Semi-inversion of Guarded Equations

Torben Æ. Mogensen

DIKU, University of Copenhagen, Denmark
torbenm@diku.dk

Abstract. An *inverse* of a program is a program that takes the output of the original program and produces its input. A *semi-inverse* of a program is a program that takes some of the input and some of the output of the original program and produces the remaining input and output. Inversion is, hence, a special case of semi-inversion.

We propose a method for inverting and semi-inverting programs written as guarded equations. The semi-inversion process is divided into four phases: Translation of equations into a relational form, refining operators, determining evaluation order for each equation of the semi-inverted functions and translation of semi-inverted functions back to the original syntax. In cases where the method fails to semi-invert a program, it can suggest which additional parts of the programs input or output are needed to make it work.

1 Introduction

Many programming problems involve making two programs that are each others semi-inverses, for example encryption and decryption of text, where the encryption program takes a clear text and a key and produces a cypher text while the decryption program takes a cypher text and key and produces the clear text. It is usually up to the programmer to ensure that the two programs are, indeed, semi-inverses and to maintain this property when the programs are modified.

Ideally, the programmer should only write one of the two programs and then let the computer derive the other. While this is not realistic for all cases (a public-key encryption function is, for example, deliberately made difficult to semi-invert), it can be useful to have systems that works only some of the time, as long as “some of the time” isn’t “nearly never”.

In general, semi-inversion means taking a program and producing a new program that as input takes part of the input and part of the output of the original program and as output produces the rest of the input and output of original program.

Not all semi-inverses of programs are well-defined: The provided parts of input and output may not be sufficient to uniquely determine the remaining parts. But it is always possible to extend an ill-defined semi-inverse to a well-defined semi-inverse by providing additional parts of the input or output. In the extreme, one can provide all of the input (and possibly some of the output) of the original program as input to the semi-inverse. This makes the output of the semi-inverse uniquely determined by its input, though there might be cases where the semi-inverse is a partial function. For example, it is possible to define a semi-inverse that takes *all* of the input and output

© Springer-Verlag Berlin Heidelberg 2005

of the original program. This semi-inverse will produce an empty output, but is defined only on input/output pairs where the output is what the original program would produce for the given input.

Our approach will ask for extra inputs to semi-inverses when it is unable to find a well-defined semi-inverse. Even semi-inverses that are mathematically well defined may require extra inputs by the method, as it may not be able to find a program that implements the mathematical semi-inverse.

2 Formalism for Semi-inverses

We start by recalling the familiar definition of inverse functions:

A partial function g is the inverse of an injective partial function f if for all x in f's domain we have that $f(x) = y \Rightarrow g(y) = x$. g is only defined on the range of f.

The equation for semi-inverses is not as simple, as we need to talk about parts of inputs and outputs.

Projections [10,9] have been used in theory about partial evaluation to talk about parts of the input. They could also be used here, but we prefer less heavy machinery, so we introduce *extractions*.

Extractions. An *extraction* for a domain α is a total and surjective function from α to a domain β. Intuitively, an extraction finds some information about its input and injects this into an output domain β in such a way that there are no redundancies.

A simple extraction is the function that takes the first component of a pair, but there are also more complex extractions, such as any total predicate (in which case the output domain is the set of boolean values). We will, in practice, use only simple extractions that "throw away" parts of tuples, including the identity function and the function that throws away all input, i.e., whose range is the one-element domain (called "`unit`" in SML and "()" in Haskell).

We call a pair of extractions $p : \alpha \rightarrow \beta$ and $q : \alpha \rightarrow \gamma$ *a division of* α if the function $(p,q) : \alpha \rightarrow \beta \times \gamma$ defined by $(p,q)(x) = (p(x), q(x))$ is bijective. Intuitively, this means that no part of (or information about) x is thrown away by both p and q and no part of (or information about) x is retained by both p and q.

Note that the bijection requirement is on the type of the arguments and results of the function, it is not required that the division is bijective on the set of valid argument/result pairs of the function. Indeed, this would generally not be possible, as the result is functionally dependent on the argument.

The semi-inverse equation. Given a partial function $f : \alpha \rightarrow \beta$ and a division (p,q) of $\alpha \times \beta$, where $p : \alpha \times \beta \rightarrow \gamma_1$ and $q : \alpha \times \beta \rightarrow \gamma_2$, we say that a partial function $g : \gamma_1 \rightarrow \gamma_2$ is a semi-inverse of f with respect to (p,q) if $f(x) = y \Rightarrow g(p(x,y)) = q(x,y)$.

Example: If $f(x,y) = x+y$ and $p((x,y),z) = (x,z)$, $q((x,y),z) = y$ then the function g defined by $g(x,z) = z - x$ is a semi-inverse of f with respect to (p,q), as $x+y = z \Rightarrow y = z - x$.

