

The universal resolving algorithm and its correctness: inverse computation in a functional language

Sergei Abramov^a, Robert Glück^{b,*}

^a*Program Systems Institute, Russian Academy of Sciences, RU-152140 Pereslavl-Zalessky, Russia*

^b*PRESTO, JST & Institute for Software Production Technology, School of Science and Engineering, Waseda University, Tokyo 169-8555, Japan*

Abstract

We present an algorithm for inverse computation in a first-order functional language based on the notion of a perfect process tree. The Universal Resolving Algorithm introduced in this paper is sound and complete, and computes each solution for which the given program terminates, in finite time. The algorithm has been implemented for TSG, a typed dialect of S-Graph, and shows some remarkable results for the inverse computation of functional programs such as a pattern matcher and an interpreter for imperative programs. © 2002 Elsevier Science B.V. All rights reserved.

1. Introduction

While standard computation is the calculation of the output of a program for a given input (“forward execution”), inverse computation is the calculation of the possible input of a program for a given output (“backward execution”). Inverse computation is an important and useful concept in many different areas. Advances in this direction have been achieved in the area of logic programming, based on solutions emerging from logic and proof theory.

But inversion is not restricted to the context of logic programming. Reversibility is an important concept in any programming language, e.g., if one direction of an algorithm is easier to define than the other, or if both directions are needed (cf. encoding/decoding). Interestingly, inversion has sparked relatively little interest in the area of functional

* Corresponding author.

E-mail addresses: abram@botik.ru (S. Abramov), glueck@acm.org (R. Glück).

¹ On leave from DIKU, Department of Computer Science, University of Copenhagen.

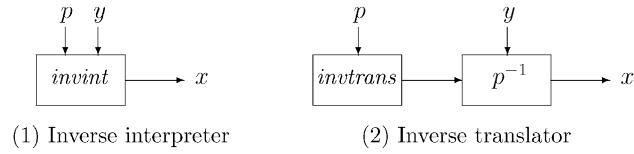


Fig. 1. Two tools for solving inversion problems.

programming (exceptions are [9,12,23,26,27,34]), even though it is an essential concept in mathematics.

We distinguish between two approaches for solving inversion problems: an inverse interpreter that performs *inverse computation* and an inverse translator that performs *program inversion*. 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 that produces p^{-1} given p , is an inverse translator (also called program inverter, Fig. 1). Applying p^{-1} to y will then determine an input x of p such that $\llbracket p \rrbracket x = y$.

As shown in [3,6], inverse computation and program inversion can be related conveniently using the Futamura projections known from partial evaluation: a program inverter is a generating extension of an inverse interpreter. In the remainder of this paper we shall focus on inverse computation.

As example of inverse computation, consider a pattern matcher which takes two arguments as input, a pattern and a text, and returns 'Success if the pattern is found in the text; 'Failure otherwise. For instance, computation with pattern "BC" and text "ABC" returns 'Success, and the same text with pattern "CB" returns 'Failure.

$$\begin{aligned} \llbracket match \rrbracket ["BC", "ABC"] &= 'Success \\ \llbracket match \rrbracket ["CB", "ABC"] &= 'Failure \end{aligned} \quad \text{standard computation}$$

Given a text "ABC", we may want to ask inverse questions such as: Which patterns *are* contained in the text, or which patterns *are not* contained in the text? To compute the answers, we can either implement new programs, in general a time consuming and error prone task, or we can use an inverse interpreter, called *ura*, to extract the answer from the pattern matcher. We do so by fixing the output to 'Success (or 'Failure) and the text to "ABC", while leaving the pattern unspecified (placeholders X_1, X_2).

$$\begin{aligned} \llbracket ura \rrbracket [match, [X_1, "ABC"], 'Success] &= ans_1 \\ \llbracket ura \rrbracket [match, [X_2, "ABC"], 'Failure] &= ans_2 \end{aligned} \quad \text{inverse computation}$$

The answers tell us which values the placeholders X_1, X_2 may take. In general, computability of the answer is not guaranteed, even with sophisticated inversion strategies. Some inversions are too resource consuming, while others are undecidable. When a program is not injective in the missing input, the answer can either be universal (all possible inputs) or existential (one of the possible inputs). We will only consider universal solutions, hence the name for our algorithm.

Most of the earlier work on this topic (e.g. [9–11,20,21]) has been 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. By contrast, our approach aims for mechanical inversion. The first observation [3] is that to do this, it suffices, in principle, to stage an inverse interpreter: via the Futamura projections this will give an inverse translator. This is convenient because inverse computation is simpler than program inversion. The second key idea is to use the notion of a *perfect process tree* [14] to systematically trace the space of possible execution paths by *standard computation*, in order to find the results of the inverse computation.

The *Universal Resolving Algorithm* (URA) introduced in this paper is sound and complete, and computes each solution (for which the given program terminates) in finite time. The algorithm has been designed for a first-order functional language with S-expressions as data structures. However, the principles and organization of inverse computation developed here are not limited to this language, but can be extended to other programming languages.

The main contributions in this paper are:

- an approach to inverse computation, its organization and structure,
- a formal specification of a URA for a first-order functional language based on the notion of a perfect process tree,
- an implementation of the algorithm and experiments with inverse computation of programs such as pattern matchers and interpreters,
- a constructive representation of sets of S-expressions allowing operations such as contractions and perfect splits.

The paper is organized as follows. Section 2 introduces the essential concepts behind the URA. Section 3 presents a first-order functional language. Section 4 presents a set representation of S-expressions and Section 5 defines a program-related extension of the set representation. Section 6 formalizes the three steps of our algorithm. Correctness is discussed in Section 7. An implementation, experiments, and termination are discussed in Sections 8–10. We conclude with a discussion of related work in Section 11 and directions for future work in Section 12. The appendix contains proofs of the main theorems. This paper is a revised and extended version of our earlier publication [5].

