# Report on an Implementation of a Semi-inverter

Torben Ægidius Mogensen

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
torbenm@diku.dk

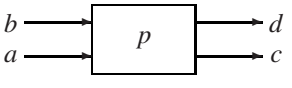
**Abstract.** Semi-inversion is a generalisation of inversion: A semi-inverse of a program takes some of the inputs and outputs of the original program and returns the remaining inputs and outputs.

We report on an implementation of a semi-inversion method. We will show some examples of semi-inversions made by the implementation and discuss limitations and possible extensions.

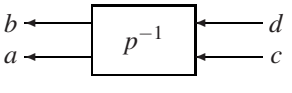
## 1 Introduction

Inversion of programs [6, 3, 8, 13, 11] is a process that transforms a program that implements a function $f$ into a program that implements the inverse function of $f$, $f^{-1}$. A related technique was used in [4] to add backtracking to a program by adding code to rewind earlier computation.

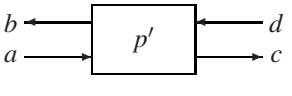
Semi-inversion [12, 15] is a generalisation of inversion that allows more freedom in the relation between the function implemented by the original program and the function implemented by the transformed program. The difference can be illustrated with the following diagrams. Assuming we have a program $p$ with two inputs and two outputs:




we can invert this into a program $p^{-1}$ by "reversing all the arrows":




With semi-inversion, we can choose to retain the orientation of some of the arrows while reversing others, to obtain a program $p'$:



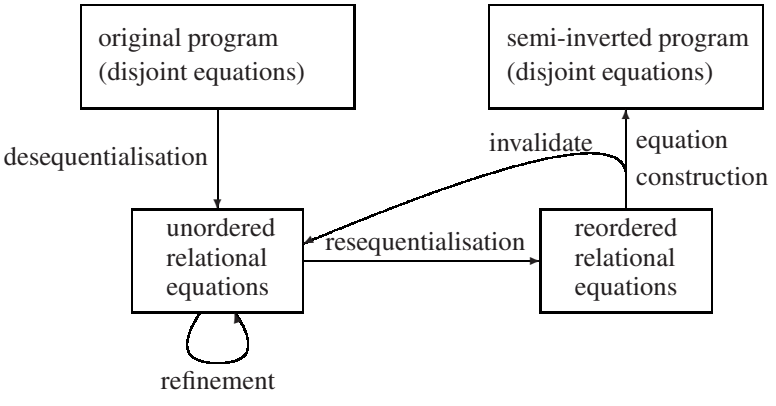

$p'$ takes one of the inputs of $p$ and one of its outputs and returns the remaining input and output. More formally, if running $p$ with inputs $(a,b)$ produces $(c,d)$, then running $p'$ with inputs $(b,c)$ produces $(a,d)$.

I. Virbitskaite and A. Voronkov (Eds.): PSI 2006, LNCS 4378, pp. 322–334, 2007.

In [12], some theory about semi-inversion and a method for it is described, but no implementation existed at the time of writing. This paper describes an implementation of the semi-inversion method described in [12], reports experiences using it on some examples and conclude by discussing limitations and possible extensions.

## 2 The Semi-inversion Transformation

The steps of the semi-inversion transformation is shown in the following diagram:




The following sections will elaborate on each of these steps, further detail can be found in [12].

We will use a simple example to illustrate the process. The program below takes two lists of equal length and returns a pair of two lists, one containing the pairwise sum of the arguments and the other the pairwise difference.

```
pm ([],[]) = ([],[]);
pm (a:as,b:bs) =
  let (ps,ms) = pm (as,bs) in
    ((a+b):ps,(a-b):ms);
```

The syntax of the language we use is similar to Haskell, but there are a few semantic differences that are not important to this example.