Not all divisions define valid semi-inverses for a function. For example, if p and q were reversed in the example above, then g would not be well-defined (there would be no unique result).

$$
\begin{array}{lll}
\mathbf{fid} & = & \textit{function identifiers} \\
\mathbf{vid} & = & \textit{variable identifiers} \\
\mathbf{cid} & = & \textit{constructor identifiers} \\
\mathbf{oid} & = & \textit{operator identifiers} \\
\mathbf{pid} & = & \textit{predicate identifiers} \\
\mathbf{num} & = & \textit{numbers} \\
 & & \\
Program & \rightarrow & Function^{+} \\
 & & \\
Function & \rightarrow & Equation^{+} \\
 & & \\
Equation & \rightarrow & \mathbf{fid}\ Pattern \mid Guard\ =\ Exp \\
 & & \\
Pattern & \rightarrow & \mathbf{num} \\
 & \mid & \mathbf{vid} \\
 & \mid & (Pattern, \ldots, Pattern) \\
 & \mid & \mathbf{cid}\ Pattern \\
 & & \\
Guard & \rightarrow & \varepsilon \\
 & \mid & \mathbf{pid}\ Exp \\
 & \mid & Guard\ \&\&\ Guard \\
 & & \\
Exp & \rightarrow & \mathbf{num} \\
 & \mid & \mathbf{vid} \\
 & \mid & (Exp, \ldots, Exp) \\
 & \mid & \mathbf{cid}\ Exp \\
 & \mid & \mathbf{fid}\ Exp \\
 & \mid & \mathbf{oid}\ Exp \\
 & \mid & \texttt{let}\ Pattern = Exp\ \texttt{in}\ Exp
\end{array}
$$

Fig. 1. Syntax

3 Language

The programming language that we use in this paper is a first-order functional language where each function is given as a set of equations using pattern-matching and guards. The equations of a function are unordered and their domains must, hence, be visibly disjoint through their patterns and guards, i.e., for any given input at most one equation will have matching pattern and guard. We allow partial functions, so it is allowed that some inputs have no matching equation. If at any point during execution no matching equation for a call can be found, the result of the entire computation in undefined.

Patterns *can* have repeated variables, which means that the matching values must be equal. A let-expression does *not* introduce a new scope, so if a variable defined in a let-pattern is defined previously in the same equation, the two definitions must define equal values, just like repeated variables in a pattern. This is different from the usual semantics of let-expressions in functional languages, but it is convenient for the inversion method, as repeated uses of a variable in the original program may become repeated definitions of it in a semi-inverted program. Translation to and from traditional first-order functional languages is not difficult.

In an equation, all variables occurring in the guards must also occur in the pattern on the left-hand side of the equation. Variables occurring in the expression must be defined in the pattern on the left-hand side of the equation or in an enclosing let-expression.

Values can be numbers, (possibly empty) tuples of values or elements of recursive datatypes in the style of ML or Haskell. The syntax is reminiscent of Haskell (in spite of the semantic differences) and can be seen in figure 1. Note that the "|" in the rule for *Equation* is part of the syntax of a guarded equation and not a grammar symbol denoting choice of several production. We have not specified the set of operators or predicates, though the latter must at least include an equality test. We also allow partial functions that return the empty tuple as predicates. These are considered to succeed if the call returns a value and fail if the function call is undefined. Such functional predicates may be constructed from "normal" functions during semi-inversion.

The syntax shows constructors, operators and predicates as prefix operators operating on one argument, which may be a tuple. When writing programs in the language, we will, for readability, sometimes use infix notation for these.

Example. A sample program is shown below.

```
i2p (0, 0) | true = nil
i2p (n, i) | n>0 = insert (i2p (n-1,i 'div' n), n, i 'mod' n)

insert (xs, n, 0) | true = n : xs
insert (x:xs, n, i) | i>0 && x/=n = x : insert (xs, n, i-1)
```

The function `i2p` takes two numbers n and i and produces a list of the numbers $1\ldots n$ permuted with the permutation with index i, where $0 \leq i < n!$ (using one of several possible enumerations of permutations).

The guard $\mathtt{x} \neq \mathtt{n}$ in the second equation for `insert` is not strictly required to make the equations disjoint, but it turns out we will need the bit of invariant it represents to make a nontrivial semi-inversion. We will discuss this issue later.

4 Semi-inversion of a Program

A semi-inverted program will consist of a set of semi-inverses of the original functions, possibly having none or several different semi-inverses of some of the original functions using different divisions.

The user specifies which semi-inverses he desires by specifying a function and division for each. These specifications forms the initial set of *desired semi-inverses*.

We may, in the course of transformation, find that we need to call other semi-inverted functions. We will, then, add their specifications to the list of desired semi-inverses and attempt to make definitions of these.

We may, also, find that we are not able to make a definition of a desired semi-inverse, as we can not uniquely define its output in terms of its input. In this case, we add the specification to a list of *invalid semi-inverses* and start over. When we want to add a semi-inverse f' to the list of desired semi-inverses, we must first check the list of invalid semi-inverses. If the specification of f' is found there, we must go back and see if we can use another (not invalidated) semi-inverse instead. If we can not go back (i.e., if f' is in the initial, user specified, top-level list of semi-inverses), semi-inversion has failed, but the information gathered by the method may suggest alternative semi-inverses that may be useful in spite of not being exactly what the user desired.

To summarize, we need to implement the following:

1. A procedure for determining if a specification of a semi-inverse is invalid, given a list of semi-inverses already found to be invalid.
2. A procedure for finding out which other semi-inverses are needed to define a desired semi-inverse.
3. A procedure to construct the definition of a valid semi-inverse.
4. A procedure for suggesting which extra inputs should be added to an invalid semi-inverse in an attempt to extend it to a valid semi-inverse.

