A Program Inverter for a Functional Language with Equality and Constructors

Robert Glück[1]* and Masahiko Kawabe[2]

[1] PRESTO, JST & Institute for Software Production Technology
Waseda University, School of Science and Engineering
Tokyo 169-8555, Japan, glueck@acm.org

[2] Waseda University, Graduate School of Science and Engineering
Tokyo 169-8555, Japan,
kawabe@futamura.info.waseda.ac.jp

Abstract. We present a method for automatic program inversion in a first-order functional programming language. We formalize the transformation and illustrate it with several examples including the automatic derivation of a program for run-length decoding from a program for run-length encoding. This derivation is not possible with other automatic program inversion methods. One of our key observations is that the duplication of values and testing of their equality are two sides of the same coin in program inversion. This leads us to the design of a new self-inverse primitive function that considerably simplifies the automatic inversion of programs.

1 Introduction

Many problems in computation can be specified in terms of computing the inverse of an easily constructed function [3]. We regard program inversion, beside program specialization and program composition, as one of the three fundamental operations of metacomputation [10]. The idea of program inversion can be traced back to reference [7]. Recent work [16] has focused on the converse of a function theorem [3], inverse computation of functional programs [2], and the transformation of interpreters into inverse interpreters by partial evaluation [9]. Despite the fundamental importance of program inversion as tool for program transformation, relatively few papers have been devoted to this topic.

Program inversion has been studied in the context of Undo facilities for editors, transaction systems, and so on (*e.g.*, [4, 5, 15]). These facilities usually rely on cumulative methods that record parts of the forward computation. The size of such a trace depends on the number of computation steps (examples are editors, debuggers, image processors). In contrast to these methods, we do not require several forward computation steps before reversing the last computation steps. We are interested in generating 'stand-alone' inverse programs from a given program. Examples are programs for encoding and decoding data when files are sent

* On leave from DIKU, Department of Computer Science, University of Copenhagen.

via networks. We consider one-to-one functions and not relations with multiple solutions. Logic programming is suited to find multiple solutions and can be regarded as inverse interpretation, but we are interested in program inversion. Our goal is to generate inverse programs. For a more detailed description of these notions, see reference [1].

The first method developed for automatic program inversion of first-order functional programs appears to be the program inverter by Korf and Eppstein [14, 8] (we call it KEinv for short). It is one of only two general-purpose automatic program inverters that have been built (the other one is InvX [13]). The key feature is the inversion of multiple functions with multiple parameters which together form an injective system of functions. KEinv uses postcondition inference heuristics as the basis for the transformation. Global inversion is based on the local invertibility of program constructs.

The contribution of this paper is an *automatic method for program inversion* of first-order function programs with constructors and equality test. One of our key observations is that the *duplication of values* and *testing of equality* are two sides of the same coin in program inversion. Based on this observation, we designed a self-inverse primitive function that considerably simplifies the inversion of programs.

Another important ingredient is the representation of a program as a logical formula in a disjunctive normal form representing atomic operations in the source program and dissolving the nesting of case-expressions. Inversion is then performed on this representation and reduces to a straightforward backward reading of the atomic formulas. As a result our method can automatically invert programs that are beyond the capability of KEinv, such as the automatic generation of a program for run-length decoding from a program for run-length encoding. However, KEinv also includes numbers and arithmetic; our capabilities in that area are less. They also use postcondition heuristics, which we have not added to our system.

These insights are an important step towards our research goals because, beside InvX, KEinv appears to be the only existing general-purpose automatic program inverter. Manual methods [7, 12, 3, 16] and semi-automatic methods [6] exist, but require ingenuity and human insight. Our goal is to achieve further automation of general-purpose program inversion.

This paper is organized as follows. First, we introduce the inversion of functions (Sect. 2) and define the source language (Sect. 3). Then we discuss our solution of the main challenges of program inversion (Sect. 4) and present our inversion method (Sect. 5). We discuss limitations and extensions (Sect. 6) and related work (Sect. 7), and then give a conclusion (Sect. 8).

2 Program Inversion

The goal of program inversion is to find an *inverse program* $q^{-1} : B \to A$ of a program $q : A \to B$ such that for all values $x \in A$ and $y \in B$ we have

$$q(x) = y \iff q^{-1}(y) = x \ . \tag{1}$$

Here, equality means strong equivalence (either both sides of an equation are defined and equal, or both sides are undefined). Observe that the definition of an inverse program does not state the properties of the inverse in isolation. The definition tells us that, if a program q terminates on input x and returns output y, then the inverse program q^{-1} terminates on y and returns x, and vice versa. This implies that both programs are *injective* (they need not be surjective or total). The definition is symmetric with respect to q and q^{-1}, which means that it is arbitrary which of the two programs we call the 'source program' and which we call the 'inverse program'. The equation does not assert properties for programs q and q^{-1} outside the domains A and B.

A program *inv* is a *program inverter* if, for all injective programs q, the result of applying *inv* to q yields an inverse program q^{-1} of q: $inv(q) = q^{-1}$. In practice, even when it is certain that an efficient inverse program q^{-1} exists, the automatic derivation of such a procedure from q by a program inverter may be difficult or impossible.[3] This paper presents a method for automatic program inversion that can automatically convert a large class of injective programs into their inverse. For instance, we will see that we can automatically obtain a program for run-length decoding from a program for run-length encoding.

3 Source Language

We are concerned with a first-order functional language. A program q is a sequence of function definitions d where the body of each definition is a term t constructed from variables, constructors, function calls and case-expressions (Fig. 1 where $m > 0$, $n \geq 0$). For simplicity, we assume that all functions are unary. The language has a call-by-value semantics.

An example is the program for run-length encoding (Fig. 3): function *pack* encodes a list of symbols as a list of symbol-number pairs, where the number specifies how many copies of a symbol have to be generated upon decoding. For instance, $pack([\text{AABCCC}]) = [\langle \text{A}\ 2\rangle\langle \text{B}\ 1\rangle\langle \text{C}\ 3\rangle]$.[4] Function *pack* maximizes the counter in each symbol-number pair: we never have an encoding like $\langle \text{C}\ 2\rangle\langle \text{C}\ 1\rangle$, but rather, always $\langle \text{C}\ 3\rangle$. This implies that the symbols in two adjacent symbol-number pairs are never equal. Fig. 4 shows the inverse function $pack^{-1}$. In the implementation, we use unary numbers. The primitive function $\lfloor\cdot\rfloor$ checks the equality of two values: $\lfloor\langle v\ v'\rangle\rfloor = \langle v\rangle$ if $v = v'$. In the absence of equality, the values are returned unchanged: $\lfloor\langle v\ v'\rangle\rfloor = \langle v\ v'\rangle$ if $v \neq v'$. This will be defined below. We assume that $\lfloor\cdot\rfloor \in$ Functions, but write $\lfloor ...\rfloor$ instead of $\lfloor\ \rfloor(...)$.

We consider only *well-formed* programs. We require that each variable occurring in a term be defined. As usual, we require that no two patterns p_i and p_j in a case-expression contain the same constructor (patterns are orthogonal) and

[3] There exists a program inverter that returns, for every q, a *trivial inverse* q^{-1} [1].