2. Principles of inverse computation

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

2.1. 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. A program that performs inverse computation is an *inverse interpreter*.

When a program p is not injective, or additional information about the input is available, we often want to restrict the search space of the input for a given output. Similarly, we may also want to specify a set 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

$$Inv(L, p, cls_{io}) = \{ (ds_{in}, d_{out}) \mid (ds_{in}, d_{out}) \in \lceil cls_{io} \rceil, \llbracket p \rrbracket_L ds_{in} = d_{out} \}, \quad (1)$$

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 answer. Note that computing an existential solution is a special case of computing a universal solution (the search stops after finding the first solution).

2.2. Inverse computation

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

$$\llbracket invint \rrbracket [p, cls_{io}] = ans, \quad (2)$$

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 input–output class cls_{io} and solution set ans . In this paper we use S-expressions known from Lisp [32] as the value domain, and represent the search space cls_{io} by *expressions with variables and restrictions*. This is a simple and elegant way to represent subsets of the value domain. (Other algorithms for inverse computation may choose other representations.)

The *Universal Resolving Algorithm* (URA) 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

$$\bigcup_i \lceil (cls_{io}/\theta_i)/\hat{r}_i \rceil = Inv(L, p, cls_{io}), \quad (3)$$

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 on the domain of free variables. The set representation and the operations on it will be defined in Section 4. Our algorithm produces a universal solution, hence the first word of its name.

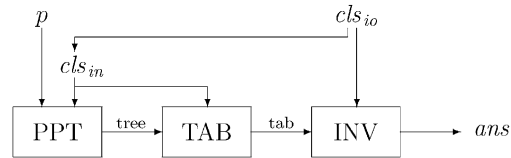


Fig. 2. Conceptual approach: three steps to inverse computation.

2.3. 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. 2; it is a refinement of box *invint* 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* [14] 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. Our 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 a program given p and cls_{in} , and then use cls_{io} to extract the solution to the backward problem. The construction of the process tree is similar to unfolding in partial evaluation where a computation is traced under partially specified input (e.g. [24]).

After introducing the source language (Section 3) and the formal foundations of our algorithm (Sections 4 and 5), we present each of the three steps in more detail (Section 6):

- (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, 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, in principle, to any computable function. Thus our method for inverse computation has full generality.

We believe the organization of inverse computation outlined above can be used for any conceivable programming language. This is supported by the fact that inverse computation is a *semantics modifier* [4], which means inverse computation can be performed in any programming language L provided we have an interpreter for L written

in the source language S of our inverse interpreter, where S is a universal language. This theoretical property [4] says nothing about the efficiency of the construction, but establishes the possibility in principle. (We shall see in Section 10 an example of inverse computation in a While-language using an interpreter for that language written in TSG.)

3. Source language

We consider the following first-order functional language, called TSG, as our source language. The language is a typed dialect of S-Graph [14]. The syntax of TSG is given by the grammar in Fig. 3 and the operational semantics by the rules in Fig. 4. An example program is shown in Fig. 11. The language has been used earlier for work on program transformation (e.g. [2,14]).

3.1. Syntax

A TSG-program is a sequence of function definitions where each definition contains the name, the parameters and the body of a function. The body of a function is a term which is either a function call **call**, a conditional **if**, or an expression e . Values d are S-expressions defined by the grammar in Fig. 5. They can be constructed by **atom**, **cons**, and tested and/or decomposed by **eqa?**, **cons?**. A program contains two types of variables. Variables $xa \in \text{PAvar}$ range over atoms DAval, variables $xe \in \text{PEvar}$ range over S-expressions Dval where DAval \subseteq Dval. The language is syntactically restricted to tail-recursion.

The first definition in a program is called *main function*. A program p is represented by a *program map* Γ which maps a function name f to the corresponding definition in p . We assume that every TSG-program p we consider is *well-formed* in the sense that every function name that appears in a call in p is defined in p , that the types of arguments and parameters are compatible, that the variables xe in **cons?** are distinct, and that every variable x used in the body of a definition q is a parameter of q or defined in an enclosing conditional.

3.2. Semantics

The evaluation of a term updates a program's *state* (t, σ) which consists of a term t and an environment σ . The meaning of each term is a *state transformation* computing the effect of the term on the state. A state with an expression e as first component is a *terminal* state; otherwise we call it a *non-terminal* state.

An *environment* $\sigma = [x_1 \mapsto d_1, \dots, x_n \mapsto d_n]$ is a sequence of typed bindings such that variables x_i are pairwise distinct, d_i are values, and $x_i \in \text{PAvar}$ implies $d_i \in \text{DAval}$ ($i = 1 \dots n$). We write $\sigma[x \mapsto d]$ to denote the environment that is just like σ except that x is bound to d . By $\sigma(x)$ we denote the value of x in σ , and by e/σ the value obtained by replacing every x occurring in e by $\sigma(x)$. If a program is well-formed, then σ in the rules of Fig. 4 defines a value for every x in e .

Grammar

$p ::= q^+$	Program
$q ::= (\mathbf{define} \ f \ x^* \ t)$	Definition
$t ::= (\mathbf{call} \ f \ e^*) \mid (\mathbf{if} \ k \ t \ t) \mid e$	Term
$k ::= (\mathbf{eqa?} \ ea \ ea) \mid (\mathbf{cons?} \ e \ xe \ xe \ xa)$	Condition
$e ::= (\mathbf{cons} \ e \ e) \mid xe \mid ea$	Expression
$ea ::= (\mathbf{atom} \ z) \mid xa$	Atomic expression
$x ::= xe \mid xa$	Typed variable

Syntax domains

$p \in \text{Program}$	$f \in \text{Fname}$	$x \in \text{Pvar}$
$q \in \text{Definition}$	$z \in \text{Symb}$	$xe \in \text{PEvar}$
$t \in \text{Term}$	$e \in \text{Pexp}$	$xa \in \text{PAvar}$
$k \in \text{Cond}$	$ea \in \text{PAexp}$	

Fig. 3. Abstract syntax of typed S-Graph (TSG).

