

Principles of Inverse Computation and the Universal Resolving Algorithm

Sergei Abramov¹ and Robert Glück^{2*}

¹ Program Systems Institute, Russian Academy of Sciences
RU-152140 Pereslavl-Zalessky, Russia,
`abram@botik.ru`

² PRESTO, JST & Institute for Software Production Technology
Waseda University, School of Science and Engineering
Tokyo 169-8555, Japan,
`glueck@acm.org`

Abstract. We survey fundamental concepts in inverse programming and present the Universal Resolving Algorithm (URA), an algorithm for inverse computation in a first-order, functional programming language. We discuss the principles behind the algorithm, including a three-step approach based on the notion of a perfect process tree, and demonstrate our implementation with several examples. We explain the idea of a semantics modifier for inverse computation which allows us to perform inverse computation in other programming languages via interpreters.

1 Introduction

While computation is the calculation of the output of a program for a given input, *inverse computation* is the calculation of the possible input of a program for a given output. The only area that deals successfully with a solution to inversion problems is logic programming [34, 36]. But solving inversion problems also makes sense outside of logic programming, for example, when an algorithm for one direction is easier to write than an algorithm for the direction in which we are interested, or when both directions of an algorithm are needed (such as encoding/decoding). Ideally, the inverse program can be derived from the forward program by *program inversion*. Given the importance of program inversion, relatively few papers have been devoted to the topic. We regard program inversion, beside program specialization and program composition, as one of the three fundamental tasks for metacomputation [16].

The idea of inverse computation can be traced back to reference [38]. We found that inverse computation and program inversion can be related conveniently [4] using the *Futamara projections* [12] which are well-known from *partial evaluation* [30]: a program inverter is a generating extension of an inverse interpreter. This insight is useful because inverse computation is simpler than program inversion. In this paper we shall focus on inverse computation.

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

We survey fundamental concepts in inverse programming and present the *Universal Resolving Algorithm* (URA), an algorithm for inverse computation in a first-order functional language, by means of examples. The emphasis in this paper is on presenting principles and foundations of inverse computation. We cover the topic starting from theoretical considerations to actual computer experiments. More details and a formal treatment of the material presented in this paper can be found in the literature [2, 4, 5]. We also draw upon results of references [15, 17, 18]. This work belongs to a line of research on supercompilation and metacomputation (*e.g.*, [48, 50, 32, 1, 20, 29, 40, 46, 43]). It is the first paper in this series that brings together, in a unified form, the results of inverse computation. We assume the reader is familiar with partial evaluation and programming languages, for example as presented in reference [30, Parts I&II] or [28].

The paper is organized as follows: Sect. 2 summarizes fundamental concepts in inverse programming; Sect. 3 explains the principles behind the Universal Resolving Algorithm; Sect. 4 illustrates the algorithm with a complete example, and Sect. 5 demonstrates our implementation; Sect. 6 discusses related work; and Sect. 7 concludes the presentation.

2 Fundamental Concepts in Inverse Programming

This section surveys fundamental concepts in inverse programming from a language-independent perspective. We present two tools, an inverse interpreter and a program inverter, for solving inversion problems, explain important properties of inverse computation and how some of the constructions can be optimized.

Notation For any program text p , written in language L , we let $\llbracket p \rrbracket_L d$ denote the application of L -program p to its input d (when the index L is unspecified, we assume that a language L is intended). The notation is strict in its arguments. When we define a program using λ -abstraction, we assume that the definition is translated into the corresponding programming language. We assume that we are dealing with universal programming languages and, thus, this translation is always possible. Equality between applications shall always mean strong (computational) equivalence: either both sides of an equation are defined and equal, or both sides are undefined. Values are S-expressions known from Lisp. We write 'A for an atom A (for numbers we omit the quote), $d_1:d_2$ for a pair of values d_1 and d_2 , and $[d_1, d_2, \dots, d_n]$ for a list $d_1:(d_2:(\dots(d_n:\text{'Nil'})\dots))$.

Tools for Solving Inversion Problems We distinguish between two tools for solving inversion problems: an inverse interpreter that performs *inverse computation* and a program inverter that performs *program inversion*. Let p be a program that computes y from x . For simplicity, we assume p is injective in x . Computation of output y by applying p to x is described by:

$$\llbracket p \rrbracket x = y \tag{1}$$

1. **Inverse interpreter:** The determination, for a given program p and output y , of an input x of p such that $\llbracket p \rrbracket x = y$ is inverse computation. A program *invint* that performs inverse computation, is an *inverse interpreter*.

$$\llbracket \text{invint} \rrbracket [p, y] = x \quad (2)$$

2. **Program inverter:** Let p^{-1} be a program that performs inverse computation for a given program p . A program *invtrans* that produces p^{-1} , is a *program inverter* (also called an inverse translator). Program p^{-1} will often be significantly faster in computing x than inverse interpretation of p by *invint*. Inversion in two stages is described by:

$$\llbracket \text{invtrans} \rrbracket p = p^{-1} \quad (3)$$

$$\llbracket p^{-1} \rrbracket y = x \quad (4)$$

Inverse computation of a program p can be performed in one stage with *invint* and in two stages with *invtrans*. Using equations (2, 3, 4) we obtain the following functional equivalence between *invint* and *invtrans*:

$$\underbrace{\llbracket \text{invint} \rrbracket [p, y]}_{\text{one stage}} = \underbrace{\llbracket \llbracket \text{invtrans} \rrbracket p \rrbracket y}_{\text{two stages}} \quad (5)$$

It is easy to see that an inverse interpreter can be defined in terms of a program inverter, and vice versa (where *sint* is a self-interpreter¹):

$$\text{invint}' \equiv \lambda[p, y]. \llbracket \text{sint} \rrbracket [\llbracket \text{invtrans} \rrbracket p, y] \quad (6)$$

$$\text{invtrans}' \equiv \lambda p. \lambda y. \llbracket \text{invint} \rrbracket [p, y] \quad (7)$$

Both definitions can be optimized using program composition as in the degeneration projections [18] (*invtrans* \rightarrow *invint*), and program specialization as in the Futamura projections [12] (*invint* \rightarrow *invtrans*). This will be explained below.

The computability of the solution is not always guaranteed, even with sophisticated inversion strategies. Some inversions are too resource consuming, while others are undecidable. For particular values p and y , no value x need exist. In fact, the existence of a value x is an undecidable question: there is no program that will always terminate if such a value does not exist.

In spite of this, an inverse interpreter exists [38] that will compute value x if it exists. Essentially, that program computes $\llbracket p \rrbracket x$ for all values x until it comes to an x such that $\llbracket p \rrbracket x = y$. Since that computation may not terminate for some x , the program needs to perform the search for x in a diagonal manner. For instance, it examines $\forall k \geq 0$ and $\forall x$ with $\text{size}(x) \leq k$ the result of computation $\llbracket p \rrbracket^k x$ where $\llbracket p \rrbracket^k x = \llbracket p \rrbracket x$ if the computation of $\llbracket p \rrbracket x$ stops in k steps; undefined otherwise. This *generate-and-test approach* will correctly find all solutions, but is too inefficient to be useful.

¹ A self-interpreter *sint* for L is an L -interpreter written in L : $\llbracket \text{sint} \rrbracket_L [p, x] = \llbracket p \rrbracket_L x$.

Even when it is certain that an efficient inverse program p^{-1} exists, the derivation of such a procedure from p by a program inverter may be difficult. Most of the work on program inversion (e.g., [7, 8, 10, 24, 25, 39, 51]) has been on program transformation by hand: specify a problem as the inverse of an easy computation, and then derive an efficient algorithm by manual application of transformation rules. Note that we can define a *trivial inverse program* p_{triv}^{-1} for any program p using an inverse interpreter *invint*:

$$p_{triv}^{-1} \equiv \lambda y. \llbracket invint \rrbracket [p, y] \quad (8)$$

Inverse Programming Programs are usually written for forward computation. In *inverse programming* [1] one writes programs so that their backwards computation produces the desired result. Inverse programming relies on the existence of an inverse interpreter or a program inverter. Given a program p that computes y from x , one solves the inversion problem for a given y :

$$\llbracket p^{-1} \rrbracket y = x \quad (9)$$

This gives rise to a *declarative style* of programming where one specifies the result rather than describes how it is computed. An example is a program r that checks whether two values x and y are in relation, and returns the corresponding Boolean value $bool \in \{\text{'True'}, \text{'False'}\}$:

$$\llbracket r \rrbracket [x, y] = bool \quad (10)$$

Although checking whether x and y are in relation is only a particular case of forward computation, it is worth considering it separately because of the large number of problems that can be formulated this way. Relations are often used in specifications as they are not directional and give preference neither to x nor to y . For example, in logic programming one writes a relation r in a subset of *first-order logic* (Horn clauses) and solves a restricted inversion problem for a given y and output 'True' by an inverse interpreter. The resolution principle [41] reduced the combinatorial growth of the search space of the early generate-and-test methods used in automated theorem proving [14, 9], and became the basis for the use of first-order logic as a tool for inverse programming [23, 34].

