ g\ x &= g\ out\ x \\ &\quad \textbf{where}\ out\ [] \quad\ = Nothing \\ &\qquad\quad out\ (a : x) = Just\ (a, x) \end{aligned}$$

Function *destroy* specifies how the intermediate list will be consumed. The polymorphic type of g guarantees that *out* destructs the intermediate lists. Then, since *unfoldr* regularly generates a list, we can directly calculate the final result from values supplied by ψ.

3 IO swapping

IO swapping is a transformation that turns call-time computations into return-time computations and *vice versa*. We introduce IO swapping rules for four kinds of functions: *foldl*, tail-recursive functions, *foldr*, and list-hylomorphisms (Meijer *et al.*, 1991; Backhouse *et al.*, 1999). We demonstrate their use in the following sections. We recommend those who are interested in examples to go to the next section and refer to this section by need.

The simplest IO swapping rule is that for *foldl*.

Theorem 3 (*IO swapping for foldl*)
The following function $foldl^{\curvearrowleft}$ is equivalent to *foldl*.

$$\begin{aligned} foldl^{\curvearrowleft}\ f\ e\ x &= \textbf{let}\ (r, []) = aux^{\curvearrowleft}_{foldl}\ x\ x\ \textbf{in}\ r \\ &\quad \textbf{where}\ aux^{\curvearrowleft}_{foldl}\ []\ x \quad\ = (e, x) \\ &\qquad\quad aux^{\curvearrowleft}_{foldl}\ (\_ : y)\ x = \textbf{let}\ (r, a : x') = aux^{\curvearrowleft}_{foldl}\ y\ x \\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\ (f\ r\ a, x') \end{aligned}$$

Proof
It is sufficient to confirm that the following equation holds for any list y that is not strictly longer than x, where $|y|$ denotes the length of list y.

$$aux^{\curvearrowleft}_{foldl}\ y\ x = (foldl\ f\ e\ (take\ |y|\ x), drop\ |y|\ x)$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

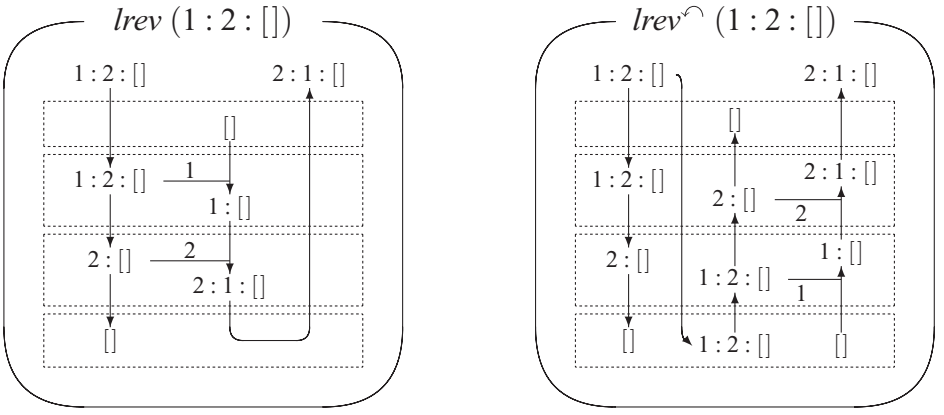

Fig. 2. Outlines of the processes of evaluating *lrev* (1 : 2 : []) and *lrev*$^{\curvearrowleft}$ (1 : 2 : []): each broken-lined box stands for a computation at a recursive step.

We prove it by induction. It is obvious when y is []. When y is $b : z$ and is not longer than x, we reason as follows:

$$
\begin{aligned}
&aux^{\curvearrowleft}_{foldl}\ (b : z)\ x = \{\ \text{definition of } aux^{\curvearrowleft}_{foldl}\ \}\\
&\quad \textbf{let } (r, a : x') = aux^{\curvearrowleft}_{foldl}\ z\ x \textbf{ in } (f\ r\ a, x')\\
&= \{\ \text{induction hypothesis (note that } z \text{ is not longer than } x)\ \}\\
&\quad \textbf{let } (a : x') = drop\ |z|\ x\\
&\qquad r = foldl\ f\ e\ (take\ |z|\ x)\\
&\quad \textbf{in } (f\ r\ a, x')\\
&= \{\ \text{claim: } foldl\ f\ e\ (x +\!\!+ [a]) = f\ (foldl\ f\ e\ x)\ a\ \}\\
&\quad \textbf{let } (a : x') = drop\ |z|\ x\\
&\quad \textbf{in } (foldl\ f\ e\ (take\ |z|\ x +\!\!+ [a]), x')\\
&= \{\ \text{definitions of } take \text{ and } drop\ \}\\
&\quad (foldl\ f\ e\ (take\ |b : z|\ x), drop\ |b : z|\ x)
\end{aligned}
$$

The claim, $foldl\ f\ e\ (x +\!\!+ [a]) = f\ (foldl\ f\ e\ x)\ a$, can be straightforwardly proved by structural induction over x. □

Theorem 3 introduces *foldl*$^{\curvearrowleft}$, that is *foldl* in the *There And Back Again* (TABA) pattern (Danvy & Goldberg, 2005). TABA is a programming pattern in which a recursive function traverses a data structure in its call time (to get *there*) and then performs another traversal in its return time (to *back again*).

For example, recall the linear-time reverse function, *lrev*. Since *lrev* is an instance of *foldl*, namely $lrev = foldl\ (\lambda r\ a \to a : r)\ []$, Theorem 3 is applicable and results in *lrev*$^{\curvearrowleft}$ as discussed in the Introduction if we inline parameter functions.

Figure 2 depicts the processes of evaluating *lrev* (1 : 2 : []) and *lrev*$^{\curvearrowleft}$ (1 : 2 : []). The broken-lined boxes stand for a computation at a recursive step, and the whole figures outline stack frames constructed through recursive calls. Down and up arrows denote data flows for accumulative arguments and return values, respectively. It might be helpful to regard these figures as data-flow graphs of attribute grammars. From this viewpoint, down and up arrows, i.e., call-time

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

and return-time computations, could be, respectively, understood as inherited and synthesized attributes.

Note that the down arrows for *lrev* exactly correspond to the up arrows in $lrev^{\curvearrowleft}$. This shows the main idea of IO swapping. IO swapping is a transformation that swaps call-time computations (down arrows) and return-time computation (up arrows) by reversing stack frames. A technical issue is the estimation of the depth of recursive calls. Since a function usually starts its computation from the top of the recursion, its IO-swapped variant should start from the bottom of the recursion. For this purpose, we adopt the TABA pattern: We traverse the first argument and start computation when it is fully destructed.

IO swapping can be extended to tail-recursive functions. We consider the higher order function *loop* as a general form of tail-recursive functions. It takes five arguments, k, p, f, e, and v, and until $p\ v$ holds, it continues the iteration with updating iteration variable v and result e to $k\ v$ and $f\ e\ v$, respectively. For example, *lrev* is an instance of *loop*: $lrev\ x = loop\ (\lambda(\_ : x) \to x)\ null\ (\lambda h\ (a : \_) \to a : h)\ []\ x$, where *null* returns *True* if the given list is empty.

Theorem 4 (*IO swapping for tail-recursive functions*)

The following two functions *loop* and $loop^{\curvearrowleft}$ are equivalent:

$$
\begin{aligned}
&loop :: (a \to a) \to (a \to Bool) \to (b \to a \to b) \to b \to a \to b\\
&loop\ k\ p\ f\ e\ v = \mathbf{if}\ p\ v\ \mathbf{then}\ e\ \mathbf{else}\ loop\ k\ p\ f\ (f\ e\ v)\ (k\ v)
\end{aligned}
$$

$$
\begin{aligned}
&loop^{\curvearrowleft} :: (a \to a) \to (a \to Bool) \to (b \to a \to b) \to b \to a \to b\\
&loop^{\curvearrowleft}\ k\ p\ f\ e\ v = \mathbf{let}\ (r, \_) = aux^{\curvearrowleft}_{loop}\ v\ v\ \mathbf{in}\ r\\
&\quad \mathbf{where}\ aux^{\curvearrowleft}_{loop}\ w\ v = \mathbf{if}\ p\ w\ \mathbf{then}\ (e, v)\\
&\qquad\qquad\qquad\qquad \mathbf{else\ let}\ (r, v') = aux^{\curvearrowleft}_{loop}\ (k\ w)\ v\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (f\ r\ v', k\ v')
\end{aligned}
$$

Proof

Consider function $pk\ v = \mathbf{if}\ p\ v\ \mathbf{then}\ Nothing\ \mathbf{else}\ Just\ (v, k\ v)$. We prove the theorem by confirming the following two claims:

$$
\begin{aligned}
loop\ k\ p\ f\ e\ v &= \{\ \text{claim 1}\ \}\\
&\quad foldl\ f\ e\ (unfoldr\ pk\ v)\\
&= \{\ \text{Theorem 3}\ \}\\
&\quad foldl^{\curvearrowleft}\ f\ e\ (unfoldr\ pk\ v)\\
&= \{\ \text{claim 2}\ \}\\
&\quad loop^{\curvearrowleft}\ k\ p\ f\ e\ v
\end{aligned}
$$

Both claims can be proved by Theorem 2. We show the latter in detail because it is more complex.

