Shifting Expression Procedures into Reverse[*]

Mark Tullsen Paul Hudak

Department of Computer Science
Yale University
New Haven CT 06520-8285
mark.tullsen@yale.edu, paul.hudak@yale.edu

Abstract

The best known approach to program transformation is the unfold/fold methodology of Burstall and Darlington: a simple, intuitive, and expressive approach which serves as the basis of many automatic program transformation algorithms (such as partial evaluation and deforestation). Unfortunately unfold/fold does not preserve total correctness and requires maintaining a transformation history of the program. Scherlis invented a similar approach, expression procedures, which solved these two problems: expression procedures preserve total correctness and require no transformation history.

Motivated by our desire to make expression procedures more expressive by eliminating their one-directional nature (they are designed to specialize but not to generalize functions), we have developed an equational specification of expression procedures, in which the essence of expression procedures is expressed in a single transformation rule. Our approach has the following advantages over expression procedures: (1) all program derivations are reversible; (2) transformations can be done which expression procedures cannot do; (3) fewer and simpler rules are used; and (4) the proof of correctness is simpler.

1 Introduction

1.1 Motivation

Program transformation has been an elusive goal of the programming language research community. We talk about it, write about it, preach about it, but in practice don't use it very much. In the functional programming community the situation is especially unfortunate, where we have the (presumably) simplest framework within which to do program transformation, and at the same time have the greatest need for it: functional programs typically run slower than imperative programs with similar functionality. So what is the problem? Why hasn't program transformation been used more in practice? We see several reasons:

[*]This research was supported in part by NSF under Grant Number CCR-9633390.

1. Program transformation is too tedious. Many steps are often needed to perform even relatively simple transformations.

2. Few good software tools exist for carrying out program transformation. Most of what is done in practice is done on paper, without formal verification.

3. There are foundational problems with some approaches to program transformation. For example, the well-known unfold/fold approach is not *safe*: terminating programs can be transformed into diverging ones.

In an attempt to bridge the gap between the theory and practice of program transformation, we are developing *PATH* (Programmer Assistant for Transforming Haskell), a program transformation system for the language Haskell [HJW92]. PATH, a programmer *assistant*, does not attempt to be fully automatic although it attempts to automate and mechanize as much as possible. For instance, PATH uses a GUI to navigate the program and apply transformations to it. PATH gives the user as much control as is safe, including letting the user write his own meta-programs (such as a deforestation algorithm). Because we give such control to the user, total correctness is considered essential.

1.2 Approaches to Program Transformation

There are a number of approaches which we could take to transforming a functional language such as Haskell. In their survey paper [PS83], Partsch and Steinbrueggen classify various approaches to program transformation into two basic approaches: (1) the schematic, or catalog, approach which is based on using a large catalog of rules, each performing a significant transformation and (2) the generative set approach, which is based on a small set of simple rules which in combination are very expressive. The Bird-Meertens Formalism (or Squiggol) [BdM97, Mee86, MFP91] is an example of the former. Unfold/fold [BD77] and expression procedures [Sch80] are examples of the latter.

Each approach has its advantages: The schematic approach can be more concise and allow for the development of powerful rules which do major transformations in a single step; the rules are symmetric, generally given in the form of transformation templates such as "$P_1 = P_2$ if C". However, a catalog may limit the possible transformations, especially in the presence of arbitrary recursive programs. A generative set approach, such as unfold/fold, is considered to be more general and works well with arbitrary recursive programs. This approach can be easier to use as there is no need to search a catalog. A disadvantage is that derivations are more verbose: many of the same steps (unfold, simplify, fold) are taken again and again.

A third approach would be to use a theorem prover into which the formal semantics (usually denotational) of the language was embedded, e.g., [Pau87]. Although this is the most general approach, it is also the most complicated: one must understand domain theory, inductive proof techniques, the logic of the theorem prover, etc.

1.3 Our Approach

Section 2 of this paper follows the evolution of the approach taken with PATH: We started off using the generative set approach because it is more general than the schematic approach and it is much simpler than theorem proving. Unwilling to use a method such as unfold/fold which does not preserve total correctness (section 2.1), we used Scherlis's expression procedures, a totally correct method (section 2.2). But while attempting to increase the power of expression procedures by eliminating their one-directional nature (section 2.3) (they were designed to specialize but not to generalize functions), we realized that the generality of expression procedures can be achieved by using a schematic rule, namely Fix-Point Fusion (section 2.4). As a result we can use the schematic approach with a very small base catalog to get a transformation system which is more powerful than expression procedures.

We then give the semantics and transformation rules of our system (section 3) and give examples of some program derivations which would be problematic with other approaches (section 4). We then show how we can very elegantly add Scherlis's "qualified expression procedures" (section 5).

2 From Unfold/Fold to Reversible Expression Procedures

In this section we compare various approaches to program transformation and introduce our reversible expression procedures.