[4] We use the shorthand notation $x : xs$ and $[\ ]$ for the constructors Cons$(x\ xs)$ and Nil. For $x_1 : x_2 : \ldots : x_n : [\ ]$ we write $[x_1 x_2 \ldots x_n]$, or sometimes $x_1 x_2 \ldots x_n$. A tuple $\langle x_1 \ldots x_n\rangle$ is a shorthand notation for an n-ary constructor $\text{C}_n(x_1 \ldots x_n)$.

Grammar

$$
\begin{array}{rll}
q & ::= d_1 \dots d_m & \text{(program)} \\
d & ::= f(x) \triangleq t & \text{(definition)} \\
t & ::= x & \text{(variable)} \\
 & \mid\ c(t_1 \dots t_n) & \text{(constructor)} \\
 & \mid\ f(t) & \text{(function call)} \\
 & \mid\ \textbf{case}\ t\ \textbf{of}\ p_1 \rightarrow t_1 \quad \dots \quad p_m \rightarrow t_m & \text{(case-expression)} \\
p & ::= c(x_1 \dots x_n) & \text{(flat pattern)}
\end{array}
$$

Syntax domains

$q \in$ Programs	$t \in$ Terms	$x \in$ Variables	$f \in$ Functions
$d \in$ Definitions	$p \in$ Patterns	$c \in$ Constructors	

Fig. 1. Abstract syntax of the source language

$$
\begin{array}{rll}
v & ::= c(v_1 \dots v_n) & \text{(value)} \\
r & ::= f(x) & \text{(redex)} \\
 & \mid\ c(x_1 \dots x_n) & \\
 & \mid\ \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow t_1 \quad \dots \quad p_m \rightarrow t_m & \\
e & ::= [\_] & \text{(evaluation context)} \\
 & \mid\ f(e) & \\
 & \mid\ c(x_1 \dots x_{i-1}\ e\ t_{i+1} \dots t_n) & \\
 & \mid\ \textbf{case}\ e\ \textbf{of}\ p_1 \rightarrow t_1 \quad \dots \quad p_m \rightarrow t_m &
\end{array}
$$

Fig. 2. Value, redex and evaluation context

that all patterns are linear (no variable occurs more than once in a pattern). For simplicity, we also require that no defined variable be redefined by a pattern.

During evaluation, a term can be decomposed into redex and evaluation context (Fig. 2). A value v is a constructor c with arguments $v_1, ..., v_n$. A redex r is a term in which the outermost construction (function call, constructor or case-expression) can be reduced without reducing subterms first. An evaluation context e is a term with a 'hole' $[\_]$.

Design Choice We assume a primitive function $\lfloor \cdot \rfloor$ defined as follows:

$$\lfloor \langle v \rangle \rfloor \stackrel{\text{def}}{=} \langle v\ v \rangle \qquad \text{(duplication)}$$

$$\lfloor \langle v\ v' \rangle \rfloor \stackrel{\text{def}}{=} \begin{cases} \langle v \rangle & \text{if } v = v' \\ \langle v\ v' \rangle & \text{if } v \neq v' \end{cases} \qquad \text{(equality test)}$$

The function duplicates a value or performs an equality test. In the former case, given a single value, a pair with identical values is returned; in the latter case, given a pair of identical values, a single value is returned; otherwise the pair is returned unchanged. There are mainly two ways of using this function.

(1) *Duplication.* A value can be duplicated by a case-expression with $\lfloor\langle t\rangle\rfloor$ as argument: the resulting value of t is bound to x_1 and x_2. These two variables can then be used in t'. This construction makes the duplication of values explicit.

$$\textbf{case } \lfloor\langle t\rangle\rfloor \textbf{ of } \langle x_1\ x_2\rangle \to t'$$

(2) *Equality test.* The equality of two values can be tested by a case-expression with $\lfloor\langle t\ t'\rangle\rfloor$ as argument: if the resulting values of t and t' are identical then t_1 is evaluated with x bound to the identical value; otherwise, x and y are bound to the values of t and t', respectively.

$$\textbf{case } \lfloor\langle t\ t'\rangle\rfloor \textbf{ of } \langle x\rangle \to t_1;\ \langle x\ y\rangle \to t_2$$

The function has the following important properties:

$$\begin{array}{rcl}
\lfloor\langle v\rangle\rfloor = \langle v_1'\ v_2'\rangle & \iff & \langle v\rangle = \lfloor\langle v_1'\ v_2'\rangle\rfloor \\
\lfloor\langle v_1\ v_2\rangle\rfloor = \langle v'\rangle & \iff & \langle v_1\ v_2\rangle = \lfloor\langle v'\rangle\rfloor \\
\lfloor\langle v_1\ v_2\rangle\rfloor = \langle v_1'\ v_2'\rangle & \iff & \langle v_1\ v_2\rangle = \lfloor\langle v_1'\ v_2'\rangle\rfloor
\end{array}$$

Thus, the function has the property $\lfloor\bullet\rfloor = \circ \iff \bullet = \lfloor\circ\rfloor$. It is its own inverse:

$$\lfloor\cdot\rfloor^{-1} = \lfloor\cdot\rfloor$$

The advantage of this unusual function definition for program inversion is that it makes it easier to deal with two sides of the same coin: the duplication of a value and the equality test. *This is one of our key observations*: the duplication of a value in a source program becomes an equality test in the inverse program, and vice versa. Also, we can deal with a call to this primitive function during inversion in exactly the same way as with a call to a user-defined function. From a programming perspective, multiple occurrences of a variable in a term or non-linear patterns can be implemented by this function (recall function *pack* in Fig. 3). The functionality provided by $\lfloor\cdot\rfloor$ can also be realized by traditional constructions (*e.g.*, a conditional with an equality [19]), but we found that this complicates the inversion rules and made this design choice.

The following diagram illustrates the duality of duplication and equality testing in forward and backward computation. Duplication implies that two variables have identical values; an equality test checks whether the values of two variables are identical and we can use a single variable instead.

$$\lfloor\langle x\rangle\rfloor = \langle x_1\ x_2\rangle \qquad\qquad \langle x\rangle = \lfloor\langle x_1\ x_2\rangle\rfloor$$

$$\begin{tikzcd}
 & & x_1 \\
x \arrow[r, "\mathrm{dup}"] & \bullet \arrow[ur] \arrow[dr] & \\
 & & x_2
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
 & & x_1 \arrow[dl] \\
x & \bullet \arrow[l, "\mathrm{eq}"'] & \\
 & & x_2 \arrow[ul]
\end{tikzcd}$$

```
$pack(s) \triangleq$ case $s$ of
                $[\,] \rightarrow [\,]$
                $c_1 : r \rightarrow$ case $pack(r)$ of
                        $[\,] \rightarrow \langle c_1\ 1\rangle : [\,]$
                        $h : t \rightarrow$ case $h$ of
                                $\langle c_2\ n\rangle \rightarrow$ case $\lfloor\langle c_1\ c_2\rangle\rfloor$ of
                                        $\langle c\rangle \rightarrow \langle c\ \mathrm{S}(n)\rangle : t$
                                        $\langle c_1'\ c_2'\rangle \rightarrow \langle c_1'\ 1\rangle : (\langle c_2'\ n\rangle : t)$
```