Condition Eqa?

$$\frac{ea_1/\sigma = ea_2/\sigma}{\sigma \vdash_{if} (\mathbf{eqa?} \ ea_1 \ ea_2) \ t_1 \ t_2 \Rightarrow (t_1, \sigma)} \quad \frac{ea_1/\sigma \neq ea_2/\sigma}{\sigma \vdash_{if} (\mathbf{eqa?} \ ea_1 \ ea_2) \ t_1 \ t_2 \Rightarrow (t_2, \sigma)}$$

Condition Cons?

$$\frac{e/\sigma = (\mathbf{cons} \ d_1 \ d_2) \quad \sigma' = \sigma[xe_1 \mapsto d_1, xe_2 \mapsto d_2]}{\sigma \vdash_{if} (\mathbf{cons?} \ e \ xe_1 \ xe_2 \ xa_3) \ t_1 \ t_2 \Rightarrow (t_1, \sigma')}$$

$$\frac{e/\sigma = (\mathbf{atom} \ z) \quad \sigma' = \sigma[xa_3 \mapsto e/\sigma]}{\sigma \vdash_{if} (\mathbf{cons?} \ e \ xe_1 \ xe_2 \ xa_3) \ t_1 \ t_2 \Rightarrow (t_2, \sigma')}$$

Terms

$$\frac{\sigma \vdash_{if} k \ t_1 \ t_2 \Rightarrow (t_i, \sigma') \quad i \in \{1, 2\}}{\vdash_T ((\mathbf{if} \ k \ t_1 \ t_2), \sigma) \Rightarrow (t_i, \sigma')}$$

$$\frac{\Gamma(f) = (\mathbf{define} \ f \ x_1 \dots x_n \ t) \quad \sigma' = [x_1 \mapsto e_1/\sigma, \dots, x_n \mapsto e_n/\sigma]}{\vdash_T ((\mathbf{call} \ f \ e_1 \dots e_n), \sigma) \Rightarrow (t, \sigma')}$$

Transition

$$\frac{\vdash_T s \Rightarrow s'}{\Vdash_\Gamma s \rightarrow s'}$$

Semantic values

$$s \in \text{PDstate} = \text{Term} \times \text{PDenv}$$

$$\sigma \in \text{PDenv} = (\text{Pvar} \times \text{Dval})^*$$

$$\Gamma \in \text{ProgMap} = \text{Fname} \rightarrow \text{Definition}$$

Fig. 4. Operational semantics of TSG-programs.

S-expressions	C-expressions
$d ::= (\mathbf{cons} \ d \ d) \mid da$	$\widehat{d} ::= (\mathbf{cons} \ \widehat{d} \ \widehat{d}) \mid Xe \mid \widehat{da}$
$da ::= (\mathbf{atom} \ z)$	$\widehat{da} ::= (\mathbf{atom} \ z) \mid Xa$
	$X ::= Xe \mid Xa$
Value domains	
$d \in \text{Dval}$	$\widehat{d} \in \text{Cexp}$
$da \in \text{DAval}$	$\widehat{da} \in \text{CAexp}$
	$Xe \in \text{CEvar}$
	$Xa \in \text{CAvar}$
	$X \in \text{Cvar}$
	$z \in \text{Symb}$

Fig. 5. S-expressions and c-expressions.

The rules in Fig. 4 define a transition relation \rightarrow from a state s to a state s' in a program represented by program map Γ . We write $s \rightarrow s'$ in infix notation and drop the Γ -index when it is clear from the context. The rules are straightforward. The rule for **call** states that a call to a function f returns a new state (t, σ') that contains the body t of f 's definition and a new environment σ' that binds each parameter x_i of f to the value obtained by e_i/σ . We can replace environment σ by a fresh environment σ' because all calls are tail-recursive and no context needs to be restored when a call returns.

The rule for **if** states that, depending on the evaluation of condition k under environment σ , a new state (t_i, σ') is formed that contains one of the two branches t_1 or t_2 , and updated environment σ' . The two rules for **eqa?** define that, depending on the equality of values ea_1/σ and ea_2/σ , a new state is formed containing term t_1 or t_2 , and unchanged environment σ . The two rules for **cons?** define that, depending on value e/σ , a new state is formed containing term t_1 or t_2 , and updated environment σ' . If value e/σ has outermost constructor **cons**, σ is extended with variables xe_1, xe_2 bound to head and tail of the value, respectively. Otherwise, σ is extended with variable xa_3 bound to atom e/σ .

We consider the *input* of a program to be the arguments of a call to the program's main function, and the *output* of a program (if it exists) to be the value returned by evaluating this call by the transition relation defined in Fig. 4. We use tuples of values $ds = [d_1, \dots, d_n] \in \text{Dvals}$ as input for programs ($0 \leq n$).