2.1 Unfold/Fold

The best known approach to program transformation is the *unfold/fold* methodology of Burstall and Darlington [BD77]. Their approach is based on six rules: (1) *unfold*: the unfolding of function calls by replacing the call with the body of the function where actual parameters are substituted for formal parameters; (2) *laws*: the use of laws about the primitives of the language; (3) *instantiation*: adding an "instance" of a function definition in which a parameter is replaced by a constant or pattern on both sides of the definition; (4) *fold*: the replacement of an expression by a function call when the function's body can be instantiated to the given expression with suitable actual parameters—this can be done with any previous definition of the function; (5) *definition*: the addition of a new function definition; and (6) *abstraction*: the introduction of a `where` clause. This methodology is extremely effective at a broad range of program transformations.

Unfortunately, the unfold/fold methodology does not preserve total correctness; both the instantiation and fold rules are unsafe. Instantiation is easily modified to be safe, but fold is problematic. For example, consider the program

```
f x = x+1
```

Since the expression `x+1` is an instance of the right-hand-side of a function definition, we can replace it with a call to the function, yielding

```
f x = f x
```

which results in a non-terminating definition for `f`. Although this example is overly simple, similar situations can arise in more subtle contexts, and thus non-termination can inadvertently be introduced.

Several approaches have been proposed to solve this problem of partial correctness. One is to suitably constrain the use of fold, for example as proposed by Kott [Kot85]. Another is to provide a separate proof of termination. Yet another is the "tick algebra" of Sands, which guarantees incremental improvement in performance to all transformations [San95b].

Besides the problem with correctness, unfold/fold has a significant inconvenience in practice: a history must be kept of all versions of the program as it is being transformed (or the user must specify which versions to keep or throw away). This history is essential because previous definitions of functions are used to give folding its power.

2.2 Expression Procedures

Motivated by the problems with unfold/fold, Scherlis proposed *expression procedures* [Sch80, Sch81]. (More recently Sands [San95a] extended this work to a higher-order non-strict language.) Scherlis's key innovation was a new procedure definition mechanism in which the left hand side of an expression procedure definition can be an arbitrary expression: thus the name "expression procedure". In addition to laws about primitive functions, three rules are used to transform programs: *abstraction*, which introduces new procedures; *composition*, which introduces new expression procedures; and *application* (or unfold), which replaces a procedure call or expression procedure call with its definition.

An example of a program derivation using expression procedures (adapted from [San95a]) follows. Suppose we have these two function definitions

```
filter p []     = []
filter p (x:xs) = if p x then x : filter p xs
                         else     filter p xs

iterate f x = x : iterate f (f x)
```

and we wish to specialize the expression

```
filter p (iterate f x)
```

Our goal is to create a new version of `iterate` which is specialized to the context "`filter p _`". The first step is to introduce an expression procedure for this context, by filling in the hole ("`_`") with each side of the definition of `iterate` using the *composition* rule:

```
filter p (iterate f x) =ep=
  filter p (x : iterate f (f x))
```

We now have an expression procedure: the left hand side is not just a function symbol applied to variables and patterns, it is an arbitrary expression. (We use = for a regular function definition and =ep= for an expression procedure definition.) Now we transform the definition of this expression procedure to obtain

```
filter p (iterate f x) =ep=
  if p x then x : filter p (iterate f (f x))
         else     filter p (iterate f (f x))
```

Next, we notice that the context appears recursively, so by introducing a new function definition using the *abstraction* rule we can transform it to

```
filter p (iterate f x) =ep= filit p f x
filit p f x =
  if p x then x : filter p (iterate f (f x))
         else     filter p (iterate f (f x))
```

Finally, we use the *application* rule to "apply" the expression procedure, obtaining

```
filter p (iterate f x) =ep= filit p f x
filit p f x =
  if p x then x : filit p f (f x)
         else     filit p f (f x)
```

So, the original expression is equal to `filit p f x`, a specialized version of `iterate`.

On the one hand, expression procedures are strictly less powerful than unfold/fold (they can be simulated by unfold/fold); however, in practice, the great majority of unfold/fold transformations can be done just as well by expression procedures. We are not aware of any *useful*[1] and *total correctness preserving* unfold/fold transformations which cannot be done by expression procedures either directly or indirectly (by finding a "common ancestor" from which to derive the two programs we wish to show equivalent).

On the other hand, expression procedures have two key advantages over unfold/fold: (1) each of the transformation rules preserves total correctness, and (2) no history needs to be maintained, as all needed information is embedded in expression procedures; and when compared to various methods of ensuring total correctness in unfold/fold, expression procedures are both easier to use and more expressive:

- Unfold/fold followed by a proof of termination: Expression procedures are simpler as they need no separate proof of termination. They are more expressive as they can transform programs which may not terminate on all inputs (i.e., for which no proof of termination exists) [Sch80].
- Unfold/fold augmented with the methods of Kott or Sands: Kott adds a number of constraints to the form of program derivations and to the laws and functions usable (so much so as to make his method unusable in practice [Fir90, San96]), and Sands requires extra machinery with his "tick" calculus[2]. Both methods are essentially using the whole transformation history to ensure total correctness. Expression procedures, in contrast, are correct at each step irrespective of any history.

[1]An example of a non-useful transformation is as follows [BD77, Zhu94]: `f x = 0` can be transformed to

```
f x = if x == 0 then 0 else f (x-1)
```

in unfold/fold, but the reverse transformation cannot be done. Expression procedures cannot transform in either direction.