Fig. 3. Program *pack*

```
$pack^{-1}(w) \triangleq$
    case $w$ of
    $[\,] \rightarrow [\,]$
    $x : y \rightarrow$ case $x$ of
            $\langle x_1\ x_2\rangle \rightarrow$ case $x_2$ of
                    $1 \rightarrow$ case $y$ of
                            $[\,] \rightarrow x_1 : pack^{-1}([\,])$
                            $o : t \rightarrow$ case $o$ of
                                    $\langle c_2'\ n\rangle \rightarrow$ case $\lfloor\langle x_1\ c_2'\rangle\rfloor$ of
                                            $\langle c_1\ c_2\rangle \rightarrow c_1 : pack^{-1}(\langle c_2\ n\rangle : t)$
                    $\mathrm{S}(n) \rightarrow$ case $\lfloor\langle x_1\rangle\rfloor$ of $\langle c_1\ c_2\rangle \rightarrow c_1 : pack^{-1}(\langle c_2\ n\rangle : y)$
```

Fig. 4. Program $pack^{-1}$

For example, in function *pack* (Fig. 3) the equality of two adjacent symbols, c_1 and c_2, is tested in the innermost case-expression. In the last line of inverse function $pack^{-1}$ (Fig. 4), symbol x_1 is duplicated. The case of non-equality of two adjacent symbols in *pack* corresponds to the case of non-equality in $pack^{-1}$. The assertion that those symbols are not equal is checked in forward and backward computation.

4 Challenges to Program Inversion

We now investigate two main challenges to automatic program inversion. Programs that satisfy the following two conditions are *well-formed for inversion* and our inversion method will always produce an inverse program.

The Inverse of a Conditional The most challenging point in program inversion is the inversion of conditionals (here, case-expressions). To calculate the input from a given output, we must know which of the m branches in a case-expression the source program took to produce that output, since *only one* of the branches was executed in the forward calculation (our language is deterministic). To make this choice in an inverse program, we must know m postconditions, R_i, one for each branch, such that for each pair of postconditions, we have: $R_i \wedge R_j = \mathit{false}$ $(1 \le i < j \le m)$. This divides the set of output values into m disjoint sets, and we can choose the correct branch by testing the given output value using the postconditions.

Postconditions that are suitable for program inversion can be derived by hand (*e.g.*, [7, 12]). In automatic program inversion they must be inferred from a source program. The program inverter KEinv [8] uses a heuristic method, and the language in which its postconditions are expressed consists of the primitive predicates available for the source language's value domain consisting of lists and integers. In general, there is no automatic method that would always find mutually exclusive postconditions, even if they exist.

Our value domain consists of constructors, and thus we decided to take a simpler approach. We formulate a criterion which can be checked locally for each function definition. We view the body of a function definition as a tree where the function head is the root, each case-expression corresponds to a node with branches, and the leaves of the tree are terms, l, consisting only of variables, constructors, and function calls. For example, in function *pack*, we have four leaves with the terms:

(a) $[\,]$ (c) $\langle c\ \mathrm{S}(n)\rangle : t$
(b) $\langle c_1\ 1\rangle : [\,]$ (d) $\langle c_1'\ 1\rangle : (\langle c_2'\ n\rangle : t)$

where (a) returns an empty list; (b) returns a non-empty list containing a single pair that has constructor 1 as its second component; (c) returns a non-empty list with a pair containing constructor S(_) as its second component, and (d) produces a list that is different from the values returned in (a–c). In short, all four branches in function *pack* return values that are pairwise disjoint. By performing pattern matching on an output value, we can decide from which of the four branches that value must have originated.

In addition to constructors and variables, a leaf term can contain function calls. For example, two leaf terms may be $\mathrm{A}(f(\mathrm{B}(x)))$ and $\mathrm{A}(f(\mathrm{C}(x)))$. Since we assume that all functions are injective, these two terms represent two different sets of values (an injective function f applied to two different input values returns two different output values). On the other hand, for leaf terms $\mathrm{A}(f(\mathrm{B}(x)))$ and $\mathrm{A}(g(\mathrm{C}(x)))$, we have not enough knowledge about f and g to decide whether their output values are different. We formalize the orthogonality test for the leaf terms of a function definition as follows:

(1) We require that, for each pair l and l' of leaf terms from different branches in a function, we have $\mathcal{P}[\![\, l,\, l' \,]\!] = \bot$. The test is defined as follows (ϵ denotes the identity substitution and for any substitution θ we have $\bot \circ \theta = \theta \circ \bot = \bot$):

$$\begin{array}{lll}
\mathcal{P}[\![\ x,\, x'\ ]\!] & = & \{x \mapsto x'\} \\
\mathcal{P}[\![\ f(l),\, f(l')\ ]\!] & = & \mathcal{P}[\![\ l,\, l'\ ]\!] \\
\mathcal{P}[\![\ c(l_1 \ldots l_n),\ c(l'_1 \ldots l'_n)\ ]\!] & = & \mathcal{P}[\![\ l_1,\, l'_1\ ]\!] \circ \cdots \circ \mathcal{P}[\![\ l_n,\, l'_n\ ]\!] \\
\mathcal{P}[\![\ c(l_1 \ldots l_n),\ c'(l'_1 \ldots l'_m)\ ]\!] & = & \perp \qquad \text{if } c \neq c' \\
\text{otherwise: } \ \mathcal{P}[\![\ l,\, l'\ ]\!] & = & \epsilon
\end{array}$$

Criterion (1) is a sufficient condition for a program to be injective if it applies to all functions defined in that program. If the postconditions of two branches are mutually exclusive, $\mathcal{P}[\![\ l,\, l'\ ]\!] = \perp$, then the sets of values they return are disjoint. Otherwise, we cannot be sure. To be more discriminating when this test is not decisive, we examine terms at nodes closer to the root. This test is more involved. Criterion (1) is sufficient for all programs in this paper except for inverting the inverse function $pack^{-1}$, which requires a refinement of the criterion (to show that the last two branches of that function have mutually exclusive postconditions).

As motivating example, consider the case where the leaf terms of two branches 'unify' (here $\mathcal{P}[\![\ y,\, y'\ ]\!] = \{y \mapsto y'\}$):

$$\begin{array}{l}
\text{(a) ...} \rightarrow \textbf{case}\ f(\mathrm{A}(x))\ \textbf{of}\ \mathrm{C}(y)\ \rightarrow y \\
\text{(b) ...} \rightarrow \textbf{case}\ f(\mathrm{B}(x'))\ \textbf{of}\ \mathrm{C}(y') \rightarrow y'
\end{array}$$

We find that (a) and (b) have mutually exclusive postconditions (the values y and y' are different because f is assumed to be injective). To summarize, when criteria (1) fails, we also check terms computing values in corresponding positions. This test is a more involved due to nested case-expressions.