### 2.1 Desequentialisation

Desequentialisation makes patterns and expressions into unordered sets of relations, where dependencies are carried through shared variables, much like in Prolog. The guarded equation $f\ P \mid G = E$ is desequentialised as follows:

$$
\begin{array}{l}
f\ (P_1,P_2) = R_1 \cup R_2 \cup R_3 \\
\text{where} \\
\quad (P_1,R_1) = I_p(P) \qquad (I_p : pattern \rightarrow var^+ \times relationset) \\
\quad (P_2,R_2) = I_e(E) \qquad (I_e : expression \rightarrow var^+ \times relationset) \\
\quad R_3 \quad\ = I_g(G) \qquad (I_g : guard \rightarrow relationset)
\end{array}
$$

The functions $I_p$, $I_e$ and $I_g$ are detailed in [12]. Our example is desequentialised as

```
pm (w,x,y,z) where {w=[], x=[], y=[], z=[]}
pm (w,x,y,z) where {w=:(a,as), x=:(b,bs), pm(as,bs,ps,ms),
                    u=+(a,b), v=-(a,b), y=:(u,ps), z=:(v,ms)}
```

Note that arguments and results of functions are not distinguished. We will collectively call inputs and outputs "in-outs".

### 2.2 Refinement

Refinement rewrites the relation set to make some relations more explicit:

- Related operators on the same arguments are combined to a single operator: $z = +(x,y),\ v = -(x,y)$ is replaced by $(z,v) = +\text{-}(x,y)$
- Equalities of tuples are split into equalities of the elements: $(x,y) = (z,v)$ is replaced by $x = z,\ y = v$

We can apply the first of the above rules to our program to get:

```
pm (w,x,y,z) where {w=[], x=[], y=[], z=[]}
pm (w,x,y,z) where {w=:(a,as), x=:(b,bs), pm(as,bs,ps,ms),
                    (u,v)=+-(a,b), y=:(u,ps), z=:(v,ms)}
```

### 2.3 Resequentialisation

Up to now, the steps do not depend on which in-outs will be arguments and results of the semi-inverted program. The next steps do, so we need to specify this. We do this by a *division* that by 1s and 0s indicate which in-outs are, respectively, inputs and outputs of the semi-inverted function. The division can also specify a name for the semi-inverted function. We will use the division `pm(0,1,0,1)=mp` to indicate that the second and last in-outs will be inputs while the first and third are outputs. The semi-inverse of `pm` will be called `mp`.

Each primitive operator has a list of possible semi-inverses, for example

$$\begin{array}{l} z = +(x,y) \qquad\quad \Rightarrow x = \text{-}(z,y),\ y = \text{-}(z,x) \\ (p,q) = +\text{-}(x,y) \Rightarrow (p,x) = +\text{-}24(q,y), \ldots, (x,y) = +\text{-}12(p,q) \end{array}$$

where $+\text{-}24(q,y)$ evaluates to $(q+y, q+2y)$ and $+\text{-}12(p,q)$ evaluates to $((p+q)/2, (p-q)/2)$. For `+-`, any two in-outs are sufficient to calculate the remaining, so there are 11 cases for `+-` (all cases where at least two in-outs are known).

Resequentialisation uses dependency analysis to list relations in possible evaluation order. Initially, the input variables are known, but as new operations are added to the sequence, more variables become known.

Given the division `pm(0,1,0,1)=mp`, the example program resequentialises to

```
pm (w,x,y,z) where {x=[], z=[], w=[], y=[]}
pm (w,x,y,z) where {x=:(b,bs), z=:(v,ms), (u,a)=+-24(b,v),
                    pm(as,bs,ps,ms), y=:(u,ps), w=:(a,as)}
```

For user-defined functions, it is initially assumed that any subset of input/output to the function can define the remaining input/output, but these assumptions may later be invalidated.

Resequentialisation may fail to succeed. If this happens for the top-level function call (that we wish to semi-invert), the whole semi-inversion process fails, but if it happens for an auxiliary function call, it may be caused by incorrectly assuming that the the input/output subset used for resequentialisation is sufficient. Hence, we mark this subset as invalid. This may require resequentialisation for other functions to be redone.

### 2.4 Constructing Equations

We now transform back from the relational form to equational form, while obeying the evaluation order found during resequentialisation. We do this in three steps:

1. Construct patterns from structural relations for new inputs.
2. Make guards from non-structural relations for new inputs.
3. Make expression from remaining relations.

The semi-inverse of the example program gives the following reconstructed equations:

```
mp([],[]) = ([],[]);
mp(b:bs,v:ms) =
  let (u,a) = +-24(b,v) in
    let (as,ps) = mp(bs,vs) in
      let y = u:ps in
        let w = a:as in (w,y);
```

After transforming each equation as described above, equations for the same function must by their patterns and guards divide the input into disjoint classes, i.e., there can be no overlap. If this is not the case, we mark the input/output subset for the function as invalid and backtrack.