Besides the technical improvements, in practice expression procedures have a simpler and more intuitive method of program derivation: with unfold/fold, the ability to add a new "eureka" definition to a program is essential; but with expression procedures, the analogous operation is selecting a recursive function and some context in which to specialize it. Thus, entering eureka definitions by hand is replaced by selecting contexts in the program. This fits well with our vision of using a GUI to apply transformations to the program.

2.3 The Reversibility Problem

Although expression procedures are an improvement over unfold/fold, they have one significant shortcoming: it is easy to specialize a function, but it is not always even possible to generalize a function! This problem, shared with unfold/fold, comes about because the transformation rules are not reversible. For instance, given this definition of `reverse'`

```
reverse' ([],ys) = ys
reverse' (x:xs,ys) = reverse'(xs,x:ys)
```

it is easy to go from

```
reverse(xs) = reverse'(xs,[])
```

to

```
reverse([])   = []
reverse(x:xs) = reverse'(xs, x:[])
```

using unfold/fold (or expression procedures), but, as noted by Burstall and Darlington, it is not possible to derive the first program from the second. Let $P_1 \Rightarrow^{ep} P_2$ signify that we can derive the program P_2 from P_1 using some sequence of expression procedure rules. Note that $\Rightarrow^{ep}$ is not symmetric, nor is $\Rightarrow^{uf}$, the comparable derives relation for unfold/fold. Even when both $P_1 \Rightarrow^{ep} P_2$ and $P_2 \Rightarrow^{ep} P_1$, the derivation associated with $P_1 \Rightarrow^{ep} P_2$ may give no insight into how to find a derivation for $P_2 \Rightarrow^{ep} P_1$.

Is reversibility that important? We believe so, for two reasons: First, adding reversibility makes the system more expressive: as in the `reverse` example, we often want to make programs shorter or more modular; and even if we want a more efficient program, we sometimes need to make it less efficient before making it more efficient (such transformations are impossible with a method—such as expression procedures—in which every transformation step preserves or increases some measure of efficiency). Secondly, reversibility is important because the system becomes simpler if each rule is reversible: the user can learn one law and use it in two directions.

Note that to get reversibility, we could simply add a rule such as this: "if $P_2 \Rightarrow P_1$ then we can transform P_1 to P_2." Such a rule, called *redefinition*, was added to unfold/fold

[2]Sands's tick calculus couldn't prove the correctness of expression procedures: this seems to have been the motivation for his paper on expression procedures [San95a].

by Burstall and Darlington to get around the problem with the `reverse` derivation above. The disadvantage of this approach is that if we have P_1 and want to transform it, we need to know what P_2 is before we start—we can't derive it directly or incrementally from P_1; also, the addition of this ad hoc rule makes the system more complex.

Instead of adding a rule, we would rather modify the rules to make them all reversible; i.e., we want to find a characterization of expression procedures in terms of one or more transformation rules.

2.4 Reversible Expression Procedures

We want to make expression procedures reversible. The compose and apply rules are inherently one-directional, but what if we were to merge into one step all the steps involved in a prototypical expression procedure transformation? There are just four key steps (as seen in the example in section 2.2):

1. the *introduction* of the expression procedure (composition),
2. the *transformation* of the body of the expression procedure,
3. the use of *abstraction* to capture the resulting recursion, and
4. *application* of the expression procedure.

These four steps can be merged as follows. We begin with a strict program context `C[]` and a function definition `f = F[f]`. *Introduction* of the expression procedure (composition) gives

```
C[f] =ep= C[F[f]]
```

which is then *transformed* into the recursive expression procedure

```
C[f] =ep= G[C[f]]
```

for some context `G[]`. After we do *abstraction* we arrive at

```
C[f] =ep= g
g = G[C[f]]
```

Finally, *application* of the expression procedure yields

```
C[f] =ep= g
g = G[g]
```

The above steps can be merged into one rule (expressing the values of `f` and `g` as the fix-points μ`f.F[f]` and μ`g.G[g]`):

$$\begin{gathered}\texttt{C[}\mu\texttt{f.F[f]]} \Rightarrow^{ep} \mu\texttt{g.G[g]} \\ \text{if} \\ \texttt{C[F[f]]} \Rightarrow^{ep} \texttt{G[C[f]]},\quad \texttt{C[]}\ \text{strict}\end{gathered}$$

This one rule replaces the three expression procedure rules—composition, abstraction, and application. We don't have reversibility yet, but if we could replace $\Rightarrow^{ep}$ with $=$ in the above rule we would have this reversible rule,

$$\begin{gathered}\texttt{C[}\mu\texttt{f.F[f]]} = \mu\texttt{g.G[g]} \\ \text{if} \\ \texttt{C[F[f]]} = \texttt{G[C[f]]},\quad \texttt{C[]}\ \text{strict}\end{gathered}$$

a theorem of Stoy [Sto77]. So, we join the company of many who have rediscovered or used this theorem [AK82, GS90, MFP91]. Our contribution is showing its connection to expression procedures. Interestingly, it is a free theorem [Wad89] of the fix-point operator μ. We take its name, Fix-Point Fusion, from Meijer et al. [MFP91] where the power of the theorem is exploited considerably: most of their transformations are instances of this one general theorem.