The importance of inverting relations stems from the fact that it is often easier to check the correctness of a solution than to find it. But solving inversion problems also makes sense outside of logic programming, for example, when an algorithm for one direction is easier to write than an algorithm for the direction in which we are interested, or when both directions of an algorithm are needed. In fact, the problem of inversion does not depend on a particular programming language. Later in this paper we will study a small functional language (Sect. 4), and show inverse computation in a small imperative language (Sect. 5.3).

A Semantics Modifier for Inverse Computation We now explain an important property of inverse computation, namely that inverse computation can

be performed in any language via an interpreter for that language [2]. Suppose we have two functionally equivalent programs p and q written in languages P and Q , respectively. For the sake of simplicity, let p and q be injective. Let $invintP$ and $invintQ$ be two inverse interpreters for P and Q , respectively. Since p and q are functionally equivalent, inverse computation of p and q using the corresponding inverse interpreter produces the same result (we say the inverse semantics is *robust* [2]):

$$(\forall x. \llbracket p \rrbracket_P x = \llbracket q \rrbracket_Q x) \Rightarrow (\forall y. \llbracket invintP \rrbracket_L [p, y] = \llbracket invintQ \rrbracket_M [q, y]) \quad (11)$$

Suppose we have a Q -interpreter written in P , but no inverse interpreter for Q . How can we perform inverse computation in Q without writing an inverse interpreter $invintQ$? The following functional equivalence holds for the Q -interpreter:

$$\llbracket intQ \rrbracket_P [q, x] = \llbracket q \rrbracket_Q x \quad (12)$$

We can immediately define a P -program q' (by fixing the program argument of $intQ$) such that q' and q are functionally equivalent, so $\llbracket q' \rrbracket_P x = \llbracket q \rrbracket_Q x$, where

$$q' \equiv \lambda x. \llbracket intQ \rrbracket_P [q, x] \quad (13)$$

Then the inversion problem of q can be solved by applying $invintP$ to q' :

$$\llbracket invintP \rrbracket_L [q', y] = x \quad (14)$$

The result x is a correct solution for the inversion problem y because q and q' are functionally equivalent and inverse computation is robust (as stated for injective programs in Eq. 11). It is noteworthy that we obtained a solution for inverse computation of a Q -program using an inverse interpreter for P and an interpreter for Q . This property of inverse computation is convenient because writing an interpreter is usually easier than writing an inverse interpreter. We show an example in Sect. 5.3 where we perform inverse computation in an imperative language using an inverse interpreter for a functional language. This indicates that the essence of an inverse semantics can be captured in a language-independent way.

Finally, we define a *semantics modifier for inverse computation* [4]. The program takes a Q -interpreter $intQ$ written in L , a Q -program q , and an output y , and computes a solution x for the given inversion problem (q, y) :

$$\llbracket invmod \rrbracket_L [intQ, q, y] = x \quad (15)$$

The inversion modifier $invmod$ makes it easy to define an inverse interpreter for any language Q , given a standard interpreter $intQ$ for that language:

$$invintQ' \equiv \lambda [q, y]. \llbracket invmod \rrbracket_L [intQ, q, y] \quad (16)$$

Such programs are called *semantics modifiers* because they modify the standard semantics of a language Q (given in the form of an interpreter $intQ$). The inversion modifier can be defined using the expressions in (13, 14). The construction

is a generalization of the classical two-level interpreter hierarchy (a theoretical exposition of non-standard interpreter hierarchies can be found in reference [2]).

$$\begin{aligned} invmod &\equiv \lambda[intQ, q, y]. \llbracket invintP \rrbracket_L [q', y] \\ \text{where } q' &\equiv \lambda x. \llbracket intQ \rrbracket_P [q, x] \end{aligned} \quad (17)$$

Optimizing the Constructions As shown in [4], inverse interpreters and program inverters can be related conveniently using the Futamura projections known from partial evaluation [30]: a program inverter is a generating extension of an inverse interpreter. Similarly, an inverse interpreter is the composition of a self-interpreter and a program inverter [17, 18]. Given one of the two tools, we can obtain, in principle, an efficient implementation of its companion tool by program specialization or by program composition. We will describe three possibilities (a, b, c) for optimizing these constructions.

Before we show these transformations, we specify the semantics of a program composer *komp* and a program specializer *spec*. For notational convenience we assume that both programs are *L*-to-*L* transformers written in *L*; for multi-language composition and specialization see [17].

$$\llbracket komp \rrbracket [p, q] = pq \text{ such that } \llbracket pq \rrbracket [x, y] = \llbracket p \rrbracket [\llbracket q \rrbracket x, y] \quad (18)$$

$$\llbracket spec \rrbracket [p, m, x_1 \dots x_m] = p_x \text{ such that } \llbracket p_x \rrbracket [y_1 \dots y_n] = \llbracket p \rrbracket [x_1 \dots x_m, y_1 \dots y_n] \quad (19)$$

a) *From program inversion to inverse computation* Let *sint* be a self-interpreter for *L*. Consider the composition of *sint* and *invtrans* in Eq. 6. Inverse computation is performed in two stages: first, program inversion of *p*; second, interpretation of *p*'s inverse program with *y*.

$$\llbracket sint \rrbracket [\llbracket invtrans \rrbracket p, y] = x \quad (20)$$

When we apply a program composer to this composition, we obtain a new program which we call *invint'* (it has the functionality of an inverse interpreter):

$$\llbracket komp \rrbracket [sint, invtrans] = invint' \quad (21)$$

$$\llbracket invint' \rrbracket [p, y] = x \quad (22)$$

This application of a program composer is known as *degeneration* [17, 18]. The transformation converts a two-stage computation into a one-stage computation (here, a program inverter into an inverse interpreter).

b) *From inverse computation to program inversion* Consider an inverse interpreter *invint* and apply it to a program *p* and a request *y* as shown in Eq. 7:

$$\llbracket invint \rrbracket [p, y] = x \quad (23)$$

When we specialize the inverse interpreter *wrt* its first argument *p* using a program specializer, we obtain a new program which we call *p'⁻¹* (it has the functionality of an inverse program of *p*):

$$\llbracket spec \rrbracket [invint, 1, p] = p'^{-1} \quad (24)$$

$$\llbracket p'^{-1} \rrbracket [y] = x \quad (25)$$

We can continue the transformation and specialize the program specializer *spec* in Eq. 24 wrt its arguments *invint* and 1. This is known as *self-application* [12] of a program specializer. Self-application is possible because a specializer, like any other program, can be subjected to program transformation. We obtain a new program which we call *invtrans'* (it has the functionality of a program inverter):

$$\llbracket spec \rrbracket [spec, 2, invint, 1] = invtrans' \quad (26)$$

$$\llbracket invtrans' \rrbracket p = p'^{-1} \quad (27)$$

This transformation of an inverse interpreter into a program inverter is interesting because inverse computation is conceptually simpler than program inversion (*e.g.*, no code generation). The applications of program specialization in Eq. 24 and Eq. 26 are a variation of the 1st and 2nd Futamura projection [12], which were originally proposed for transforming interpreters into translators (and pioneered in the area of partial evaluation [30]). We see that the same principle applies to the transformation of inverse interpreters into program inverters.

c) Porting inverse interpreters Let *invmod* be an inversion modifier for *P*, and use it to perform inverse computation of a *Q*-program *q* via a *Q*-interpreter *intQ* written in *L* (as shown in Eq. 15):

$$\llbracket invmod \rrbracket [intQ, q, y] = x \quad (28)$$

When we specialize the modifier wrt its argument *intQ*, we obtain a new program which we call *invintQ''* (it has the functionality of an inverse interpreter for *Q*):

$$\llbracket spec \rrbracket [invmod, 1, intQ] = invintQ'' \quad (29)$$

$$\llbracket invintQ'' \rrbracket [q, y] = x \quad (30)$$

We see that program specialization has the potential to make inversion modifiers more efficient. This is important for a practical application of the modifiers because inverse interpretation of a *Q*-program using an interpreter *intQ* as mediator between languages *P* and *Q* adds additional computational overhead. The equations can be carried further [4], for instance, when we specialize *spec* in Eq. 29 wrt its arguments *invmod* and 1, we obtain a program which converts an interpreter *intQ* into an inverse interpreter *invintQ''*. Or, when we specialize *invmod* in Eq. 28 wrt its arguments *intQ* and *q*, we obtain an *L*-program *q⁻¹* (it has the functionality of an inverse program of *Q*-program *q*).