In our example, the equations are clearly disjoint, so the semi-inverse is valid. The semi-inverter additionally rewrites special operators like `+-24` into "normal" operations and unfolds trivial or linear let-definitions. The actual output from the semi-inverter (after addition of a few newlines) is:

```
mp ([],[]) = ([],[]);
mp (b : bs,e_10_ : ms) =
  let (as,ps) = (mp (bs,ms)) in
    let a = (e_10_+b) in (a : as,(a+b) : ps);
```

## 3 Design Details

[12] leaves some implementation details unspecified. We have chosen the following:

**Refinement.** Addition and subtraction are combined, as shown above. Additionally, the constraints `p = /(x,y), q = %(x,y)`[1] are combined to `(p,q) = /%(x,y)`,

[1] Where `%` is the remainder operator.

which when `p, q` and `y` are known can be semi-inverted to `x = p*y+q`. Equalities between tuples are split, and if a variable is equated with a tuple, other occurrences of the variable are replaced by the tuple.

**Resequentialisation.** When there are several possible relations that can be selected during resequentialisation, the following priorities are used:

1. Tests with all parameters known.
2. Primitive operators with sufficient instantiation for semi-inversion.
3. Calls to already desired semi-inverses of user-defined functions, including the one that is currently being resequentialised.
4. Other calls to user-defined functions.

**Backtracking.** If the semi-inverter fails to semi-invert a desired semi-inverse, it prints a message, marks the semi-inverse invalid and starts over. Sometimes, it may find other semi-inverses that can be used instead, otherwise the message may help the user rewrite the program to get better results.

Starting over is clearly not the most efficient way of doing back-tracking, but in our experience the semi-inverter rarely back-tracks very much, if at all.

**Determining disjointedness of equations.** First, the variables are renamed to make the two equations use disjoint variables. Next, the patterns are unified. If unification fails, the equations are disjoint, otherwise the unifying substitution is applied to the guards. If the conjunction of the guards can never be true, the equations are disjoint. To test the guard, intervals of the possible values of integer variables are maintained, and if an interval becomes empty, the guard can't become true. A few additional cases of unsatisfiable constraints such as $e/=e$ are also considered.

There is room for improvement, as conjunctions like `x<y && y<x` are not recognised as unsatisfiable. The general problem of determining non-overlap of guards is undecidable, so there will always be cases that aren't handled.

Resequentialisation of a single equation is quadratic in the worst case, but back-tracking can make semi-inversion take exponential time, as a large fraction of the exponentially many possible divisions of a function can be tried. In the (admittedly small) examples we have tried, little backtracking occurs, so we don't believe this to be a problem in practice.

## 4 A More Ambitious Example: Multiplication of Binary Numbers

The semi-inverter can semi-invert the primitive multiplication operator into a division operator when one argument and the result are known.

But can we repeat this if we instead represent numbers as lists of bits?

The binary multiplication algorithm can be described by the recursive equations:

$$\begin{aligned} 1 &\times y = y \\ 2x &\times y = 2(x \times y) \\ (2x+1) &\times y = 2(x \times y) + y \end{aligned}$$

Note that we use 1 as a base case, as multiplication by zero doesn't have a unique left-inverse.

The last equation uses addition, so our first step is to semi-invert addition of binary numbers to obtain subtraction. To make the result of the subtraction unambiguous, we assume that numbers do not have leading zeroes (zero is represented by an empty list of bits).

In the addition above, the first argument is always at least as large as the second. This allows us to simplify the algorithm a bit, so addition (for little-endian lists of bits) looks like:

```
add(m,n) = adc(m,n,0);

adc(as,[],0) = as;
adc(as,[],1) = inc(as);
adc(a:as,b:bs,c) =
  let (s,c1) = add3(a,b,c) in s:adc(as,bs,c1);

add3(0,0,0) = (0,0);
add3(0,0,1) = (1,0);
add3(0,1,0) = (1,0);
add3(0,1,1) = (0,1);
add3(1,0,0) = (1,0);
add3(1,0,1) = (0,1);
add3(1,1,0) = (0,1);
add3(1,1,1) = (1,1);

inc [] = [1];
inc (0:as) | as/=[] = 1:as;
inc (1:as) = 0 : inc as;
```

Note that absence of leading zeroes is explicitly tested in `inc`. This test implicitly informs the semi-inverter of the invariant, which allows the semi-inverter to use the invariant to determine disjointedness of equations. It is often required to assert invariants to make the semi-inverter work, we shall see more of this when we tackle the multiplication rules.