In order to use Theorem 2, we work out a *destroy*-form of $foldl^{\curvearrowleft}$. This is not difficult. It is sufficient to abstract out all the destruction (i.e., pattern-matching in

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

this case) of the input. Then we obtain the following function $foldlD^{\curvearrowleft}$.

$$\begin{aligned} foldlD^{\curvearrowleft}\ f\ e\ \psi\ x = {} & \textbf{let}\ (r, x') = aux\ x\ x\ \textbf{in case}\ \psi\ x'\ \textbf{of}\ Nothing \to r \\ & \textbf{where}\ aux\ y\ x = \textbf{case}\ \psi\ y\ \textbf{of}\ Nothing \to (e, x) \\ & \qquad Just\ (\_, y') \to \textbf{let}\ (r, x'') = aux\ y'\ x \\ & \qquad\qquad Just\ (a, x') = \psi\ x'' \\ & \qquad\qquad \textbf{in}\ (f\ r\ a, x') \end{aligned}$$

Note that $destroy\ (foldlD^{\curvearrowleft}\ f\ e)\ x$ yields the following after inlining the parameters.

$$\begin{aligned} destroy\ (foldlD^{\curvearrowleft}\ f\ e)\ x = {} & \textbf{let}\ (r, x') = aux\ x\ x\ \textbf{in case}\ x'\ \textbf{of}\ [] \to r \\ & \textbf{where}\ aux\ y\ x = \textbf{case}\ y\ \textbf{of}\ [] \to (e, x) \\ & \qquad (\_ : y') \to \textbf{let}\ (r, x'') = aux\ y'\ x \\ & \qquad\qquad a : x' = x'' \\ & \qquad\qquad \textbf{in}\ (f\ r\ a, x') \end{aligned}$$

Therefore, $foldl^{\curvearrowleft}\ f\ e\ x = destroy\ (foldlD^{\curvearrowleft}\ f\ e)\ x$ holds. Moreover, $foldlD^{\curvearrowleft}\ f\ e$ has the polymorphic type that Theorem 2 requires. Thus, Theorem 2 is applicable,

$$\begin{aligned} & foldl^{\curvearrowleft}\ f\ e\ (unfoldr\ pk\ v) \\ & \quad = \{\ destroy\ \text{form}\ \} \\ & \quad\ destroy\ (foldlD^{\curvearrowleft}\ f\ e)\ (unfoldr\ pk\ v) \\ & \quad = \{\ \text{Theorem 2}\ \} \\ & \quad\ \textbf{let}\ (r, v') = aux\ v\ v\ \textbf{in}\ \underline{\textbf{case}\ p\ v'\ \textbf{of}\ True} \to r \\ & \qquad \textbf{where}\ aux\ w\ v = \textbf{case}\ p\ w\ \textbf{of}\ True \to (e, v) \\ & \qquad\qquad False \to \textbf{let}\ (r, v') = aux\ (k\ w)\ v \\ & \qquad\qquad \textbf{in}\ \underline{\textbf{case}\ p\ v'\ \textbf{of}\ False} \to (f\ r\ v', k\ v') \end{aligned}$$

Now it is sufficient to confirm that the underlined pattern-matching always succeeds.

Let $k^0\ v = v$ and $k^{n+1}\ v = k^n\ (k\ v)$. Assume that *aux* above performs n recursive calls. From the definition, $p\ (k^{n-1}\ v) = True$ and $p\ (k^m\ v) = False$ for all $0 \leqslant m < n-1$. Now observe the second component of the return value. Starting from the input, v, we repeatedly apply k to it, with confirming that it does not satisfy p, and after $n-1$ times of applications of k, it reaches the top of the recursion and confirms that it satisfies p. This observation proves that all of the underlined pattern-matching succeeds. □

Although the auxiliary function, $aux^{\curvearrowleft}_{loop}$, is not in the TABA pattern, the essence of Theorem 4 is very similar to Theorem 3. $aux^{\curvearrowleft}_{loop}$ uses its first argument for moving to the bottom of the recursion, and then it simulates the computation of *loop* in its return-time computation.

We have seen that call-time computations can be simulated by return-time computations. The following theorem shows that the converse also holds.

Theorem 5 (*IO swapping for foldr*)
The following function $foldr^{\curvearrowleft}$ is equivalent to *foldr*.

$$\begin{aligned} foldr^{\curvearrowleft}\ f\ e\ x = {} & \textbf{let}\ (r, []) = aux^{\curvearrowleft}_{foldr}\ x\ x\ e\ \textbf{in}\ r \\ & \textbf{where}\ aux^{\curvearrowleft}_{foldr}\ []\ x\ h = (h, x) \\ & \qquad aux^{\curvearrowleft}_{foldr}\ (\_ : y)\ x\ h = \textbf{let}\ (r, a : x') = aux^{\curvearrowleft}_{foldr}\ y\ x\ (f\ a\ h) \\ & \qquad\qquad \textbf{in}\ (r, x') \end{aligned}$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

Proof
First, note that $foldr\ f\ e\ x$ is equivalent to $foldl\ (flip\ f)\ e\ (reverse\ x)$, where $flip\ f\ x\ y = f\ y\ x$. Bird and Wadler (1988) call it the third duality theorem. Now we reason as follows:

$$\begin{aligned}
foldr\ f\ e\ x &= \{\ \text{third duality theorem}\ \} \\
&\quad foldl\ (flip\ f)\ e\ (reverse\ x) \\
&= \{\ reverse = lrev = lrev^{\curvearrowleft}\ \} \\
&\quad foldl\ (flip\ f)\ e\ (lrev^{\curvearrowleft}\ x) \\
&= \{\ \text{definition of } lrev^{\curvearrowleft}\ \} \\
&\quad \mathbf{let}\ (r, []) = aux^{\curvearrowleft}_{lrev}\ x\ x\ \mathbf{in}\ foldl\ (flip\ f)\ e\ r \\
&= \{\text{claim: } (\mathbf{let}\ (r, x') = aux^{\curvearrowleft}_{lrev}\ y\ x\ \ \mathbf{in}\ (foldl\ (flip\ f)\ h\ r, x')) \\
&= aux^{\curvearrowleft}_{foldr}\ y\ x\ h\ \}\ foldr^{\curvearrowleft}\ f\ e\ x
\end{aligned}$$

We prove the claim by structural induction over y. The claim obviously holds when y is []. When $y = \_ : z$, we reason as follows:

$$\begin{aligned}
aux^{\curvearrowleft}_{foldr}\ (_ : z)\ x\ h\ &= \{\ \text{definition of } aux^{\curvearrowleft}_{foldr}\ \} \\
&\quad \mathbf{let}\ (r, a : x') = aux^{\curvearrowleft}_{foldr}\ z\ x\ (f\ a\ h)\ \ \mathbf{in}\ (r, x') \\
&= \{\ \text{induction hypothesis}\ \} \\
&\quad \mathbf{let}\ (r', a : x') = aux^{\curvearrowleft}_{lrev}\ z\ x\ \ \mathbf{in}\ (foldl\ (flip\ f)\ (f\ a\ h)\ r', x') \\
&= \{\ \text{definitions of } foldl \text{ and } flip\ \} \\
&\quad \mathbf{let}\ (r', a : x') = aux^{\curvearrowleft}_{lrev}\ z\ x\ \ \mathbf{in}\ (foldl\ (flip\ f)\ h\ (a : r'), x') \\
&= \{\ \text{definition of } aux^{\curvearrowleft}_{lrev}\ \} \\
&\quad \mathbf{let}\ (a : r', x') = aux^{\curvearrowleft}_{lrev}\ (_ : z)\ x\ \ \mathbf{in}\ (foldl\ (flip\ f)\ h\ (a : r'), x')
\end{aligned}$$

Finally, we confirm the termination of $foldr^{\curvearrowleft}$. Note that $aux^{\curvearrowleft}_{foldr}$ is a circular program (Bird, 1984): It uses a part of the second component of its result, a, for computing its third argument. Circularity may lead to nontermination in general, but it is not harmful in this case, because the second component of the result can be calculated without knowing the third argument. □

Like $foldl^{\curvearrowleft}$, the auxiliary function of $foldr^{\curvearrowleft}$ uses the TABA pattern: It moves to the bottom of the recursion by using the first argument, and then it starts consuming the input lists and accumulates its result in its third argument.

Because $foldr^{\curvearrowleft}$ is a circular program, we should take much care of how their computations go so as to avoid nontermination (Bird, 1984). Yet, now that we know that $foldr^{\curvearrowleft}$ terminates, it is safe to see the effect of IO swapping from the perspective of attribute grammars, as shown in Figure 3.

Figure 3 compares two identity functions, $foldr\ (:)\ []$ and $foldr^{\curvearrowleft}\ (:)\ []$. The left picture is standard, and the right one describes the data flow that $foldr^{\curvearrowleft}\ (:)\ []$ composes. Note that down and up arrows correspond to call-time and return-time computations, respectively. Actually, we have so defined "call-time" and "return-time" that they are functional synonyms of "inherited" and "synthesized" in attribute grammars.