Definition 1 (*Program evaluation*). Let p be a well-formed TSG-program with main function $q = (\mathbf{define} \ f \ x_1 \dots x_n \ t)$ and let $ds = [d_1, \dots, d_n] \in \text{Dvals}$. Define *initial state* $s^\circ(p, ds) \stackrel{\text{def}}{=} (t_0, \sigma_0)$ where $t_0 = (\mathbf{call} \ f \ x_1 \dots x_n)$ and $\sigma_0 = [x_1 \mapsto d_1, \dots, x_n \mapsto d_n]$. Then *program evaluation* $\llbracket \cdot \rrbracket$ is defined by

$$\llbracket p \rrbracket ds \stackrel{\text{def}}{=} \begin{cases} e/\sigma & \text{if } s^\circ(p, ds) \rightarrow^* (e, \sigma) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

4. Set representation of S-expressions

This section introduces a set representation of S-expressions and related operations such as substitution and concretization, contraction and splitting. We need this to define

CR-pairs	Restrictions
$\hat{cr} ::= \langle \hat{cc}, \hat{r} \rangle$	$\hat{r} ::= neq^*$
$\hat{cc} ::= \hat{d}$ (see Fig. 5)	$neq ::= (\hat{da} \# \hat{da}) \mid \text{contra}$
Value domains	
$\hat{cr} \in \text{CRpair}$	$\hat{r} \in \text{Restr}$
$\hat{cc} \in \text{Ccon}$	$neq \in \text{Neq}$

Fig. 6. CR-pairs and restrictions.

inverse computation for a language with S-expressions. A simple and elegant way to represent subsets of a value domain is to use *variables*, *expressions with variables* and *restrictions on variables*.

4.1. S-expressions

We use S-expressions known from Lisp as the value domain for our programs. The syntax of S-expressions is given by the grammar in Fig. 5. Values are built recursively from an infinite set of symbols using **atom** and **cons** as constructors. A value $d \in \text{Dval}$ is *ground*. We will often use 'z as shorthand for (**atom** z).

4.2. Representing sets of S-expressions

Expressions with variables, called *c-expressions* (Fig. 5), represent sets of S-expressions by means of two types of variables: *ca-variables* Xa and *ce-variables* Xe , where variables Xa range over DAval , and variables Xe range over Dval .

To further refine our set representation we introduce restrictions on variables (Fig. 6). A *restriction* is a set of non-equalities defining a set of values a ca-variable Xa must not be equal to. A *non-equality* can be expressed between ca-variables and atoms. We need restrictions on ca-variables because our language can test atoms for non-equality.² Finally, we form pairs of c-expressions and restrictions, short *cr-pairs* (Fig. 6). These are our main methods for representing and manipulating infinite sets of values constructively. Later, when we define inverse computation, we shall see how they are used.

In this section we use the term *c-construction* only for c-expressions. In Section 5 we will extend it to include program-related constructions such as environment and state. Since these notions depend on our programming language, we will discuss them later. We should stress that we distinguish between expressions in a program and the values they construct, and between program variables and c-variables in the set representation. Even though these entities may look similar, they have different functions and purposes. For example, we need operations on the set representation which are not directly present in a program. For notational convenience we indicate entities containing c-variables by a hat ($\hat{}$).

² An extension to express non-equalities between ce-variables can be found in [38].

Definition 2 (*c-expression*). A *c-expression* $\widehat{d} \in \text{Cexp}$ is an expression built from constructors **cons**, **atom**, and c-variables X_e, X_a as defined in Fig. 5. By $\text{var}(\widehat{d})$ we denote the set of all c-variables occurring in \widehat{d} .

Definition 3 (*c-construction*). A *c-construction* $\widehat{cc} \in \text{Ccon}$ is a c-expression. We define $\text{Ccon} = \text{Cexp}$.³

Definition 4 (*Non-equality, restriction*). A *non-equality* $\text{neq} \in \text{Neq}$ is an unordered pair $(\widehat{da}_1 \# \widehat{da}_2)$ with $\widehat{da}_1, \widehat{da}_2 \in \text{CAexp}$, or the symbol **contra** (Fig. 6). A *restriction* $\widehat{r} \in \text{Restr}$ is a finite set of non-equalities. By $\text{var}(\widehat{r})$ we denote the set of all ca-variables occurring in \widehat{r} .

Definition 5 (*Tautology, contradiction*). A *tautology* is a non-equality of the form $(\widehat{da}_1 \# \widehat{da}_2) \in \text{Neq}$ where $\widehat{da}_1, \widehat{da}_2$ are ground and $\widehat{da}_1 \neq \widehat{da}_2$. A *contradiction* is either a non-equality of the form $(\widehat{da} \# \widehat{da}) \in \text{Neq}$ or the symbol **contra**. By *Tauto* and *Contra* we denote the set of tautologies and the set of contradictions, respectively.

Definition 6 (*cr-pair*). A *cr-pair* $\widehat{cr} \in \text{CRpair}$ is a pair $\langle \widehat{cc}, \widehat{r} \rangle$ where $\widehat{cc} \in \text{Ccon}$ is a c-construction and $\widehat{r} \in \text{Restr}$ is a restriction (Fig. 6). By $\text{var}(\widehat{cr})$ we denote the set of c-variables occurring in \widehat{cr} : $\text{var}(\widehat{cr}) = \text{var}(\widehat{cc}) \cup \text{var}(\widehat{r})$.

Example 7. The following expressions are cr-pairs:

$$\begin{aligned}\widehat{cr}_1 &= \langle (\text{cons } X_a (\text{cons } X_e 'Z)), \emptyset \rangle \\ \widehat{cr}_2 &= \langle (\text{cons } X_a (\text{cons } X_a 'Z)), \emptyset \rangle \\ \widehat{cr}_3 &= \langle (\text{cons } X_a (\text{cons } X_a 'Z)), \{ (X_a \# 'A) \} \rangle \\ \widehat{cr}_4 &= \langle (\text{cons } X_{a_1} (\text{cons } X_{a_2} 'Z)), \{ (X_{a_1} \# X_{a_2}) \} \rangle.\end{aligned}$$

The values a ca-variable (X_a, \dots) can take must satisfy all non-equalities in a restriction. Thus, the following simplifications can be performed on a restriction: (i) any tautology can be removed because it does not limit the domain of any ca-variable, (ii) a restriction containing a contradiction can be replaced by $\{\text{contra}\}$ because no value can satisfy the contradiction (the domain of ca-variables is empty). This is stated by the following definition.

