Semi-Inversion of Functional Parameters

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.

Previous papers by the author have described semi-inversion for a first-order functional language. We now extend semi-inversion to handle functions as parameters.

We start by summarising the steps of the semi-inversion transformation for first-order functional languages and then describes how these steps can be extended to handle functions as parameters and illustrate this by a running example.

It turns out that, even with the fairly modest extension of the language, the resequentialisation step of the semi-inversion transformation is considerably complicated.

We conclude by comparison with related work and discussion of future developments.

Categories and Subject Descriptors D [*3*]: 4

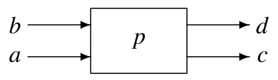
General Terms Algorithms, Languages

1. Introduction

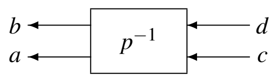
Inversion of programs [Glück and Kawabe(2004), Dijkstra(1978), Gries(1981), Mu et al.(2004)Mu, Hu, and Takeichi, Knapen(1993), Nishida et al.(2001)Nishida, Sakai, and Sakabe] is a process that transforms a program that implements a function f into a program that implements the inverse function of f, i.e., f^{-1}. A related technique was used in [Floyd(1967)] to add backtracking to a program by adding code to rewind earlier computation. A generalisation called "partial inversion" [Romanenko(1988), Romanenko(1991), Nishida et al.(2005)Nishida, Sakai, and Sakabe] assumes that the result and some of the input is available and creates a program that given these will compute the remaining input.

Semi-inversion [Mogensen(2005), Mogensen(2007)] is a generalisation of both of these 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:

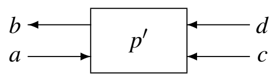
PEPM'08 January 7–8, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-977-7/08/0001...$5.00



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

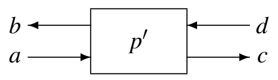


With partial inversion inversion, some of the incoming arrows can retain their orientation, but the out-going arrows are all reversed:



The relation between p and p' is that if $(c,d) = p(a,b)$ then $b = p'(a,c,d)$. If the program has two inputs and a single output, the partial inverses of the program correspond to the mathematical notion of left and right inverses of a binary operator.

With semi-inversion, we can choose to retain the orientation of any subset of the arrows of the arrows while reversing the rest:



The relation between p and p' is now that if $(c,d) = p(a,b)$ then $(b,c) = p'(a,d)$. Semi-inversion thus generalises both inversion and partial inversion.

In [Mogensen(2005)], some theory about semi-inversion and a method for it is described, but no implementation existed at the time of writing. A later paper, [Mogensen(2007)], reports experiences with an implementation of the methods described in the earlier paper and extends the method to better handle tail-recursion by rewriting tail-recursive functions to loops.

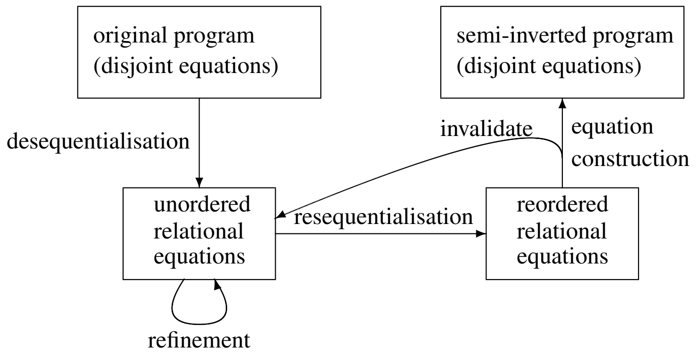
The present paper extends the semi-inversion method to handle functions as parameters to other functions, i.e., second-order functions. We start by summarising the semi-inversion method by showing the steps on a simple example. We then describe how this method can be extended to handle functions as parameters, show some examples and, finally, discuss how the methods presented in this paper might be extended to cover full higher-order programs (i.e., closures).

2. The semi-inversion transformation

We start by summarising the semi-inversion transformation as it applies to first-order programs, as shown in [Mogensen(2005), Mogensen(2007)].

We will subsequently in section 3 extend this method to handle functional parameters.

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 [Mogensen(2005)] and [Mogensen(2007)], which also describes how Haskell-style guarded equations are handled. Since guards are not an issue in this paper, we simplify the following description by omitting these

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 (bs,as) in
    ((a+1):ps,(a-b):ms);
```

Note that the recursive call to pm has swapped the arguments.

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 Haskell-style equation $f\ P = E$ is desequentialised as follows:

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

The functions I_p and I_e are detailed in [Mogensen(2005)]. 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(bs,as,ps,ms),
         u=+(a,1), 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 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) to indicate that the second and last in-outs for pm will be inputs while the first and third are outputs.

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

$$\begin{array}{lcl} z = +(x,y) & \Rightarrow & x = -(z,y),\ y = -(z,x) \\ z = -(x,y) & \Rightarrow & x = +(z,y),\ y = -(x,z) \end{array}$$

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. The same function can be resequentialised for several different divisions.

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