(1') Given the ith branch, let l_i be its leaf term and let S_i be the set of all pattern-argument pairs of case-expressions. Given the ith and jth branch, if $\mathcal{P}[\![\ l_i,\, l_j\ ]\!] \neq \perp$, then perform the following check. The two branches have mutually exclusive postconditions if

$$\langle S_i, S_j, l_i, l_j, \mathcal{P}[\![\ l_i,\, l_j\ ]\!]\rangle \overset{*}{\hookrightarrow} \langle S'_i, S'_j, l'_i, l'_j, \perp\rangle$$

where $\langle S_i, S_j, l_i, l_j, \theta\rangle \hookrightarrow \langle S_i \setminus \{\langle p\ l'_i\rangle\}, S_j \setminus \{\langle p\theta\ l'_j\rangle\}, l'_i, l'_j, \theta \circ \mathcal{P}[\![\ l'_i,\, l'_j\ ]\!]\rangle$
if there exists p such that $\langle p\ l'_i\rangle \in S$ and $\langle p\theta\ l'_j\rangle \in S'$

For example, the 3rd and 4th branch in function $pack^{-1}$ have mutually exclusive postconditions because in one step with $\hookrightarrow$ we have $\mathcal{P}[\![\ \lfloor\langle x_1\ c'_2\rangle\rfloor,\ \lfloor\langle x_1\rangle\rfloor\ ]\!] = \perp$:

$$\left\langle \begin{array}{c}
\{\,...\,, \langle\langle c_1\ c_2\rangle\ \lfloor\langle x_1\ c'_2\rangle\rfloor\rangle\},\ \{\,...\,, \langle\langle c_1\ c_2\rangle\ \lfloor\langle x_1\rangle\rfloor\rangle\} \\
c_1 : pack^{-1}(\langle c_2\ n\rangle : t),\ c_1 : pack^{-1}(\langle c_2\ n\rangle : y) \\
\{c_1 \mapsto c_1, c_2 \mapsto c_2, \ldots\}
\end{array}\right\rangle \hookrightarrow \left\langle \begin{array}{c}
\{\,...\,\},\ \{\,...\,\} \\
\lfloor\langle x_1\ c'_2\rangle\rfloor,\ \lfloor\langle x_1\rangle\rfloor \\
\perp
\end{array}\right\rangle$$

Dead Variables Another challenging point in program inversion is when input values are discarded. Consider the selection function *first* defined by

$$\mathit{first}(x) \triangleq \textbf{case } x \textbf{ of } h{:}t \to h$$

Besides the fact that the function is not injective, the value of variable t is lost. When we invert such a program, we have to guess 'lost values'. In general, there are infinitely many possible guesses. The function may be invertible with respect to a precondition that ensures a constant value for t. Detecting suitable preconditions when values are lost is beyond the scope of our method. We adopted a straightforward solution which we call the "preservation of values" requirement.

(2) We require that each defined variable be used once in each branch.

Thus, a variable's value is always part of the output, and the only way to "diminish the amount of the output" is to reduce pairs into singletons by $\lfloor\langle t\ t'\rangle\rfloor$. But, in this case, no information is lost because both values need to be identical.

5 A Method for Program Inversion

We now present a method for the automatic inversion of programs that are well-formed for inversion. Our method uses logical formulas as internal representation. It consists of a frontend $\mathcal{A}[\![\ \cdot\ ]\!]$ and a backend $\mathcal{Z}[\![\ \cdot\ ]\!]$ that translate from a function to a formula and vice versa. Inversion $\mathcal{I}_0[\![\ \cdot\ ]\!]$ itself is then performed on the formula by backward reading. To simplify inversion of a program, inversion is carried out on a formula, rather than on the source program. The structure of the program inverter is shown in Fig. 5. We now give its definition and explain each of its components in the remainder of this section.

Definition 1 (program inverter). *Let q be a program well-formed for inversion. Then* program inverter $[\![\cdot]\!]^{-1}_{pgm}$ *is defined by*

$$[\![q]\!]^{-1}_{pgm} \stackrel{\text{def}}{=} \left\{[\![d]\!]^{-1} \mid d \in q\right\} \quad \textit{where} \quad [\![d]\!]^{-1} \stackrel{\text{def}}{=} \mathcal{Z}[\![\ \mathcal{I}_0[\![\ \mathcal{A}[\![\ d\ ]\!]\ ]\!]\ ]\!]$$

5.1 Translating between Function and Formula

The translation of a function to a logical formula makes it easier to invert the function (other representations are possible). During the translation, each construction is decomposed into a formula. A formula ϕ is constructed from atomic formulas, conjunctions and disjunctions. An atomic formula a is either a predicate that marks a variable x as input $\text{in}(x)$ or as output $\text{out}(y)$, an equality representing a function call $f(x){=}y$, a constructor application $c(x_1 \ldots x_n){=}y$, or a pattern matching $x{=}c(y_1 \ldots y_n)$.

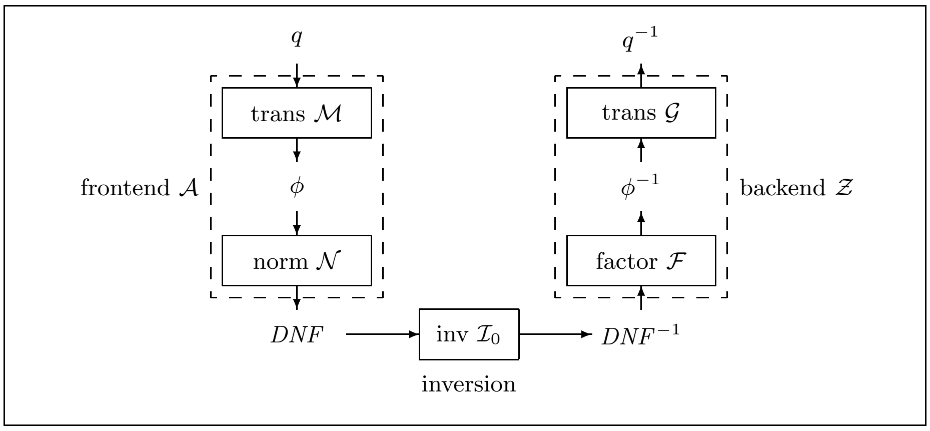

Fig. 5. Structure of the program inverter

$$\begin{aligned} \phi &::= \; a \;\mid\; \phi_1 \wedge \phi_2 \;\mid\; \phi_1 \vee \phi_2 \\ a &::= \; \text{in}(x) \;\mid\; \text{out}(y) \;\mid\; f(x){=}y \;\mid\; c(x_1 \ldots x_n){=}y \;\mid\; x{=}c(y_1 \ldots y_n) \end{aligned}$$

An atomic formula can be thought of as representing an ‘atomic operation’ (function call, constructor application, pattern matching). As a convention, the left-hand side of an equation is defined only in terms of input variables (here x) and the right-hand side is defined only in terms of output variables (here y). The intended forward reading of the equalities is from left to right. Atomic formulas are connected by conjunctions (representing a sequence of atomic operations), disjunctions (representing branches of a case-expression).

Definition 2 (frontend, backend). *Let d be a definition in a program well-formed for inversion. Then,* frontend $\mathcal{A}[\![\cdot]\!]$ *and* backend $\mathcal{Z}[\![\cdot]\!]$ *are defined by*

$$\mathcal{A}[\![d]\!] \overset{\text{def}}{=} \mathcal{N}_0[\![\mathcal{M}_0[\![d]\!]]\!]$$
$$\mathcal{Z}[\![d]\!] \overset{\text{def}}{=} \mathcal{G}_0[\![\mathcal{F}_0[\![d]\!]]\!]$$

Composition $\mathcal{Z}[\![\mathcal{A}[\![d]\!]]\!]$ returns the same definition modulo renaming of variables and reordering of nested case-expressions that do not depend on each other. The composition of the translations is a semantics-preserving transformation. The frontend and backend are conventional translators from functions to formulas, and vice versa.