With the division `add(0,1,1) = sub` (which says that we know the second argument and the result of `add` and that the semi-inverse should be named `div`) we get the following output from the semi-inverter:

```
sub (n,e_75_) = (adc_0111 (n,0,e_75_));

adc_0111 ([],0,as) = as;
adc_0111 ([],1,e_78_) = (inc_01 e_78_);
adc_0111 (b : bs,c,s : e_82_) =
  let (a,c1) = (add3_01110 (b,c,s)) in a : (adc_0111 (bs,c1,e_82_));

add3_01110 (0,0,0) = (0,0);
add3_01110 (0,1,1) = (0,0);
add3_01110 (1,0,1) = (0,0);
```

```
add3_01110 (1,1,0) = (0,1);
add3_01110 (0,0,1) = (1,0);
add3_01110 (0,1,0) = (1,1);
add3_01110 (1,0,0) = (1,1);
add3_01110 (1,1,1) = (1,1);

inc_01 [1] = [];
inc_01 1 : as | as/=[] = 0 : as;
inc_01 0 : e_91_ = 1 : (inc_01 e_91_);
```

The test for absence of leading zeroes in `inc` is now a guard that makes the two first rules for `inc_01` disjoint. Note that the semi-inverter automatically finds the required semi-inverses of `adc`, `add3` and `inc`, using names that indicate the division used.

Getting multiplication to semi-invert is a bit more tricky. We start with a straight rewrite of the equations for multiplication into the syntax of the language:

```
mul([1],bs) = bs;
mul(0:as,bs) = 0:mul(as,bs);
mul(1:as,bs) | as/=[] = add(0:mul(as,bs),bs);
```

Note that the guard `as/=[]` is added to ensure non-overlapping equations. We give the semi-inverter this program and a division `mul(0,1,1) = div`. The result is an error-message:

```
Overlap:
div (bs,bs)
div (bs,0 : e_4_)
```

The semi-inverter has found that (at least) two equations of the semi-inverse of `mul` overlaps. This isn't too surprising, as the known argument `bs` isn't used to select equations, so we have only the result to do this, and the right-hand sides of the equations for `mul` are not clearly disjoint. Letting the first argument of `mul` be the known argument doesn't help either, as this makes both arguments to `add` unknown, so clearly no semi-inverse of this can be found. Instead, we must make pattern-matching on `bs` while still keeping it as one of the arguments to the addition.

We have three cases: `[1]`, `0:bs` and `1:bs`, where `bs/=[]` in the last case. The two first cases are so simple that we don't need to special-case on `as`, but instead use mirror-images of the rules for `as = [1]` and `as = 0:as'`. The remaining case gives us the three original rules specialised for `1:bs` with `bs/=[]`:

```
mul(as,[1]) = as;
mul(as,0:bs) = 0:mul(as,bs);
mul([1],1:bs) | bs /= [] = 1:bs;
mul(0:as,1:bs) | bs/=[] = 0:mul(as,1:bs);
mul(1:as,1:bs) | as/=[] && bs/=[] = add(0:mul(as,1:bs),1:bs);
```

In the last rule, we can unroll a step of the addition, so we obtain:

```
mul(as,[1]) = as;
mul(as,0:bs) = 0:mul(as,bs);
mul([1],1:bs) | bs /= [] = 1:bs;
mul(0:as,1:bs) | bs/=[] = 0:mul(as,1:bs);
mul(1:as,1:bs) | as/=[] && bs/=[] = 1:add(mul(as,1:bs),bs);
```

We are nearly there, but the semi-inverter reports overlap between the third and last equation:

```
Overlap:
div (1 : bs,1 : bs) | bs/=[]
div (1 : bs,1 : e_23_) | bs/=[]
```