Fix-Point Fusion can be used in both directions:

$$\begin{array}{ll}\texttt{C[}\mu\texttt{f.F[f]]} \Rightarrow \mu\texttt{g.G[g]} & \text{specialization (fusion)} \\ \mu\texttt{g.G[g]} \Rightarrow \texttt{C[}\mu\texttt{f.F[f]]} & \text{generalization (fission)}\end{array}$$

To do fusion, `C[]` and `F[]` are known, and `G[]` is desired; so the premise is proven by finding a derivation `C[F[f]]` $\Rightarrow$ `G[C[f]]`. To do fission, `G[]` is known, the user provides `C[]`, and `F[]` is desired; so the premise is proven by finding a derivation `G[C[f]]` $\Rightarrow$ `C[F[f]]`. (Had an extra "redefinition" rule been added to expression procedures, the user would also need to know the answer, `F[]`, before proceeding.)

So, we have accomplished our goal: we can do expression procedure transformations with one reversible rule, Fix-Point Fusion. The advantages to using Fix-Point Fusion as a replacement for expression procedures are as follows:

- We now have a symmetric derives relation and need no ad hoc rules to get reversibility. (And henceforth we use $=$ rather than $\Rightarrow$.) Thus the system is simpler than expression procedures would be with an extra rule for reversibility: there is *one* rule which the user uses to both specialize and generalize, rather than an extra rule added to a set of "one directional" rules.
- We do not need to extend our base language with expression procedures. Although we would not need to actually implement expression procedures to use them (they are removed in the final program) we would need to add expression procedures to the operational semantics of the language, which complicates reasoning about the transformation system. With our approach there is just one theorem to prove.
- Derivations are now structured in a goal-directed fashion. Program derivations are structured as 1) *goal*: a function and its context are specified; and 2) *sub-goal*: the derivation is developed which satisfies the sub-goal (thereby synthesizing the new definition). Besides clearly indicating the goal of each transformation, this allows all the sub-goals of an unreachable goal to be removed easily when the goal is removed. With unfold/fold and expression procedures the derivations *can* be, although seldom are, much more unstructured.

By showing how to do expression procedure transformations with a single transformation rule we have integrated two different approaches to doing program transformation: the schematic approach and the generative set approach (as described in the introduction). Chin and Darlington [CD89], seeing the need to integrate these two approaches, added schematic rules to unfold/fold along with a method to generate new schematic rules using unfold/fold. In contrast, our method of integration is to use the schematic approach in which we include a law which gives us the expressiveness of one generative set approach.

With Fix-Point Fusion we believe we have a solid basis for a totally correct, simple, and expressive transformation system. We will next describe the language and transformation rules of our system in detail.

3 The Language and Transformation Rules

3.1 The Language

Our language is a typed, non-strict functional language with products, sums, integers, and an explicit fix-point operator (i.e., un-sugared Haskell)[3]. The syntax of programs is as follows:

```
e = λv.e | v | e e
  | (e,e) | fst e | snd e
  | L e | R e | case e of L v -> e; R v -> e
  | n | p
  | μv.e

n ∈ integer constants
v ∈ variables
p ∈ strict primitive functions over integers
```

We consider only well-typed programs. The operational semantics is given by applying the reduction rules (in Figure 1) from left to right with a leftmost outermost reduction strategy.

The following syntactic sugar is used here:

```
                []  ≡ L 0
           e1 : e2  ≡ R (e1,e2)
       [e1,...,en]  ≡ e1:(... en:[])
let v = e1 in e2  ≡ (λv.e2) e1
         λ(x,y).e  ≡ λv.e{fst v/x}{snd v/y}
         μ(x,y).e  ≡ μv.e{fst v/x}{snd v/y}
if e1 then e2 else e3  ≡
  case e1 of L v -> e2; R v -> e3
```

We use $e\{e_1/v\}$ to represent the term e where free occurrences of v are replaced by e_1; The set of free variables in expression `e` is denoted by `fv(e)`. A *context* (ranged over by `C`,`D`,`E`,`F`,`G`) is an expression with zero or more holes, `[]`. `C[e]` is the expression with `C[]`'s holes replaced by `e`. A context `C[]` *traps* a variable `v` if at any hole in `C[]` the variable `v` is in scope. A context `C[]` is strict if for all closed expressions, `e`, and substitutions, σ, `e` diverges implies that σ`(C[e])` diverges. $e_1 \to e_2$ signifies that e_1 reduces to e_2 in one step and $e_1 \to^+ e_2$ signifies that e_1 reduces to e_2 in one or more steps. More formally, $\to$ is the least relation satisfying the reduction rules (Figure 1) and that is closed under $e_1 \to e_2 \Rightarrow C[e_1] \to C[e_2]$. We also define `*` as follows (note that `f*g` is strict if `f` and `g` are strict):

```
(f*g)(x,y) = (f x, g y)
```

3.2 The Transformation Rules

The transformation rules include the reduction rules as given in Figure 1 plus the additional rules given in Figure 2. (The notion of equivalence in these rules, =, is observational equivalence at base types, here just integers.)