About the transformers The equations shown above say nothing about the efficiency of the transformed programs. This depends on the transformation power of each transformer. Indeed, each equation can be realized using a trivial program composer or a trivial program specializer:

$$komp_{triv} \equiv \lambda[p, q].\lambda[x, y].\llbracket p \rrbracket [\llbracket q \rrbracket x, y] \quad (31)$$

$$spec_{triv} \equiv \lambda[p, m, x_1, \dots, x_m].\lambda[y_1, \dots, y_n].\llbracket p \rrbracket [x_1, \dots, x_m, y_1, \dots, y_n] \quad (32)$$

In practice, we expect the transformers to be non-trivial. A non-trivial program composer generates an efficient composition of p and q by removing intermediate data structures, redundant computations, and other interface code (e.g., [50, 52]). A non-trivial program specialization recognizes which of p 's computations can be precomputed so as to yield an efficient residual program p_x (e.g., [30]).

The question raised by the equations above is operational: to what extent can the constructions be optimized by existing transformers? If it is not possible with current methods, then this is a challenge for future work on program transformation.

3 Principles of Inverse Computation

This section presents the concepts behind the Universal Resolving Algorithm. We discuss the inverse semantics of programs and the key concepts of the algorithm.

Inverse Semantics of Programs The determination, for a program p written in programming language L and output d_{out} , of an input ds_{in} such that $\llbracket p \rrbracket_L ds_{in} = d_{out}$ is inverse computation. When a program p is not injective, or additional information about the input is available, we may want to restrict the search space of the input for a given output. Similarly, we may also want to specify a set of output values, instead of fixing a particular value. We do so by specifying the input and output domains using an *input-output class* cls_{io} . A class is a finite representation of a possibly infinite set of values. Let $\lceil cls_{io} \rceil$ be the set of values represented by cls_{io} , then a correct solution Inv to an inversion problem is specified by

$$Inv(L, p, cls_{io}) = \{ (ds_{in}, d_{out}) \mid (ds_{in}, d_{out}) \in \lceil cls_{io} \rceil, \llbracket p \rrbracket_L ds_{in} = d_{out} \} \quad (33)$$

where L is a programming language, p is an L -program, and cls_{io} is an input-output class. The *universal solution* $Inv(L, p, cls_{io})$ for the given inversion problem is the largest subset of $\lceil cls_{io} \rceil$ such that $\llbracket p \rrbracket_L ds_{in} = d_{out}$ for all elements (ds_{in}, d_{out}) of this subset. An *existential solution* picks one of the elements of the universal solution as an answer. Note that computing an existential solution is a special case of computing a universal solution (the search stops after finding the first solution).

Inverse Computation In general, inverse computation using an inverse interpreter *invint* for L takes the form

$$\llbracket invint \rrbracket [p, cls_{io}] = ans \quad (34)$$

where p is an L -program and cls_{io} is an input-output class. We say, cls_{io} is a *request* for inverse computation of L -program p . When designing an algorithm for inverse computation, we need to choose a concrete representation of the input-output class cls_{io} and the solution set ans . In this paper we use S-expressions known from Lisp as the value domain, and represent the search space cls_{io} by

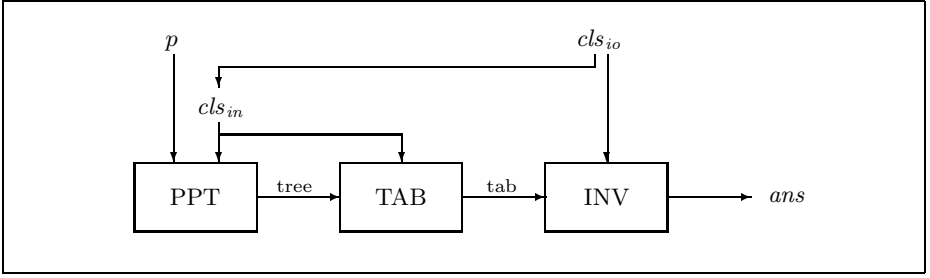


Fig. 1. Conceptual approach: three steps to inverse computation

expressions with variables and restrictions. This is a simple and elegant way to represent subsets of our value domain. (Other algorithms for inverse computation may choose other representations.)

The *Universal Resolving Algorithm* (URA) [5, 3] is an algorithm for inverse computation in a first-order functional language. The answer produced by URA is a set of substitution-restriction pairs $ans = \{(\theta_1, \hat{r}_1), \dots\}$ which represents set *Inv* for the given inversion problem. More formally, the correctness of the answer produced by URA is given by

$$\bigcup_i [(cls_{io}/\theta_i)/\hat{r}_i] = Inv(L, p, cls_{io}) \quad (35)$$

where $(cls_{io}/\theta_i)/\hat{r}_i$ narrows the pairs of values represented by cls_{io} by applying substitution θ_i to cls_{io} and adding restriction \hat{r}_i to the domains of the free variables. The set representation and the operations on it will be explained in Sect. 4. Our algorithm produces a universal solution, hence the first word of its name.

An Approach to Inverse Computation Inverse computation can be organized into three steps: walking through a perfect process tree (PPT), tabulating the input and output (TAB), and extracting the answer to the inversion problem from the table (INV). This organization is shown in Fig. 1. In practice, the three steps can be carried out in a single phase. However, we shall not be concerned with different implementation techniques in this paper.

Our approach is based on the notion of a *perfect process tree* [15] which represents the computation of a program with *partially specified input* (class cls_{in} taken from cls_{io}) by a tree of all possible computation traces. Each fork in a perfect tree partitions the input class cls_{in} into disjoint and exhaustive subclasses. The algorithm then constructs, breadth-first and lazily, a perfect process tree for a given program p and input class cls_{in} . Note that we first construct a forward trace of the computation given p and cls_{in} , and then extract the solution to the backward problem using cls_{io} . The construction of a process tree is similar to unfolding in partial evaluation where a computation is traced under partially specified input (*cf.* [28]).

In the next section we present each of the three steps in more detail:

1. **Perfect Process Tree:** tracing program p under standard computation with input class cls_{in} taken from cls_{io} .
2. **Tabulation:** forming the table of input-output pairs from the perfect process tree and class cls_{in} .
3. **Inversion:** extracting the answer for the desired output given by cls_{io} from the table of input-output pairs.

Since our method for inverse computation is sound and complete [5], and since the source language of our algorithm is a universal programming language, which follows from the fact that the Universal Turing Machine can be programmed in it, we can apply inverse computation to any computable function. Thus our method for inverse computation has full generality.

4 A Complete Example

As an example, consider a program that replaces symbols in a symbol list by a new value given in a replacement list. Program ‘findrep’ in Fig. 2 is tail-recursive and returns the new symbol list in reversed order. For instance, applying the program to symbol list $s = [1, 2]$ and replacement list² $rr = [2, 3]$ returns the reversed symbol list $[3, 1]$ where 2 has been replaced by 3:

$$[\text{findrep}] [[1, 2], [2, 3]] = [3, 1] .$$

The program is written in TSG, a first-order functional language. The language is a typed dialect of S-Graph [15]. The body of a function consists of a term which is either a function call, a conditional or an expression. The calls are restricted to tail-recursion. Values can be tested and/or decomposed in two ways. Test **eqa?** checks the equality of two atoms, and (**cons?** x h t a) works like pattern matching: if the value of variable x is a pair $d_1:d_2$, then variable h is bound to head d_1 and variable t to tail d_2 ; otherwise the value (an atom) is bound to variable a . By ‘ $_$ ’ we denote anonymous variables.

An Inversion Problem Suppose we have output $[3, 1]$, and we want to find all possible inputs which produce this output. To show a complete example of inverse computation, let us limit the search space to lists of length two. We specify the symbol list as a list of two *atoms*, $[Xa_1, Xa_2]$, and the replacement list as a list of two *S-expressions*, $[Xe_3, Xe_4]$. Placeholders Xa_i, Xe_i are called *configuration variables* (c-variables). We specify the input and output domains by the input-output class

$$cls_{io} = \langle (\underbrace{[[Xa_1, Xa_2], [Xe_3, Xe_4]]}_{\widehat{ds}_{in}}, \underbrace{[3, 1]}_{\widehat{d}_{out}}), \emptyset \rangle$$

² In general, $rr = [u_1, v_1, \dots, u_n, v_n]$ where u_i is a symbol and v_i its replacement.

where \widehat{ds}_{in} is the partially specified input, \widehat{d}_{out} is the desired output, and the restriction on the domain of c-variables is empty (\emptyset). Inverse computation with URA takes the form:

$$\llbracket ura \rrbracket [\text{findrep}, cls_{io}] = ans .$$

As the answer *ans* of inverse computation, we expect a (possibly infinite) sequence of substitution-restriction pairs for the c-variables occurring in cls_{io} (cf. Eq. 35).