```
pm_0101 (w,x,y,z)
  where [x=[], z=[], w=[], y=[]]
pm_0101 (w,x,y,z)
  where [x=:(b,bs), z=:(v,ms), a=+(v,b), u=+(a,1),
         pm_1001(bs,as,ps,ms), y=:(u,ps), w=:(a,as)]

pm_1001 (w,x,y,z)
  where [w=[], z=[], x=[], y=[]]
pm_1001 (w,x,y,z)
  where [w=:(a,as), z=:(v,ms), u=+(a,1), b=-(a,v),
         pm_0101(bs,as,ps,ms), x=:(b,bs), y=:(u,ps)]
```

Note that pm is called with two different divisions, so two resequentialisations are made. These will be two (mutually recursive) functions in the semi-inverted program.

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, as described below.

There might not be any evaluation order that can make all variables known, so 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, the failure may have been caused by incorrectly assuming that the input/output division used for resequentialisation is sufficient. Hence, we mark this division as invalid. If already completed resequentialisations for other functions depends on the now invalidated division, we redo the resequentialisation of these. This may invalidate these also, possibly all the way up to the top-level function.

2.3 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 two steps:

1. Construct patterns from structural relations for new inputs.
2. Make expression from remaining relations.

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

```
pm_0101([],[]) = ([],[]);
pm_0101(b:bs,v:ms) =
  let a = v+b in
    let u = a+1 in
      let (as,ps) = pm_1001(bs,vs) in
        let y = u:ps in
          let w = a:as in (w,y);

pm_1001([],[]) = ([],[]);
pm_1001(a:as,v:ms) =
  let u = a+1 in
    let b = a-v in
      let (bs,ps) = pm_0101(as,ms) in
        let x = b:bs in
          let y = u:ps in (x,y);
```

After transforming each equation as described above, equations for the same function must by their patterns divide the input into disjoint classes, i.e., there can be no overlap. This ensures that the semi-inverse defines a function and not a one-to-many or many-to-many relation and that the order in which the equations appear doesn't matter.[1] If the equations are not disjoint, the semi-inverted function is not valid, so we mark the input/output subset for the function as invalid and backtrack, like we do if resequentialisation fails.

In our example, the equations are clearly disjoint, so the semi-inverses are valid. The semi-inverter additionally unfolds trivial or linear let-definitions, so the actual output is like the following (we have, for readability, renamed variables and added line breaks):

```
pm_0101([],[]) = ([],[]);
pm_0101(b:bs,v:ms) =
  let a = v+b in
    let (as,ps) = pm_1001(bs,vs) in
      (a:as,(a+1):ps);

pm_1001([],[]) = ([],[]);
pm_1001(a:as,v:ms) =
  let (bs,ps) = pm_0101(as,ms) in
    ((a-v):bs,(a+1):ps);
```

This concludes the summary of the first-order transformation.

3. Extending to functional parameters

We add two new constructs to the expressions in the language:

fun *f* creates a functional value from a function *f*. Since all function definitions are global and uncurried, no closure is constructed – the functional value is simply a reference to a globally declared function.

x @ *E* applies a functional value *x* to an argument *E*.

A further restriction is that functional values can not be returned from functions, only passed as arguments, so we are working with second-order functions only. We will discuss lifting this restriction in section 5.

3.1 Running example

Consider the following program:

```
main(xs,ys) = zipWith2(fun sub,xs,ys);

zipWith2(f,[],[]) = [];
zipWith2(f,a1:as,b1:bs) =
    f@(a1,b1) : zipWith2(f,bs,as);

sub(x,y) = x-y;
```

Assume we want to semi-invert `main` with `xs` and the result known but `ys` unknown, i.e., with the division `main(1,0,1)`.

We observe that when `main` calls `zipWith2`, the first and second parameters and the result is known, but the third parameter is not, i.e., we use the division `zipWith2(1,1,0,1)`.

[1] An alternative is to add backtracking, but that can have serious implications for efficiency, so we won't.

Inside `zipWith2` with this division, we call the functional parameter `f` with the division `f(1,0,1)` and `zipWith2` with the division `zipWith2(1,0,1,1)`, so we need one more variant of `zipWith2`. Inside this variant, `f` is called with the division `f(0,1,1)` and `zipWith2` with the division `zipWith2(1,1,0,1)`, the latter of which we have already seen.

Note that `f` is applied in two places with two different divisions, so we need two different semi-inverses of the functions that `f` can be bound to (in this case, `sub`), and we need to give both semi-inverses of these as parameters to the semi-inverses of `zipWith2`.

The complete semi-inverted program is:

```
main_101(xs,zs) =
  zipWith2_1101((fun sub_101,fun sub_011),xs,zs);

zipWith2_1101((f_101,f_011),[],[]) = [];
zipWith2_1101((f_101,f_011),a1:as,c1:cs) =
   f_101@(a1,c1) :
       zipWith2_1011((f_101,f_011),as,cs);

zipWith2_1011((f_101,f_011),[],[]) = [];
zipWith2_1011((f_101,f_011),b1:bs,c1:cs) =
   f_011@(b1,c1) :
       zipWith2_1101((f_101,f_011),bs,cs);

sub_101(x,z) = x-z;