[3]Previously our examples used recursion equations, but now we use an explicit fix-point operator in order to make our laws clearer.

Instantiation. The Instantiation rule lets us move strict contexts into case expressions (and in the reverse direction, out). It corresponds to the instantiation rule in unfold/fold (which only goes in one direction). The context `C[]` must be strict: as Sands [San95a] has noted, this corresponds to the conditions described by Runciman et al. [RFJ90] for safe instantiation in a non-strict language.

In actual use the strictness condition needed by the Instantiation and Fix-Point Fusion rules is often detected syntactically: the contexts defined by `S`, an extension of reduction contexts, are strict (where `e` is as defined above):

```
S = [] | S e | p S S | p e S | p S e
  | fst S | snd S
  | case S of L x-> e; R x-> e
```

It is interesting to note that Instantiation happens to be another free theorem, derivable from the type of case.

Eta. We have eta rules for functions, sums, *and* products. The language differs from un-sugared Haskell by having an unlifted product, not a lifted product; however, a lifted product can be had by simply wrapping a constructor around the unlifted product.

Fix-Point Fusion. This rule gives us the ability to transform recursive definitions. As noted previously, the Fix-Point Fusion rule lets us do transformations possible with expression procedures. It requires that the context `C[]` be strict. This strictness condition is necessary, otherwise non-termination could be introduced.

Fix-Point Expansion. This rule enables us to expand, or inline, the definition of a recursive definition inside itself.

4 Examples of Program Derivations

In this section we give some examples of program derivations. Although each of these examples is by necessity short, each demonstrates a transformation which is problematic using other approaches.

4.1 Infinite Lists

Here we derive a program which is an infinite list. Expression procedures are capable of doing this, but unfold/fold with a proof of termination would not. This is because the standard technique of using a well-founded ordering to prove termination of *functions* [Fir90, MW79] is inapplicable for recursively defined *data structures.* We assume that `map` and `succ` are predefined.

```
map succ (μones.1:ones)
________________{Fusion} (= μtwos.?) _______________
∀ones.
  map succ (1:ones)
=                                            {reduce}
  succ 1 : map succ ones
____________________________________________________
=
μtwos. (succ 1) : twos
=                                            {reduce}
μtwos. 2 : twos
```

$$
\begin{array}{rcll}
\texttt{p}\ n_1 \ldots n_n & \to & \texttt{n} & \text{where } \texttt{n} = [\![\texttt{p}]\!]\ n_1 \ldots n_n \\
\texttt{fst (e1,e2)} & \to & \texttt{e1} & \\
\texttt{snd (e1,e2)} & \to & \texttt{e2} & \\
\texttt{case L e1 of L v -> e2; R v -> e3} & \to & \texttt{e2}\{\texttt{e1/v}\} & \\
\texttt{case R e1 of L v -> e2; R v -> e3} & \to & \texttt{e3}\{\texttt{e1/v}\} & \\
(\lambda\texttt{v.e1) e2} & \to & \texttt{e1}\{\texttt{e2/v}\} & \\
\mu\texttt{v.e1} & \to & \texttt{e1}\{\mu\texttt{v.e1/v}\} &
\end{array}
$$

Figure 1: Reduction Rules

Instantiation:

$$
\begin{array}{c}
\texttt{C[case e of L x -> e1; R x -> e2]} \quad = \quad \texttt{case e of L x -> C[e1]; R x -> C[e2]} \\
\text{if } \texttt{C[]} \text{ strict, } \texttt{fv(e)} \text{ not trapped by } \texttt{C[]}, \text{ and } \texttt{x} \notin \texttt{fv(C[])}
\end{array}
$$

Eta:

$$
\begin{array}{rcll}
\texttt{e} & = & \lambda\texttt{v.e v} & \text{if } \texttt{e} :: \alpha \to \beta \text{ and } \texttt{v} \notin \texttt{fv(e)} \\
\texttt{e} & = & \texttt{(fst e, snd e)} & \text{if } \texttt{e} :: (\alpha, \beta) \\
\texttt{e} & = & \texttt{case e of L x -> L x; R x -> R x} & \text{if } \texttt{e} :: \alpha \mid \beta
\end{array}
$$

Fix-Point Fusion:

$$
\frac{\forall \texttt{f.C[F[f]]} = \texttt{G[C[f]]}}{\texttt{C[}\mu\texttt{f.F[f]]} = \mu\texttt{g.G[g]}} \quad \text{if} \quad \texttt{C[]} \text{ strict, } \texttt{f}, \texttt{g} \text{ neither free in nor trapped by } \texttt{F[]} \text{ or } \texttt{G[]}
$$

Fix-Point Expansion:

$$
\texttt{e} \quad = \quad \mu\text{f.F[f]} \qquad \text{if } \texttt{e} \to^{+} \texttt{F[e]}
$$

Figure 2: Additional Transformation Rules

A word of explanation is needed about the format of our derivation given here: The horizontal line labeled $\{Fusion\}$ marks where we start to derive the premise of Fix-Point Fusion. This sub-derivation is indented and when the premise is shown we know that the program above the sub-derivation is equal to the program below it. The `(=` μ`twos.?)` corresponds to what the user might enter to give a name to the variable bound by the new μ.