Now, like the case of Figure 2, we could find a correspondence between two pictures: The down and up arrows in the left picture, respectively, correspond to the up and down arrows in the center of the right picture; the order of the broken-lined

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

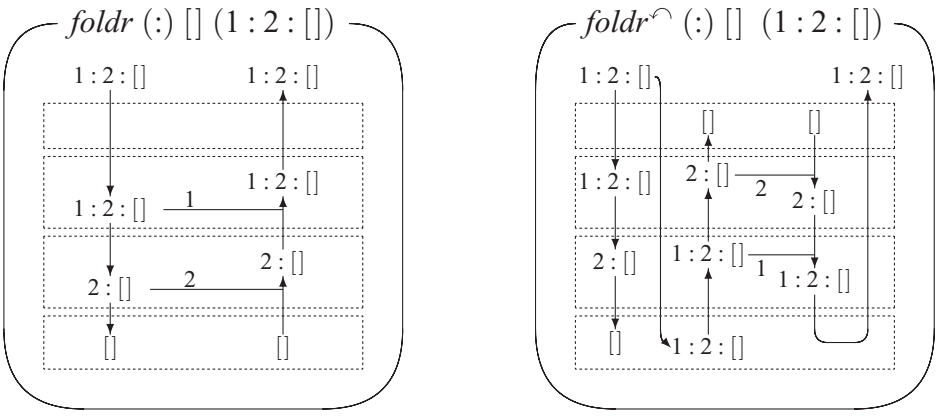

Fig. 3. Outlines of the processes of evaluating *foldr* (:) [] (1 : 2 : []) and *foldr*↶ (:) [] (1 : 2 : []): each broken-lined box stands for a set of data-flow dependencies that a recursive step imposes.

boxes are reverted. Therefore, we can see that *foldr*↶ simulates the computation of *foldr* in its call-time.

Like an extension from Theorem 3 to Theorem 4, we can generalize Theorem 5. We consider functions called list-hylomorphisms (Meijer *et al.*, 1991; Backhouse *et al.*, 1999). List hylomorphisms are captured by the following function *hylo* that takes five arguments: k, p, f, e, and v. Their roles are similar to those of *loop*. k, p, and v take charge of the iteration, and f and e calculate the final result.

Theorem 6 (*IO swapping for list-hylomorphisms*)
The following two functions *hylo* and *hylo*↶ are equivalent.

$$
\begin{aligned}
&hylo :: (a \to a) \to (a \to Bool) \to (a \to b \to b) \to b \to a \to b\\
&hylo\ k\ p\ f\ e\ v = \textbf{if}\ p\ v\ \textbf{then}\ e\ \textbf{else}\ f\ v\ (hylo\ k\ p\ f\ e\ (k\ v))
\end{aligned}
$$

$$
\begin{aligned}
&hylo^{\curvearrowleft} :: (a \to a) \to (a \to Bool) \to (a \to b \to b) \to b \to a \to b\\
&hylo^{\curvearrowleft}\ k\ p\ f\ e\ v = \textbf{let}\ (r, \_) = aux\ v\ v\ e\ \textbf{in}\ r\\
&\quad \textbf{where}\ aux\ w\ v\ h = \textbf{if}\ p\ w\ \textbf{then}\ (h, v)\\
&\qquad\qquad\qquad\qquad \textbf{else let}\ (r, v') = aux\ (k\ w)\ v\ (f\ v'\ h)\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (r, k\ v')
\end{aligned}
$$

Proof
The proof is very similar to that of Theorem 4: We split *hylo* into *foldr* and *unfoldr*, introduce *foldr*↶ by Theorem 5, apply the *destroy*/*unfoldr*-rule, and confirm that constraints introduced by pattern-matching hold. In addition, we note that *aux* is terminating, although it is circular. □

As with Theorem 5, Theorem 6 derives a function that simulates return-time computations at call time by using circularity.

On one hand, Theorems 3 and 4 derive nonaccumulative functions from *foldl* and *loop* that are accumulative, and thus possibly improve manipulability. We

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

demonstrate their effect in Sections 4–7. On the other hand, $foldr^{\curvearrowleft}$ and $hylo^{\curvearrowleft}$ introduced in Theorems 5 and 6 seem not suitable for manipulation because they are accumulative and moreover have circularity. Therefore, we use them backward, i.e., we derive *foldr* or *hylo* from their IO-swapped variants, in Section 8.

We remark on the computational costs of functions IO-swapping derives. As Figures 2 and 3 indicate, the IO-swapped variant does very similar computation to the original, and therefore, their asymptotic time complexities are usually the same.[1] For example, if f and e can be evaluated in constant time, $foldl^{\curvearrowleft}\ f\ e\ x$ runs in time proportional to the length of x. However, IO-swapped variants are usually slower and consume more spaces. They may be non-tail-recursive, although the original ones are tail-recursive. They do multiple traversals over the input; in particular, Theorems 4 and 6 derive functions that invoke k twice as many as the original, which could make the IO-swapped variant twice as slow when k is very costly. Moreover, they are slower because of the additional accumulative parameters and/or return values, and the circularity.

4 Deforesting accumulative functions

Gray code (reflected binary code) is a representation of binary numbers. It is useful for reducing signal errors because two successive numbers differ in only one bit. For example, by three-bits Gray codes, integers from 0 to 7 are respectively represented as 000, 001, 011, 010, 110, 111, 101, and 100.

Consider encoding natural numbers to Gray code. We can naturally develop a function, say *itog*, for this purpose by composing two functions *itoa* and *grayCode*: *itoa* calculates the binary representation of a given natural number with the highest bit first and the lowest bit last; *grayCode* transforms it to Gray code by taking exclusive disjunctions between each bit and its previous bit (the bit preceding the first bit is 0). In the resulting list, the initial occurrences of 0s are omitted. Therefore, applying *itog* to 0, 1, 2, 3, 4, 5, 6, and 7 yields [], [1], [1, 1], [1, 0], [1, 1, 0], [1, 1, 1], [1, 0, 1], and [1, 0, 0], respectively.

$$
\begin{aligned}
&itog\ n = grayCode\ (itoa\ n)\\
&\quad \textbf{where}\ itoa\ v = i2a\ v\ []\\
&\qquad\qquad i2a\ v\ s\ = \textbf{if}\ v \equiv 0\ \textbf{then}\ s\ \textbf{else}\ i2a\ (div\ v\ 2)\ (mod\ v\ 2 : s)\\
&\qquad\qquad grayCode\ s = gc\ s\ 0\\
&\qquad\qquad gc\ []\ \_\ \ \ = []\\
&\qquad\qquad gc\ (a : s)\ f = xor\ a\ f\ : gc\ s\ a
\end{aligned}
$$

Here *xor* calculates an exclusive disjunction, and *div* and *mod* respectively calculate the quotient and the remainder of the integer division.

[1] Their asymptotic time complexities can be different. For instance, the *head* function written by *foldr* runs in O(1) time by virtue of the lazy evaluation, but its IO-swapped variant requires time proportional to the length of the input list.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

Our goal is to deforest the intermediate list passed between *itoa* and *grayCode*. However, naive unfolding–folding cannot deforest it

$$
\begin{aligned}
&\mathit{itog}\ n = \{\ \text{unfolding } \mathit{itog} \text{ and } \mathit{itoa}\ \} \\
&\qquad \mathit{grayCode}\ (\mathit{i2a}\ n\ []) \\
&\quad = \{\ \text{let } \mathit{i2g}^*\ v\ s = \mathit{grayCode}\ (\mathit{i2a}\ v\ s), \text{ and folding } \mathit{i2g}^*\ \} \\
&\qquad \mathit{i2g}^*\ n\ [] \\
&\mathit{i2g}^*\ v\ s = \{\ \text{unfolding } \mathit{i2g}^* \text{ and } \mathit{i2a}\ \} \\
&\qquad \mathit{grayCode}\ (\textbf{if}\ v \equiv 0\ \textbf{then}\ s\ \textbf{else}\ \mathit{i2a}\ (\mathit{div}\ v\ 2)\ (\mathit{mod}\ v\ 2 : s)) \\
&\quad = \{\ \text{distributing } \mathit{grayCode} \text{ to each branch}\ \} \\
&\qquad \textbf{if}\ v \equiv 0\ \textbf{then}\ \mathit{grayCode}\ s\ \textbf{else}\ \mathit{grayCode}\ (\mathit{i2a}\ (\mathit{div}\ v\ 2)\ (\mathit{mod}\ v\ 2 : s)) \\
&\quad = \{\ \text{folding } \mathit{i2g}^*\ \} \\
&\qquad \textbf{if}\ v \equiv 0\ \textbf{then}\ \mathit{grayCode}\ s\ \textbf{else}\ \mathit{i2g}^*\ (\mathit{div}\ v\ 2)\ (\mathit{mod}\ v\ 2 : s)
\end{aligned}
$$