Set Representation Expressions with variables, called *c-expressions*, represent sets of values by means of two types of variables: *ca-variables* Xa_i which range over atoms, and *ce-variables* Xe_i which range over S-expressions.

For instance, consider the input class $cls_{in} = \langle \widehat{ds}_{in}, \emptyset \rangle$ which we take from cls_{io} above. The set of values represented by cls_{in} , denoted $\llbracket cls_{in} \rrbracket$, is

$$\llbracket cls_{in} \rrbracket = \{ \llbracket [da_1, da_2], [d_3, d_4] \rrbracket \mid da_1, da_2 \in \text{DAval}, d_3, d_4 \in \text{Dval} \}$$

where DAval is the set of all atoms and Dval is the set of all values (S-expressions). In our example, cls_{io} and cls_{in} contain an empty restriction (\emptyset). In general, a *restriction* is a finite set of non-equalities on ca-variables. A *non-equality* takes the form $(Xa_i \# Xa_j)$ or $(Xa_i \# 'A)$, where 'A is an arbitrary atom. It specifies a set of values to which a ca-variable must not be equal. Suppose we add restriction $\{(Xa_1 \# Xa_2)\}$ to cls_{in} , then the set of values represented by the new class is narrowed to

$$\llbracket cls'_{in} \rrbracket = \{ \llbracket [da_1, da_2], [d_3, d_4] \rrbracket \mid da_1, da_2 \in \text{DAval}, d_3, d_4 \in \text{Dval}, da_1 \neq da_2 \} .$$

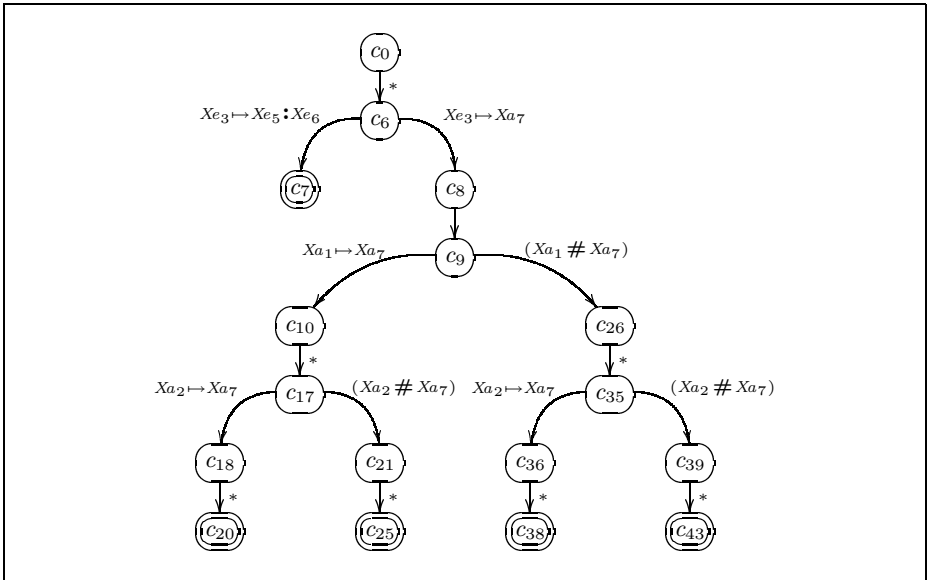
To restrict a class to represent an empty set of values, we add to it a *contradiction* $(Xa_1 \# Xa_1)$: there exist no value which satisfies this non-equality. Later, when we construct a perfect process tree, we shall see how restrictions are used. We need only non-equalities on ca-variables because our language can only test atoms for non-equality.³ C-expressions and restrictions are our main method for manipulating infinite sets of values using a finite representation.

1. Perfect Process Trees A computation is a linear sequence of states and transitions. Each state and transition in a deterministic computation is fully defined. Tracing a program with partially specified input (cls_{in}) may confront us with tests that depend on unspecified values (represented by c-variables), and we have to consider the possibility that either branch is entered with some input value. This leads to traces that branch at conditionals that depend on unspecified values. Tracing a computation is called *driving* in supercompilation [50]; the method used below is *perfect driving* [15]. (A variant is *positive driving* [46].)

To trace a program with partially specified input we use *configurations* of the form $\langle (t, \widehat{\sigma}), \widehat{r} \rangle$ where t is a program term, $\widehat{\sigma}$ an environment binding program

³ An extension to express non-equalities between ce-variables can be found in [43].

<pre> (define findrep [s, rr] t₁ (call scan [s, rr, []])) (define scan [s, rr, out] t₂ (if (cons? s h s _) t₃ (if (cons? h _ _ a) t₄ 'Error:atom_expected t₅ (call find [a, s, rr, rr, out])) t₆ out)) </pre>	<pre> (define find [a, s, r, rr, out] t₇ (if (cons? r h r _) t₈ (if (cons? h _ _ b) t₉ 'Error:atom_expected t₁₀ (if (cons? r c r _) t₁₁ (if (eqa? a b) t₁₂ (call scan [s, rr, c:out]) t₁₃ (call find [a, s, r, rr, out])) t₁₄ 'Error:pair_expected)) t₁₅ (call scan [s, rr, a:out]))) </pre>
--	---

Fig. 2. TSG-program ‘findrep’**Fig. 3.** Perfect process tree for program ‘findrep’

variables to c-expressions, and \hat{r} is a restriction on the domain of c-variables. We call $\hat{\sigma}$ a *c-environment* to stress that this environment binds program variables to c-expressions instead of values.

Let us trace our example program. We begin with a tree whose single node is labeled with the initial configuration c_0 . The initial term t_0 is a call to function findrep, the initial c-environment binds program variables s and rr to the corresponding c-expressions, and the initial restriction is empty.

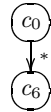
$$c_0 = \langle \underbrace{\langle (\text{call findrep } [s, rr]) \rangle}_{\text{term } t_0}, \underbrace{\langle [s \mapsto [Xa_1, Xa_2], rr \mapsto [Xe_3, Xe_4]] \rangle}_{\text{c-environment}}, \underbrace{\langle \emptyset \rangle}_{\text{restr.}} \rangle$$

c	t	a	s	b	h	c	r	rr	out	\widehat{r}
c_0	t_0		$[Xa_1, Xa_2]$					$[Xe_3, Xe_4]$		\emptyset
c_6	t_8	Xa_1	$[Xa_2]$		Xe_3		$[Xe_4]$	$[Xe_3, Xe_4]$	$[]$	\emptyset
<u>c_7</u>	t_9	Xa_1	$[Xa_2]$		$Xe_5:Xe_6$		$[Xe_4]$	$[(Xe_5:Xe_6), Xe_4]$	$[]$	\emptyset
c_8	t_{10}	Xa_1	$[Xa_2]$	Xa_7	Xa_7		$[Xe_4]$	$[Xa_7, Xe_4]$	$[]$	\emptyset
c_9	t_{11}	Xa_1	$[Xa_2]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[]$	\emptyset
c_{10}	t_{12}	Xa_7	$[Xa_2]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[]$	\emptyset
c_{17}	t_{11}	Xa_2	$[]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[Xe_4]$	\emptyset
c_{18}	t_{12}	Xa_7	$[]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[Xe_4]$	\emptyset
<u>c_{20}</u>	t_6		$[]$					$[Xa_7, Xe_4]$	$[Xe_4, Xe_4]$	\emptyset
c_{21}	t_{13}	Xa_2	$[]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[Xe_4]$	\widehat{r}_{21}
<u>c_{25}</u>	t_6		$[]$					$[Xa_7, Xe_4]$	$[Xa_2, Xe_4]$	\widehat{r}_{21}
c_{26}	t_{13}	Xa_1	$[Xa_2]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[]$	\widehat{r}_{26}
c_{35}	t_{11}	Xa_2	$[]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[Xa_1]$	\widehat{r}_{26}
c_{36}	t_{12}	Xa_7	$[]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[Xa_1]$	\widehat{r}_{26}
<u>c_{38}</u>	t_6		$[]$					$[Xa_7, Xe_4]$	$[Xe_4, Xa_1]$	\widehat{r}_{26}
c_{39}	t_{13}	Xa_2	$[]$	Xa_7	Xa_7	Xe_4	$[]$	$[Xa_7, Xe_4]$	$[Xa_1]$	\widehat{r}_{39}
<u>c_{43}</u>	t_6		$[]$					$[Xa_7, Xe_4]$	$[Xa_2, Xa_1]$	\widehat{r}_{39}
where $t_0 = (\text{call findrep } [s, rr])$ $\widehat{r}_{21} = \{(Xa_2 \# Xa_7)\}$ $\widehat{r}_{26} = \{(Xa_1 \# Xa_7)\}$ $\widehat{r}_{39} = \{(Xa_1 \# Xa_7), (Xa_2 \# Xa_7)\}$										

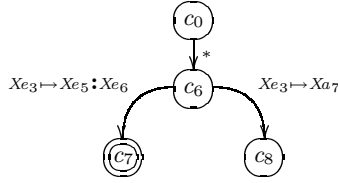
Fig. 4. Selected configurations for program ‘findrep’

All subsequent configurations at branching nodes are listed in Fig. 4. Each row shows the name of the configuration, the term, the contents of the c-environment and the restriction. Underlined names denote terminal nodes. If a program variable does not occur in a c-environment then its entry is empty.