Given these, we can keep updating the lists of desired and invalid semi-inverses until all desired semi-inverses are definable in terms of predefined operators and other desired semi-inverses and and none of them are in the list of invalid semi-inverses. We will, eventually, reach such a state as we can, in the worst case, add all of the inputs of the original functions as extra inputs to the semi-inverses, which makes them trivially valid.

Example. We will semi-invert `i2p` from figure 3 with a division (p,q), where $p(n,i,xs) = (n,xs)$ and $q(n,i,xs) = i$, i.e., making a function `i2p'` that takes a number n and permutation xs of the numbers $1 \ldots n$ and returns i, where i is the index of the permutation. We initialize the list of desired semi-inverses to hold one semi-inverse of `i2p` with the division (p,q) as defined above.

4.1 Desequentialisation

The remaining parts of the transformation are simplified if we don't have nested expressions or patterns and if the order of evaluation is less explicit, so we first translate each equation to an unordered set of relations between tuples of variables. We call this process *desequentialisation*.

Figure 2 shows the syntax of the relational form and figure 3 shows the desequentialisation of a single equation. I translates an equation into an equation in the relational form. I_p translates a pattern into a set of relations and I_e translates an expression into a set of relations. I_v is an auxiliary function that is used when a singleton variable is required. If necessary, it adds an extra relation that binds a non-variable to a new variable. The guards are left unchanged by this transformation. Note that constructed syntax is shown in `typewriter font` to distinguish from meta-level syntax used to define the translation function (which is shown as "normal" text). Function calls are translated into calls of relations between input and output variables (using the function name as name for the relation). Guards are already relational, so they are just added to to the set of relations.

Note the similarity between the relations for patterns and structure-building expressions: It is not *a priori* given which side defines the other. The same will be true for some of the other relations: They can be read either as defining the left-side variables

$$\begin{array}{lcl}
\textbf{fid} & = & \textit{function identifiers} \\
\textbf{vid} & = & \textit{variable identifiers} \\
\textbf{cid} & = & \textit{constructor identifiers} \\
\textbf{oid} & = & \textit{operator identifiers} \\
\textbf{pid} & = & \textit{predicate identifiers} \\
\textbf{num} & = & \textit{numbers} \\
 & & \\
Program & \rightarrow & Function^{+} \\
 & & \\
Function & \rightarrow & Equation^{+} \\
 & & \\
Equation & \rightarrow & \textbf{fid}\ Vars\ \texttt{where}\ Relation^{*} \\
 & & \\
Vars & \rightarrow & \textbf{vid} \\
 & | & (\textbf{vid},\ldots,\textbf{vid}) \\
 & & \\
Relation & \rightarrow & \textbf{vid} = \textbf{num} \\
 & | & Vars = Vars \\
 & | & \textbf{vid} = \textbf{cid}\ Vars \\
 & | & \textbf{fid}\ Vars \\
 & | & \textbf{pid}\ Vars \\
 & | & Vars = \textbf{oid}\ Vars
\end{array}$$

Fig. 2. Syntax of relational form

in terms of the right-side variables or *vice versa*. They can even be read "sideways", determining any subset of the variables in terms of the rest (like semi-inverses). This undirected view of the relations will be the essence of the semi-inversion method.

Example. The example program from section 3 is shown below in relational form.

```
i2p (x1,x2,x3) where
  {x1 = 0, x2 = 0, x3 = nil}
i2p (n,i,x1) where
  {n>0, insert (x2,n,x3,x1), i2p (x4,x5,x2), x4 = sub1 n,
   x5 = div (i,n), x3 = mod (i,n)}

insert (xs,n,x1,x2) where
  {x1 = 0, x2 = : (n,xs)}
insert (x1,n,i,x2) where
  {x1 = : (x,xs), x2 = : (x,x3), i>0, x/=n, insert (xs,n,x4,x3), x4 = sub1 i}
```

Note that `n-1` has been translated as `sub1 n` instead of using a binary subtraction operator. While this is not strictly necessary, it will make the example a bit less cumbersome.

4.2 Refining Operators

In order to make semi-inversion of operators possible more often, we can refine the operators in different ways:

$$\begin{array}{ll}
f & \in \textbf{fid} = \textit{function identifiers} \\
x, y, x_i & \in \textbf{vid} = \textit{variable identifiers} \\
c & \in \textbf{cid} = \textit{constructor identifiers} \\
o & \in \textbf{oid} = \textit{operator identifiers} \\
k & \in \textbf{num} = \textit{numbers}
\end{array}$$

$$\begin{array}{ll}
I & : \textit{Equation} \rightarrow \textit{Equation}' \\
I[f\ P \mid G\ =\ E] & = f\ (V_i \cup V_o)\ \texttt{where}\ (R_i \cup R_o \cup R_g) \\
 & \quad \text{where } (R_i, V_i) = I_p[P],\ \ (R_o, V_o) = I_e[E],\ \ R_g = I_g[G]
\end{array}$$