We have got the following program. The intermediate list still remains,

$$
\begin{aligned}
&\mathit{itog}\ n = \mathit{i2g}^*\ n\ [] \\
&\quad \textbf{where}\ \mathit{i2g}^*\ v\ s = \textbf{if}\ v \equiv 0\ \textbf{then}\ \mathit{grayCode}\ s\ \textbf{else}\ \mathit{i2g}^*\ (\mathit{div}\ v\ 2)\ (\mathit{mod}\ v\ 2 : s)
\end{aligned}
$$

The situation is similar to *revMap′* in the Introduction. Though several methods have been proposed for this problem (Kühnemann, 1998; Correnson *et al.*, 1999; Nishimura, 2004; Voigtländer, 2004; Voigtländer & Kühnemann, 2004; Katsumata & Nishimura, 2008), we use IO swapping. We apply Theorem 4 to *itoa* and obtain its IO-swapped variant, say $\mathit{itoa}^{\curvearrowleft}$.

$$
\begin{aligned}
&\mathit{itoa}^{\curvearrowleft}\ v = \textbf{let}\ (r, \_) = \mathit{i2a}^{\curvearrowleft}\ v\ v\ \textbf{in}\ r \\
&\quad \textbf{where}\ \mathit{i2a}^{\curvearrowleft}\ w\ v = \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\
&\qquad\qquad\qquad\qquad \textbf{else let}\ (r, v') = \mathit{i2a}^{\curvearrowleft}\ (\mathit{div}\ w\ 2)\ v \\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{mod}\ v'\ 2 : r, \mathit{div}\ v'\ 2)
\end{aligned}
$$

IO swapping moves the list construction from the call-time to the return-time; then we can deforest it by unfolding–folding,

$$
\begin{aligned}
&\mathit{itog}\ n = \{\ \text{unfolding } \mathit{itog}\ \} \\
&\qquad \mathit{grayCode}\ (\mathit{itoa}\ n) \\
&\quad = \{\ \text{Theorem 4}\ \} \\
&\qquad \mathit{grayCode}\ (\mathit{itoa}^{\curvearrowleft}\ n) \\
&\quad = \{\ \text{unfolding } \mathit{grayCode} \text{ and } \mathit{itoa}^{\curvearrowleft}\ \} \\
&\qquad \mathit{gc}\ (\textbf{let}\ (r, \_) = \mathit{i2a}^{\curvearrowleft}\ n\ n\ \textbf{in}\ r)\ 0 \\
&\quad = \{\ \text{let } \mathit{i2g}\ w\ v\ f = \textbf{let}\ (r, v') = \mathit{i2a}^{\curvearrowleft}\ w\ v\ \textbf{in}\ (\mathit{gc}\ r\ f, v'), \text{ and folding } \mathit{i2g}\ \} \\
&\qquad \textbf{let}\ (r, \_) = \mathit{i2g}\ n\ n\ 0\ \textbf{in}\ r
\end{aligned}
$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

$$
\begin{aligned}
i2g\ w\ v\ f = &\ \{\ \text{unfolding } i2g \text{ and } i2a^{\curvearrowleft}\ \} \\
&\ \textbf{if}\ w \equiv 0\ \textbf{then}\ (gc\ []\ f, v) \\
&\ \textbf{else let}\ (r, v') = i2a^{\curvearrowleft}\ (div\ w\ 2)\ v\ \textbf{in}\ (gc\ (mod\ v'\ 2 : r)\ f, div\ v'\ 2) \\
= &\ \{\ \text{unfolding } gc\ \} \\
&\ \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\
&\ \textbf{else let}\ (r, v') = i2a^{\curvearrowleft}\ (div\ w\ 2)\ v \\
&\qquad \textbf{in}\ (xor\ (mod\ v'\ 2)\ f : gc\ r\ (mod\ v'\ 2), div\ v'\ 2) \\
= &\ \{\ \text{folding } i2g\ \} \\
&\ \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\
&\ \textbf{else let}\ (r', v') = i2g\ (div\ w\ 2)\ v\ (mod\ v'\ 2) \\
&\qquad \textbf{in}\ (xor\ (mod\ v'\ 2)\ f : r', div\ v'\ 2)
\end{aligned}
$$

We have obtained the following program that does not construct any intermediate list:

$$
\begin{aligned}
&itog\ n = \textbf{let}\ (r, \_) = i2g\ n\ n\ 0\ \textbf{in}\ r \\
&\quad \textbf{where}\ i2g\ w\ v\ f = \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\
&\qquad\qquad\qquad\qquad \textbf{else let}\ (r', v') = i2g\ (div\ w\ 2)\ v\ (mod\ v'\ 2) \\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (xor\ (mod\ v'\ 2)\ f : r', div\ v'\ 2)
\end{aligned}
$$

Note that $i2g$ is a circular program: from the second component of its result, v', which retains the inputted decimal number, it calculates its third argument, $mod\ v'\ 2$, which will be used to calculate the next bit. This circularity does not introduce nontermination, because the third argument is unnecessary for calculating the second result. It was pointed out (Nishimura, 2003; Nishimura, 2004; Voigtländer, 2004) that introduction of circularities enables us to deforest accumulative functions. Indeed, IO swapping derived a circular program.

5 Removing higher order value in accumulation

Next, we consider a cooperation between IO swapping and the shortcut deforestation.

We rewrite *grayCode* and *itoa* using *foldr* and *build*, respectively.

$$
\begin{aligned}
&grayCode\ s = foldr\ (\lambda a\ r\ f \to xor\ a\ f : r\ a)\ (\lambda\_ \to [])\ s\ 0 \\
&itoa\ v = build\ (\lambda c\ n \to i2a'\ v\ n\ c) \\
&\quad \textbf{where}\ i2a'\ v\ s\ c = \textbf{if}\ v \equiv 0\ \textbf{then}\ s\ \textbf{else}\ i2a'\ (div\ v\ 2)\ (c\ (mod\ v\ 2)\ s)\ c
\end{aligned}
$$

Then we can apply the shortcut deforestation.

$$
\begin{aligned}
itog\ v = &\ \{\ foldr/build \text{ forms}\ \} \\
&\ foldr\ (\lambda a\ r\ f \to xor\ a\ f : r\ a)\ (\lambda\_ \to [])\ (build\ (\lambda c\ n \to i2a'\ v\ n\ c))\ 0 \\
= &\ \{\ \text{Theorem 1}\ \} \\
&\ i2a'\ v\ (\lambda\_ \to [])\ (\lambda a\ r\ f \to xor\ a\ f : r\ a)\ 0
\end{aligned}
$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

After inlining the parameter functions, we obtain the following program:

$$\begin{array}{l} itog\ n = i2g'\ n\ (\lambda\_ \to [])\ 0 \\ \quad \textbf{where}\ i2g'\ v\ s = \textbf{if}\ v \equiv 0\ \textbf{then}\ s \\ \qquad\qquad\qquad\qquad \textbf{else}\ i2g'\ (div\ v\ 2)\ (\lambda f \to xor\ (mod\ v\ 2)\ f : s\ (mod\ v\ 2)) \end{array}$$

The intermediate list is eliminated. However, instead of it, $i2g'$ accumulates a rather complicated function closure.

There are transformations that eliminate such function closures and thereby obtain a simpler program (Nishimura, 2003; Nishimura, 2004; Katsumata & Nishimura, 2008). We demonstrate that IO swapping can eliminate the closure.

Since $i2g'$ is tail-recursive, Theorem 4 is applicable and yields the following program:

$$\begin{array}{l} itog\ n = \textbf{let}\ (r, \_) = i2g'^{\curvearrowleft}\ n\ n\ \textbf{in}\ r\ 0 \\ \quad \textbf{where}\ i2g'^{\curvearrowleft}\ v\ n = \textbf{if}\ v \equiv 0\ \textbf{then}\ ((\lambda\_ \to []), n) \\ \qquad\qquad\qquad\qquad \textbf{else let}\ (s, v') = i2g'^{\curvearrowleft}\ (div\ v\ 2)\ n \\ \qquad\qquad\qquad\qquad\quad \textbf{in}\ ((\lambda f \to xor\ (mod\ v'\ 2)\ f : s\ (mod\ v'\ 2)), div\ v'\ 2) \end{array}$$

For removing the function closure in the result, we consider another function $ai2g^{\curvearrowleft}$ that supplies an additional argument to $i2g'^{\curvearrowleft}$.

$$ai2g^{\curvearrowleft}\ w\ v\ f = \textbf{let}\ (r, s) = i2g'^{\curvearrowleft}\ w\ v\ \textbf{in}\ (r\ f, s)$$

Then, $itog\ n = \textbf{let}\ (r, \_) = ai2g^{\curvearrowleft}\ n\ n\ 0\ \textbf{in}\ r$ holds. Moreover, $ai2g^{\curvearrowleft}$ does not require function closures. In fact, $ai2g^{\curvearrowleft}$ is $i2g$ that we derived in the previous section, as the following calculation shows:

$$\begin{array}{l} ai2g^{\curvearrowleft}\ w\ v\ f = \{\ \text{unfolding } ai2g^{\curvearrowleft} \text{ and } i2g'^{\curvearrowleft}\ \} \\ \qquad\qquad \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\ \qquad\qquad \textbf{else let}\ (s, v') = i2g'^{\curvearrowleft}\ (div\ w\ 2)\ v \\ \qquad\qquad\quad \textbf{in}\ (xor\ (mod\ v'\ 2)\ f : s\ (mod\ v'\ 2), div\ v'\ 2) \\ \qquad = \{\ \text{folding } ai2g^{\curvearrowleft}\ \} \\ \qquad\qquad \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\ \qquad\qquad \textbf{else let}\ (s', v') = ai2g^{\curvearrowleft}\ (div\ w\ 2)\ v\ (mod\ v'\ 2) \\ \qquad\qquad\quad \textbf{in}\ (xor\ (mod\ v'\ 2)\ f : s', div\ v'\ 2) \end{array}$$

The underlying observation is that higher order results can be removed by supplying additional arguments, and therefore we can remove higher order accumulations by transforming them to higher order results. Indeed we did it by using IO swapping.