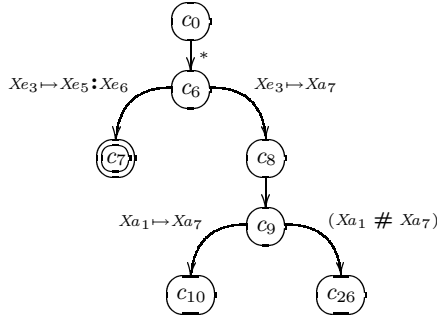
Tracing starts in the root, and then proceeds with one of the successor configurations depending on the shape of the values in the c-environment. The first test we encounter in our program is (**cons?** s h s -) in term t_2 . It tests whether the value of program variable s is a pair. Since s is bound to list $[Xa_1, Xa_2]$, the test can be decided. After the first few steps we obtain a linear sequence of configurations from c_0 to c_6 :



The next test is (**cons?** h - - b) in term t_8 of configuration c_6 . Since program variable h is bound to c-variable Xe_3 (Fig. 4, c_6), which represents an arbitrary value, we have to consider two possibilities: Xe_3 is a pair of the form $Xe_5:Xe_6$ or an atom Xa_7 . These assumptions lead to two new configurations, c_7 and c_8 . They are expressed by attaching substitutions $Xe_3 \mapsto Xe_5:Xe_6$ and $Xe_3 \mapsto Xa_7$ to the corresponding edges. We call such a pair a *splitted*.



Configuration c_7 is a terminal node with output $t_9 = \text{'Error:atom_expected'}$. Tracing configuration c_8 one step leads to configuration c_9 . The test **(eqa? a b)** in term t_{11} of configuration c_9 depends on two unknown atoms represented by Xa_1 and Xa_7 (Fig. 4, c_9). Assuming that both atoms are identical, expressed by substitution $Xa_1 \mapsto Xa_7$, leads to configuration c_{10} . Assuming that the atoms are not equal, expressed by restriction $(Xa_1 \# Xa_7)$, leads to configuration c_{26} :



Repeating these steps leads to a finite tree (in general, the tree may be infinite). Informally, the perfect process tree represents all computation traces of program `findrep` with cls_{in} , where branchings in the tree correspond to different assumptions about the c-variables in cls_{in} . Each configuration in the tree represents a set of computation states, and each edge corresponds to a set of transitions. In each branching configuration, two contractions are performed which split the set of states into two disjoint sets (see [29]). The complete tree is shown in Fig. 3.

We use only *perfect splits* in the construction of a process tree, hence its name. A perfect split guarantees that no elements will be lost, and no elements will be added when partitioning a set. The four perfect splits needed for tracing TSG-programs are

1. $(\kappa_{id}, \kappa_{contra})$
2. $([Xa_1 \mapsto 'A], \{(Xa_1 \# 'A)\})$
3. $([Xa_1 \mapsto Xa_2], \{(Xa_1 \# Xa_2)\})$
4. $([Xe_3 \mapsto Xa^\diamond], [Xe_3 \mapsto Xe_h^\diamond : Xe_t^\diamond])$

where κ_{id} is an empty substitution, κ_{contra} is a restriction with a contradiction and $'A$ is an arbitrary atom. C-variables Xa_1 , Xa_2 and Xe_3 occur in the c-expression we split, and Xa^\diamond , Xe_h^\diamond and Xe_t^\diamond are “fresh” c-variables.

2. Tabulation To build a table of input-output pairs, we follow each path from the root of the tree to a terminal node. All contractions encountered on such a

path are applied to cls_{in} , and the new class cls_i is entered in the table (Fig. 5) together with the output expression \widehat{d}_i of the corresponding terminal node.

Each class cls_i in the table represents a set of input values all of which lead to an output value that lies in \widehat{d}_i . Since all splits in the tree are perfect, $\lceil cls_{in} \rceil$ is partitioned into disjoint sets of input values: $\lceil cls_i \rceil \cap \lceil cls_j \rceil = \emptyset$ where $0 < i < j$. This means that each input value lies, at most, in one class cls_i . Note that the partitioning can also be carried out while constructing the tree. In general, when the tree has infinitely many terminal nodes, the table contains infinitely many entries.

Input class $cls_i = \langle \widehat{ds}_i, \widehat{r}_i \rangle$	Output expression \widehat{d}_i
$\langle \llbracket [Xa_1, Xa_2], [(Xe_5: Xe_6), Xe_4] \rrbracket, \emptyset \rangle$	'Error:atom_expected'
$\langle \llbracket [Xa_7, Xa_7], [Xa_7, Xe_4] \rrbracket, \emptyset \rangle$	$[Xe_4, Xe_4]$
$\langle \llbracket [Xa_7, Xa_2], [Xa_7, Xe_4] \rrbracket, \{(Xa_2 \# Xa_7)\} \rangle$	$[Xa_2, Xe_4]$
$\langle \llbracket [Xa_1, Xa_7], [Xa_7, Xe_4] \rrbracket, \{(Xa_1 \# Xa_7)\} \rangle$	$[Xe_4, Xa_1]$
$\langle \llbracket [Xa_1, Xa_2], [Xa_7, Xe_4] \rrbracket, \{(Xa_1 \# Xa_7), (Xa_2 \# Xa_7)\} \rangle$	$[Xa_2, Xa_1]$

Fig. 5. Tabulation of program ‘findrep’ for $cls_{in} = \langle \llbracket [Xa_3, Xa_4], [Xe_3, Xe_4] \rrbracket, \emptyset \rangle$

3. Inversion Finally, we extract the solution to our inversion problem by intersecting each entry in the table with the given io-class cls_{io} . This can be done by unifying each pair $(\widehat{ds}_i, \widehat{d}_i)$ with $(\widehat{ds}_{in}, \widehat{d}_{out})$, where $cls_{io} = \langle (\widehat{ds}_{in}, \widehat{d}_{out}), \widehat{r}_{io} \rangle$. If the unification succeeds with the most general unifier θ , and the restriction $\widehat{r} = (\widehat{r}_i \cup \widehat{r}_{io})/\theta$ contains no contradiction, then pair (θ, \widehat{r}) is in the answer *ans*.

Three examples are given in Fig. 6. They show the application of URA to three input-output classes and the corresponding solutions. In the first case, given output $\widehat{d}_{out} = [3, 1]$, we obtain three substitution-restriction pairs from the five entries in the table in Fig. 6 (unification of $[3, 1]$ with $[Xe_4, Xe_4]$ does not succeed, neither does unification of $[3, 1]$ with ‘Error:atom_expected’).

In the second case, given output $\widehat{d}_{out} = \text{'Error:pair_expected'}$, the answer tells us which input values lead to the undesired error (namely, when Xe_3 in the replacement list is a pair of values). In the last case, given output $\widehat{d}_{out} = [1]$, the answer is empty. There exists no input in $\lceil cls_{in} \rceil$ which leads to output $[1]$.

Two Important Operations Two key operations in the development of a process tree are:

1. Applying perfect splits at branching configurations.
2. Cutting infeasible branches in the tree.

Let us discuss these two operations. The second operation, *cutting infeasible branches*, is important because an infeasible branch in a process tree is either non-terminating, or terminating in an unreachable node. When infeasible branches

1. $\llbracket \text{ura} \rrbracket [\text{findrep}, \langle \langle [[Xa_1, Xa_2], [Xe_3, Xe_4]], [3, 1] \rangle, \emptyset \rangle] = [$
 $([Xa_1 \mapsto Xa_7, Xa_2 \mapsto 3, Xe_3 \mapsto Xa_7, Xe_4 \mapsto 1], \{(3 \# Xa_7)\}),$
 $([Xa_1 \mapsto 1, Xa_2 \mapsto Xa_7, Xe_3 \mapsto Xa_7, Xe_4 \mapsto 3], \{(1 \# Xa_7)\}),$
 $([Xa_1 \mapsto 1, Xa_2 \mapsto 3, Xe_3 \mapsto Xa_7, Xe_4 \mapsto Xe_4], \{(1 \# Xa_7), (3 \# Xa_7)\})]$
2. $\llbracket \text{ura} \rrbracket [\text{findrep}, \langle \langle [[Xa_1, Xa_2], [Xe_3, Xe_4]], \text{'Error:atom_expected'} \rangle, \emptyset \rangle] = [$
 $([Xa_1 \mapsto Xa_1, Xa_2 \mapsto Xa_2, Xe_3 \mapsto Xe_5:Xe_6, Xe_4 \mapsto Xe_4], \emptyset)]$
3. $\llbracket \text{ura} \rrbracket [\text{findrep}, \langle \langle [[Xa_1, Xa_2], [Xe_3, Xe_4]], [1] \rangle, \emptyset \rangle] = []$

Fig. 6. Solutions for program ‘findrep’ and three requests

are not detected at branching points, the correctness of the solution can be guaranteed, but inverse computation becomes less terminating and less efficient. The risk of entering non-terminating branches which are infeasible makes inverse computation less terminating (but completeness of the solution can be preserved). A terminal node reached via an infeasible branch can only be associated with an empty set of input (but soundness of the solution is preserved). In both cases unnecessary work is performed.