But we can see that `e_32_`, which is the result of the addition in the last line, must be different from `bs`, so we can add this as an assertion:

```
mul(as,[1]) = as;
mul(as,0:bs) = 0:mul(as,bs);
mul([1],1:bs) | bs /= [] = 1:bs;
mul(0:as,1:bs) | bs/=[] = 0:mul(as,1:bs);
mul(1:as,1:bs) | as/=[] && bs/=[] =
  let xs = add(mul(as,1:bs),bs)
    in let True() = xs/=bs in 1:xs;
```

With this test in place, we can call the semi-inverter with the division `mul(0,1,1) = div` and successfully obtain a semi-inverse (which divides its second argument by its first argument):

```
div ([1],as) = as;
div (0 : bs,0 : e_66_) = (div (bs,e_66_));
div (1 : bs,1 : bs) | bs/=[] = [1];
div (1 : bs,0 : e_77_) | bs/=[] = 0 : (div (1 : bs,e_77_));
div (1 : bs,1 : xs) | bs/=[] && xs/=bs =
  let as = (div (1 : bs,(add_011 (bs,xs)))) in
    let (True ()) = (as/=[]) in 1 : as;
```

The above is taken straight from the output of the semi-inverter, only adding a few newlines and omitting the definition of `add_011`, which is identical to `sub`.

Note that the assertion we inserted in the `mul` function is now part of the guard of the last equation to `div`. This test (that ensures disjoint equations) corresponds to the test that in the traditional algorithm for binary division stops the repeated doubling of the divisor. Note, also, that the pattern-matching on `as` has become an assertion.

Since the division is a semi-inverse of multiplication, it is only defined when the divisor divides evenly into the other argument.

This development shows that semi-inversion in its present state is far from being something you can just use without thought. This is similar to how you will often need to rewrite programs (using *binding-time improvements* [9, 5, 1]) to get good results (or even termination) from partial evaluators.

## 5 Tail-Recursive Programs

Programs with tail-recursive functions can give problems for the semi-inversion transformation.

Glück and Kawabe [6] observe that tail-calls after inversion are similar to left-recursive productions, and that overlapping equations are similar to overlapping productions in a grammar. They suggest using methods inspired by LR-parsing to solve these problems, just like LR-parsing handles the equivalent problems in grammars. Their method might work also for semi-inversion, but a simpler method is sufficient for most cases. Consider the reverse function:

```
reverse(xs) = rev(xs, []);

rev([], acc) = acc;
rev(x : xs, acc) = rev(xs, x : acc);
```

The essense of the problem is that, with the current method, the call-trees of the original and semi-inverted programs are isomorphic, the only difference being the order of children of each node. But to invert the above, we really want to run the iteration in `rev` backwards, i.e., change the call-tree. Glück and Kawabe's solution is to apply a program transformation to the inverted program to change the call-tree, but we propose to apply a simpler transformation prior to (semi-)inversion instead. The idea is that tail-recursion can be seen as iteration, so we introduce an explicit iteration construct in the language: `loop` $f$ $e$ calls $f$ with the value $v_0$ of $e$ as argument. If no rule of $f$ matches, it returns $v_0$ unchanged, but if a rule matches, it applies $f$ to $v_0$ to get $v_1$ and repeats the procedure, i.e., checks if there is a matching rule for $v_1$ and so on. In short, the loop applies $f$ repeatedly until no rule matches, at which point it returns the current value of the argument.

We can rewrite the `reverse` function to use this loop construct:

```
reverse(xs) = let ([], acc) = loop revStep (xs, []) in acc;

revStep(x : xs,acc) = (xs, x : acc);
```

Note that `revStep` corresponds to the recursive case of `rev`, but instead of making the tail-recursive call, it just returns what would have been the arguments to the call. The non-recursive call has now been incorporated into the function that does the looping. Notably, both the initialization and termination patterns are part of this function.