As another example, we consider the summation in the continuation-passing style.

$$\begin{array}{l} sumCPS\ x = sumC\ x\ (\lambda v \to v) \\ \quad \textbf{where}\ sumC\ []\ k\ \ \ \ = k\ 0 \\ \qquad\qquad sumC\ (a : x)\ k = sumC\ x\ (\lambda v \to k\ (a + v)) \end{array}$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

Note that $sumCPS\ x = foldl\ (\lambda k\ a\ v \to k\ (a+v))\ (\lambda v \to v)\ x\ 0$. Therefore, Theorem 3 is applicable and yields the following program:

$$\begin{aligned} sumCPS^{\curvearrowleft}\ x = {} & \mathbf{let}\ (k, []) = sumC^{\curvearrowleft}\ x\ x\ \mathbf{in}\ k\ 0 \\ & \mathbf{where}\ sumC^{\curvearrowleft}\ []\ x \quad = (\lambda v \to v, x) \\ & \quad sumC^{\curvearrowleft}\ (\_ : y)\ x = \mathbf{let}\ (k, a : x') = sumC^{\curvearrowleft}\ y\ x \\ & \qquad\qquad\qquad\qquad \mathbf{in}\ (\lambda v \to k\ (a + v), x') \end{aligned}$$

Now we can remove the higher order result by supplying an additional argument

$$sumC'^{\curvearrowleft}\ y\ x\ v = \mathbf{let}\ (k, x') = sumC^{\curvearrowleft}\ y\ x\ \mathbf{in}\ (k\ v, x')$$

We calculate a recursive definition of $sumC'^{\curvearrowleft}$ as follows:

$$\begin{aligned} sumC'^{\curvearrowleft}\ []\ x\ v = {} & \{\ \text{unfolding } sumC'^{\curvearrowleft}\ \} \\ & \mathbf{let}\ (k, x') = sumC^{\curvearrowleft}\ []\ x\ \mathbf{in}\ (k\ v, x') \\ = {} & \{\ \text{unfolding } sumC^{\curvearrowleft}\ \} \\ & (v, x) \end{aligned}$$

$$\begin{aligned} sumC'^{\curvearrowleft}\ (\_ : y)\ x\ v = {} & \{\ \text{unfolding } sumC'^{\curvearrowleft} \text{ and } sumC^{\curvearrowleft}\ \} \\ & \mathbf{let}\ (k, a : x') = sumC^{\curvearrowleft}\ y\ x\ \mathbf{in}\ (k\ (a + v), x') \\ = {} & \{\ \text{folding } sumC'^{\curvearrowleft}\ \} \\ & \mathbf{let}\ (r, a : x') = sumC'^{\curvearrowleft}\ y\ x\ (a + v)\ \mathbf{in}\ (r, x') \end{aligned}$$

We have developed the following first-order program:

$$\begin{aligned} sumCPS'^{\curvearrowleft}\ x = {} & \mathbf{let}\ (r, []) = sumC'^{\curvearrowleft}\ x\ x\ 0\ \mathbf{in}\ r \\ & \mathbf{where}\ sumC'^{\curvearrowleft}\ []\ x\ v \quad = (v, x) \\ & \quad sumC'^{\curvearrowleft}\ (\_ : y)\ x\ v = \mathbf{let}\ (r, a : x') = sumC'^{\curvearrowleft}\ y\ x\ (a + v) \\ & \qquad\qquad\qquad\qquad \mathbf{in}\ (r, x') \end{aligned}$$

This is also a terminating circular program. Nevertheless, it is different from the familiar summation function. We will revisit it in Section 8.

6 Inverting accumulative function

The next transformation is program inversion, which generates a program that takes an output and returns the input of the original program. We consider *parse* that parses expressions written in the reverse Polish notation. For example, $parse$ [“21”, “5”, “36”, “+”, “*”] $= Op$ “*” (Val “21”) (Op “+” (Val “5”) (Val “36”)). The following is a program for *parse*, in which we use predicate *isOp* that checks for operators

$$\begin{aligned} parse\ x = {} & \mathbf{let}\ [e] = ps\ x\ []\ \mathbf{in}\ e \\ & \mathbf{where}\ ps\ []\ st \quad = st \\ & \quad ps\ (a : x)\ st = \mathbf{if}\ isOp\ a\ \mathbf{then}\ \mathbf{let}\ (e_1 : e_2 : st') = st \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ ps\ x\ (Op\ a\ e_2\ e_1 : st') \\ & \qquad\qquad\qquad\qquad \mathbf{else}\ ps\ x\ (Val\ a : st) \end{aligned}$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

We would like to derive its inverse, *unparse*, such that *unparse* (*parse* x) $= x$. But it is nontrivial. Consider the following reasoning:

$$\begin{aligned} &\mathit{unparse}\ (\mathit{parse}\ x) = x \Rightarrow \{\ \text{assume that } x = a : x' \text{ and } \neg(\mathit{isOp}\ a);\ \text{unfolding}\ \} \\ &\qquad\qquad \mathit{unparse}\ (\mathit{ps}\ x'\ [\mathit{Val}\ a]) \mid \neg(\mathit{isOp}\ a) = a : x' \end{aligned}$$

Here we get stuck. It is unclear what *ps* x' [*Val* a] will result in, and therefore we cannot know when *unparse* should result in $a : x'$. Several methods have been proposed (Glück & Kawabe, 2005; Mogensen, 2006; Matsuda *et al.*, 2010) for solving this problem.

By swapping the accumulative parameter and the result, IO swapping makes it apparent what kinds of outputs are generated from each branch and thereby makes inversion easier. Let us see its effect. First, we decompose *parse* as *parse* $x =$ *unwrap* (*ps* x []) where *unwrap* $[r] = r$. Since *ps* is an instance of *foldl*, Theorem 3 is applicable and results in the following:

$$\begin{aligned} \mathit{parse}\ x = {} & \mathbf{let}\ (r', []) = \mathit{ps}^{\curvearrowleft}\ x\ x\ \mathbf{in}\ \mathit{unwrap}\ r' \\ & \mathbf{where}\ \mathit{ps}^{\curvearrowleft}\ []\ x = ([], x) \\ & \quad \mathit{ps}^{\curvearrowleft}\ (\_ : y)\ x = \mathbf{let}\ (st, a : x') = \mathit{ps}^{\curvearrowleft}\ y\ x \\ & \qquad\qquad \mathbf{in\ if}\ \mathit{isOp}\ a\ \mathbf{then\ let}\ (e_1 : e_2 : st') = st \\ & \qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (\mathit{Op}\ a\ e_2\ e_1 : st', x') \\ & \qquad\qquad\ \mathbf{else}\ (\mathit{Val}\ a : st, x') \end{aligned}$$

The derived function may appear to be unsuitable for inversion because $ps^{\curvearrowleft}$ discards its first argument and therefore is not injective. This is not a problem. Recall that IO swapping introduces the first argument only for counting the depth of the recursion. In other words, the first argument is not important for inversion. Therefore, we work out function *ups* that satisfies the following equation

$$\mathit{ups}\ (\mathit{ps}^{\curvearrowleft}\ y\ x) = x$$

It is indeed sufficient for our purpose, as the following calculation confirms:

$$\begin{aligned} \mathit{unparse}\ (\mathit{parse}\ x) = x \Leftrightarrow {} & \{\ \text{unfolding } \mathit{parse}\ \} \\ & \mathit{unparse}\ (\mathbf{let}\ (r', []) = \mathit{ps}^{\curvearrowleft}\ x\ x\ \mathbf{in}\ \mathit{unwrap}\ r') = x \\ \Leftrightarrow {} & \{\ \text{definition of } \mathit{ups}\ \} \\ & \mathit{unparse}\ (\mathbf{let}\ (r', []) = \mathit{ps}^{\curvearrowleft}\ x\ x\ \mathbf{in}\ \mathit{unwrap}\ r') = \mathit{ups}\ (\mathit{ps}^{\curvearrowleft}\ x\ x) \\ \Leftrightarrow {} & \{\ \text{definition of } \mathit{unwrap} \text{ and simplification}\ \} \\ & \mathit{unparse}\ r = \mathit{ups}\ ([r], []) \end{aligned}$$