For instance, the two steps of converting function *pack* to a formula are shown in Fig. 8 (to save space, commas (,) denote conjunctions). First, *pack* is converted into a nested formula by $\mathcal{M}_0[\![\cdot]\!]$. The formal parameter s is marked as input, in(s), and the outermost case-expression is decomposed into a conjunction containing $s{=}[\,]$ and another containing $s{=}c_1 : r$. Further study of the formula

Function-to-formula:

$$\begin{aligned}
\mathcal{M}_0\llbracket\ f(x) \triangleq t\ \rrbracket &= \mathrm{in}(x) \wedge \mathcal{M}\llbracket\ t\ \rrbracket \\
\mathcal{M}\llbracket\ y\ \rrbracket &= \mathrm{out}(y) \\
\mathcal{M}\llbracket\ e[\ f(x)\ ]\ \rrbracket &= f(x){=}\hat{y} \wedge \mathcal{M}\llbracket\ e[\ \hat{y}\ ]\ \rrbracket \\
\mathcal{M}\llbracket\ e[\ c(x_1 \ldots x_n)\ ]\ \rrbracket &= c(x_1 \ldots x_n){=}\hat{y} \wedge \mathcal{M}\llbracket\ e[\ \hat{y}\ ]\ \rrbracket \\
\mathcal{M}\llbracket\ e[\ \mathbf{case}\ x\ \mathbf{of}\ \{c_j(y_{j1} \ldots y_{jn_j}) \to t_j\}_{j=1}^m\ ]\ \rrbracket & \\
&= \bigvee_{j=1}^{m} x{=}c_j(y_{j1} \ldots y_{jn_j}) \wedge \mathcal{M}\llbracket\ e[\ t_j\ ]\ \rrbracket
\end{aligned}$$

Formula-to-function:

$$\begin{aligned}
\mathcal{G}_0\llbracket\ \mathrm{in}(x) \wedge \phi\ \rrbracket &= f^{-1}(x) \triangleq \mathcal{G}\llbracket\ \phi\ \rrbracket \\
\mathcal{G}\llbracket\ \mathrm{out}(y)\ \rrbracket &= y \\
\mathcal{G}\llbracket\ f(x){=}y \wedge \phi\ \rrbracket &= \mathcal{G}\llbracket\ \phi\ \rrbracket\{y \mapsto f(x)\} \\
\mathcal{G}\llbracket\ c(x_1 \ldots x_n){=}y \wedge \phi\ \rrbracket &= \mathcal{G}\llbracket\ \phi\ \rrbracket\{y \mapsto c(x_1 \ldots x_n)\} \\
\mathcal{G}\llbracket\ \bigvee_{j=1}^{m} x{=}c_j(x_{j1} \ldots x_{jn_j}) \wedge \phi_j\ \rrbracket &= \mathbf{case}\ x\ \mathbf{of}\ \{c_j(x_{j1} \ldots x_{jn_j}) \to \mathcal{G}\llbracket\ \phi_j\ \rrbracket\}_{j=1}^m
\end{aligned}$$

Fig. 6. Translation from function to formula, and vice versa

reveals how other constructions in the program are represented by atomic formulas. Second, the formula is normalized into a disjunctive normal form by $\mathcal{N}_0\llbracket \cdot \rrbracket$. The original nesting is dissolved and all branches are 'parallel', separated only by a disjunction at the top level. Now each branch contains the entire path from the root ($\mathrm{in}(s)$) to the leaf (*e.g.*, $\mathrm{out}(q)$). Note that several auxiliary variables had to be introduced because each construct was represented by several atomic formulas, *e.g.*, variable b to connect the call $\mathit{pack}(r){=}b$ with the use of its result.

Translations The two translations, $\mathcal{M}_0\llbracket \cdot \rrbracket$ and $\mathcal{G}_0\llbracket \cdot \rrbracket$, between a function and a formula are shown in Fig. 6. $\mathcal{M}_0\llbracket \cdot \rrbracket$ converts a function to a formula (using redex and evaluation context to decompose a term in accordance with its operational semantics; see Fig. 2) and $\mathcal{G}_0\llbracket \cdot \rrbracket$ translates a formula to a function. Nested constructor applications are decomposed into single constructor applications when they are converted into a formula (recall the definition of redex and evaluation context in Fig. 2). When converting a formula to a function, single constructor applications are nested. We use the notation $\hat{x}$ to denote a fresh variable; expression $t_1\{y \mapsto t_2\}$ denotes the result of replacing all occurrences of variable y in term t_1 by term t_2. Note that there is exactly one occurrence of variable y in term t_1 due to the fact that the source program is well-formed.

$$\begin{aligned}
\mathcal{F}_0\llbracket\ \bigvee_{j=1}^{m} \mathrm{in}(x_j) \wedge \phi_j\ \rrbracket &= \mathrm{in}(\hat{x}) \ \wedge\ \mathcal{F}\llbracket\ \bigvee_{j=1}^{m} \phi_j\{x_j \mapsto \hat{x}\}\ \rrbracket \\
\mathcal{F}\llbracket\ \mathrm{out}(y)\ \rrbracket &= \mathrm{out}(y) \\
\mathcal{F}\llbracket\ \bigvee_{j=1}^{m} f(x){=}y_j \wedge \phi_j\ \rrbracket &= f(x){=}\hat{y} \ \wedge\ \mathcal{F}\llbracket\ \bigvee_{j=1}^{m} \phi_j\{y_j \mapsto \hat{y}\}\ \rrbracket \\
\mathcal{F}\llbracket\ \bigvee_{j=1}^{m} c(x_1 \ldots x_n){=}y_j \wedge \phi_j\ \rrbracket &= c(x_1 \ldots x_n){=}\hat{y} \ \wedge\ \mathcal{F}\llbracket\ \bigvee_{j=1}^{m} \phi_j\{y_j \mapsto \hat{y}\}\ \rrbracket \\
\mathcal{F}\llbracket\ \bigvee_{j=1}^{m} x{=}c_j(y_{j1} \ldots y_{jn_j}) \wedge \phi_j\ \rrbracket &= \bigvee_{j=1}^{m'} x{=}c'_j(y'_{j1} \ldots y'_{jn'_j}) \ \wedge\ \mathcal{F}\llbracket\ \phi'_j\ \rrbracket
\end{aligned}$$

$$\text{where } \bigvee_{j=1}^{m} x{=}c_j(y_{j1} \ldots y_{jn_j}) \wedge \phi_j \ \overset{*}{\rightsquigarrow}\ \bigvee_{j=1}^{m'} x{=}c'_j(y'_{j1} \ldots y'_{jn'_j}) \wedge \phi'_j$$

with $\rightsquigarrow$ to search for mutually exclusive postconditions:

$$\begin{aligned}
&(x{=}c(y_1 \ldots y_n) \wedge \phi_1) \ \vee\ (x{=}c(y'_1 \ldots y'_n) \wedge \phi_2) \ \vee\ \phi \\
&\quad \rightsquigarrow\ x{=}c(\hat{y}_1 \ldots \hat{y}_n) \wedge (\phi_1\{y_i \mapsto \hat{y}_i\}_{i=1}^{n} \vee \phi_2\{y'_i \mapsto \hat{y}_i\}_{i=1}^{n}) \ \vee\ \phi
\end{aligned}$$

Note: Left-side variables ($x, x_1, \ldots$) must not occur in ϕ_j. Some of the c_j can be identical, but all c'_j are different. The order of atomic formulas in conjunctions is ignored.