sub_011(y,z) = z+y;
```

This example shows two complications when semi-inverting in the presence of functional parameters:

1. One functional parameter in the original program may become several functional parameters in the semi-inverted program – one for each division at which the functional parameter is used. We combine these as a single parameter which holds a tuple of functional values. In the example, in both semi-inverses of `zipWith2` the functional parameter `f` is used at two different divisions (`(1,0,1)` and `(0,1,1)`), so `zipWith2_1101` and `zipWith2_1011` both have have the pair `(f_101,f_011)` as parameters.
2. When we handle a call to a function *g* , we know which parameters and results of *g* are known, so we can start resequentialising *g* at this division. But it is not until we have made this resequentialisation that we know at which divisions *g*'s functional parameters are called. This information needs to be propagated back to the place *g* was called, so the required semi-inverses of *g*'s functional parameters are passed to it. In our example, when we resequentialise `main` we need to know how the semi-inverse of `zipWith2` uses `f` in order to know which semi-inverses of `sub` we need to pass to it.

   The information flow about desired semi-inverses of functional parameters is, hence, from a called function to its caller, where in the first-order case, the information flow is only from caller to callee.

We will address these issues in the subsequent sections, where we extend the method sketched in section 2 to handle functional parameters.

3.2 Desequentialisation

Desequentialisation is trivially extended:

- An expression of the form **fun** *f* is translated into a relation $y = \texttt{fun}\ f$, where *y* is used to connect the expression to its context. So a call `h(fun f,x)` is replaced by the relations `y`

= `fun f, h(y,x,z)`, where `z` represents the result of the call to `f`.

- A higher-order application g @ $(x_1,\dots,x_n)$ is translated to the relation g @ $(x_1,\dots,x_n,z)$, respectively, where z holds the result of the call. Note that, as for first-order calls, we combine inputs and output to a higher-order call in a single list of in-outs.

Our motivating example from section 3.1 is desequentialised to

```
main(xs,ys,zs)
  where {g = fun sub, zipWith2(g,xs,ys,zs)}

zipWith2(f,as,bs,cs)
  where {as=[], bs=[], cs=[]}
zipWith2(f,as0,bs0,cs0)
  where {as0=:(a1,as), bs0=:(b1,bs),
         f@(a1,b1,c1), zipWith2(f,bs,as,cs),
         cs0=:(c1,cs)}

sub(x,y,z)
  where {z=x-y}
```

3.3 Resequentialisation

This is where it becomes somewhat tricky.

Resequentialisation of $x = \texttt{fun}\ f$ may seem trivial, as f is known, so x will be known too. However, what we need is not really f, but a collection of semi-inverses of f. Exactly which semi-inverses of f we need depends on how x is used.

In our example, we must first resequentialise `zipWith2` at the division `zipWith2(1,1,0,1)` to find the divisions for `g = fun sub` in `main`. These turn out to be `{g_101 = fun sub_101, g_011 = fun sub_011}`. How this is found, we will see below.

An application x @ $(y_0,\dots,y_n)$ of a variable x holding a functional value to an argument/result tuple will be resequentialised as x_{div} @ $(y_0,\dots,y_n)$, where x_{div} is the variant of x that has the division corresponding to the known/unknown status of the variables in $(y_0,\dots,y_n)$. For example, the application `f@(a1,b1,c1)` in `zipWith2_1101` is resequentialised to `f_101@(a1,b1,c1)` because `a1` and `c1` are known while `b1` is not. Similarly, `f@(b1,a1,c1)` in `zipWith2_1011` is resequentialised to `f_011@(b1,a1,c1)`.

A variable x holding a functional value that is used as a parameter to a call or application will have to be replaced by a parameter holding a tuple of variables, one for each different application of x inside the called function and, possibly, more if x inside the called function is also used as a parameter to another call.

Inside `zipWith2_1101`, `f` is applied to the division `(1,0,1)` and the call to `zipWith2_1011` adds another use of `f` at the division `(0,1,1)`, so we replace the formal parameter `f` of `zipWith2_1101` with a pair: `(f_101, f_011)`. `zipWith2_1011` has (due to mutual recursion) the same uses of `f` as `zipWith2_1101`, so it also replaces `f` by `(f_101 f_011)`.

In a call to `zipWith2_1101`, the actual parameter corresponding to the original formal parameter `f` will have to be replaced by a pair corresponding to the two divisions for `f`. In `main`, the actual parameter is `g`, so the call `zipWith2(g,xs,ys,zs)` is resequentialised to `zipWith2_1101((g_101,g_011),xs,ys,zs)`. Similarly, the call from `zipWith2_1101` to `zipWith2(f,bs,as,cs)` is resequentialised to `zipWith2_1101((f_101,f_011),bs,as,cs)`.

In section 2.2, data flow is forwards-only: When we resequentialise a function body, we generate demands for desired semi-inverses of the functions called in the body, so the demands go from caller to callee.

Now, we also have information flow from callee to caller: The callee provides information about the uses of is functional parameters. To provide this information to its caller, the callee needs to get information from the functions it calls, and so on.