4.2 Fission: The Generalization of Recursive Definitions

The reverse of fusion—bringing a context and a function together—is fission, which splits a function into a context and a function. This cannot in general be done with expression procedures and was the original motivation for our work. Suppose, for example, that we wish to generalize `mapsucc` to the standard `map` function. (To make the derivation smaller we use the `map` which is defined only on infinite lists—a derivation for the standard `map` is the same number of steps but a larger program would be displayed at each step.)

```
 μmapsucc.λxs.succ (head xs) : mapsucc (tail xs)
 ─────────────{Fission} (= (μmap.?) succ) ─────────────
 ∀map.
   λxs.succ (head xs) : map succ (tail xs)
 =                                      {abstract succ}
   (λh.λxs.h (head xs) : map h (tail xs)) succ
 ──────────────────────────────────────────────────────
=
 (μmap.λh.λxs.h (head xs) : map h (tail xs))
   succ
```

If we read the derivation from bottom to top, we simply have the specialization of `map succ`. Here, when we do Fix-Point Fusion—or in this case, Fix-Point Fission—we use a similar notation but the

```
(= (μmap.?) succ)
```

(which corresponds to what the user would enter) gives the form of the desired program.

4.3 Tupling

This example does tupling, or loop fusion. It takes a two-pass average program and derives a one pass algorithm. This transformation is normally beyond the scope of fully automated strategies such as deforestation or partial evaluation (which is not surprising for a user-guided transformation system).

```
let sum = μsum.λxs.case xs of
                    []     -> 0;
                    x:xs' -> x + sum xs'
    len = μlen.λxs.case xs of
                    []     -> 0;
                    x:xs' -> 1 + len xs'

 λys.sum ys / len ys
=                                          {abstract}
 λys.let (s,l) = (sum ys, len ys) in s/l
=                                          {abstract}
 λys.let (s,l) = (λxs.(sum xs, len xs)) ys
     in s/l
```

Now we want to specialize

```
λxs.(sum xs, len xs)
```

but neither `sum` nor `len` is in a strict context, so expression procedures cannot specialize this program nor can Fix-Point Fusion be directly applied here. One way to get around this is to add a construct to the language by which we can express a strict product; but even for cases in which the product is not strict, we can continue with the help of the derived rules *FPF2* and *ABIDES* (derived rules whose definitions and derivations are in the Appendix).

```
  λxs.(sum xs, len xs)
  ________________{FPF2} (= μsumlen.?) ______________
  ∀sum,len.
    λxs.(case xs of []->0; x:xs'->x+sum xs',
         case xs of []->0; x:xs'->1+len xs')
  =                                         {ABIDES}
    λxs.case xs of
          []    ->(0,0);
          x:xs'->(x+ sum xs',1+ len xs')
  =                                    {abstraction}
    λxs.case xs of
          []    ->(0,0);
          x:xs'->let (s,l) = (sum xs',len xs')
                  in  (x+s,1+l)
  =                                    {abstraction}
    λxs.case xs of
          []    ->(0,0);
          x:xs'->
            let (s,l) = (λxs.(sum xs,len xs)) xs'
            in  (x+s,1+l)
  ___________________________________________________
=
  μsumlen.λxs.case xs of
                []    -> (0,0);
                x:xs' -> let (s,l) = sumlen xs'
                         in  (x+s,1+l)
```

The rule *FPF2* is a variation on Fix-Point Fusion which is applicable when we have two recursive definitions. Interestingly, it is also a free theorem of μ: we get Fix Point Fusion (*FPF*) when we use a binary relation, we get *FPF2* when we use a ternary relation.

4.4 Mutually Recursive Expression Procedures

It is not obvious that Fix-Point Fusion along with the other laws can do all transformations that are possible with expression procedures: Can we do all transformations done by combining expression procedure rules in ways other than the prototypical order: composition, laws, abstraction, then application? In the future we would like to prove this; but for now, here is an expression procedure derivation that seems difficult to do with Fix-Point Fusion because the uses of composition, abstraction, and application are completely intertwined. Given this

```
f = F[f,g]
g = G[f,g]
```

and assuming

```
∀f,g. C[F[f,g]] => A[C[f],D[g]]
∀f,g. D[G[f,g]] => B[C[f],D[g]],
```

with expression procedures we can do this:

```
  C[f] =ep= C[F[f,g]]
  D[g] =ep= D[G[f,g]]
 ⇒                                      {assumption}
  C[f] =ep= A[C[f],D[g]]
  D[g] =ep= B[C[f],D[g]]
 ⇒                                   {abstract twice}
  C[f]  =ep= f'
  f' = A[C[f],D[g]]
  D[g] =ep= g'
  g' = B[C[f],D[g]]
 ⇒                {apply C[f] twice; apply D[g] twice}
  C[f]  =ep= f'
  f' = A[f',g']
  D[g] =ep= g'
  g' = B[f',g']
```