Fig. 7. Factorization

Normalization After translating a function to a formula, we convert the formula into its *disjunctive normal form* (DNF) by normalization $\mathcal{N}_0\llbracket \cdot \rrbracket$. We can think of this as 'flattening' all nested case-expressions. This normalization of a logical formula is standard and not shown here. Recall that in the disjunctive normal form, all disjunctions are at the top level, connecting formulas consisting of conjunctions of atomic formulas. An example is shown in Fig. 8, where the formula obtained by translating function *pack* to a formula is converted to its disjunctive normal form.

Factorization Before translating a formula to a function definition, we need to factor out common atomic formulas from disjunctions. This is necessary because patterns in case-expressions cannot be nested in our language. This reintroduces nested case-expressions in a program. The factorization $\mathcal{F}\llbracket \cdot \rrbracket$ is shown in Fig. 7.

If all m disjunctions in a DNF contain a call to the same function $f(x){=}y_j$ using the same argument x, then we factor this call out of all m branches and replace all output variables y_j by the same fresh variable $\hat{y}$. Similarly, if all m disjunctions contain the constructor application $c(x_1 \ldots x_n){=}y_j$. The most involved part is when we meet pattern matchings $x{=}c_j(y_{j1} \ldots y_{jn_j})$. Here, some of the constructors c_j may be identical, and we only want to factor those common patterns out of the disjunctions. In Fig. 7, this is achieved by an auxiliary func-

1) Function-to-formula translation:

$$\mathrm{in}(s),\ \left(\begin{array}{l} \quad s{=}[\,],\ [\,]{=}a,\ \mathrm{out}(a) \\ \vee\ s{=}c_1{:}r,\ \mathit{pack}(r){=}b, \\ \quad \left(\begin{array}{l} \quad b{=}[\,],\ 1{=}e,\ \langle c_1\ e\rangle{=}f,\ [\,]{=}z,\ f{:}z{=}g,\ \mathrm{out}(g) \\ \vee\ b{=}h{:}t,\ h{=}\langle c_2\ n\rangle,\ \langle c_1\ c_2\rangle{=}u,\ \lfloor u\rfloor{=}v, \\ \quad \left(\begin{array}{l} \quad v{=}\langle c\rangle,\ \mathrm{S}(n){=}i,\ \langle c\ i\rangle{=}j,\ j{:}t{=}k,\ \mathrm{out}(k) \\ \vee\ v{=}\langle c_1'\ c_2'\rangle,\ 1{=}l,\ \langle c_1'\ l\rangle{=}m,\ \langle c_2'\ n\rangle{=}o, \\ \quad o{:}t{=}p,\ m{:}p{=}q,\ \mathrm{out}(q) \end{array}\right) \end{array}\right) \end{array}\right)$$

2) Disjunctive normal form:

$$\begin{array}{l} \quad \mathrm{in}(s),\ s{=}[\,],\ [\,]{=}a,\ \mathrm{out}(a) \\ \vee\ \mathrm{in}(s),\ s{=}c_1{:}r,\ \mathit{pack}(r){=}b,\ b{=}[\,],\ 1{=}e,\ \langle c_1\ e\rangle{=}f,\ [\,]{=}z,\ f{:}z{=}g,\ \mathrm{out}(g) \\ \vee\ \mathrm{in}(s),\ s{=}c_1{:}r,\ \mathit{pack}(r){=}b,\ b{=}h{:}t,\ h{=}\langle c_2\ n\rangle,\ \langle c_1\ c_2\rangle{=}u,\ \lfloor u\rfloor{=}v,\ v{=}\langle c\rangle, \\ \quad \mathrm{S}(n){=}i,\ \langle c\ i\rangle{=}j,\ j{:}t{=}k,\ \mathrm{out}(k) \\ \vee\ \mathrm{in}(s),\ s{=}c_1{:}r,\ \mathit{pack}(r){=}b,\ b{=}h{:}t,\ h{=}\langle c_2\ n\rangle,\ \langle c_1\ c_2\rangle{=}u,\ \lfloor u\rfloor{=}v,\ v{=}\langle c_1'\ c_2'\rangle, \\ \quad 1{=}l,\ \langle c_1'\ l\rangle{=}m,\ \langle c_2'\ n\rangle{=}o,\ o{:}t{=}p,\ m{:}p{=}q,\ \mathrm{out}(q) \end{array}$$

3) Inversion:

$$\begin{array}{l} \quad \mathrm{out}(s),\ [\,]{=}s,\ a{=}[\,],\ \mathrm{in}(a) \\ \vee\ \mathrm{out}(s),\ c_1{:}r{=}s,\ \mathit{pack}^{-1}(b){=}r,\ [\,]{=}b,\ e{=}1,\ f{=}\langle c_1\ e\rangle,\ z{=}[\,],\ g{=}f{:}z,\ \mathrm{in}(g) \\ \vee\ \mathrm{out}(s),\ c_1{:}r{=}s,\ \mathit{pack}^{-1}(b){=}r,\ h{:}t{=}b,\ \langle c_2\ n\rangle{=}h,\ u{=}\langle c_1\ c_2\rangle,\ \lfloor v\rfloor{=}u, \\ \quad \langle c\rangle{=}v,\ i{=}\mathrm{S}(n),\ j{=}\langle c\ i\rangle,\ k{=}j{:}t,\ \mathrm{in}(k) \\ \vee\ \mathrm{out}(s),\ c_1{:}r{=}s,\ \mathit{pack}^{-1}(b){=}r,\ h{:}t{=}b,\ \langle c_2\ n\rangle{=}h,\ u{=}\langle c_1\ c_2\rangle,\ \lfloor v\rfloor{=}u, \\ \quad \langle c_1'\ c_2'\rangle{=}v,\ l{=}1,\ m{=}\langle c_1'\ l\rangle,\ o{=}\langle c_2'\ n\rangle,\ p{=}o{:}t,\ q{=}m{:}p,\ \mathrm{in}(q) \end{array}$$

4) Factorization:

$$\mathrm{in}(w),\ \left(\begin{array}{l} \quad w{=}[\,],\ [\,]{=}s,\ \mathrm{out}(s) \\ \vee\ w{=}x{:}y,\ x{=}\langle x_1\ x_2\rangle, \\ \quad \left(\begin{array}{l} \quad x_2{=}1,\ \left(\begin{array}{l} \quad y{=}[\,],\ [\,]{=}b,\ \mathit{pack}^{-1}(b){=}r,\ x_1{:}r{=}s,\ \mathrm{out}(s) \\ \vee\ y{=}o{:}t,\ o{=}\langle c_2'\ n\rangle,\ \langle x_1\ c_2'\rangle{=}v,\ \lfloor v\rfloor{=}u, \\ \quad u{=}\langle c_1\ c_2\rangle,\ \langle c_2\ n\rangle{=}h,\ h{:}t{=}b, \\ \quad \mathit{pack}^{-1}(b){=}r,\ c_1{:}r{=}s,\ \mathrm{out}(s) \end{array}\right) \\ \vee\ x_2{=}\mathrm{S}(n),\ \langle x_1\rangle{=}v,\ \lfloor v\rfloor{=}u,\ u{=}\langle c_1\ c_2\rangle,\ \langle c_2\ n\rangle{=}h, \\ \quad h{:}y{=}b,\ \mathit{pack}^{-1}(b){=}r,\ c_1{:}r{=}s,\ \mathrm{out}(s) \end{array}\right) \end{array}\right)$$