$$\begin{array}{ll}
I_p & : \textit{Pattern} \rightarrow \textit{RelationSet} \times \textit{Vars} \\
I_p[(P_1, \ldots, P_n)] & = (R_1 \cup \cdots \cup R_n,\ (x_1, \ldots, x_n)) \\
 & \quad \text{where } (R_1, x_1) = I_v(I_p[P_1]),\ \cdots,\ (R_n, x_n) = I_v(I_p[P_n]) \\
I_p[k] & = (\{x = k\}, x) \quad \text{where } x \text{ is a new variable} \\
I_p[y] & = (\{\}, y) \\
I_p[c\ P] & = (R \cup \{x = c\ V\}, x) \\
 & \quad \text{where } (R, V) = I_p[P] \text{ and } x \text{ is a new variable}
\end{array}$$

$$\begin{array}{ll}
I_g & : \textit{Guard}^* \rightarrow \textit{RelationSet} \\
I_g[\varepsilon] & = \{\} \\
I_g[p\ E] & = R \cup \{p\ V\} \\
 & \quad \text{where } (R, V) = I_e[E] \text{ and } x \text{ is a new variable} \\
I_g[G_1\ \&\&\ G_2] & = I_g[G_1] \cup I_g[G_2]
\end{array}$$

$$\begin{array}{ll}
I_e & : \textit{Exp} \rightarrow \textit{RelationSet} \times \textit{Vars} \\
I_e[k] & = (\{x = k\}, x) \quad \text{where } x \text{ is a new variable} \\
I_e[y] & = (\{\}, y) \\
I_e[(E_1, \ldots, E_n)] & = (R_1 \cup \cdots \cup R_n,\ (x_1, \ldots, x_n)) \\
 & \quad \text{where } (R_1, x_1) = I_v(I_e[E_1]),\ \cdots,\ (R_n, x_n) = I_v(I_e[E_n]) \\
I_e[c\ E] & = (R \cup \{x = c\ V\}, x) \\
 & \quad \text{where } (R, V) = I_e[E] \text{ and } x \text{ is a new variable} \\
I_e[f\ E] & = (R \cup \{f\ (V \cup x)\}, x) \\
 & \quad \text{where } (R, V) = I_e[E] \text{ and } x \text{ is a new variable} \\
I_e[o\ E] & = (R \cup \{x = o\ V\}, x) \\
 & \quad \text{where } (R, V) = I_e[E] \text{ and } x \text{ is a new variable} \\
I_e[\texttt{let}\ P = E_1\ \texttt{in}\ E_2] & = (R_0 \cup R_1 \cup R_2 \cup \{V_0 = V_1\}, V_2) \\
 & \quad \text{where } (R_0, V_0) = I_p[P],\ \ (R_1, V_1) = I_e[E_1],\ \ (R_2, V_2) = I_e[E_2]
\end{array}$$

$$\begin{array}{ll}
I_v & : \textit{RelationSet} \times \textit{Vars} \rightarrow \textit{RelationSet} \times \textbf{vid} \\
I_v[(R, V)] & = (R, V) \quad \text{if } V \in \textbf{vid} \\
I_v[(R, V)] & = (R \cup \{x = V\}, x) \quad \text{if } V \notin \textbf{vid} \\
 & \quad \text{where } x \text{ is a new variable}
\end{array}$$

Fig. 3. Translation to relational form

- The guard of an equation may restrict the range of inputs to an operator enough that it makes semi-inversion possible more often. For example, if the guard ensures x is even, we can refine the relation $z = x\ \texttt{div}\ 2$ to $x = 2 * y$, where any of the two variables can define the other.
- Related operators with the same inputs may be combined into one operator that is easier to semi-invert. For example, $z = \texttt{div}\ (x, y)$ and $v = \texttt{mod}\ (x, y)$ can be com-

bined into one relation $(z,v) = \texttt{divmod}\ (x,y)$, using an operator `divmod`, that returns both fraction and remainder after division by y. Given this, we can from z, v and y find x, which can't be found from any one of the two relations individually even when all of z, v and y are known.

- Relations of the form $(x_1,\ldots,x_n) = (y_1,\ldots,y_n)$ are translated into separate relations $x_1 = y_1,\ldots,x_n = y_n$.

We may combine several of these approaches, for example by joining two related operators and then use the guard to restrict the range.

Example. The operators used in the program are `sub1`, `div` and `mod`. `sub1` requires no refinement as any of the two variables defines the other. `div` and `mod`, on the other hand, can be combined as they both take the same arguments. This means we replace the relations `x5 = div (i,n), x3 = mod (i,n)` with `(x5,x3) = divmod (i,n)`.

4.3 Resequentialisation

When semi-inverting an relational equation $f\ V_f$ `where` R with respect to a division (p,q), where p and q are functions from the tuple V_f to the input and output tuples of the semi-inverted equation, we need to check if a thusly defined semi-inverse is valid, i.e., if the value of $q(V_f)$ is uniquely defined from the value of $p(V_f)$ through the relations R. Hence, we need to determine which variables can be computed from which others, starting from $p(V_f)$. We do this by *resequentialisation*, which orders relations by data dependency. We keep a list of known variables K and a list S of relations from R that depend only on the variables in K. At the end, the relations in S will be in a valid evaluation order.

```
$K = p(V_f)$  (treated as a set of variables)
$S = \varepsilon$
while there is a relation $r$ in $R$ that is determined by $K$
   $S := S, r;$
   $R := R \setminus \{r\}$
   $K := K \cup Vars(r)$
```

A relation r is determined by K if all variables in the relation can be defined through variables that are already contained in K. For structural relations (tuples and constructors), this means that all variables on any one side of $=$ must be defined. Predicates normally need all variables in order to be defined, but equality predicates can be resolved if any one side is defined.