However, this is just as easy to do with Fix-Point Fusion if we express the mutual recursion explicitly. Given this

```
(f,g) = μ(f,g).(F[f,g],G[f,g])
```

we can specialize both definitions at the same time as follows:

```
  (C[f],D[g])
 =                                          {def. of *}
  (C*D)(f,g)
  ____________{Fusion} (= μ(f',g').?) ______________
  ∀f,g.
    (C*D)(F[f,g],G[f,g])
    =                                       {def. of *}
    (C[F[f,g]],D[G[f,g]])
    =                                     {assumption}
    (A[C[f],D[g]], B[C[f],D[g]])
    =                                       {def. of *}
    (A*B)(C[f],D[g])
    =                                       {def. of *}
    (A*B)((C*D)(f,g))
  __________________________________________________
 =
  μ(f',g').(A*B)(f',g')
 =                                   {syntactic sugar}
  (f',g') where f' = A[f',g'] and g' = B[f',g']
```

5 Qualified Expression Procedures

In his thesis [Sch80] Scherlis noted that expression procedures allow us to specialize recursive functions in a syntactic context, but do not allow us to specialize functions based on non-syntactic information. For instance, expression procedures can specialize `f` in the syntactic context "`f x 0`" but couldn't take advantage of "`x > y`" in the specialization of "`f x y`".

To take advantage of the non-syntactic information available, Scherlis extended his system to support "qualified expression procedures"[4]. A qualified expression procedure looks like this

```
{p} e1 =ep= e2
```

in which `p` is a boolean valued expression similar to a precondition. In the transformation of the definition, we can assume `p` is true; the qualified expression procedure may only be applied where the qualifier is true.

Thanks to the non-strict semantics of our language and our schematic approach to expression procedures, we get the power of qualified expression procedures without having to add any ad hoc constructs to the language. Let's say we have an `assert` function defined as

```
assert p e = if p then e else error
```

[4]With these we get the power of Generalized Partial Computation [FN88, Tak91]. The importance of this extra information for specialization of programs is discussed in Sørenson et al. [SGJ94].

(where `error` is equivalent to $\perp$) for which we add some syntactic sugar:

```
{p} e = assert p e
{p _} e = {p e} e
```

With `assert` and some simple laws about it, we get the power of qualified expression procedures. Some easily proven laws regarding assertions are as follows:

Introducing/Eliminating Assertions

```
if p then a else b = if p then {p} a else b
if p then a else b = if p then a else {not p} b
```

Manipulating Assertions

```
{if p then p2 else p3} if p then a else b
                      =
        if p then {p2} a else {p3} b

       {p} = {p} {q}       if p => q
       {p} = {q}           if p = q
  {p} C[e] = C[{p} e]      if C[] strict
```

Using Assertions

```
               {e1=e2} D[e1] = {e1=e2} D[e2]
    {p} D[if p then a else b] = {p} D[a]
{not p} D[if p then a else b] = {not p} D[b]
```

Assertions can be used either as a run-time construct or as a construct which is introduced and removed during transformation. Nothing needs to be added to the language with our approach, nor do our transformation rules require any change, we just add the assertion laws. Assertions and Fix-Point Fusion allow us to bring pre-conditions into a recursion and to bring post-conditions out of a recursion as seen in following two laws, which follow directly from Fix-Point Fusion:

Where `f` = μ`f.`λ`x.F[f]` and `p` is strict.

Pre-condition:

$$\frac{\lambda\texttt{x.\{p x\} F[f]} = \lambda\texttt{x.\{p x\} F[}\lambda\texttt{x.\{p x\} f x]}}{\lambda\texttt{x.\{p x\} f x} = \mu\texttt{f.}\lambda\texttt{x.\{p x\} F[f]}}$$

Post-condition:

$$\frac{\lambda\texttt{x.F[}\lambda\texttt{x.\{p \_\} (f x)]} = \lambda\texttt{x.\{p \_\} F[f]}}{\texttt{f} = \lambda\texttt{x.\{p \_\} (f x)}}$$

6 Conclusion

In our attempt to increase the expressiveness of expression procedures we have made these contributions:

- We have shown that the essence of expression procedure transformations is Fix-Point Fusion. Thus, the power of expression procedures can be achieved with a schematic program transformation rule.
- As a result, we have integrated two different approaches to doing program transformation: the schematic approach and the generative set approach.
- We can do transformations which neither expression procedures nor safe restrictions of unfold/fold can do. This is a consequence of the reversibility inherent in the schematic approach.
- We have improved on the work of Scherlis and Sands. By doing expression procedures in one step, we gain simplicity over their approaches: (1) we have a simpler proof of correctness since we need only prove one law and need not show that a set of laws preserves consistency and progressiveness; (2) we can dispense with Sands's restriction on where we can perform abstraction in expression procedures (this restriction is motivated by the proof of correctness); and (3) we need not give a semantics to expression procedure definitions.
- We show how, with our approach, we get the expressiveness of Scherlis's qualified expression procedures by simply adding assertion laws.