The correctness of the solution cannot be guaranteed without the first operation, *applying perfect splits*, because the missing information can lead to a situation where an empty set of input cannot be detected, neither during the development of the tree nor in the solution. Suppose we reach a terminal node, then the input class cls_i associated with that terminal node may tell us that there exists an input which computes a certain output, even though this is not true.

In short, the first operation is essential to guarantee the correctness of the solution and the second operation improves the termination and efficiency of the algorithm. When building a process tree, we perform both operations (see example ‘findrep’ above). This is possible because the set representation we use expresses structural properties of values (S-expressions) and all tests on it which occur during the development of a process tree are decidable.

Extensions to other value domains may involve the use of constraint systems [37] or theorem proving as in [13]. When the underlying logic of the set representation is not decidable for certain logic formulas (or too time consuming to prove), infeasible branches cannot always be detected. For example, this is the case when we use a perfect tree developer for Lisp based on generalized partial computation (GPC) [13]. In this case, the solution of inverse computation may contain elements which represent empty sets of input, but the solution is correct.

Termination In general, inverse computation is undecidable, so an algorithm for inverse computation cannot be sound, complete, and terminating at the same time. Our algorithm is sound and complete with respect to the solutions defined by a given program, but not always terminating. If a source program terminates on a given input and produces the desired output, our algorithm will find that input. Each such input will be found in finite time [5].

<pre> (define inorder [t] (call tree [t, [], []])) (define tree [t, rest, out] (if (cons? t l cr leaf) (call tree [l, (cr:rest), out]) (call center [rest, (leaf:out)]))) </pre>	<pre> (define center [rest, out] (if (cons? rest cr rest _) (if (cons? cr center r _) (call tree [r, rest, (center:out)]) 'Error:tree_expected) out)) </pre>
---	--

Fig. 7. TSG-program ‘inorder’

Inverse computation does not always terminate because the search for inputs can continue infinitely, even when the number of inputs that lead to the desired output is finite (*e.g.*, the search for a solution continue along an infinite branch in the process tree). Our algorithm terminates *iff* the process tree is finite [5].

Algorithmic Aspects The algorithm for inverse computation is fully implemented [5] in Haskell and serves our experimental purposes quite well. The program constructs, breadth-first and lazily, a perfect process tree. In particular, Haskell’s lazy evaluation strategy allowed us to use a modular approach very close to the organization shown in Fig. 1 (where the development of the perfect process tree, the tabulation, and the inversion of the table are conveniently separated). A more efficient implementation may merge these modules, and other designs may make the algorithm ‘more’ terminating (*e.g.*, by detecting certain finite solution sets or cutting more infinite branches).

In general, since the Universal Resolving Algorithm explores sets of program traces, inverse computation of a program p using it will be more efficient than the generate-and-test approach [38] (which generates all possible ground input and tests the corresponding output), but less efficient than running the corresponding (non-trivial) inverse program p^{-1} . This corresponds to the familiar trade-off between interpretation and translation of programs.

5 Demonstration

This section shows three examples of inverse computation including inverse computation of an interpreter for a small imperative programming language.

5.1 Inorder Traversal of a Binary Tree

Consider a program that traverses a binary tree and returns a list of its nodes. Given a list of nodes, inverse computation then produces all binary trees that lead to that list. The classic example is to construct a binary tree given its inorder and preorder traversal [8, 51]. Our example uses the inorder traversal [25].

The program ‘inorder’ is written in TSG (Fig. 7). It performs an inorder traversal of a binary tree and, for simplicity, returns the list of nodes in reversed

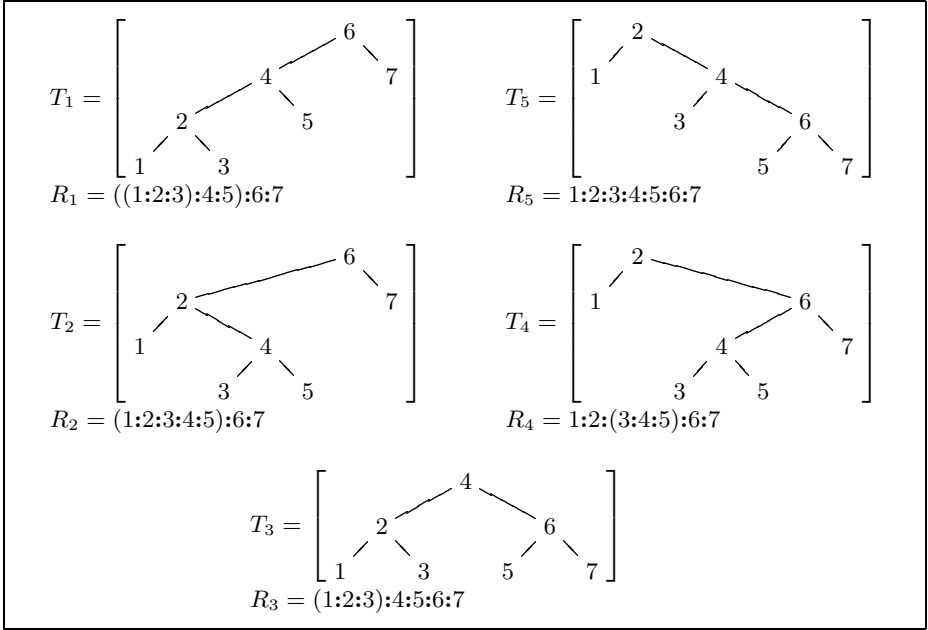


Fig. 8. Program ‘inorder’: representation of binary trees

order. Binary trees are represented by S-expressions. For example, $1:2:3$ represents a binary tree with left child 1, root 2 and right child 3. The representation (R_i) of all five binary trees (T_i) that can be built from seven nodes is shown in Fig. 8. For all $R_i, i = 1..5$, we have:

$$\llbracket \text{inorder} \rrbracket R_i = [7, 6, 5, 4, 3, 2, 1]$$

In other words, the inorder traversal of all binary trees T_i returns the same node list. Given this node list, inverse computation of the program ‘inorder’ by URA returns all five binary trees which can be built from that node list. URA finds all answers, but does not terminate. It continues the search after all answers have been found. The time⁴ to find the five binary trees is 0.13 sec.

$$\begin{aligned} \llbracket \text{ura} \rrbracket [\text{inorder}, \langle ([Xe_1], [7, 6, 5, 4, 3, 2, 1]), \emptyset \rangle] = [\\ & ([Xe_1 \mapsto ((1:2:3):4:5):6:7], \emptyset), \quad ([Xe_1 \mapsto (1:2:3:4:5):6:7], \emptyset), \\ & ([Xe_1 \mapsto (1:2:3):4:5:6:7], \emptyset), \quad ([Xe_1 \mapsto 1:2:(3:4:5):6:7], \emptyset), \\ & ([Xe_1 \mapsto 1:2:3:4:5:6:7], \emptyset), \\ & \dots \end{aligned}$$

⁴ All running times for Intel Pentium-IV-1.4GHz, RAM 512MB, MS Windows Me, Red Hat Cygwin 1.3.4-2 and The Glorious Glasgow Haskell Compilation System 5.02.1.

```

(define isWalk[w, g]
  (call wloop[w, g, [], w]))

(define wloop[w, g, pw, cc]
  (if (cons? w wh wt a_)
    (if (cons? wh e_ e_ ac)
      'Error:atom_expected
      (call check_pw[pw, ac, wt, g, pw, cc]))
    'True))

(define check_cc[x, ac, w, g, pw, cc]
  (if (cons? x xh xt a_)
    (if (cons? xh e_ e_ ac)
      'Error:bad_graph_definition
      (if (eqa? ax ac)
        (call new_cc[g, ac, w, g, pw])
        (call check_cc[xt, ac, w, g, pw, cc])))
    'False))

(define check_pw[x, ac, w, g, pw, cc]
  (if (cons? x xh xt a_)
    (if (cons? xh e_ e_ ac)
      'Error:internal
      (if (eqa? ax ac)
        'False
        (call check_pw[xt, ac, w, g, pw, cc]))))
    (call check_cc[cc, ac, w, g, pw, cc])))

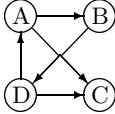
(define new_cc[x, ac, w, g, pw]
  (if (cons? x xh xt a_)
    (if (cons? xh xhh cc a_)
      (if (cons? xhh e_ e_ ac)
        'Error:bad_graph_definition
        (if (eqa? ax ac)
          (call wloop[w, g, (ac:pw), cc])
          (call new_cc[xt, ac, w, g, pw])))
      'Error:bad_graph_definition)
    'False))