Arithmetic operators are treated as relations between input and output. For example $z = x + y$ is a relation between three variables.

Semi-inverting an operator is possible when all variables can be uniquely determined from those that are known. For the $z = x + y$ example, this is true when any two of the variables are known. For all standard operators, we list the subsets of inputs and outputs that can make the remaining defined, each with the name of the corresponding semi-inverse operator. For example, the operator $+$ will be described by the list

$[z = +(x,y),\ x = -(z,y),\ y = -(z,x)]$. What this means is that the first relation in the list can be replaced by any of the other relations.

For function calls, we don't know *a priori* which semi-inverses of the function are valid. We will, hence, assume that any non-empty subset of the variables in the relation is enough to define the remaining variables, *unless* this subset corresponds to a semi-inverse in the list of invalid semi-inverses. If we later need to invalidate the semi-inverse, we must redo the resequentialisation.

If, at the end, K contains all variables in the relation set (or even just in $q(V_f)$), we know that we can evaluate $q(V_f)$ from $p(V_f)$, so we can semi-invert the current equation for f with respect to (p,q), assuming the called semi-inverses are all valid. We add the called semi-inverses to the list of desired semi-inverses, so we will check their validity later.

If, on the other hand, at the end of resequentialisation there are variables from $q(V_f)$ that are not contained in K, the semi-inverse is not valid. So we add it to the list of invalid semi-inverses and start over.

Example. We recall that we want to semi-invert `i2p` with the first argument and the result known. In the relational form in section 4.1, this means the first and last parameter.

Starting with the first equation for `i2p`, we initialise K to $\{$`x1, x3`$\}$, so we can immediately add `x1 = 0` and `x3 = nil` to S. As one side of the equality `x2 = 0` is known, we can add this too and add `x2` to K, which concludes resequentialisation of this equation with S equal to

```
x1 = 0, x3 = nil, x2 = 0
```

The second equation starts with $K = \{$`n, x1`$\}$. We can immediately add `n>0` to S. The list of valid semi-inverses of the `sub1` operator is `[y = sub1 x, x = add1 y]`, so we can add `x4 = sub1 n` to S and `x4` to K. At this point, we can only add semi-inverses of functions to the list. We choose `insert`, where we know two of the four parameters. Hence, we add `insert` with known second and last parameter to the list of desired semi-inverses, add `insert (x2,n,x3,x1)` to S and `x2, x3` to K. We can then add `i2p (x4,x5,x2)` to S and `x5` to K. We already have this semi-inverse in the list of desired semi-inverses, so we continue.

We replaced `x5 = div (i,n)` and `x3 = mod (i,n)` by `(x5,x3) =divmod(i,n)`, where the semi-inverses of `divmod` are described by `[(z,v) = divmod (x,y), x = MLA (z,y,v)]` with `MLA (z,y,v)` meaning `z*y+v`, provided `0`$\leq$`v<y` (and undefined otherwise). Hence, we can add `(x5,x3) = divmod(i,n)` to S and i to K. This concludes the resequentialisation of the second equation for `i2p` with S equal to

```
n>0, x4 = sub1 n, insert (x2,n,x3,x1), i2p (x4,x5,x2), (x5,x3) = divmod(i,n)
```

We added a semi-inverse of `insert` with second and fourth parameters known to the list of desired semi-inverses, so we need to resequentialise the equations for that. Skipping details, we get the sequences

```
x2 = : (n,xs), x1 = 0
x2 = : (x,x3), x/=n, insert (xs,n,x4,x3), x1 = : (x,xs), x4 = sub1 i, i>0
```

for the two equations. We have now resequentialised all desired semi-inverses.

4.4 Translation Back into Guarded Equation Form

When we have resequentialised all desired semi-inverses, we translate each equation from relational form back into "normal" syntax.

Resequentialisation gives us a list of relations that defines a possible order of evaluation. Together with a division, we wish to translate this into an equation in the original syntax, *i.e.*, into something of the form $f'\ P \mid G' = E$ where P is a pattern for $p(V_f)$, E is an expression that defines the value of $q(V_f)$ in terms of the variables in P, and the guard G' is a guard using these same variables.

We start with the patterns. P will be created from $p(V_f)$ by finding a relation where a variable in $p(V_f)$ is related to a structure (tuple or constructor application), substituting this structure for the variable in the pattern, and then repeating this process for the variables in the substructure.

$$
\begin{array}{ll}
x, y, x_i & \in \textbf{vid} = \textit{variable identifiers} \\
c & \in \textbf{cid} = \textit{constructor identifiers} \\
 & \\
M_p & : \textit{Relation}^* \rightarrow \textit{Vars} \rightarrow \textit{Pattern} \times \textit{Relation}^* \\
M_p[S](x_1,\ldots,x_n) & = ((p_1,\ldots,p_n), S_n) \\
 & \quad \text{where } (p_1, S_1) = M_p[S]x_1, \ldots, (p_n, S_n) = M_p[S_{n-1}]x_n) \\
M_p[\{x = k\}, S]x & = (k, S) \\
M_p[\{x = (x_1,\ldots,x_n)\}, S]x & = M_p[S](x_1,\ldots,x_n) \\
M_p[\{x = c\ x_1\}, S]x & = (c\ (p_1), S_1)) \\
 & \quad \text{where } (p_1, S_1) = M_p[S]x_1) \\
M_p[\{x = y\}, S]x & = M_p[S]y \\
M_p[r, S]x & = (p_1, (r, S_1)) \\
 & \quad \text{where } (p_1, S_1) = M_p[S]x \\
M_p[\varepsilon]x & = (x, \varepsilon)
\end{array}
$$