```
$desired := \{(\texttt{main}, div_{main})\}$
$completed := \{\}$
$invalidated := \{\}$
$repeat$
  $get\ (f, div)\ from\ desired$
  $desired := desired \setminus \{(f, div)\}$
  $if\ (f, div, uses, eqs) \in completed\ then$
    $uses_0 := uses$
    $completed := completed \setminus \{(f, div, uses, eqs)\}$
  $else$
    $uses_0 := \{\}$
  $endif$
  $uses_1 := uses_0$
  $eqs_1 := \{\}$
  $invalidate := false$
  $for\ rels \in f's\ definition\ do$
    $case\ resequentialiseRels(rels, div, uses_1)\ of$
      $SOME\ (rels', uses):$
        $eqs_1 := eqs_1 @ [rels']$
        $uses_1 := uses_1 \cup uses$
      $NONE:$
        $invalidate := true$
    $endcase$
  $endfor$
  $if\ fail\ then$
    $invalidated := invalidated \cup \{(f, div)\}$
  $else$
    $completed := completed \cup \{(f, div, uses_1, eqs_1)\}$
    $if\ uses_1 \neq uses_0\ then$
      $desired := desired\ \cup \{(f, div)\}$
                           $\cup \{\text{all semi-inverses that call } (f, div)\}$
    $endif$
  $endif$
$until\ desired = \{\}$
```

Figure 1. Algorithm for resequentialisation

As functions are recursive, we may need to know how a function is resequentialised in order to resequentialise it. This suggests that we use a fixed-point iteration:

The first time we encounter a call at a particular division, we assume that the called function does not use its functional parameters. If we then, when we resequentialise the called function, find that it does use its functional parameters, we store this information with its division and redo the resequentialisation of its caller. We repeat this until no changes in the stored information occurs.

The algorithm is sketched is figure 1 and explained in more detail below.

We work with three sets:

- A set of desired semi-inverses, which holds function/division pairs that we have processed calls to, but which we have not yet resequentialised. Initially this set holds only the top-level function at the desired division.
- A set of completed semi-inverses, which holds function/division pairs that we have resequentialised. Each of these holds the resequentialised relations for the function and information about how it uses its functional parameters. The set of completed semi-inverses is initially empty.

- A set of invalidated semi-inverses, which holds function/division pairs that we have tried, but failed to resequentialise. This is also initially empty.

When we take a function/division pair (f,d) out of the set of desired semi-inverses, we resequentialise the body of f using the division d while keeping track of uses of functional variables. If (f,d) is already in the set of completed semi-inverses, we initialise the uses with the uses stored there.

When we in the body need to resequentialise a call to g at division d', we look for (g,d') in the set of completed semi-inverses. If we find a semi-inverse matching (g,d'), it has information about how it uses its functional parameters. We add this information to the uses of the functional variables used in f as arguments to the call to g. If there is no completed semi-inverse for (g,d'), we check the set of invalidated semi-inverses. If (g,d') is in this set, we can not resequentialise it, so we try to add another relation to the sequence first, in the hope that more variables will become known. If there is no relation we can choose, we invalidate (f,d) by adding it to the set of invalidated semi-inverses. If (g,d') is in neither the set of completed semi-inverses nor in the set of invalidated semi-inverses, we add it to a set of desired semi-inverses. Since we don't know how (g,d') will use its functional parameters, we assume it doesn't, so the uses of functional variables in the body of f is unchanged.

When we complete reseqentialisation of the body of f *except* the relations of the form $x = \texttt{fun}\ g$, we use the set of uses of x (i.e., the different divisions at which x is used) to resequentialise $x = \texttt{fun}\ g$ to a set of relations, one for each different use of x. These relations are added to the front of the resequentialised body of f, so it will appear before all relations that use x. If f has a functional parameter of its own, say h, we add the information of the uses of h to (f,d) when we add this to the set of completed semi-inverses.

If one of the following three things happen:

1. (f,d) is invalidated.
2. (f,d) was not already in the set of completed semi-inverses, and the information about uses of h that we add with (f,d) in the set of completed semi-inverses is non-empty.
3. (f,d) was already in the set of completed semi-inverses but the uses of h changes.

previous resequentialisations of semi-inverses that called (f,d) have been done under false assumptions, so we need to redo these. We do this by looking through the set of completed semi-inverses for functions that call (f,d) and add each to the set of desired semi-inverses (but still keeping then in the set of completed semi-inverses, so the information about how they use their own functional parameters can still be used). Additionally, we also add (f,d) itself to the set of desired semi-inverses, so the equations for (f,d) can get the same tuple of functional parameters.[2]

Eventually, we will either invalidate the top-level function at its desired division or the set of desired semi-inverses becomes empty. In the first case, we have failed to semi-invert the program, and must report an error. In the latter case, semi-inversion is successful and we can proceed to constructing equations for the resequentialised functions in the set of completed semi-inverses.

To make this process more clear, we will apply it to the example in some detail.

[2] In the current implementation, we simply add *all* completed semi-inverses to the set of desired semi-inverses whenever anything changes. This is less efficient, but easier to do.

3.3.1 Resequentialising the example

Initially, the set of desired semi-inverses contains only the `main` function at the division `(1,0,1)`. In the body, we can only select the call to `zipWith2` (as definitions of functional variables must be treated last). The pair `(zipWith2,(1,1,0,1))` is not found in the set of completed semi-inverses, so we add it to the list of desired semi-inverses. Since there are no known uses of the functional parameter, we use an empty tuple as argument. We now resequentialise `g = fun sub`, but since there are no known uses of `g`, it doesn't generate any relations. We have now completed resequentialisation of `(main,(1,0,1))` and add it to the set of completed semi-inverses with the definition

```
main_101(xs,ys,zs)
  where [zipWith2_1101((),xs,ys,zs)]
```

Since there are no functional parameters to `main`, there is no information about uses of functional parameters. The set of desired semi-inverses now contains the pair `(zipWith2,(1,1,0,1))`, so we resequentialise its definition. The first equation is trivial, as we can select the relations in any order. Since there are no uses of the functional parameter in the body of this equation and we don't have information from previous resequentialisations that contradict this, we use the empty tuple for the functional parameter.

In the second equation, we know `as0` and `cs0`, we can immediately select the relations `as0=:(a1,as)` and `cs0=:(c1,cs)`, which add `a1, as, c1` and `cs` to the set of known variables. We then select the call to `zipWith2` at the division `(1,0,1,1)`, which is not in the set of completed semi-inverses, so we add it to the set of desired semi-inverses.

Since we have no information about uses of the functional parameter of `(zipWith2,(1,1,0,1))`, we replace the functional argument by the empty tuple. We now also have `bs` in the set of known variables. We next select the application `f@(a1,b1,c1)` at the division `(1,0,1)` (which we will record in the set of uses) and, finally, select `bs0=:(b1,bs)`. We now add the pair `(zipWith2,(1,1,0,1))` to the set of completed semi-inverses with the definition