Now let us develop a recursive definition of *ups*.

$$\begin{aligned} &\mathit{ups}\ (\mathit{ps}^{\curvearrowleft}\ []\ x) = x \Leftrightarrow \{\ \text{unfolding } \mathit{ps}^{\curvearrowleft}\ \} \\ &\qquad\qquad \mathit{ups}\ ([], x) = x \\ &\mathit{ups}\ (\mathit{ps}^{\curvearrowleft}\ (\_ : y)\ x) = x \\ &\quad \Leftrightarrow \{\ \text{definition of } \mathit{ups}\ \} \\ &\quad\ \ \mathit{ups}\ (\mathit{ps}^{\curvearrowleft}\ (\_ : y)\ x) = \mathit{ups}\ (\mathit{ps}^{\curvearrowleft}\ y\ x) \\ &\quad \Leftrightarrow \{\ \text{assume } (e_1 : e_2 : st', a : x') = \mathit{ps}^{\curvearrowleft}\ y\ x \text{ and } \mathit{isOp}\ a;\ \text{unfolding } \mathit{ps}^{\curvearrowleft}\ \} \\ &\quad\ \ \mathit{ups}\ (\mathit{Op}\ a\ e_2\ e_1 : st', x') \mid \mathit{isOp}\ a = \mathit{ups}\ (e_1 : e_2 : st', a : x') \end{aligned}$$

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

$$
\begin{array}{l}
ups\ (ps^{\curvearrowleft}\ (\_ : y)\ x) = x \\
\quad \Leftrightarrow \{\ \text{definition of } ups\ \} \\
\quad\quad ups\ (ps^{\curvearrowleft}\ (\_ : y)\ x) = ups\ (ps^{\curvearrowleft}\ y\ x) \\
\quad \Leftrightarrow \{\ \text{assume } (st, a : x') = ps^{\curvearrowleft}\ y\ x \text{ and } \neg(isOp\ a);\ \text{unfolding } ps^{\curvearrowleft}\ \} \\
\quad\quad ups\ (Val\ a : st, x') \mid \neg(isOp\ a) = ups\ (st, a : x')
\end{array}
$$

We have obtained the following program:

$$
\begin{array}{l}
unparse\ r = ups\ ([r], []) \\
\quad \textbf{where}\ ups\ ([], x) = x \\
\quad\quad\quad ups\ (Op\ a\ e_2\ e_1 : st', x') \mid isOp\ a = ups\ (e_1 : e_2 : st', a : x') \\
\quad\quad\quad ups\ (Val\ a : st, x') \mid \neg(isOp\ a) = ups\ (st, a : x')
\end{array}
$$

This is certainly an unparsing program for the reverse Polish notation.

7 IO swapping applied to tree-traversing functions

So far we have considered linearly recursive functions because IO swapping is only applicable to them. By linearizing the recursion (Wand, 1980; Boiten, 1992), we can apply IO swapping to nonlinearly recursive functions. Although linearization usually yields complicated programs, this approach sometimes leads to successful manipulations.

As an example, let us fuse $revflat = lrev \circ flatten$. Function *lrev* is that seen in the Introduction, and *flatten* gathers leaves in a tree,

$$
\begin{array}{l}
flatten\ t = flat\ t\ [] \\
\quad \textbf{where}\ flat\ (Leaf\ a)\ h = a : h \\
\quad\quad\quad flat\ (Fork\ l\ r)\ h = flat\ l\ (flat\ r\ h)
\end{array}
$$

Since both *lrev* and *flatten* are accumulative, their fusion is nontrivial. The situation is very similar to *grayCode* discussed in Sections 4 and 5: unfolding-folding cannot deforest the intermediate list, and the *foldr/build* rule yields a program that accumulates complicated closures, whereas more sophisticated methods (Kühnemann, 1998; Correnson *et al.*, 1999; Nishimura, 2003; Nishimura, 2004; Voigtländer, 2004; Voigtländer & Kühnemann, 2004; Katsumata & Nishimura, 2008) can deal with it.

We linearize *flat* by introducing a stack

$$
\begin{array}{l}
flatten\ t = flat_1\ [t]\ [] \\
\quad \textbf{where}\ flat_1\ []\ h = h \\
\quad\quad\quad flat_1\ (Leaf\ a : st)\ h = flat_1\ st\ (a : h) \\
\quad\quad\quad flat_1\ (Fork\ l\ r : st)\ h = flat_1\ (r : l : st)\ h
\end{array}
$$

Now that $flat_1$ is a linear tail-recursive function, IO swapping can deforest the intermediate list. We apply Theorem 4 for removing the accumulative computation.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

For clarity, we introduce two additional auxiliary functions, *addVal* and *nextStack*.

$$
\begin{array}{l}
\mathit{flatten}\ t = \textbf{let}\ (r, \_) = \mathit{flat}_1^{\curvearrowleft}\ [t]\ [t]\ \textbf{in}\ r \\
\quad \textbf{where}\ \mathit{flat}_1^{\curvearrowleft}\ []\ t' \qquad = ([], t') \\
\qquad\qquad \mathit{flat}_1^{\curvearrowleft}\ (s' : st')\ t' = \textbf{let}\ (h, s : st) = \mathit{flat}_1^{\curvearrowleft}\ (\mathit{nextStack}\ s'\ st')\ t' \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\mathit{addVal}\ s\ h, \mathit{nextStack}\ s\ st) \\
\qquad\qquad \mathit{addVal}\ (\mathit{Leaf}\ a)\ k\ = a : k \\
\qquad\qquad \mathit{addVal}\ (\mathit{Fork}\ l\ r)\ k = k \\
\qquad\qquad \mathit{nextStack}\ (\mathit{Leaf}\ a)\ st\ = st \\
\qquad\qquad \mathit{nextStack}\ (\mathit{Fork}\ l\ r)\ st = r : l : st
\end{array}
$$

Then, we fuse *lrev* with *flatten* by using the unfolding–folding method.

$$
\begin{array}{l}
\mathit{revflat}\ t = \{\ \text{unfolding } \mathit{revflat}, \mathit{lrev}, \text{ and } \mathit{flatten}\ \} \\
\qquad\qquad \mathit{aux}_{\mathit{lrev}}\ (\textbf{let}\ (r, \_) = \mathit{flat}_1^{\curvearrowleft}\ [t]\ [t]\ \textbf{in}\ r)\ [] \\
\qquad\quad = \{\text{let } \mathit{rf}\ \mathit{st}'\ t'\ k = (\textbf{let}\ (r, st) = \mathit{flat}_1^{\curvearrowleft}\ \mathit{st}'\ t'\ \textbf{in}\ (\mathit{aux}_{\mathit{lrev}}\ r\ k, st)), \\
\qquad\qquad \text{and folding } \mathit{rf}\}\ \textbf{let}\ (r, \_) = \mathit{rf}\ [t]\ [t]\ []\ \textbf{in}\ r
\end{array}
$$

$$
\begin{array}{l}
\mathit{rf}\ []\ t'\ k = \{\ \text{unfolding } \mathit{rf} \text{ and } \mathit{flat}_1^{\curvearrowleft}\ \} \\
\qquad\qquad (\mathit{aux}_{\mathit{lrev}}\ []\ k, t') \\
\qquad\quad = \{\ \text{unfolding } \mathit{aux}_{\mathit{lrev}}\ \} \\
\qquad\qquad (k, t')
\end{array}
$$

$$
\begin{array}{l}
\mathit{rf}\ (s' : st')\ t'\ k = \{\ \text{unfolding } \mathit{rf}, \mathit{flat}_1^{\curvearrowleft}, \text{ and } \mathit{addVal}\ \} \\
\qquad\qquad\qquad \textbf{let}\ (h, s : st) = \mathit{flat}_1^{\curvearrowleft}\ (\mathit{nextStack}\ s'\ st')\ t' \\
\qquad\qquad\qquad \textbf{in case}\ s\ \textbf{of}\ \mathit{Fork}\ l\ r \to (\mathit{aux}_{\mathit{lrev}}\ h\ k, \mathit{nextStack}\ s\ st) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{Leaf}\ a \to (\mathit{aux}_{\mathit{lrev}}\ (a : h)\ k, \mathit{nextStack}\ s\ st) \\
\qquad\qquad = \{\ \text{unfolding } \mathit{aux}_{\mathit{lrev}}\ \} \\
\qquad\qquad\qquad \textbf{let}\ (h, s : st) = \mathit{flat}_1^{\curvearrowleft}\ (\mathit{nextStack}\ s'\ st')\ t' \\
\qquad\qquad\qquad \textbf{in case}\ s\ \textbf{of}\ \mathit{Fork}\ l\ r \to (\mathit{aux}_{\mathit{lrev}}\ h\ k, \mathit{nextStack}\ s\ st) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{Leaf}\ a \to (\mathit{aux}_{\mathit{lrev}}\ h\ (a : k), \mathit{nextStack}\ s\ st) \\
\qquad\qquad = \{\ \text{folding } \mathit{rf} \text{ and } \mathit{addVal}\ \} \\
\qquad\qquad\qquad \textbf{let}\ (h', s : st) = \mathit{rf}\ (\mathit{nextStack}\ s'\ st')\ t'\ (\mathit{addVal}\ s\ k) \\
\qquad\qquad\qquad \textbf{in}\ (h', \mathit{nextStack}\ s\ st)
\end{array}
$$

We have developed the following program:

$$
\begin{array}{l}
\mathit{revflat}\ t = \textbf{let}\ (r, \_) = \mathit{rf}\ [t]\ [t]\ []\ \textbf{in}\ r \\
\quad \textbf{where}\ \mathit{rf}\ []\ t'\ k \qquad = (k, t') \\
\qquad\quad \mathit{rf}\ (s' : st')\ t'\ k = \textbf{let}\ (h', s : st) = \mathit{rf}\ (\mathit{nextStack}\ s'\ st')\ t'\ (\mathit{addVal}\ s\ k) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (h', \mathit{nextStack}\ s\ st)
\end{array}
$$

As expected, the derived program does not have any intermediate list except for the stack. Note that *rf* is circular: its accumulative argument, *addVal s k*, uses its return value, *s*. As in the case of *itog*, we obtained a circular function by fusing accumulative functions.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

8 Manipulating circular functions

In the previous section, we have performed deforestation and obtained a program for *revflat*. However, it is not satisfactory. The program traverses the input tree twice, and moreover it has a circularity. We would like to develop a simpler program.