Fig. 4. From relations to pattern

Figure 4 shows how a pattern and a reduced list of relations is made from a variable or tuple of variables and a relation list. The rules are applied in precedence from top to bottom. If no other rule applies, the last is used, so the variable itself is used as pattern. Note that this procedure may result in a variable occurring several times in a pattern, but this is O.K., as we allow nonlinear patterns.

Relations in S that only contain variables from the pattern can be considered as tests and will be part of the new guard, so we remove such relations from the set and build an expression from the remaining relations. Figure 5 shows how the new guard and the reduced list of relations are constructed by M_g and $M_{\overline{g}}$, respectively, from the relation list S and the set V of variables in the pattern p returned by M_p.

Figure 6 shows how we build an expression given a list of relations R and a tuple of desired variables V_{out} (where $V_{out} = q(V_f)$). To choose the correct semi-inverses for operators and function calls, we maintain a list of known variables. This is initialised to be the set of variables contained in the pattern returned by M_p.

All intermediate results are bound in let-expressions, but (for readability) some of these can later be unfolded, e.g., when the bound variable is used only once.

$$\begin{array}{lll}
M_g, M_{\overline{g}} & : \mathit{Relation}^* \to \mathit{Vars} \to \mathit{Relation}^* & \\
M_g[\varepsilon]V & = \mathit{true} & \\
M_g[r, S]V & = r \;\&\&\; M_g[S]V & \text{if the variables in } r \text{ are contained in } V \\
M_g[r, S]V & = M_g[S]V & \text{if the variables in } r \text{ are not all contained in } V \\
 & & \\
M_{\overline{g}}[\varepsilon]V & = \varepsilon & \\
M_{\overline{g}}[r, S]V & = r, M_{\overline{g}}[S]V & \text{if the variables in } r \text{ are not all contained in } V \\
M_{\overline{g}}[r, S]V & = M_{\overline{g}}[S]V & \text{if the variables in } r \text{ are contained in } V
\end{array}$$

Fig. 5. From relations to guard and relations

Combining all of the above, we get that if we have a function `f`, a list of relations S, a set of input variables V_{in} and a set of output variables V_{out}, we construct an equation `f` $p \mid g = e$, where

$$\begin{array}{ll}
(p, S_1) & = M_p[S]V_{in} \\
V & = \text{the set of variables in } p \\
g & = M_g[S_1]V \\
S_2 & = M_{\overline{g}}[S_1]V \\
e & = M_e[S_2]V
\end{array}$$

Example. If we apply the translation function to the sequences found in section 4.3, we get the equations shown below.

```
i2p' (0,nil) | true = 0
i2p' (n,x1) | n>0 =
  let x4 = sub1 n in
    let (x2,x3) = insert' (n,x1) in
      let x5 = i2p' (x4,x2) in
         let i = MLA (x5,n,x3) in i

insert' (n,n:xs) | true = (xs, 0)
insert' (n,x:x3) | x/=n =
  let (xs,x4) = insert' (n,x3) in
    let x1 = (x:xs) in
      let i = add1 x4 in
        let true = i>0 in (x1, i)
```

We can unfold some of the let-expressions, replace prefix operators with infix operators and eliminate the redundant test to get a more readable result as shown here:

```
i2p' (0,nil) | true = 0
i2p' (n,x1) | n>0 =
    let (x2,x3) = insert' (n,x1) in i2p' (n-1,x2) * n + x3
```