```
zipWith2_1101((),as,bs,cs)
  where [as=[], bs=[], cs=[]]
zipWith2_1101((f_101),as0,bs0,cs0)
  where [as0=:(a1,as), cs0=:(c1,cs),
         zipWith2_1011((),bs,as,cs), f_101@(a1,b1,c1),
         bs0=:(b1,bs)]
```

and the information that it uses its first parameter at the division `(1,0,1)`. Note that the functional parameter is replaced by a tuple of one element, corresponding the single application of `f` in the body.

The added use information contradicts the assumption about uses in `(zipWith2,(1,1,0,1))` that we made when resequentialising `main`. Hence, we have to redo resequentialisation of the `main` function. We achieve this by adding `(main,(1,0,1))` to the set of desired semi-inverses. We also add `(zipWith2,(1,1,0,1))`, since the change in the set of uses makes the first equation inconsistent with the use information.

We now select `(zipWith2,(1,0,1,1))` from the set of desired semi-inverses. The first equation is, again, trivial. In the second, we can select `bs0=:(b1,bs)` and `cs0=:(c1,cs)` (giving us `b1, bs, c1` and `cs`). Next, we can select the application `f@(a1,b1,c1)`, which adds `(0,1,1)` to the uses of `f`. Then select the call to `zipWith2` at the division `(1,1,0,1)`. This is already in the set of completed semi-inverses and gives `{(1,0,1)}` as the set of uses of its functional (first) argument. After adding `as0=:(a1,as)`, we have completed resequentialisation of the definitions for (zipWith2,(1,0,1,1)), so we add it to the set of completed semi-inverse with the equations

```
zipWith2_1011((),as,bs,cs)
  where [bs=[], cs=[], as=[]]
zipWith2_1011((f_101,f_011),as0,bs0,cs0)
  where [bs0=:(b1,bs), cs0=:(c1,cs),
         f_011@(a1,b1,c1),
         zipWith2_1101((f_101,f_011),bs,as,cs),
         as0=:(a1,as)]
```

and the information that f is used at the divisions {(1,0,1), (0,1,1)}. Since the resequentialisation of (zipWith2,(1,1,0,1)) assumed differently, we add it back to the set of desired semi-inverses. Like before, we must also redo resequentialisation of the first equation for (zipWith2,(1,0,1,1)), so we also add (zipWith2,(1,0,1,1)) to the set of desired semi-inverses.

The new resequentialisation of (zipWith2,(1,1,0,1)) only changes the uses of the functional parameter to {(1,0,1), (0,1,1)}, which gives the following new definition:

```
zipWith2_1101((f_101,f_011),as,bs,cs)
  where [as=[], bs=[], cs=[]]
zipWith2_1101((f_101,f_011),as0,bs0,cs0)
  where [as0=:(a1,as), cs0=:(c1,cs),
         zipWith2_1011((f_101,f_011),bs,as,cs),
         f_101@(a1,b1,c1),
         bs0=:(b1,bs)]
```

Since (zipWith2,(1,0,1,1)) and (main,(1,0,1)) were resequentialised under different assumptions, we must add them to the set of desired semi-inverse. (They are already there, though).

Redoing resequentialisation of (zipWith2,(1,0,1,1)) changes the first equation, but doesn't change the use information, so we now only need to resequentialise (main,(1,0,1)). The only thing that changes is the set of uses of g, which is now {(1,0,1), (0,1,1)}. This changes the resequentialisation of g = fun sub to the two relations g_101 = fun sub_101, g_011 = fun sub_011. We must now add (sub,(1,0,1)) and (sub,(0,1,1)) to the set of desired semi-inverses. All in all, the new resequentialisation of (main,(1,0,1)) becomes

```
main_101(xs,ys,zs)
  where [g_101 = fun sub_101, g_011 = fun sub_011,
         zipWith2_1101((g_101,g_011),xs,ys,zs)]
```

Resequentialisation of (sub,(1,0,1)) and (sub,(0,1,1)) is straight-forward and adds no new desired semi-inverses, so after doing this, we have an empty set of desired semi-inverses. The complete resequentialised program is