Definition 8 (*Simplification*). Let $\widehat{r} \in \text{Restr}$, then define simplification by

$$\text{simplify}(\widehat{r}) \stackrel{\text{def}}{=} \begin{cases} \{\text{contra}\} & \text{if } \widehat{r} \cap \text{Contra} \neq \emptyset \\ \widehat{r} \setminus \text{Tauto} & \text{otherwise.} \end{cases}$$

Definition 9 (*Addition of restrictions*). Let $\widehat{r}_1, \widehat{r}_2 \in \text{Restr}$ be restrictions, then define *addition of restrictions* \widehat{r}_1 and \widehat{r}_2 by the associative operation

$$\widehat{r}_1 + \widehat{r}_2 \stackrel{\text{def}}{=} \text{simplify}(\widehat{r}_1 \cup \widehat{r}_2).$$

³ Later in Section 5 we extend domain Ccon with program-related constructions: c-state \widehat{s} , c-binding \widehat{b} , c-environment \widehat{e} , c-sequence \widehat{ds} , and c-pair \widehat{dd} .

$$\begin{aligned}
&\text{CR-pair:} \\
&\quad \langle \widehat{cc}, \widehat{r} \rangle / \theta = \langle \widehat{cc} / \theta, \widehat{r} / \theta \rangle \\
&\text{C-expression:} \\
&\quad X / \theta = \begin{cases} \theta(X) & \text{if } X \in \text{dom}(\theta) \\ X & \text{otherwise} \end{cases} \\
&\quad (\text{atom } z) / \theta = (\text{atom } z) \\
&\quad (\text{cons } \widehat{d}_1 \widehat{d}_2) / \theta = (\text{cons } \widehat{d}_1 / \theta \widehat{d}_2 / \theta) \\
&\text{Non-equality:} \\
&\quad \text{contra} / \theta = \text{contra} \\
&\quad (\widehat{da}_1 \# \widehat{da}_2) / \theta = (\widehat{da}_1 / \theta \# \widehat{da}_2 / \theta) \\
&\text{Restriction:} \\
&\quad \widehat{r} / \theta = \text{simplify}(\{ \text{neq} / \theta \mid \text{neq} \in \widehat{r} \})
\end{aligned}$$

Fig. 7. Substitutions \widehat{cc}/θ , \widehat{d}/θ , neq/θ and \widehat{r}/θ .

We require that all restrictions we consider are simplified, and we include *simplify* in two operations where tautologies or contradictions can occur: adding restrictions and performing a substitution on non-equalities (Definitions 9 and 14).

4.3. Substitution and concretization

In this section we define substitution and concretization on cr-pairs. The application of a *substitution* θ to a cr-pair \widehat{cr} is shown in Fig. 7. Substitutions will be used to define *concretization* of a cr-pair, $[\widehat{cr}]$, which is the set of S-expressions represented by \widehat{cr} . We now define these notions more precisely.

Definition 10 (Substitution). A *substitution* $\theta = [X_1 \mapsto \widehat{d}_1, \dots, X_n \mapsto \widehat{d}_n]$ is a sequence of typed bindings such that c-variables X_i are pairwise distinct, \widehat{d}_i are c-expressions, and $X_i \in \text{CAvar}$ implies $\widehat{d}_i \in \text{CAexp}$, $i = 1 \dots n$. A substitution θ is *ground* if all \widehat{d}_i are ground. By $\text{dom}(\theta)$ we denote the set $\{X_1, \dots, X_n\}$, and by CCsub the set of all substitutions.

Definition 11 (Substitution on c-construction). Let $\widehat{cc} \in \text{Ccon}$ be a c-construction and let $\theta = [X_1 \mapsto \widehat{d}_1, \dots, X_n \mapsto \widehat{d}_n] \in \text{CCsub}$ be a substitution, then the result of *applying* θ to \widehat{cc} , denoted \widehat{cc}/θ , is the c-construction obtained by replacing every occurrence of X_i in \widehat{cc} by \widehat{d}_i for every $X_i \mapsto \widehat{d}_i$ in θ . We define the operation to be left-associative: $(\widehat{cc}/\theta_1)/\theta_2 = \widehat{cc}/\theta_1/\theta_2$.

Proposition 12 (Equivalence of substitution on c-construction). Let θ_1, θ_2 be substitutions and let \widehat{cc} be a c-construction, then

$$(\widehat{cc}/\theta_1 = \widehat{cc}/\theta_2) \Leftrightarrow (\forall X \in \text{var}(\widehat{cc}). (X/\theta_1 = X/\theta_2)).$$

Definition 13 (Full substitution). Let \widehat{cc} be a c-construction (or restriction, or cr-pair) and let θ be a substitution. Then θ is a *full substitution for* \widehat{cc} iff θ is ground and $\text{var}(\widehat{cc}) \subseteq \text{dom}(\theta)$. By $\text{FS}(\widehat{cc})$ we denote the set of all full substitutions for \widehat{cc} .

Below we define substitution for restrictions and cr-pairs, and properties of these operations. Again, we include *simplify* to remove tautologies and to detect contradictions that may be induced by a substitution.