$$
\begin{array}{ll}
x, y, x_i & \in \mathbf{vid} = \textit{variable identifiers} \\
c & \in \mathbf{cid} = \textit{constructor identifiers} \\
f & \in \mathbf{fid} = \textit{function identifiers} \\
o & \in \mathbf{oid} = \textit{operator identifiers} \\
o & \in \mathbf{pid} = \textit{predicate identifiers} \\
\\
M_e & : \textit{Relation}^* \rightarrow \mathbf{vid}^* \rightarrow \textit{Exp} \\
M_e[S]V & = V_{out} \quad \text{if } V_{out} \subseteq V \\
M_e[x = k, S]V & = \mathtt{let}\ x = k\ \mathtt{in}\ M_e[S](\{x\} \cup V) \\
M_e[x = y, S]V & = \mathtt{let}\ x = y\ \mathtt{in}\ M_e[S](\{x\} \cup V) \quad \text{if } y \in V \\
M_e[x = y, S]V & = \mathtt{let}\ y = x\ \mathtt{in}\ M_e[S](\{y\} \cup V) \quad \text{if } x \in V \\
M_e[x = (y_1, \ldots, y_n), S]V & = \mathtt{let}\ x = (y_1, \ldots, y_n)\ \mathtt{in}\ M_e[S](\{x\} \cup V) \\
 & \qquad \text{if } \{y_1, \ldots, y_n\} \subseteq V \\
M_e[x = (y_1, \ldots, y_n), S]V & = \mathtt{let}\ (y_1, \ldots, y_n) = x\ \mathtt{in}\ M_e[S](\{y_1, \ldots, y_n\} \cup V) \\
 & \qquad \text{if } x \in V \\
M_e[x = c\,Y, S]V & = \mathtt{let}\ x = c\,Y\ \mathtt{in}\ M_e[S](\{x\} \cup V) \quad \text{if } Y \subseteq V \\
M_e[x = c\,Y, S]V & = \mathtt{let}\ c\,Y = x\ \mathtt{in}\ M_e[S](Y \cup V) \quad \text{if } x \in V \\
M_e[Y_2 = o\,Y_1, S]V & = \mathtt{let}\ Z_2 = o'\,Z_1\ \mathtt{in}\ M_e[S](Z_2 \cup V) \\
 & \qquad \text{where } Z_1 \subseteq (Y_1 \cup Y_2) \cap V,\ Z_2 = (Y_1 \cup Y_2) \setminus Z_1 \\
 & \qquad \text{and } o' \text{ is a semi-inverse of } o \text{ with inputs corresponding to } Z_1 \\
M_e[f\,Y, S]V & = \mathtt{let}\ Z_2 = f'\,Z_1\ \mathtt{in}\ M_e[S](Z_2 \cup V) \\
 & \qquad \text{where } Z_1 \subseteq Y \cap V,\ Z_2 = Y \setminus Z_1 \\
 & \qquad \text{and } f' \text{ is a valid semi-inverse of } f \text{ with inputs corresponding to } Z_1 \\
M_e[p\,(y_1, \ldots, y_n), S]V & = \mathtt{let\ true} = p\,(y_1, \ldots, y_n)\ \mathtt{in}\ M_e[S]V
\end{array}
$$

Fig. 6. From relations to expression

```
insert' (n,n:xs) | true = (xs, 0)
insert' (n,x:x3) | x/=n =
  let (xs,x4) = insert' (n,x3) in (x:xs, x4+1)
```

4.5 Joining Equations

The language demands that the equations of a function must have disjoint domains through their patterns and guards. There is no guarantee that this will be true of the semi-inverted equations, even if it was true for the original equations.

If two or more equations of a semi-inverted function have overlapping domains, we can do several things:

- See if we can refine the guards by considering the domains of operators: If an operator is applied to variables in the pattern and the operation is not defined on all possible values, we can add a guard that restrict the variables to the defined domain of the operator.
- If a predicate is part of the expression, it and all required let-bindings of variables used in the predicate can be copied into the guard.
- Invalidate the semi-inverse.

The first of these options is preferable, but it only rarely works.

The second option may cause duplicated work, so we will use this only if the duplicated code doesn't involve function calls.

Invalidating the semi-inverse is simple (we already have a mechanism for that) and can always be done, but requires that we rerun part of the transformation, see below, so we use it as a last resort.

Note that the presence of overlapping domains may be undecidable if the guards are nontrivial. Hence, we may sometimes reject equations where there is no overlap because we are unable to see this. Alternatively, we can restrict guards to a form that makes disjointedness decidable. This would require the extraction of guards from relations to only extract guards in this form.

4.6 Adding Extra Arguments to Make Semi-inverses Valid

In the simplest case, the user specifies a number of desired semi-inverses, we desequentialise, resequentialise and find that the specified semi-inverses are valid, possibly in the process adding a few more required semi-inverses to the set and validating these. We then translate back, and if the domains of the equations are disjoint, we are done.

If, however, resequentialisation or joining of equations find that a desired semi-invariant is not valid, we add it to the list of invalid semi-inverses. We then rerun resequentialisation for all semi-inverses in the desired list and do the subsequent back-translation and joining of equations, repeating all of this as needed. If we find that we are still unable to define the top-level user-specified semi-inverses, we need to have extra inputs added to these in order to make them valid.

In the extreme, we can move all remaining parts of the input of the original function from the output of the semi-inverse to its input, in which case the semi-inverse will surely be valid: All intermediate variables can be computed from the original input, and the patterns and guards used to distinguish the original equations will also distinguish the equations of the semi-inverse.

But we will usually want to add as little of the original input as possible as extra input to the semi-inverse. Fortunately, when we discover that a semi-inverse in invalid, we will often have information that is useful in deciding which part of the input to add:

– If resequentialisation fails to sequentialise all relations in the relation set, we can look at the variables not in K. If one of these represents a part of the original input and is enough to make other variables defined (possibly through an as yet not validated semi-inverse function), this is an obvious choice for additional input to the semi-inverse.
– If we find that the equations for a semi-inverse do not have disjoint domains, a part of the input that would make them disjoint is an obvious additional parameter. This may be a variable that is used in the guards of the original equations or a variable that correspond to a part of the input where patterns made the original equations disjoint.

Example. The equations of `i2p'` are disjoint as `n` is 0 in the first equation and required to be greater than 0 in the second. The equations of `insert'` are disjoint as the first element of the list is equal to n in the first equation and required to be different from n in the second. Note that this exploits the "unneeded" guard x≠n from the original definition of insert. The guard represents part of a non-trivial invariant of i2p: The permuted lists do not have repeated elements.