Note that *revflat* is an instance of $hylo^{\curvearrowleft}$ in Theorem 6. Therefore, we can eliminate the circularity by transforming *revflat* to its equivalent *hylo* form. Theorem 6 yields the following simpler program that contains no circularity:

$$\begin{aligned} &revflat\ t = rf_1^{\curvearrowright}\ [t] \\ &\quad \textbf{where}\ rf_1^{\curvearrowright}\ [] \quad\ = [] \\ &\qquad\qquad rf_1^{\curvearrowright}\ (s : st) = addVal\ s\ (rf_1^{\curvearrowright}\ (nextStack\ s\ st)) \end{aligned}$$

This circularity removal is not magic. A circularity is a situation where an argument depends on a return value. Then by swapping call-time and return-time computations, IO swapping transforms a circularity to a situation where a return value depends on an argument.

As another example, recall $sumCPS'^{\curvearrowleft}$ is discussed in Section 5. $sumCPS'^{\curvearrowleft}$ is an instance of $foldr^{\curvearrowleft}$ in Theorem 5, and therefore the theorem enables us to remove the circularity and results in the following usual summation function:

$$\begin{aligned} sum\ [] &= 0 \\ sum\ (a : x) &= a + sum\ x \end{aligned}$$

Nevertheless, IO swapping often fails to transform circular programs to noncircular ones. For instance, recall *itog* that we derived in Sections 4 and 5. We repeat the program for convenience,

$$\begin{aligned} &itog\ n = \textbf{let}\ (r, \_) = i2g\ n\ n\ 0\ \textbf{in}\ r \\ &\quad \textbf{where}\ i2g\ w\ v\ f = \textbf{if}\ w \equiv 0\ \textbf{then}\ ([], v) \\ &\qquad\qquad\qquad\qquad \textbf{else let}\ (r, v') = i2g\ (div\ w\ 2)\ v\ (mod\ v'\ 2) \\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\ (xor\ (mod\ v'\ 2)\ f : r, div\ v'\ 2) \end{aligned}$$

It has a circularity: in the argument of $i2g$, $mod\ v'\ 2$, uses a part of the return value.

None of the IO swapping rules are applicable for this program. Even so we can consider swapping call-time and return-time computations based on the core idea of IO swapping. Observe that the first argument of *i2g* is only used for moving to the bottom of the recursion. Then it starts the computation by using the initial input as if it simulates computations in arguments at return time. Based on this observation we may think of the following $i2g^{\dagger}$, which is in fact equivalent to *i2g*

$$\begin{aligned} &itog\ n = \textbf{let}\ (r', \_) = i2g^{\dagger}\ n\ []\ \textbf{in}\ r' \\ &\quad \textbf{where}\ i2g^{\dagger}\ v'\ r = \textbf{if}\ v' \equiv 0\ \textbf{then}\ (r, 0) \\ &\qquad\qquad\qquad\qquad \textbf{else let}\ (r', f) = i2g^{\dagger}\ (div\ v'\ 2)\ (xor\ (mod\ v'\ 2)\ f : r) \\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\ (r', mod\ v'\ 2) \end{aligned}$$

Now function *mod* uses an argument rather than a return value. However, $i2g^{\dagger}$ is a circular program. Result f is used in an argument. This is because IO swapping derives a circularity from a situation where a return value depends on an argument.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

9 Related work and concluding remarks

We introduced IO swapping that swaps call-time and return-time computations and demonstrated its usefulness for manipulating accumulative, in particular, tail-recursive functions.

IO swapping originates from the TABA programming pattern devised by Danvy and Goldberg (2005). While usual recursive functions traverse data structures in the call time, those in the TABA pattern do in the return time. Several interesting programs could be developed by using the TABA pattern, including $lrev^{\curvearrowleft}$, $foldl^{\curvearrowleft}$, $foldr^{\curvearrowleft}$, $sumCPS'^{\curvearrowleft}$, and *revflat*. More examples could be found in the paper by Danvy and Goldberg (2005) and in a textbook by Roy and Haridi (2004) (Exercise 16).

We originally formulated IO swapping rules for systematically developing non-trivial programs in the TABA pattern (Morihata *et al.*, 2006). Indeed, the original IO swapping rule, Theorem 3, relies on the TABA pattern. Later, we noticed that its core idea, swapping call-time and return-time computations, could have further impacts: traversals of data structures are not essential (Theorem 4); we can relate usual programs to circular programs (Theorems 5 and 6); and furthermore, deriving the TABA-like programming patterns is sometimes effective for manipulating accumulative programs. Such possibilities were partially suggested in the preliminary report (Morihata *et al.*, 2006). In this paper, we demonstrated the effect of IO swapping in program manipulations.

The first author's (Morihata, 2006) Master's thesis contains other applications of IO swapping, including developments of programs in the TABA pattern and deforestation for circular programs. In addition, it contains a more generic IO-swapping rule that can derive $i2g^{\dagger}$ from $i2g$. The generic rule requires a nontrivial precondition and its proof is intricate; moreover, it seems that there are few interesting applications in practice. Therefore, we have introduced simpler rules in Section 3 together with their proofs in a transformational manner. It is worth noting that Theorems 4 and 6 specify the functions whose IO-swapped variant has no circularities; thus, they are sufficient for eliminating circularities.

As mentioned in the Introduction, accumulative functions are known to be less suitable for program manipulations, and several methods have been proposed. IO swapping is neither more powerful nor simpler than the existing methods. However, though they have been developed independently, IO swapping enables us to uniformly apply several kinds of manipulations.

Wadler (1990) proposed a deforestation method based on the unfolding–folding. It often fails to deforest intermediate structures produced in accumulative parameters (Chin, 1994; Kühnemann, 1999). In order to resolve this issue, Kühnemann (1998) and Correnson *et al.* (1999) proposed deforestation methods that first translate functional programs to attribute grammars and then apply a method of composing attribute grammars. Voigtländer (2004) and Voigtländer and Kühnemann (2004) developed more direct methods that avoid the translation to attribute grammars. Essentially, they could deal with all the deforestation examples presented here if we neglect technical differences caused by their formalisms. In particular, they could

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press

derive $i2g^{\dagger}$ seen in Section 8 for *grayCode*. Moreover, they are much simpler, especially when dealing with nonlinear recursive functions. It is worth noting that we also borrowed some intuition from attribute grammars, as seen in Figure 2, and proposed IO swapping that swaps the roles of inherited and synthesized attributes and thereby making them more equally manipulable.

Another well-known method of deforestation is shortcut deforestation (Gill *et al.*, 1993; Takano & Meijer, 1995; Gill, 1996; Svenningsson, 2002). Applying shortcut deforestation to a composition of two accumulative functions has the problem of yielding functions that construct complicated function closures. Nishimura (2003, 2004) and Katsumata and Nishimura (2008) tackled this problem and formulated higher order removal methods. Their methods could derive $i2g^{\dagger}$ and *sum* from $i2g'$ and *sumCPS*, respectively, if we additionally apply a kind of copy propagation; unfortunately, its connections to the reverse CPS transformation were not noticed.

Accumulative functions are harder to apply inversion. Glück and Kawabe (2005) and Matsuda *et al.* (2010) proposed parsing-based methods for dealing with them. Our inversion can be seen as a variant of the method by Mogensen (2006) who considered a loop combinator and introduced a rule for inverting it.

Circular programs were first invented by Bird (1984), and their manipulation is a topic of active research (Voigtländer, 2004; Fernandes *et al.*, 2007; Fernandes & Saraiva, 2007; Katsumata & Nishimura, 2008; Pardo *et al.*, 2009; Fernandes *et al.*, 2011). Most of them consider developing circular programs from composition of functions, or eliminating circularities by decomposing functions, while IO swapping only concerns one function.