```
main_101(xs,ys,zs)
  where [g_101 = fun sub_101, g_011 = fun sub_011,
         zipWith2_1101((g_101,g_011),xs,ys,zs)]

zipWith2_1101((f_101,f_011),as,bs,cs)
  where [as=[], bs=[], cs=[]]
zipWith2_1101((f_101,f_011),as0,bs0,cs0)
  where [as0=:(a1,as), cs0=:(c1,cs),
         zipWith2_1011((f_101,f_011),bs,as,cs),
         f_101@(a1,b1,c1),
         bs0=:(b1,bs)]

zipWith2_1011((f_101,f_011),as,bs,cs)
  where [bs=[], cs=[], as=[]]
zipWith2_1011((f_101,f_011),as0,bs0,cs0)
  where [bs0=:(b1,bs), cs0=:(c1,cs),
         f_011@(a1,b1,c1),
         zipWith2_1101((f_101,f_011),bs,as,cs),
         as0=:(a1,as)]

sub_101(x,y,z) where [y=x-z]

sub_011(x,y,z) where [x=z+y]
```

3.4 Constructing equations

Constructing equations from the resequentialised equations adds no significant complications. From a functional value definition of the form

$$x \;=\; \texttt{fun}\, f_{div},\, rs$$

we construct

$$\texttt{let}\, x = \;\texttt{fun}\, f_{div}\; \texttt{in}\; e$$

where e is the expression for the relation sequence rs. From an application of the form

$$g_{div}\; @\; (x_0,\ldots,x_n),\, rs$$

we construct

$$\texttt{let}\; (z_0,\ldots,z_m) \;=\; g_{div} @\; (y_0,\ldots,y_k)\; \texttt{in}\; e$$

where $(z_0,\ldots,z_m)$ and $(y_0,\ldots,y_k)$ is the division into input and output variables of $(x_0,\ldots,x_n)$ corresponding to the division div of g_{div} and, again, e is the expression for the relation sequence rs.

Applying this to the resequentialised program yields the following:

```
main_101(xs,zs) =
  let g_101 = fun sub_101 in
    let g_011 = fun sub_011 in
      let ys = zipWith2_1101((g_101,g_011),xs,zs) in
        ys

zipWith2_1101((f_101,f_011),[],[]) =
  let bs = [] in bs
zipWith2_1101((f_101,f_011),a1:as,c1:cs) =
  let bs = zipWith2_1011((f_101,f_011),as,cs) in
    let b1 = f_101@(a1,c1) in
      let bs0 = b1:bs in
        bs0

zipWith2_1011((f_101,f_011),[],[]) =
  let as = [] in as
zipWith2_1011((f_101,f_011),b1:bs,c1:cs)
  let a1 = f_011@(b1,c1) in
    let as = zipWith2_1101((f_101,f_011),bs,cs) in
      let as0 = a1:as in
        as0

sub_101(x,z) =
  let y = x-z in y

sub_011(y,z) =
  let x = z+y in x
```

We can, as in the first-order case, unfold linear let-expressions. If we do that for our example program, we get the semi-inverted program shown in section 3.1.

3.5 Summary

Most phases of the semi-inversion method are trivially extended, but resequentialisation requires nontrivial changes to handle functional parameters. The changes centre on propagating information about use of functional parameters back to their definitions.

4. Other examples

The method is implemented as an extension to the semi-inverter reported in [Mogensen(2007)]. We have tried it on some other examples, some of which we show below.

4.1 Map

A program using the map function:

```
main l =  map(fun add1, map(fun double, l));

map(f, []) = [];
map(f, a:as) = f @ a : map(f, as);

add1 x = x+1;

double x = 2*x;
```

has been fully inverted to the following program:

```
main_oi e_53_ =
  map_foi (fun double_oi,
           map_foi (fun add1_oi, e_53_));

map_foi (f_oi, []) = [];
map_foi (f_oi, e_57_ : e_58_) =
  (f_oi@e_57_) : map_foi (f_oi, e_58_);

double_oi e_61_ = e_61_/2;

add1_oi e_60_ = e_60_-1;
```

The shown is actual output from the semi-inverter (except that newlines have been added to fit the column), hence the uninformative naming of variables. The divisions for functions is shown as f for functional parameters, i for input parameters and o for output parameters. These were shown as 1, 0 and 1, respectively, in the handwritten examples above. Note that map_foi is identical to map (except for renaming), which proves that the inverse of mapping a function is mapping the inverse of the function, i.e., that $(map\ f)^{-1} = map\ f^{-1}$.

4.2 Scan

The scan function folds a function on all prefixes of a list. It can be defined as:

```
scan(f,a,[]) = [];
scan(f,a,b:bs) = let c = f@(a,b) in c:scan(f,c,bs);
```

If we add a main function and a function to use as argument:

```
main(xs) = scan(fun add,0,xs);

add(x,y) = x+y;
```

and semi-invert main with the division (0,1), we get:

```
main_oi e_31_ = scan_fioi (fun add_ioi, 0, e_31_);

scan_fioi (f_ioi, a, []) = [];
scan_fioi (f_ioi, a, c : e_36_) =
  (f_ioi@(a, c)) : scan_fioi (f_ioi, c, e_36_);

add_ioi (x, e_38_) = e_38_-x;
```

Note how both scan and add have been semi-inverted, even though the main function is fully inverted. If we rename a bit for readability, we can see the "unscan" function more clearly:

```
unscan(g,a,[]) = [];
unscan(g,a,c:cs) = g@(a,c) : unscan (g,c,cs);
```

where we have the relation that if $\mathtt{c} = \mathtt{f}(\mathtt{a},\mathtt{b}) \Leftrightarrow \mathtt{b} = \mathtt{g}(\mathtt{a},\mathtt{c})$ then $\mathtt{cs} = \mathtt{scan}(\mathtt{f},\mathtt{i},\mathtt{as}) \Leftrightarrow \mathtt{as} = \mathtt{unscan}(\mathtt{g},\mathtt{i},\mathtt{cs})$.

5. Future work

We will look at some limitations of the method as it is described so far, suggesting possible solutions for each.

5.1 Requirements on functional values

An issue we have skirted in section 3.3 is what to do if a semi-inverse of f required by $x = \mathtt{fun}\ f$ is invalidated.

Let us, for example, look at the following (rather contrived) example:

```
main(a,b) = h(fun p,a,b);

p(a,b) = a/2+b;

h(f,a,b) = (f@(a,b), f@(b,a));
```

where / is truncating integer division.

The first thing we note is that p can be successfully semi-inverted with the divisions (1,0,1) and (1,1,1) and giving the semi-inverses

```
p_101(a,c) = c-a/2;

p_111(a,b,c) = let True() = c=a/2+b in ();
```

where the latter has converted the calculation into a assertion.

Note that p can not be semi-inverted with the division (0,1,1), as there is no unique inverse for integer division by two.

If we semi-invert main with the division (1,0,1), we call h with the division (1,1,0,1).

Desequentialising h yields

```
h(f,a,b,e) where
 {f@(a,b,c), f@(b,a,d), e=(c,d)}
```

Resequentialising this with the division (1,1,0,1) can choose either call first, yielding the following two alternative resequentialisations:

```
h_1101((f_011,f_111),a,b,e) where
 [e=(c,d), f_011@(b,a,d), f_111@(a,b,c)]

h_1101((f_101,f_111),a,b,e) where
 [e=(c,d), f_101@(a,b,c), f_111@(a,b,d)]
```

Note that, though the two resequentialisations use the same division, their uses of the functional parameter differ. Let us assume that we had chosen the first resequentialisation for h_1101. The main function would find that p is required at the division (0,1,1), but semi-inversion of that fails. In the current implementation, this will invalidate main at the current division, so semi-inversion of the program fails.

But semi-inversion is actually possible: If we had chosen the second resequentialisation for h_1101, the requirement for p can be satisfied, and we would get the following semi-inverse program:

```
main_101(a,c) =
  h_1101((fun p_101,fun p_111),a,c);

p_101(a,c) = c-a/2;

p_111(a,b,c) = let True() = c=a/2+b in ();
```

```
h_1101((f_101,f_111),a,(c,d)) =
  f_111@(a,f_101@(a,c),d);
```

So, instead of invalidating (main,(1,0,1)), we should retry resequentialisation of (h,(1,1,0,1)) in the hope of getting different uses of its functional parameter.

Note, moreover, that if we define

```
q(b,a) = a/2+b;
```

and make main call h once with p as parameter and once with q as parameter, but at the same division for both calls, then we would need both resequentialisations of (h,(1,1,0,1)) – one for use with the call where p is parameter and the other for use where q is parameter.

In essence, we would want uses of functional parameters to become a part of the division, so a semi-inverse for each set of uses can be made. You can argue that the example is somewhat contrived, and that for "real programs" the choice of resequentialisation is unlikely to affect the uses of functional parameters, so the current behaviour might not really be a problem in practise.

See section 6 for an alternative solution to this problem.

5.2 Functions returning functions as result

The next step is to allow functions to return functions. This complicates the data flow during resequentialisation somewhat, as uses of functional values need to be propagated from the context of a call to its body.

This complex two-way flow of information might indicate a need for a different way of propagating information. For example, it might be useful to do depth-first resequentialisation: When resequentialisation encounters a call, the called function is immediately resequentialised instead of just registered for later resequentialisation. This way, uses of functional parameters can be returned directly. Some kind of fixed-point iteration is still required for recursive calls, though, and contexts of calls that return functions might not be known at the time the call is handled, so iteration is required here also.

Another alternative could be to treat the information flow problem as a type inference for a non-standard type system, where types are annotated with (sets of) divisions.

The same semi-inversed function can be used in several contexts with different uses of its result, so either all these must be combined to a single set of uses, or different semi-inverses must be constructed for each different context, which means that the uses of functional results becomes part of the division. This is similar to the situation described in section 5.1.

6. Related work

The introduction mentions related work on inversion and semi-inversion of first-order functional or imperative languages, so I will not repeat this here.

Andersen [Andersen(2006)] has worked on extending Glück and Kawabe's inversion method [Glück and Kawabe(2004)] to higher-order functions. The basic approach is to first transform the program into first-order form, invert this and then (possibly) transform the inverted program back into higher-order form. Andersen investigates the traditional first-order transformation method [Reynolds(1972)], where a closure is represented as a tag identifying the function paired with a representation of the environment of the closure and where application of a functional value is replaced by a case of the tag and in each branch a first-order call to the function with the representation of the environment as an additional argument. Andersen finds that a flow-analysis that determines which closures are possible at each application is required to invert even a simple application of the map function, and that if there are more than one possible tag in the case analysis (i.e., if there is more than one possible functional argument), inversion requires disjoint co-domains of the corresponding argument functions (so you can determine the function from its result). Hence, Andersen's method can not handle the double application of map shown in section 4.1 (where add1 and double have overlapping co-domains). This limitation is a direct consequence of doing full inversion: Everything that is a parameter in the original program must become a result in the inverted program. So a program that uses map must fully invert the map function. And since map takes a function and a list and produces a list, the inverse of map must take a list and return both a function and a list.

This is not required in semi-inversion, where, as seen in section 4.1, map is semi-inverted even though the program as a whole is fully inverted. The semi-inverse of map is equivalent to map itself, but takes the inverse function of the originally mapped function as argument.

Given the limitations of our extension to functional parameters, transformation to first-order form is relatively simple: You just specialise functions that take functional parameters to all possible values of these parameters, essentially partially evaluating ([Jones et al.(1993)Jones, Gomard, and Sestoft]) the program with functional parameters as static parameters and non-functional parameters as dynamic parameters. This is guaranteed to terminate, as there are only finitely many functions that can be passed as parameters. So we could do this transformation and then semi-invert the result using the first-order semi-inverter to obtain a first-order semi-inverted program.

Using the example from section 3.1, we can transform the higher-order program into the following first-order program:

```
main(xs,ys) = zipWith2sub(xs,ys);

zipWith2sub([],[]) = [];
zipWith2sub(a1:as,b1:bs) =
    sub(a1,b1) : zipWith2sub(bs,as);

sub(x,y) = x-y;
```

which (using the first-order method) semi-inverts to

```
main_101(xs,zs) = zipWith2sub_101(xs,zs);

zipWith2sub_101([],[]) = [];
zipWith2sub_101(a1:as,c1:cs) =
   sub_101(a1,c1) : zipWith2sub_011(as,cs);

zipWith2sub_011([],[]) = [];
zipWith2sub_011(b1:bs,c1:cs) =
   sub_011(b1,c1) : zipWith2sub_101(bs,cs);

sub_101(x,z) = x-z;

sub_011(y,z) = z+y;
```

Doing "firstification" also solves the problem about requirements for functional values mentioned in section 5.1: By specialising the h function twice with p and q as the value of the functional parameter, we get a first-order program that is easily semi-inverted, since the two differently specialised versions of h can be resequentialised differently.

However, in addition to being less academically satisfying and less likely to generalise to full higher-order functions, the firstification approach can lead to very large semi-inverted programs: Transformation to first-order form can in the worst case yield an exponentially larger program.

In the example, specialisation only creates one specialised instance of `zipWith2`, so no blow-up happens. But if we apply firstification to the program in section 4.1, we will get two specialised `map` functions and, hence, two semi-inverses, where the higher-order method produces only one. And if we, for example, have a function that takes N functional parameters and calls itself recursively with a permutation of these parameters, the number of specialised variants will be equal to the period of the permutation, which can be exponential in N.

That said, semi-inversion itself can lead to exponentially larger programs, as the same function can be semi-inverted with exponentially many different divisions. Neither case of exponential blow-up is very likely to occur in practise, though.

7. Conclusion

We have described a method for semi-inversion of functional programs with simple functional parameters, i.e., second-order functions. In spite of the limitations on how functional values are used, they add considerable complexity to the semi-inversion transformation, as information about known and required parameters flows both forwards and backwards in the program, where first-order semi-inversion requires only forwards flow.

The method has been implemented as an extension to the semi-inverter reported in [Mogensen(2007)]. The modifications to the resequentialisation phase were substantial, but the other phases were fairly easy to extend. We have tried the semi-inverter on a number of small, but non-trivial examples.

The lack of a proper way of handling invalidated functional values mentioned in section 5.1 means that it is crucial that we don't add requirements that are not needed, in particular that we don't apply functional values to too few known arguments. To reduce the risk of this, we only select functional value applications when no other relation can be selected, i.e., when as many variables as possible are known. This has worked for the examples tried so far, but the issue needs to be addressed. We will also continue to work on the other issues discussed in section 5.

Allowing only second-order functions limits the number of interesting examples for semi-inversion, so we hope in the future to extend the method to full closures.

References

[Andersen(2006)] Jesper Andersen. A method for inversion of a higher-order programming language. Master's thesis, DIKU, University of Copenhagen, June 2006.

[Dijkstra(1978)] 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.

[Floyd(1967)] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14 (4):636–644, 1967. ISSN 0004-5411.

[Glück and Kawabe(2004)] 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.

[Gries(1981)] David Gries. *The Science of Programming*, chapter 21 Inverting Programs, pages 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[Jones et al.(1993)Jones, Gomard, and Sestoft] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[Knapen(1993)] Ed Knapen. *Relational programming, program inversion and the derivation of parsing algorithms*. Master's thesis, Eindhoven University of Technology, 1993.

[Mogensen(2005)] Torben Æ. Mogensen. Semi-inversion of guarded equations. In *GPCE'05*, Lecture Notes in Computer Science 3676, pages 189–204. Springer-Verlag, 2005.

[Mogensen(2007)] Torben Æ. Mogensen. Report on an implementation of a semi-inverter. In *PSI'06*, Lecture Notes in Computer Science 4378, pages 322–334. Springer-Verlag, 2007.

[Mu et al.(2004)Mu, Hu, and Takeichi] 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.

[Nishida et al.(2001)Nishida, Sakai, and Sakabe] Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. Generation of inverse term rewriting systems for pure treeless functions. In Yoshihito Toyama, editor, *The International Workshop on Rewriting in Proof and Computation*, 2001.

[Nishida et al.(2005)Nishida, Sakai, and Sakabe] Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. Partial inversion of constructor term rewriting systems. In Jürgen Geisl, editor, *Term Rewriting and Applications 2005*, volume 3467 of *LNCS*, pages 264–278. Springer, 2005.

[Reynolds(1972)] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740. ACM, 1972.

[Romanenko(1988)] 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.

[Romanenko(1991)] A. Y. Romanenko. Inversion and metacomputation. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 12–22. New York: ACM, 1991.