Definition 14 (*Substitution on restriction*). Let $\theta \in \text{CCsub}$ and let $\hat{r} \in \text{Restr}$, then the result of applying θ to \hat{r} , denoted \hat{r}/θ , is defined by

$$\hat{r}/\theta \stackrel{\text{def}}{=} \text{simplify}(\{\text{neq}/\theta \mid \text{neq} \in \hat{r}\}).$$

Proposition 15 ($(/)$ distributive for $(+)$). *Substitution $(/)$ is distributive with respect to the addition of restrictions: $(\hat{r}_1 + \hat{r}_2)/\theta = (\hat{r}_1/\theta) + (\hat{r}_2/\theta)$.*

Due to the use of *simplify* in Definition 14, the result of \hat{r}/θ is either a contradiction, which means it is impossible to satisfy the new restriction, or a new set of non-equalities from which all tautologies have been removed.⁴ Let *neq* be a non-equality such that $\text{var}(\text{neq}) = \emptyset$. According to Definition 5, *neq* is either a tautology or a contradiction, and we can prove the following proposition.

Proposition 16 (Full substitution on restriction). *Let $\hat{r} \in \text{Restr}$ be a restriction and let $\theta \in \text{FS}(\hat{r})$ be a full substitution for \hat{r} , then either $\hat{r}/\theta = \emptyset$ or $\hat{r}/\theta = \{\text{contra}\}$.*

Definition 17 (*Substitution on cr-pair*). Let $\hat{cr} = \langle \hat{cc}, \hat{r} \rangle \in \text{CRpair}$ be a cr-pair and $\theta \in \text{CCsub}$ be a substitution, then the result of applying θ to \hat{cr} , denoted \hat{cr}/θ , is defined by

$$\hat{cr}/\theta \stackrel{\text{def}}{=} \langle \hat{cc}/\theta, \hat{r}/\theta \rangle.$$

Proposition 18 (Equivalence of substitution on cr-pair). *Let $\theta_1, \theta_2 \in \text{CCsub}$ be substitutions and let $\hat{cr} \in \text{CRpair}$ be a cr-pair, then⁵*

$$(\forall X \in \text{var}(\hat{cr}). (X/\theta_1 = X/\theta_2)) \Rightarrow (\hat{cr}/\theta_1 = \hat{cr}/\theta_2).$$

Definition 19 (\circ of substitutions). Let $\theta_1, \theta_2 \in \text{CCsub}$ be substitutions, then define *superposition of substitution θ_2 on θ_1* by

$$\theta_1 \circ \theta_2 \stackrel{\text{def}}{=} \{X \mapsto ((X/\theta_1)/\theta_2) \mid X \in (\text{dom}(\theta_1) \cup \text{dom}(\theta_2))\}.$$

Proposition 20 (\circ properties). *Let $\theta_1, \theta_2 \in \text{CCsub}$ be substitutions, then $(\theta_1 \circ \theta_2) \in \text{CCsub}$, and for all cr-pairs $\hat{cr} \in \text{CRpair}$: $\hat{cr}/(\theta_1 \circ \theta_2) = (\hat{cr}/\theta_1)/\theta_2$.*

⁴ Even though from a formal point of view it is not necessary to remove all tautologies, it is convenient to check for empty set after applying a full substitution (cf. Proposition 16).

⁵ Not an equivalence as in Proposition 12. For example, let $\theta_1 = \{X \mapsto 'A'\}$, $\theta_2 = \{X \mapsto 'B'\}$, and $\hat{r} = \{(X \# 'A'), (X \# 'B')\}$, then $\hat{r}/\theta_1 = \hat{r}/\theta_2 = \text{contra}$, but $X/\theta_1 \neq X/\theta_2$.

We are now in the position to define concretization of a cr-pair formally. As a result of our definitions above, it is easy to decide when a cr-pair represents an empty set of values. This is stated in the proposition below.

Definition 21 (*cr-concretization*). The set of data represented by cr-pair $\langle \widehat{cc}, \widehat{r} \rangle \in \text{CRpair}$, denoted $\lceil \langle \widehat{cc}, \widehat{r} \rangle \rceil$, is defined by

$$\lceil \langle \widehat{cc}, \widehat{r} \rangle \rceil \stackrel{\text{def}}{=} \{ \widehat{cc} / \theta \mid \theta \in FS(\langle \widehat{cc}, \widehat{r} \rangle), \widehat{r} / \theta = \emptyset \}.$$

Proposition 22 (*cr-pair represents empty set*). Let $\langle \widehat{cc}, \widehat{r} \rangle \in \text{CRpair}$, then

$$(\lceil \langle \widehat{cc}, \widehat{r} \rangle \rceil = \emptyset) \Leftrightarrow (\widehat{r} = \{\text{contra}\}).$$

Example 23. The cr-pairs from Example 7 represent the following sets of values:

$$\begin{aligned} \lceil \widehat{cr}_1 \rceil &= \{ (\text{cons } da \ (\text{cons } d \ 'Z)) \mid da \in \text{DAval}, d \in \text{Dval} \} \\ \lceil \widehat{cr}_2 \rceil &= \{ (\text{cons } da \ (\text{cons } da \ 'Z)) \mid da \in \text{DAval} \} \\ \lceil \widehat{cr}_3 \rceil &= \{ (\text{cons } da \ (\text{cons } da \ 'Z)) \mid da \in \text{DAval}, da \neq 'A \} \\ \lceil \widehat{cr}_4 \rceil &= \{ (\text{cons } da_1 \ (\text{cons } da_2 \ 'Z)) \mid da_1, da_2 \in \text{DAval}, da_1 \neq da_2 \}. \end{aligned}$$

4.4. Contraction and splitting

To narrow the set of values represented by a cr-pair, we introduce contractions. A *contraction* κ is either a substitution θ or a restriction \widehat{r} . A *split* is a pair of contractions (κ_1, κ_2) that partitions a set of values into two disjoint sets. A *perfect split* guarantees that no elements will be lost, and no elements will be added when partitioning a set. Later, we will use perfect splits in the construction of process trees, hence the name perfect process tree.