```

Fig. 9. TSG-program 'isWalk'

5.2 Acyclic Walk in a Directed Graph

As another example, consider a program that checks whether a walk can be realized without cycles in a directed graph. The program is written as a predicate that takes a walk and a graph as input, and returns 'True', 'False', or an error atom as output. The program 'isWalk' is written in TSG (Fig. 9). It takes a walk as a list of nodes, and a directed graph as an association list where the first element is the node and the remaining elements are the reachable nodes:

gr = [['A', 'B', 'C'],		[[isWalk]] [['A'], gr] = 'True
['B', 'D'],		[[isWalk]] [['A', 'B', 'D'], gr] = 'True
['C'],		[[isWalk]] [['A', 'B', 'D', 'A'], gr] = 'False
['D', 'A', 'C']]		[[isWalk]] [['B', 'D'], gr] = 'Error:atom_expected

Given a graph and one of the outputs, inverse computation finds all walks in the graph which produce the desired output. We consider two tasks for inverse computation. Here the perfect process tree is always finite.

Task 1: Find all cycle-free walks in graph gr. URA enumerates all 17 answers and terminates. The time to find the answers is 0.15 sec. By convention a walk Xe_1 is terminated by an atom 'Nil'. The answers show that, in fact, it can be any atom without changing the output of the program.

$$\begin{aligned}
 \llbracket ura \rrbracket [\text{isWalk}, \langle ([Xe_1, \text{gr}], \text{'True}), \emptyset \rangle] = [\\
 ([Xe_1 \mapsto Xa_4], \emptyset), \quad ([Xe_1 \mapsto \text{'A':}Xa_{10}], \emptyset),
 \end{aligned}$$

$$\begin{aligned}
& ([Xe_1 \mapsto 'B: Xa_{10}], \emptyset), & ([Xe_1 \mapsto 'C: Xa_{10}], \emptyset), \\
& ([Xe_1 \mapsto 'D: Xa_{10}], \emptyset), & ([Xe_1 \mapsto 'A: 'B: Xa_{16}], \emptyset), \\
& ([Xe_1 \mapsto 'A: 'C: Xa_{16}], \emptyset), & ([Xe_1 \mapsto 'D: 'A: Xa_{16}], \emptyset), \\
& ([Xe_1 \mapsto 'B: 'D: Xa_{16}], \emptyset), & ([Xe_1 \mapsto 'D: 'C: Xa_{16}], \emptyset), \\
& ([Xe_1 \mapsto 'A: 'B: 'D: Xa_{22}], \emptyset), & ([Xe_1 \mapsto 'B: 'D: 'A: Xa_{22}], \emptyset), \\
& ([Xe_1 \mapsto 'D: 'A: 'B: Xa_{22}], \emptyset), & ([Xe_1 \mapsto 'D: 'A: 'C: Xa_{22}], \emptyset), \\
& ([Xe_1 \mapsto 'B: 'D: 'C: Xa_{22}], \emptyset), & ([Xe_1 \mapsto 'A: 'B: 'D: 'C: Xa_{28}], \emptyset), \\
& ([Xe_1 \mapsto 'B: 'D: 'A: 'C: Xa_{28}], \emptyset)]
\end{aligned}$$

Task 2: Find all values which do not represent cycle-free walks in the directed graph gr. URA enumerates 54 answers (classes not shown) and, interestingly, also terminates. These classes represent all (infinitely many) values which do not represent cycle-free walks in gr. The time to find the answers is 0.17 sec.

$$\llbracket ura \rrbracket [\text{isWalk}, \langle ([Xe_1, \text{gr}], \text{'False}), \emptyset \rangle] = [\dots \text{omitted} \dots]$$

5.3 Interpreter for an Imperative Language

Consider the small imperative programming language MP [45] with assignments, conditionals, and while-loops. An MP-program consists of a parameter list, a variable declaration and a sequence of statements. The value domain is the set of S-expressions. An MP-program operates over a global store. The semantics is conventional Pascal-style semantics.

We implemented an MP-interpreter, intMP, in TSG (309 lines of pretty-printed program text; 30 functions in TSG) and rewrote all three example programs in MP (findrep, inorder, isWalk). The three MP-programs are shown in Figs. 10, 11, and 12. The text of the MP-interpreter is too long to be included.

In order to compare the application of URA to the three MP-programs (via the MP-interpreter) with the direct application of URA to the corresponding TSG-programs, we repeated the six experiments from the sections above.

The direct application of URA to a TSG-program *prog* takes the form:

$$\llbracket ura \rrbracket [prog, \langle (\widehat{ds}_{in}, \widehat{d}_{out}), \emptyset \rangle] = ans.$$

Two-level inverse computation of the corresponding MP-program *progMP* via the MP-interpreter intMP is performed by the following application which is an implementation of Eq. 15:

$$\llbracket ura \rrbracket [\text{intMP}, \langle ([progMP, \widehat{ds}_{in}], \widehat{d}_{out}), \emptyset \rangle] = ans'.$$

The values for *progMP*, \widehat{ds}_{in} , \widehat{d}_{out} , and the running times (in seconds) for one-level (t_1) and two-level (t_2) inverse computations are as follows:

```

program
  parameters input, find_rep;
  variables f_r, what, head, result;
begin
  while (input) begin
    head := car(input);
    input := cdr(input);
    f_r := find_rep;
    while (f_r) begin
      what := car(f_r);
      f_r := cdr(f_r);
      if (isEqual?(head, what)) begin
        head := car(f_r);
        f_r := []; /* break f_r-loop */
      end else f_r := cdr(f_r);
    end;
    result := cons(head, result);
  end;
return result;
end.

```

Fig. 10. MP-program ‘findrepMP’

$progMP$	\hat{d}_{in}	\hat{d}_{out}	t_1	t_2	t_2/t_1
findrep	$[[Xa_1, Xa_2], [Xe_3, Xe_4]]$	$[3, 1]$	0.15	0.36	2.40
findrep	$[[Xa_1, Xa_2], [Xe_3, Xe_4]]$	<code>'Error:atom_expected</code>	0.11	0.33	3.00
findrep	$[[Xa_1, Xa_2], [Xe_3, Xe_4]]$	$[1]$	0.08	0.33	4.13
inorder	$[Xe_1]$	$[7, 6, 5, 4, 3, 2, 1]$	0.13	3.55	27.31
isWalk	$[Xe_1, gr]$	<code>'True</code>	0.15	5.00	33.33
isWalk	$[Xe_1, gr]$	<code>'False</code>	0.17	5.12	30.12

In each case, the answer ans' obtained via the MP-interpreter was identical to the answer ans obtained by the direct application of URA – modulo reordering of elements in the printed list. Such a close correspondence between the answers may not always be the case, but as our experiments indicate, it is not infeasible either. As our results show, the answer of two-level inverse computation can have the same quality as the answer of one-level inverse computation. On the other hand, and this is what can be expected, the extra level of interpretation increases the running time by an order of magnitude. Optimization by program specialization outlined in Sect. 2 has the potential to reduce the running times (*e.g.*, as suggested in Eq. 29). This will be a task for future work.

```

program
  parameters t;
  variables rest, out;
begin
  while (true) begin
    if (t) begin
      rest := cons(cdr(t), rest); /* center and right */
      t := car(t); /* left */
    end else begin /* t is leaf */
      out := cons(t, out);
      if (rest)
        if (car(rest)) begin
          out := cons(car(car(rest)), out); /* center */
          t := cdr(car(rest)); /* right */
          rest := cdr(rest);
        end else return 'Error:tree_expected
      else return out;
    end;
  end;
end.

```

Fig. 11. MP-program 'inorderMP'

6 Related Work

An early result [48] of inverse computation in a functional language was obtained in 1972 when *driving* [50], a unification-based program transformation technique, was shown to perform subtraction given an algorithm for binary addition (see [1, 22]). The Universal Resolving Algorithm presented in this paper is derived from *perfect driving* [15] and is combined with a mechanical extraction of the answers (*cf.* [1, 42]) giving our algorithm the power comparable to SLD-resolution, but for a first-order, functional language (*cf.* [20]). The complete algorithm is given in [5]. A program analysis for inverting programs which makes use of *positive driving* [46] was proposed in [44]. The use of driving for theorem proving is studied in [49] and its relation to partial evaluation in [29]. Reference [21] contains a detailed bibliography on driving and supercompilation.