We can now invert `reverse into`

```
reverse2 acc = let (xs, []) = loop revStep_0011 ([], acc) in xs;

revStep_0011 (xs, x : acc) = (x : xs, acc);
```

The loop has been inverted by swapping argument and result and inverting the iteration-step function in the usual way. We can, finally, transform the result back to using a tail-recursive function by making the let-expression of the calling function into the base case for the recursion, mirroring the transformation from tail-recursive function to loop:

```
reverse2 acc = revStep_0011 ([], acc);

rev_0011 (xs, []) = xs;
rev_0011 (xs, x : acc) = rev_001(x : xs, acc);
```

It is interesting to note that the non-recursive rule of `rev_0011` comes from the parameters to the call of the original `rev` and vice-versa. Hence, in retrospect it does not seem reasonable to expect inversion of the program by tansforming each function in isolation. Transforming into the loop form is a way of bringing together the information needed for the successful transformation.

Note, also, that the resulting tail-recursive function has disjoint equations, as required. If we can not rewrite the loops in a (semi-)inverted program back into loop-free disjoint equations, the semi-inversion is not valid, in which case we must backtrack and possibly eventually fail to do (semi-) inversion at all.

As an example of this, consider the function

```
revapp(xs, ys) = rev(xs, ys);

rev([], acc) = acc;
rev(x : xs, acc) = rev(xs, x : acc);
```

Note that `rev` is the same as before. In loop form, this becomes

```
revapp(xs, ys) = let ([], acc) = loop revStep (xs, ys) in acc;

revStep(x : xs,acc) = (xs, x : acc);
```

which we can invert initially to

```
revapp_001(acc) = let (xs, ys) = loop revStep_0011 ([], acc) in (xs, ys);

revStep_0011 (xs, x : acc) = (x : xs, acc);
```

When we rewrite the loop back into functional form, we get

```
revapp_001(acc) = rev_0011 ([], acc);

rev_0011 (xs, ys) = (xs, ys);
rev_0011 (xs, x : acc) = rev_0011 (x : xs, acc);
```

whis has overlapping patterns for `rev_0011`. It should not really be a surprise that we can't invert `revapp`, as it is not injective. But note that the difference is not in the form of the tail-recursive function, but in its calling context. Hence, it is clear that we can not treat a tail-recursive function separate from its context.

We can predict whether a loop can be converted back to functional form after inversion by looking at the original call to the tail-recursive function: If the argument to the call is disjoint from the right-hand sides of the equations for the tail-recursive function, this will make the equations of the inverted function disjoint too. So we can choose to

only rewrite to loop form only calls that have this property. This requirement is similar to the requirements for loops in [8], where the precondition of a loop is required to not overlap the postcondition of the loop body.

It does not seem possible to semi-invert a loop other than by full inversion or no inversion at all.

Not all tail-recursion is easily rewritten to a loop. Indirect tail-recursion will require more extensive rewriting, as will tail-recursive function with several non-recursive rules. Multiple recursive rules are no problem, as long as their inverses are disjoint equations.

At the moment, we have not implemented the transformations between tail-recursive functions and loops, but the example above (and a few more complicated examples) written with explicit loops in the source text have been transformed by the current implementation.

This includes inverting a compiler from queue-machine code [14] to syntax trees to get a compiler from syntax tres to queue machine code[2]. In a queue machine, instructions take their operands from the front of a queue and put their results at the end of the queue. Building a syntax tree is easily done by letting the queue contain subtrees rather than values and let instructions build larger subtrees from the operator in the instruction and the subtrees in the queue:

```
fromQueue(prog) = let ([],[t]) = loop fq (prog,[]) in t;

fq(Const n : p, q) = (p, Num n : q);
fq(Bop op : p, q) = let (a,q1) = dequeue q in
                      let (b,q2) = dequeue q1 in
                        (p, Binop (op,a,b) : q2);
fq(Uop op: p, q) = let (a,q1) = dequeue q in
                     (p, Unop (op, a) : q1);

dequeue([x]) = (x,[]);
dequeue(x:xs) | xs/=[] = let (y,ys) = dequeue(xs) in
                           (y,x:ys);
```

The queue (`q`) is represented as a list where the front of the queue is the end of the list. `dequeue` returns a pair of the head of the queue and the rest of it. Inverting with the division `fromQueue(0,1)=toQueue;` produces the following compiler from syntax trees to queue code:

```
toQueue t = let (prog, []) = loop fq_0011 ([], [t]) in prog;

fq_0011 (p, (Num n) : q) = ((Const n) : p, q);
fq_0011 (p, (Binop (op, a, b)) : q2) =
   ((Bop op) : p, dequeue_011 (a, dequeue_011 (b, q2)));
fq_0011 (p, (Unop (op, a)) : q1) =
   ((Uop op) : p, dequeue_011 (a, q1));
```