Fig. 8. Translation, normalization, backward reading, and factorization of *pack*

$$
\begin{aligned}
\mathcal{I}_0[\![\ \bigvee_{j=1}^{m} \phi_j\ ]\!] &= \bigvee_{j=1}^{m} \mathcal{I}_1[\![\ \phi_j\ ]\!] \\
\mathcal{I}_1[\![\ \bigwedge_{j=1}^{m} \phi_j\ ]\!] &= \bigwedge_{j=1}^{m} \mathcal{I}[\![\ \phi_j\ ]\!]
\end{aligned}
\qquad
\begin{aligned}
\mathcal{I}[\![\ \text{in}(x)\ ]\!] &= \text{out}(x) \\
\mathcal{I}[\![\ \text{out}(y)\ ]\!] &= \text{in}(y) \\
\mathcal{I}[\![\ f(x){=}y\ ]\!] &= f^{-1}(y){=}x \\
\mathcal{I}[\![\ c(x_1 \ldots x_n){=}y\ ]\!] &= y{=}c(x_1 \ldots x_n) \\
\mathcal{I}[\![\ x{=}c(y_1 \ldots y_n)\ ]\!] &= c(y_1 \ldots y_n){=}x
\end{aligned}
$$

Fig. 9. Inversion: backward reading

tion. These steps are repeated until all common atomic formulas are factored out of the disjunctions. This prepares for the construction of the inverse program. An example of factorization can be seen in Fig. 8. For instance, the matchings $l{=}1$ and $e{=}1$ contained in the second and fourth conjunctions, respectively, could be factored out into $x_2{=}1$. In the inverse program $pack^{-1}$, this becomes one of the patterns in a case-expression (Fig. 4). An inspection of the example reveals similar correspondences and factorizations.

5.2 Inversion by Backward Reading

Atomic formulas are easily inverted by reading the intended meaning backwards. The idea of 'backward reading' programs can be found in [7, 12]. We follow this method for atomic formulas. The rules for our formula representation are shown in Fig. 9. The inverse of in(x) is out(x) and vice versa; the inverse of function call $f(x){=}y$ is $f^{-1}(y){=}x$; the inverse of constructor application $c(x_1 \ldots x_n){=}y$ is pattern matching $y{=}c(x_1 \ldots x_n)$ and vice versa. As we recall from Sect. 3, a convenient detail is that primitive function $\lfloor\cdot\rfloor$ is its own inverse: $\lfloor\cdot\rfloor^{-1} = \lfloor\cdot\rfloor$. Thus, we can write the inverse of $\lfloor x\rfloor{=}y$ immediately as $\lfloor y\rfloor{=}x$. Observe that the inversion performs no unfold/fold on functions. It terminates on all formulas. We see that global inversion of a program is based on the local invertibility of atomic formulas (also called 'compositional program inversion').

The result of backward reading the normal form is shown in Fig. 8. Compare the disjunctive normal form of function *pack* before and after the transformation. The order of atomic formulas is unchanged, their order does not matter in a conjunction, but each atomic formula is inverted according to the rules in Fig. 9. For instance, $pack(r){=}b$ becomes $pack^{-1}(b){=}r$. According to our convention, input and output variables switched sides. The treatment of calls to recursive functions is surprisingly easy.

After the backward reading, the formula is converted into a program by $\mathcal{Z}[\![\ \cdot\ ]\!]$. Inversion was successful; the inverse program $pack^{-1}$ is shown in Fig. 4. We have automatically produced an unpack function from a pack function. For instance, to unpack a packed symbol list: $pack^{-1}([\langle\text{A } 2\rangle\langle\text{B } 1\rangle\langle\text{C } 3\rangle]) = [\text{AABCCC}]$.

5.3 Termination, Correctness, Non-Degradation

Program inverter $[\![q]\!]^{-1}_{pgm}$ terminates on every program q that is well-formed for inversion. It is easy to see by structural induction that each of the transformations terminates (frontend, inversion, backend) for all its respective inputs (function, formula). The correctness of the transformation follows from the correctness of the frontend and backend, and the correctness of the backward reading.

Due to our particular method of inverting programs, inverse programs have the following property: if the program takes n steps to compute an output value y from an input value x, then the inverse program takes n steps to compute x from y. The downside of this property is that our inverse programs are never faster; the upside is that the efficiency is never degraded. This is only the case for our inversion method because we perform no unfold/fold steps (it does not hold for program inversion in general).

To see this, we examine the inversion of formulas. We assume that the computation of each atomic formula requires one unit of time. The inversion rules only redirect the intended computation; they neither add nor omit atomic operations. All source programs are injective, which means that the backward computation has to follow the reversed path of the forward computation to produce the input from the output. There is no unfold/fold or generalization that could shorten or extend a computation path. The translation in the frontend and in the backend does not add or omit operations. Thus, the computation path of forward and backward computation are identical.

6 More Examples

This section shows more examples of automatic program inversion using our method. Consider the following three examples.

(1) The first example is the inversion of a program containing two function definitions (Fig. 10). Function *fib* computes two neighboring Fibonacci numbers. For example, $\mathit{fib}(0) = \langle 1\ 1\rangle$, $\mathit{fib}(1) = \langle 1\ 2\rangle$, $\mathit{fib}(2) = \langle 2\ 3\rangle$, and so on. The definition of *fib* is injective. Function *plus* is defined by $\mathit{plus}(\langle x\ y\rangle) = \langle x\ x+y\rangle$. Duplication and equality tests are realized by $\lfloor\cdot\rfloor$. As usual, integers are represented by unary numbers. Automatic inversion of both functions is successful and produces the two inverse functions fib^{-1} and plus^{-1} (Fig. 10). For instance, we can now compute $\mathit{fib}^{-1}(\langle 34\ 55\rangle) = 8$ and $\mathit{plus}^{-1}(\langle 3\ 5\rangle) = \langle 3\ 2\rangle$.

(2) The second example is the inversion of function *mir* which appends to an input list the reversed input list (Fig. 11). For instance, $\mathit{mir}(\text{ABC}) = \text{ABCCBA}$. The auxiliary function *tailcons* appends a value at the end of a list. If we assume that the inverse of *tailcons*, $\mathit{tailcons}^{-1}$, is given, automatic inversion is successful. It produces function mir^{-1} (Fig. 11). Therefore, to undo the mirroring operation, we can use $\mathit{mir}^{-1}(\text{ABCCBA}) = \text{ABC}$. The function is undefined when applied to an incorrect input, *e.g.*, $\mathit{mir}^{-1}(\text{ACBA})$.