Definition 24 (*Contraction*). A contraction $\kappa \in \text{Contr}$ is either a substitution $\theta \in \text{CCsub}$ or a restriction $\widehat{r} \in \text{Restr}$.

Definition 25 (*Contracting*). The result of contracting cr-pair $\langle \widehat{cc}, \widehat{r} \rangle \in \text{CRpair}$ by contraction $\kappa \in \text{Contr}$, denoted $\langle \widehat{cc}, \widehat{r} \rangle / \kappa$, is a cr-pair defined by

$$\langle \widehat{cc}, \widehat{r} \rangle / \kappa \stackrel{\text{def}}{=} \begin{cases} \langle \widehat{cc}, \widehat{r} \rangle / \kappa & \text{if } \kappa \in \text{CCsub} \\ \langle \widehat{cc}, \widehat{r} + \kappa \rangle & \text{if } \kappa \in \text{Restr}. \end{cases}$$

For notational convenience we also define

$$\widehat{r} / \kappa \stackrel{\text{def}}{=} \begin{cases} \widehat{r} / \kappa & \text{if } \kappa \in \text{CCsub} \\ \widehat{r} + \kappa & \text{if } \kappa \in \text{Restr}. \end{cases}$$

Theorem 26 (*Contracting implies subset*). Let $\widehat{cr} \in \text{CRpair}$ be a cr-pair and let $\kappa \in \text{Contr}$ be a contraction, then

$$\lceil \widehat{cr} / \kappa \rceil \subseteq \lceil \widehat{cr} \rceil.$$

It is easy to show that the relation in Theorem 26 holds for all cr-pairs \hat{cr} and for all contractions κ . That is, a contraction κ never enlarges the set represented by a cr-pair. For convenience, we define two special contractions, κ_{id} and κ_{contra} .

Definition 27 (κ_{id} , κ_{contra} contractions). Define two special contractions: identity $\kappa_{\text{id}} \stackrel{\text{def}}{=} [\] \in \text{CCsub}$ and contradiction $\kappa_{\text{contra}} \stackrel{\text{def}}{=} \{\text{contra}\} \in \text{Restr}$.

It is easy to show that for all $\hat{cr} \in \text{CRpair}$:

$$[\hat{cr}/\kappa_{\text{id}}] = [\hat{cr}] \quad \text{and} \quad [\hat{cr}/\kappa_{\text{contra}}] = \emptyset.$$

The following identities are useful when applying a series of contractions to cr-pairs. They will be useful, among others in the correctness proofs.

Proposition 28 (Combination of contractions). *Let $\theta_1, \theta_2 \in \text{CCsub}$ be substitutions and let $\hat{r}_1, \hat{r}_2 \in \text{Restr}$ be restrictions, then we have the identities*

$$\begin{aligned} (SS \rightarrow S) : \quad & \forall \hat{cr} \in \text{CRpair} . \hat{cr}/\theta_1/\theta_2 = \hat{cr}/(\theta_1 \circ \theta_2) \\ (RR \rightarrow R) : \quad & \forall \hat{cr} \in \text{CRpair} . \hat{cr}/\hat{r}_1/\hat{r}_2 = \hat{cr}/(\hat{r}_1 + \hat{r}_2) \\ (RS \rightarrow SR) : \quad & \forall \hat{cr} \in \text{CRpair} . \hat{cr}/\hat{r}_1/\theta_2 = \hat{cr}/\theta_2/(\hat{r}_1/\theta_2) \end{aligned}$$

We define the split of a cr-pair by a pair of contractions. Splits play a key role when tracing a computation with partially specified input. In particular, we are interested in the so-called perfect splits because of their clean theoretical properties.

Definition 29 (Split). A split $sp \in \text{Split}$ is an unordered pair (κ_1, κ_2) where $\kappa_1, \kappa_2 \in \text{Contr}$.

Definition 30 (Perfect splitting). A split $(\kappa_1, \kappa_2) \in \text{Split}$ is *perfect* for $\hat{cr} \in \text{CRpair}$ if (κ_1, κ_2) partitions $[\hat{cr}]$ into two sets $[\hat{cr}/\kappa_1]$ and $[\hat{cr}/\kappa_2]$ such that

$$[\hat{cr}/\kappa_1] \cup [\hat{cr}/\kappa_2] = [\hat{cr}] \quad \text{and} \quad [\hat{cr}/\kappa_1] \cap [\hat{cr}/\kappa_2] = \emptyset.$$

Theorem 31 (Perfect splits). *For all cr-pairs $\langle \hat{cc}, \hat{r} \rangle \in \text{CRpair}$ the following four splits are perfect:*

- (1) $(\kappa_{\text{id}}, \kappa_{\text{contra}})$
- (2) $([Xa_1 \mapsto da], \{(Xa_1 \# da)\})$
- (3) $([Xa_1 \mapsto Xa_2], \{(Xa_1 \# Xa_2)\})$
- (4) $([Xe_3 \mapsto Xa^\diamond], [Xe_3 \mapsto (\text{cons } Xe_h^\diamond Xe_t^\diamond)])$

where $Xa_1, Xa_2, Xe_3 \in \text{var}(\hat{cc})$, $Xa^\diamond, Xe_h^\diamond, Xe_t^\diamond \notin \text{var}(\hat{cc}) \cup \text{var}(\hat{r})$, $da \in \text{DAval}$.

Remark. We use notation $^\diamond$ to denote fresh c-variables for $\langle \hat{cc}, \hat{r} \rangle$.