Logic programming inherently supports inverse computation [34]. The use of an appropriate inference procedure permits the determination of any computable answer [36]. It is not surprising that the capabilities of logic programming provided the foundation for many applications [47] in artificial intelligence, program verification and logical reasoning. Connections between logic programming, inverse computation and meta-computation are discussed in [49, 1]; see also [21, 4]. Driving and partial deduction, a technique for program specialization in logic programming, were related in [20].

```

program
  parameters w, g;
  variables c, cc, pw, x, cont;
begin
  pw := []; /* prefix of w */
  cc := w;
  while (w) begin
    c := car(w); /* current node */
    x := pw;
    while (x) /* check loops: c 'elem' pw */
      if (isEqual?(car(x), c)) then return 'False
      else x := cdr(x);
    x := cc;
    cont := true;
    while (cont) /* check: c 'elem' cc */
      if (x) then
        if (isEqual?(car(x), c)) then cont := false
        else x := cdr(x);
      else return 'False;
    x := g;
    cont := true;
    while (cont) /* search c in g, compute next cc */
      if (x) then
        if (isEqual?(car(car(x)), c) then begin
          cc := cdr(car(x)); /* next c must be 'elem' cc */
          cont := false;
        end
        else x := cdr(x);
      else return 'False;
    pw := cons(c, pw);
    w := cdr(w);
  end;
  return 'True;
end.

```

Fig. 12. MP-program 'isWalkMP'

Similar to ordinary programming, there exists no single programming paradigm that would satisfy all needs of inverse programming. New languages emerge as new problems are approached. It is therefore important to develop inversion methods also outside the domain of logic programming. Recently, work in this direction has been done on the integration of the functional and logic programming paradigm using narrowing, a unification-based goal-solving mechanism [26]; for a survey see [6].

Most of the work on program inversion deals with imperative languages (*e.g.*, [8, 10, 24, 25, 39, 51]). Inversion of functional programs has been stud-

ied in [7, 11, 27, 31, 33, 39, 42]. Experiments with program inversion using specialization systems are reported in [22, 40, 19, 35].

7 Conclusion

We presented two tools for inverse programming (inverse interpreter, inverse translator) and showed how they can be related by program specialization and program composition. We described an algorithm for inverse computation in a first-order functional language, discussed its organization and structure, and illustrated our implementation with several examples. Our results show that inverse computation can be performed in a functional programming language and ported to other programming languages via interpreters. We demonstrated this idea for a fragment of an imperative programming language.

The Universal Resolving Algorithm is fully implemented in Haskell which serves our experimental purposes quite well. In particular, Haskell's lazy evaluation strategy allowed us to use a modular approach very close to the theoretical definition of the algorithm (where the development of a perfect process trees, the tabulation, and the inversion of the table are conveniently separated). Clearly, more efficient implementations exist.

Methods for detecting finite solution sets and cutting infinite branches can make the algorithm 'more' terminating which will be useful for a practical application of the method. Techniques from program transformation and logic programming may prove to be useful in this context. Since it is impossible to design an algorithm for inverse computation that always terminates, even when a terminating inverse program is known to exist, we can only expect approximate solutions that do 'better' in terms of efficiency and termination. More work is needed to study the portability of the algorithm to realistic programming languages as suggested by the semantics modifier approach.

Acknowledgements The authors thank the reviewers for thorough and careful reading of the submitted paper and for providing many helpful comments.

References

- [1] S. M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).
- [2] S. M. Abramov, R. Glück. Combining semantics with non-standard interpreter hierarchies. In S. Kapoor, S. Prasad (eds.), *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, LNCS 1974, 201–213. Springer-Verlag, 2000.
- [3] S. M. Abramov, R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse, J. N. Oliveira (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 1837, 187–212. Springer-Verlag, 2000.
- [4] S. M. Abramov, R. Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12(2):171–211, 2001.

- [5] 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.
- [6] E. Albert, G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.
- [7] R. Bird, O. de Moor. *Algebra of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
- [8] W. Chen, J. T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15:1–13, 1990.
- [9] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
- [10] E. W. Dijkstra. EWD671: Program inversion. In *Selected Writings on Computing: A Personal Perspective*, 351–354. Springer-Verlag, 1982.
- [11] D. Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*, 219–221. Morgan Kaufmann, Inc., 1985.
- [12] Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
- [13] Y. Futamura, K. Nogi. Generalized partial computation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 133–151. North-Holland, 1988.
- [14] P. C. Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of Research and Development*, 4:28–35, 1960.
- [15] R. Glück, A. V. Klimov. Occam’s razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, A. Rauzy (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.
- [16] 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.
- [17] R. Glück, A. V. Klimov. A regeneration scheme for generating extensions. *Information Processing Letters*, 62(3):127–134, 1997.
- [18] R. Glück, A. V. Klimov. On the degeneration of program generators by program composition. *New Generation Computing*, 16(1):75–95, 1998.
- [19] R. Glück, M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification (extended abstract). In D. Bjørner, M. Broy, A. V. Zamulin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1755, 93–100. Springer-Verlag, 2000.
- [20] R. Glück, M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, 165–181. Springer-Verlag, 1994.
- [21] R. Glück, M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 137–160. Springer-Verlag, 1996.
- [22] R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, 286–287. ACM Press, 1990.
- [23] C. Green. Application of theorem proving to problem solving. In D. E. Walker, L. M. Norton (eds.), *Int. Joint Conference on Artificial Intelligence (IJCAI-69)*, 219–239. William Kaufmann, Inc., 1969.
- [24] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

- [25] D. Gries, J. L. A. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In E. W. Dijkstra (ed.), *Formal Development of Programs and Proofs*, 37–42. Addison Wesley, 1990.
- [26] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
- [27] P. G. Harrison, H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.
- [28] J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.
- [29] N. D. Jones. The essence of program transformation by partial evaluation and driving. In N. D. Jones, M. Hagiya, M. Sato (eds.), *Logic, Language and Computation*, LNCS 792, 206–224. Springer-Verlag, 1994.
- [30] N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
- [31] H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications (RTA '89)*, LNCS 355, 564–568. Springer-Verlag, 1989.
- [32] A. V. Klimov, S. A. Romanenko. Metavychislitel' dlja jazyka Refal. Osnovnye ponjatija i primery. (A metaevaluator for the language Refal. Basic concepts and examples). Preprint 71, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987. (in Russian).
- [33] R. E. Korf. Inversion of applicative programs. In *Int. Joint Conference on Artificial Intelligence (IJCAI-81)*, 1007–1009. William Kaufmann, Inc., 1981.
- [34] R. Kowalski. Predicate logic as programming language. In J. L. Rosenfeld (ed.), *Information Processing 74*, 569–574. North-Holland, 1974.
- [35] M. Leuschel, T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi (ed.), *Logic-Based Program Synthesis and Transformation. Proceedings*, LNCS 1817, 62–81. Springer-Verlag, 2000.
- [36] J. W. Lloyd. *Foundations of Logic Programming. Second, extended edition*. Springer-Verlag, 1987.
- [37] K. Marriott, P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
- [38] J. McCarthy. The inversion of functions defined by Turing machines. In C. E. Shannon, J. McCarthy (eds.), *Automata Studies*, 177–181. Princeton University Press, 1956.
- [39] 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.
- [40] A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, M. Broy, I. V. Pototosin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1181, 249–260. Springer-Verlag, 1996.
- [41] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
- [42] A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 427–444. North-Holland, 1988.
- [43] J. P. Secher, M. H. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, A. Zamulin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1755, 113–127. Springer-Verlag, 2000.

- [44] J. P. Secher, M. H. Sørensen. From checking to inference via driving and DAG grammars. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 41–51. ACM Press, 2002.
- [45] P. Sestoft. The structure of a self-applicable partial evaluator. Technical Report 85/11, DIKU, University of Copenhagen, Denmark, Nov. 1985.
- [46] M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
- [47] L. Sterling, E. Shapiro. *The Art of Prolog. Second Edition*. MIT Press, 1994.
- [48] V. F. Turchin. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal). In *Teorija Jazykov i Metody Programmirovaniya (Proceedings of the Symposium on the Theory of Languages and Programming Methods)*, 31–42, 1972. (In Russian).
- [49] V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker, J. van Leeuwen (eds.), *Automata, Languages and Programming*, LNCS 85, 645–657. Springer-Verlag, 1980.
- [50] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
- [51] J. L. A. van de Snepscheut. *What Computing is All About*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
- [52] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.