Future work on PATH will include designing and implementing the user-interface, implementing various meta-programs, building a useful catalog of rules, and applying the system to more realistically sized programs. Other avenues of research related to the work here would be (1) developing a proof that our transformation rules really are as expressive as unrestricted expression procedures, (2) investigating the relative power of our system compared to unfold/fold, and (3) investigating the limits of our approach compared to, for instance, a theorem prover with fix-point induction.

Acknowledgements. We would like to thank David Sands for many helpful comments on an earlier version of this paper.

References

[AK82] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, April 1982.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[CD89] Wei Ngan Chin and John Darlington. Schematic rules within unfold/fold approach to program transformation. In *TENCON '89: Fourth IEEE Region 10 International Conference*, 1989.

[Fir90] M. A. Firth. *A Fold/Unfold Transformation System for a Non-Strict Language*. PhD thesis, University of York, 1990.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjorner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[GS90] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 633–674. North-Holland, Amsterdam, 1990.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[Kot85] L. Kott. Unfold/fold transformations. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 412–433. CUP, 1985.

[Mee86] L. Meertens. Algorithmics - towards programming as a mathematical activity. In J. W. de Bakker, E. M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986. CWI Monographs, volume 1.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Verlag, June 1991. LNCS 523.

[MW79] Zohar Manna and Richard Waldinger. Synthesis: Dreams → programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, July 1979.

[Pau87] L. C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge Univ. Press, 1987.

[PS83] H. Partsch and R. Steinbrueggen. Program transformation systems. *ACM Computing Surveys*, 15:199–236, 1983.

[RFJ90] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language: An approach to instantiation. In K. Davis and R. J. M. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 133–141, London, UK, 1990. Springer-Verlag. British Computer Society Workshops in Computing Series.

[San95a] D. Sands. Higher-order expression procedures. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 1995.

[San95b] D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 221–232. ACM Press, 1995.

[San96] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[Sch80] W.L. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, California, August 1980. Stanford Computer Science Report STAN-CS-80-818.

[Sch81] W.L. Scherlis. Program improvement by internal specialization. In *Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, January 1981*, pages 41–49. ACM, 1981.

[SGJ94] M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 485–500. Springer-Verlag, 1994.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Tak91] A. Takano. Generalized partial computation for a lazy functional language. *ACM SIGPLAN Notices*, 26(9):1–11, 1991. Proceedings of Conference on Partial Evaluation and Semantics-Based Program Manipulation, New Haven, CT.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture.* Springer Verlag, 1989.

[Zhu94] Hong Zhu. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, January 1994.

A Derived Rules

A.1 FPF2

Rule:

$$\frac{\texttt{C[] strict},\quad \forall\texttt{d,e. C[(D[d],E[e])] = G[C[(d,e)]]}}{\texttt{C[(}\mu\texttt{d.D[d],}\mu\texttt{e.E[e])]} = \mu\texttt{g.G[g]}}$$

Derivation:

Assuming

```
C[] strict
∀d,e. C[(D[d],E[e])] = G[C[(d,e)]]
```

We get

```
  C[(μd.D[d], μe.E[e])]
=                                                  {SPLIT}
  C[μ(d,e).(D*E)(d,e)]
 ________________{Fusion} (= μg.?) _______________________
 ∀d,e.
   C[(D*E)(d,e)]
 =                                                 {def. of *}
   C[(D[d],E[e])]
 =                                                 {assumption}
   G[C[(d,e)]]
 _________________________________________________________
=
  μg.G[g]
```

A.2 SPLIT

Rule:

```
μ(d,e).(D*E)(d,e) = (μd.D[d], μe.E[e])
```

Derivation:

```
 let x = μ(d,e).(D*E)(d,e)

  fst x
=                                                {def. of x}
  fst (μ(d,e).(D*E)(d,e))
=                                            {syntactic sugar}
  fst (μz.(D*E) z)
  ________________{Fusion} (= μd.?) _________________
  ∀z.
    fst ((D*E) z)
    =                                        {eta expansion}
    fst ((D*E) (fst z,snd z))
    =                                            {def. of *}
    fst (D[fst z],E[snd z])
    =                                            {reduction}
    D[fst z]
  ___________________________________________________
=
  μd.D[d]
```

And similarly we get

```
  snd x = μe.E[e]
```

Then we get

```
  x
=                                            {eta expansion}
  (fst x, snd x)
=                                           {above two laws}
  (μd.D[d], μe.E[e])
```

A.3 ABIDES

Rule:

```
          (case e of L x->a1; R y->a2,
           case e of L x->b1; R y->b2)
                       =
     case e of L x->(a1,b1); R y->(a2,b2)
```

Derivation:

```
  (case e of L x->a1; R y->a2,
   case e of L x->b1; R y->b2)
=                    {inverse reductions for fst and snd}
  (case e of L x->fst(a1,b1); R y->fst(a2,b2),
   case e of L x->snd(a1,b1); R y->snd(a2,b2))
=                          {reverse instantiation, twice}
  (fst (case e of L x->(a1,b1); R y->(a2,b2)),
   snd (case e of L x->(a1,b1); R y->(a2,b2)))
=                                      {eta contraction}
  case e of L x->(a1,b1); R y->(a2,b2)
```