It is known that mechanical reasoning about accumulative programs is difficult (Boyer & Moore, 1975; Boyer *et al.*, 1976; Giesl, 2000; Giesl *et al.*, 2007). Induction is a main method of mechanical reasoning, and accumulative parameters make it difficult to provide appropriate induction hypotheses. For instance, recall *revMap* discussed in the Introduction and reconsider proving $map\ f\ (lrev\ x) = revMap$. We saw that $aux_{revMap}\ f\ x\ h = map\ f\ (aux_{lrev}\ x\ h)$ was not an appropriate induction hypothesis; in fact, an appropriate one is $aux_{revMap}\ f\ x\ (map\ f\ h) = map\ f\ (aux_{lrev}\ x\ h)$. To resolve this difficulty, program transformations, called *deaccumulation*, have been proposed (Boyer *et al.*, 1976; Giesl, 2000; Kühnemann *et al.*, 2001; Giesl *et al.*, 2007). Deaccumulation derives nonaccumulative programs from accumulative ones. For example, it derives *reverse* from *lrev*. Deaccumulation techniques are not designed for program manipulation, and they do not consider efficiency of derived programs. They may significantly make programs slower as the case of reverse functions. IO swapping usually does not change asymptotic time complexity.

Acknowledgments

We are grateful to several people for their encouragement. Olivier Danvy advised us to seek more applications and suggested some examples. Jeremy Gibbons recommended us proving the IO swapping theorems calculationally and explained an idea for it. He also gave useful comments on this paper. Varmo Vene and

Alberto Pardo emphasized their interest in applying to nonlinear recursions. Their encouragement led to further development of IO swapping. The transformational proof of Theorem 3 is given by Shin-Cheng Mu. We are also grateful to anonymous reviewers for their instructive comments and suggestions of references.

References

Backhouse, R. C., Jansson, P., Jeuring, J. & Meertens, L. G. L. T. (1999) Generic programming: An introduction. In *Advanced Functional Programming, Third International School, Braga, Portugal, Revised Lectures*. Lecture Notes in Computer Science, 1608, pp. 28–115. Berlin, Germany: Springer-Verlag.

Bird, R. S. (1984) Using circular programs to eliminate multiple traversals of data. *Acta Inf.* **21**, 239–250.

Bird, R. S. & Wadler, P. (1989) *Introduction to Functional Programming*. Saddle River, NJ: Prentice-Hall.

Boiten, E. A. (1992) Improving recursive functions by inverting the order of evaluation. *Sci. Comput. Program.* **18**(2), 139–179.

Boyer, R. S. & Moore, J. S. (1975) Proving theorems about Lisp functions. *J. ACM*. **22**(1), 129–144.

Boyer, R. S., Moore, J. S. & Shostak, R. E. (1976) Primitive recursive program transformations. In *Proceedings of POPL'76: Conference Record of the Third ACM Symposium on Principles of Programming Languages*, Atlanta, Georgia. Pittsburgh, PA: ACM Press, pp. 171–174.

Burstall, R. M. & Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*. **24**(1), 44–67.

Chin, W.-N. (1994) Safe fusion of functional expressions II: Further improvements. *J. Funct. Program.* **4**(4), 515–555.

Correnson, L., Duris, É., Parigot, D. & Roussel, G. (1999) Declarative program transformation: A deforestation case-study. In *Proceedings of International Conference on Principles and Practice of Declarative Programming (PPDP'99), Paris, France*. Lecture Notes in Computer Science, vol. 1702. Berlin, Germany: Springer-Verlag, pp. 360–377.

Danvy, O. & Goldberg, M. (2005) There and back again. *Fundam. Inform.* **66**(4), 397–413.

Fernandes, J. P., Pardo, A. & Saraiva, J (2007) A shortcut fusion rule for circular program calculation. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, Freiburg, Germany. Pittsburgh, PA: ACM, pp. 95–106.

Fernandes, J. P. & Saraiva, J. (2007) Tools and libraries to model and manipulate circular programs. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Nice, France. Pittsburgh, PA: ACM, pp. 102–111.

Fernandes, J. P., Saraiva, J., Seidel, D. & Voigtländer, J. (2011) Strictification of circular programs. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*, Austin, TX, USA. Pittsburgh, PA: ACM, pp. 131–140.

Giesl, J. (2000). Context-moving transformations for function verification. In *Proceedings of the 9th International Workshop on Logic Programming Synthesis and Transformation (LOPSTR'99)*, selected papers. Lecture Notes in Computer Science, vol. 1817. Berlin, Germany: Springer-Verlag, pp. 293–312.

Giesl, J., Kühnemann, A. & Voigtländer, J. (2007) Deaccumulation techniques for improving provability. *J. Log. Algebr. Program.* **71**(2), 79–113.

Gill, A. (1996) *Cheap Deforestation for Non-Strict Functional Languages*. PhD. thesis, Department of Computing Science, Glasgow University, Glasgow, UK.

Gill, A., Launchbury, J. & Peyton Jones, S. (1993) A short cut to deforestation. In *Proceedings of FPCA'93 Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark. New York, NY: ACM Press, pp. 223–232.

Glück, R. & Kawabe, M. (2005) A method for automatic program inversion based on LR(0) parsing. *Fundam. Inform.* **66**(4), 367–395.

Katsumata, S. & Nishimura, S. (2008) Algebraic fusion of functions with an accumulating parameter and its improvement. *J. Funct. Program.* **18**(5–6), 781–819.

Kühnemann, A. (1998) Benefits of tree transducers for optimizing functional programs. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, Chennai, India. Lecture Notes in Computer Science, vol. 1530. Berlin, Germany: Springer-Verlag, pp. 146–157.

Kühnemann, A. (1999) Comparison of deforestation techniques for functional programs and for tree transducers. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, Tukuba, Japan. Lecture Notes in Computer Science, vol. 1722. Berlin, Germany: Springer-Verlag, pp. 114–130.

Kühnemann, A., Glück, R. & Kakehi, K. (2001) Relating accumulative and non-accumulative functional programs. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, Utrecht, the Netherlands. Lecture Notes in Computer Science, vol. 2051. Berlin, Germany: Springer-Verlag, pp. 154–168.

Matsuda, K., Mu, S.-C., Hu, Z. & Takeichi, M. (2010) A grammar-based approach to invertible programs. In *Proceedings of the 19th European Symposium on Programming Languages and Systems (ESOP 2010), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2010)*, Paphos, Cyprus. Lecture Notes in Computer Science, vol. 6012. Berlin, Germany: Springer-Verlag, pp. 448–467.

Meijer, E., Fokkinga, M. M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture,* Cambridge, MA, USA . Lecture Notes in Computer Science, vol. 523. Berlin, Germany: Springer-Verlag, pp. 124–144.

Mogensen, T. Æ. (2006) Report on an implementation of a semi-inverter. In *Proceedings of 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI 2006)*, Novosibirsk, Russia, revised papers. Lecture Notes in Computer Science, vol. 4378. Berlin, Germany: Springer-Verlag, pp. 322–334.

Morihata, A. (2006) *Relationship between Arguments and Results of Recursive Functions.* Master's thesis, Graduate School of Information Science and Technology, University of Tokyo, Japan.

Morihata, A., Kakehi, K., Hu, Z. & Takeichi, M. (2006) Swapping arguments and results of recursive functions. In *Proceedings of the 8th International Conference on Mathematics of Program Construction (MPC 2006)*, Kuressaare, Estonia. Lecture Notes in Computer Science, vol. 4014. Berlin, Germany: Springer-Verlag, pp. 379–396.

Nishimura, S. (2003) Correctness of a higher-order removal transformation through a relational reasoning. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, Beijing, China. Lecture Notes in Computer Science, vol. 2895. Berlin, Germany: Springer-Verlag, pp. 358–375.

Nishimura, S. (2004) Fusion with stacks and accumulating parameters. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'04)*, Verona, Italy. New York, NY: ACM Press, pp. 101–112.

Pardo, A., Fernandes, J. P. & Saraiva, J. (2009) Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2009)*, Savannah, GA, USA. New York, NY: ACM Press, pp. 81–90.

Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge, UK: Cambridge University Press.

Roy, P. V. & Haridi, S. (2004) *Concepts, Techniques, and Models of Computer Programming*. Cambridge, MA: MIT Press.

Svenningsson, J. (2002) Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, Pittsburgh, Pennsylvania, USA. New York, NY: ACM Press, pp. 124–132.

Takano, A. & Meijer, E. (1995) Shortcut deforestation in calculational form. In *Proceedings of SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture (conference record of FPCA'95)*, La Jolla, CA, USA. New York, NY: ACM Press, pp. 306–313.

Voigtländer, J. (2004) Using circular programs to deforest in accumulating parameters. *Higher-Order Symb. Comput.* **17**(1–2), 129–163.

Voigtländer, J. & Kühnemann, A. (2004) Composition of functions with accumulating parameters. *J. Funct. Program.* **14**(3), 317–363.

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.* **73**(2), 231–248.

Wand, M. (1980) Continuation-based program transformation strategies. *J. ACM*. **27**(1), 164–180.

https://doi.org/10.1017/S0956796812000111 Published online by Cambridge University Press