[2] Thanks to Rustan Leino of Microsoft Research for suggesting this problem.

```
dequeue_011 (x, []) = [x];
dequeue_011 (y, x : ys) = let xs = dequeue_011 (y, ys) in
                              let True () = xs/=[] in x : xs;
```

`dequeue_011` is now an enqueue operation, but on a reversed queue.

## 6 Conclusion

As the multiplication example shows, it is sometimes necessary to rewrite programs and add assertions of invariants to get successful semi-inversion. This example, though small, is rather complex, so we don't expect nearly as much rewriting to be necessary for the majority of programs. Even so, semi-inversion will probably remain a tool for experts until the technology matures.

Ideally, a semi-inversion method should discover such invariants, but it is unrealistic to expect it to always do so, as discovery of nontrivial invariants is uncomputable. As a consequence, it may sometimes be necessary to provide such invariants as extra information to the semi-inversion process. Adding such redundant assertions is conceptually similar to using *binding time improvements* [9,5,1] to improve the result of partial evaluation.

In addition to making invariants explicit, it may be necessary to combine several user-defined functions, just like some predefined operators are combined at the refinement stage. For example, the function that returns the last element of a list can not be inverted on its own, nor can the function that returns all but the last element of a list. In combination, they can, but only if the functions are merged into a single function (such as `dequeue` above). Such merging of functions is called *tupling* and can be automated [2, 16].

Another non-trivial program that has been semi-inverted by the semi-inverter is an interpreter of the invertible stack language used in [6]. The interpreter was semi-inverted by specifying the program and output as known while the input is unknown. The result is an *inverse interpreter* [7]. This example did not require any assertion of invariants or similar tricks but, admittedly, the language was designed by Glück and Kawabe to be easily invertible.

Tail-recursive functions are in the semi-inverter handled by transformation to explicit loops, and they can only be left unchanged or fully inverted.

Another limitation of the current method is that it only works on first-order equations. We are currently working on extending it to handle simple cases of higher-order functions.

## References

1. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10. ACM Press, 1992.
2. Wei-Ngan Chin and Hu Zhenjiang. Towards a modular program derivation via fusion and tupling. In *Proceedings of the first ACM SIGPLAN Conference on Generators and Components*, pages 140–155. ACM Press, 2002.

3. Edsger W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction: International Summer School*, LNCS 69, pages 54–57. Springer-Verlag, 1978.
4. Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, 1967.
5. Robert Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–19. ACM Press, 2002.
6. Robert Glück and Masahiko Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming. Proceedings*, LNCS 2998, pages 291–306. Springer-Verlag, 2004.
7. Robert Glück, Youhei Kawada, and Takuya Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 10–19. ACM Press, 2003.
8. David Gries. *The Science of Programming*, chapter 21 Inverting Programs, pages 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
9. Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In [10], pages 83–100, 1991.
10. Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990. Workshops in Computing*. Springer-Verlag, August 1991.
11. Ed Knapen. *Relational programming, program inversion and the derivation of parsing algorithms*. Master's thesis, Eindhoven University of Technology, 1993.
12. Torben Æ. Mogensen. Semi-inversion of guarded equations. In *GPCE'05*, Lecture Notes in Computer Science 3676, pages 189–204. Springer-Verlag, 2005.
13. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In Dexter Kozen, editor, *Mathematics of Program Construction. Proceedings*, LNCS 3125, pages 289–313. Springer-Verlag, 2004.
14. Bruno Richard Preiss. *Data Flow on a Queue Machine*. PhD thesis, Department of Electrical Engineering, University of Toronto, 1987. 185 pp.
15. A. Y. Romanenko. The generation of inverse functions in Refal. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 427–444. North-Holland, 1988.
16. Jens Peter Secher. Driving in the jungle. In Olivier Danvy and Andrzej Filinsky, editors, *Programs as Data Objects. Proceedings*, LNCS 2053, pages 198–217. Springer-Verlag, 2001.