The relations between the original and semi-inverted functions are:

$$\begin{array}{ll}\texttt{i2p}(n,i) = p & \Leftrightarrow \texttt{i2p'}(n,p) = i \\ \texttt{insert}(xs,n,i) = ys & \Leftrightarrow \texttt{insert'}(n,ys) = (xs,i)\end{array}$$

5 Conclusion

The main new contribution of this paper is working on semi-inverses, where most previous work has focused on true inverses. This is enabled by a number of smaller contributions: Using a relational intermediate form with no *a priori* order of evaluation and a resequentialisation analysis to discover a valid order of evaluation for a semi-inverse and determining if one such exist. Additionally, the method, when it fails, can provide guidance to the user for finding extra information that can lead to useful semi-inverses as alternatives to those initially specified.

Our definition of extractions is also, we believe, new. It is used similarly to the way *projections* [10,9] have been used for describing incomplete input in partial evaluation, but it is a better fit to strict languages as these don't normally work on the partially undefined values that projections yield.

Due to space constraints, the example did not show backtracking on invalidated semi-inverses. If true inversion of the program from figure 3 is attempted, the method will find that the desired semi-inverse of insert is not valid (due to overlapping equations), so backtracking is made to the top level, where extra inputs are requested. Adding n as the extra input gives the semi-inverse shown above.

The example required an invariant of the input to the semi-inverse to be specified in order to work. This invariant was a property of the output of the original program, but became a property of the input to the semi-inverse. As such it is to be expected that tests are required in the semi-inverse to verify that the input actually has this property.

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. In the example, the invariant concerns one of the original function parameters, so adding a guard to the original program was easy. If the invariant concerns variables that do not occur in a pattern, it may be necessary to add a "partial identity function", i.e., a function that returns its input as result but is only defined on values that obey the invariant. Turchin [14] call such partial identity functions "filters". Adding such redundant guards or filters is conceptually similar to using *binding time improvements* [6,3,1] to improve the result of partial evaluation.

5.1 Related Work

Prolog [13] and similar languages have long had the ability to run programs backward or partly backward, so each program is its own inverse and semi-inverse. Prolog relies on backtracking and may fail to terminate when not run with sufficient variables instantiated. We avoid both backtracking and nontermination by requiring extra inputs to be added when there isn't sufficient information to uniquely choose an equation, but in doing so may fail to produce a semi-inverse program in some cases where Prolog is able to find solutions with a similar subset of inputs specified. The relational form used as an intermediate step in our transformation also has a resemblance to Prolog. Inversion of relational programs is investigated in [8] and [11], but the relational form is quite different, as the programs are variable-free many-to-many relations constructed from relational combinators, which makes inversion relatively simple.

Other work on inverting programs [4,2,5] also avoid backtracking by requiring deterministic choice in the inverted programs. These methods can, however, only make true inverses and simply give up if they can't find a non-backtracking inverse program. In the main, these methods work by direct inversion of all data and control flow, so they are not easily extended to semi-inversion, where data and control flow is partly forwards and partly backwards.

We have been able to find one work, [12], that mentions the possibility of transforming a program to a semi-inverse (there called a partial inverse) and shows an example of semi-inverting multiplication on unary numbers to division. The method can introduce backtracking (and does so in the example). Our method can semi-inverse multiplication of unary numbers to non-backtracking divison.[1]

5.2 Future Work

A prototype implementation of an early version of the semi-inversion method presented here has been implemented as a student project. It is able to do the example semi-inversion shown in this paper, but lacks backtracking on invalid semi-inverts.

Some of the transformation steps used in this paper, such as the order in which variables are defined in the semi-inverted programs, are under-specified. Good heuristics for these steps need to be found.

The method for obtaining disjoint equations after semi-inversion is fairly crude, and it is plausible that the more advanced methods used in [4] could be applied. However, semi-inversion can often do with less powerful methods than complete inversion, partly because some of the original inputs may be retained so guards of the original program can be reused and partly because one might be able to find another semi-inverse to use instead of the one that failed, an option not available for full inversion.

Larger, more realistic, examples need to be examined, such as deriving a decryption function from an encryption function as mentioned in the introduction. We can not expect the method to be amenable to encryption methods based on prime numbers, as their invertibility often rely on nontrivial number theoretic properties, but it is possible that the method can work with cyphers that work by manipulating bitstrings. This needs to be determined, however.

[1] This requires constraining one argument of the multiplier to be non-zero, as division by a possibly zero value will cause overlapping equations in the semi-inverse.

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. 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.
3. 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.
4. 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.
5. David Gries. *The Science of Programming*, chapter 21 Inverting Programs, pages 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
6. Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In [7], pages 83–100, 1991.
7. Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990. Workshops in Computing*. Springer-Verlag, August 1991.
8. Ed Knapen. *Relational programming, program inversion and the derivation of parsing algorithms*. Master's thesis, Eindhoven University of Technology, 1993.
9. J. Launchbury. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.
10. John Launchbury. Projections for specialisation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.
11. 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.
12. 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.
13. Leon Sterling and Ehud Shapiro. *The Art of Prolog. Second Edition*. MIT Press, Cambridge, Massachusetts, 1994.
14. Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, LNCS 85, pages 645–657. Springer-Verlag, 1980.