(3) Finally, all four inverse programs (pack^{-1}, fib^{-1}, plus^{-1}, mir^{-1}) can be inverted again and we get their original definitions back (modulo renaming of

```
$\mathit{fib}(x)$          $\triangleq$ case $x$ of $0 \to \lfloor\langle 1\rangle\rfloor$
                                 $\mathrm{S}(y) \to$ case $\mathit{fib}(y)$ of
                                            $\langle u\ v\rangle \to$ case $\mathit{plus}(\langle v\ u\rangle)$ of $\langle w\ z\rangle \to \lfloor\langle w\ z\rangle\rfloor$
$\mathit{fib}^{-1}(a)$     $\triangleq$ case $\lfloor a\rfloor$ of
                   $\langle b\rangle \to$ case $b$ of $1 \to 0$
                   $\langle w\ z\rangle \to$ case $\mathit{plus}^{-1}(\langle w\ z\rangle)$ of $\langle v\ u\rangle \to \mathrm{S}(\mathit{fib}^{-1}(\langle u\ v\rangle))$
$\mathit{plus}(z)$         $\triangleq$ case $z$ of
                   $\langle x\ y\rangle \to$ case $y$ of $0 \to \lfloor\langle x\rangle\rfloor$
                                              $\mathrm{S}(w) \to$ case $\mathit{plus}(\langle x\ w\rangle)$ of $\langle u\ v\rangle \to \lfloor\langle u\ \mathrm{S}(v)\rangle\rfloor$
$\mathit{plus}^{-1}(h)$    $\triangleq$ case $\lfloor h\rfloor$ of $\langle x\rangle \to \langle x\ 0\rangle$
                                 $\langle u\ e\rangle \to$ case $e$ of
                                              $\mathrm{S}(v) \to$ case $\mathit{plus}^{-1}(\langle u\ v\rangle)$ of $\langle x\ w\rangle \to \langle x\ \mathrm{S}(w)\rangle$
```

Fig. 10. Functions *fib* and *plus* and their inverse functions

```
$\mathit{mir}(x)$       $\triangleq$ case $x$ of $[\,] \to [\,]$
                              $h\!:\!t \to$ case $\lfloor\langle h\rangle\rfloor$ of $\langle h_1\ h_2\rangle \to h_1\!:\!\mathit{tailcons}(\langle \mathit{mir}(t), h_2\rangle)$
$\mathit{mir}^{-1}(g)$  $\triangleq$ case $g$ of $[\,] \to [\,]$
                              $h_1\!:\!e \to$ case $\mathit{tailcons}^{-1}(e)$ of
                                      $\langle c, h_2\rangle \to$ case $\lfloor\langle h_1\ h_2\rangle\rfloor$ of $\langle h\rangle \to h\!:\!\mathit{mir}^{-1}(c)$
```

Fig. 11. Function *mir* and its inverse function

variables and reordering of nested case-expressions).

A limitation of the present method is the lack of postcondition heuristics that can use global information about the output of a function. For example, using the well-formed criteria from Sect. 4, function *tailcons* is not well-formed for inversion because criterion (1) is not enough to show that the output sets are different (the first branch is a singleton list; the second branch returns a list with at least two elements since *tailcons* always returns a non-empty list). Owing to the postcondition heuristics in KEinv, *tailcons* can be inverted with KEinv, while we cannot perform this inversion. This leaves room for future work.

```
$\mathit{tailcons}(a) \triangleq$ case $a$ of
            $\langle x\ y\rangle \to$ case $x$ of $[\,] \to y\!:\![\,]$
                                    $u\!:\!v \to u\!:\!\mathit{tailcons}(\langle v\ y\rangle)$
```

Similarly, tail-recursive functions are not well-formed according to criterion (1) in Sect. 4 because one of the branches contains only a function call, while other branches will contain variables or constructors. For example, the tail-recursive version of function *reverse* can not be inverted with the present method. This requires a non-local criterion for testing invertibility.

7 Related Work

The method presented in this paper is based on the principle of global inversion based on local invertibility [7, 12]. The work was inspired by KEinv [14, 8]. In contrast to KEinv, our method can successfully deal with equality and duplication of variables. Most studies on functional languages and program inversion have involved program inversion by hand (*e.g.*, [16]). They may be more powerful at the price of automation. This is the usual trade-off. Inversion based on Refal graphs [18, 11, 20, 17] is related to the present method in that both use atomic operations for inversion; a more detailed comparison will be a topic of future work. An algorithm for inverse computation can be found in [1, 2]. It performs inverse computation also on programs that are not injective.

8 Conclusion

We presented a method suited for automatic program inversion in a functional language with constructors and equality. A key observation was that value duplication and equality are two sides of the same coin in program inversion. Based on this observation, we introduced a self-inverse primitive function that simplifies the inversion considerably. We introduced well-formedness for inversion based on orthogonal output domains. To simplify inversion of a program, inversion is carried out on the disjunctive normal form of a logical formula representing atomic operations in our languages, rather than on the source language. The inversion method is split into a frontend and backend and inversion proper. This clean and modular structure of the inversion makes it easy to verify the correctness of our method.

Tasks for future work include the refinement of our well-formedness criteria. We have seen two examples where it excluded a program that could be inverted by other methods. We have presented our method using a small functional language and based on several simplifying assumptions, but we believe that our results can be ported to other functional languages, such as Lisp, at the price of somewhat more involved inversion rules. We have not included integer arithmetic and have represented integers by unary numbers. This worked surprisingly well, for example, for inverting a version of Fibonacci numbers. However, adding arithmetic will make our method more practical. A possible extension might involve constraint systems for which well-established theories exist.

Acknowledgements We are grateful to Alberto Pettorossi and Maurizio Proietti for valuable comments on an earlier version of our inversion method. The example of Fibonacci inversion and other technical improvements are a result of these discussions. Thanks to the anonymous reviewers for their feedback.

References

1. S. M. Abramov, R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. Schmidt, I. H. Sudborough (eds.), *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, 269–295. Springer-Verlag, 2002.
2. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
3. R. Bird, O. de Moor. *Algebra of Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
4. J. S. Briggs. Generating reversible programs. *Software Practice and Experience*, 17:439–453, 1987.
5. C. D. Carothers, K. S. Perumalla, R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM TOMACS*, 9(3):224–253, 1999.
6. J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(1):1–46, 1981.
7. E. W. Dijkstra. Program inversion. In F. L. Bauer, M. Broy (eds.), *Program Construction: International Summer School*, LNCS 69, 54–57. Springer-Verlag, 1978.
8. D. Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*, 219–221. Morgan Kaufmann, Inc., 1985.
9. R. Glück, Y. Kawada, T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 10–19. ACM Press, 2003.
10. R. Glück, A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl (ed.), *Cybernetics and Systems '94*, Vol. 2, 1563–1570. World Scientific, 1994.
11. R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, 286–287. ACM Press, 1990.
12. D. Gries. *The Science of Programming*, chapter 21 Inverting Programs, 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
13. H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications. Proceedings*, LNCS 355, 564–568. Springer-Verlag, 1989.
14. R. E. Korf. Inversion of applicative programs. In *Int. Joint Conference on Artificial Intelligence (IJCAI-81)*, 1007–1009. William Kaufmann, Inc., 1981.
15. G. B. Leeman. A formal approach to undo operations in programming languages. *ACM TOPLAS*, 8(1):50–97, 1986.
16. S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 2386, 209–232. Springer-Verlag, 2002.
17. A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, M. Broy, I. V. Pottosin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1181, 249–260. Springer-Verlag, 1996.

18. A. Y. Romanenko. Inversion and metacomputation. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 12–22. ACM Press, 1991.
19. M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
20. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.