5. Program-related extension of the set representation

We extend our set representation to include program-related constructions. These notions are language dependent and relate to the operational semantics of our programming language.

First, we extend Definition 3 (c-construction) to include also the structures *c-pair*, *c-sequence*, *c-environment*, *c-binding*, and *c-state* as shown in Fig. 8. Second, we extend substitution to these structures (Fig. 9). All definitions and results from Section 4 remain valid. In particular, Theorem 31 (perfect splits) holds for the extended set of c-constructions.

We use the new c-constructions to define three cr-pairs, called *io-class*, *class*, *configuration*, which play a crucial role in inverse computation. An *io-class* represents a request for inverse computation, a class the partially specified input of a program, and a configuration a set of states. We introduce a relation \leq for cr-pairs, in particular for classes, and show how to represent a subclass by a single substitution–restriction pair. We say $\hat{c}r'$ is a *subclass* of $\hat{c}r$ if classes $\hat{c}r' \leq \hat{c}r$.

Definition 32 (*Class, io-class*). A cr-pair $\langle \hat{d}s, \hat{r} \rangle$ is a *class* and a cr-pair $\langle (\hat{d}s, \hat{d}), \hat{r} \rangle$ is an *io-class* where $\hat{d}s \in \text{Cexp}$ and $\hat{d} \in \text{Cexp}$. We denote the domains by *Class* and *IOClass*, respectively. By *in* and *io* we denote two operations defined by $\text{in}(\langle (\hat{d}s, \hat{d}), \hat{r} \rangle) \stackrel{\text{def}}{=} \langle \hat{d}s, \hat{r} \rangle$ and $\text{io}(\langle \hat{d}s, \hat{r} \rangle, \hat{d}) \stackrel{\text{def}}{=} \langle (\hat{d}s, \hat{d}), \hat{r} \rangle$.

C-constructions		
$\widehat{c}c ::= \widehat{d} \mid \widehat{d}\widehat{d} \mid \widehat{d}s \mid \widehat{\sigma} \mid \widehat{b} \mid \widehat{s}$	$(\widehat{d}$ defined in Fig. 5)	
$\widehat{d}\widehat{d} ::= (\widehat{d}s, \widehat{d})$	C-pair	
$\widehat{d}s ::= [\widehat{d}^*]$	C-sequence	
$\widehat{\sigma} ::= [\widehat{b}^*]$	C-environment	
$\widehat{b} ::= xe \mapsto \widehat{d} \mid xa \mapsto \widehat{d}a$	C-binding	
$\widehat{s} ::= (t, \widehat{\sigma})$	C-state	
Value domains		
$\widehat{d} \in \text{Cexp}$	$\widehat{d}\widehat{d} \in \text{Cpairs}$	$\widehat{b} \in \text{PCbind}$
$\widehat{d}s \in \text{Cexps}$	$\widehat{\sigma} \in \text{PCenv}$	$\widehat{s} \in \text{PCstate}$

Fig. 8. A program-related extension of c-constructions.

C-pair:	$(\hat{d}s, \hat{d})/\theta = (\hat{d}s/\theta, \hat{d}/\theta)$
C-sequence:	$[\hat{d}_1, \dots, \hat{d}_n]/\theta = [\hat{d}_1/\theta, \dots, \hat{d}_n/\theta]$
C-environment:	$[\hat{b}_1, \dots, \hat{b}_n]/\theta = [\hat{b}_1/\theta, \dots, \hat{b}_n/\theta]$
C-binding:	$(x \mapsto \hat{d})/\theta = (x \mapsto \hat{d}/\theta)$
C-state:	$(t, \hat{\sigma})/\theta = (t, \hat{\sigma}/\theta)$

Fig. 9. Substitution applied to program-related c-constructions.

Definition 33 (\preceq relation). Let $\widehat{cr}, \widehat{cr}' \in \text{CRpair}$ be cr-pairs, then define a reflexive and transitive relation on cr-pairs by

$$(\widehat{cr}' \preceq \widehat{cr}) \Leftrightarrow (\exists n \geq 0. \exists \kappa_1, \dots, \kappa_n \in \text{Contr}. \widehat{cr}' = \widehat{cr} / \kappa_1 \dots / \kappa_n).$$

Theorem 34 (\preceq implies \subseteq). Let $\widehat{cr}, \widehat{cr}' \in \text{CRpair}$ be cr-pairs, then

$$(\widehat{cr}' \preceq \widehat{cr}) \Rightarrow (\lceil \widehat{cr}' \rceil \subseteq \lceil \widehat{cr} \rceil).$$

Theorem 35 ((θ, \widehat{r}) -representation). Let $\widehat{cr}, \widehat{cr}' \in \text{CRpair}$ be cr-pairs such that $\widehat{cr}' \preceq \widehat{cr}$, then $\exists (\theta, \widehat{r}). \widehat{cr}' = \widehat{cr} / \theta / \widehat{r}$.

According to Definition 33, for all (\preceq)-related cr-pairs ($\widehat{cr}' \preceq \widehat{cr}$) there exists a sequence $\kappa_1, \dots, \kappa_n$ such that $\widehat{cr}' = \widehat{cr} / \kappa_1 \dots / \kappa_n$. We can always add an empty substitution $\theta_{\text{id}} = []$ and an empty restriction $\widehat{r}_{\text{id}} = \emptyset$ without changing \widehat{cr}' . According to Proposition 28 we can simplify the sequence of substitutions (S) and restrictions (R) in Eq. (4) to a single substitution–restriction pair (θ, \widehat{r}) .

$$\widehat{cr}' = \widehat{cr} \underbrace{/ \theta_{\text{id}}}_{S} \underbrace{/ \kappa_1 \dots / \kappa_n}_{\{S|R\}^*} \underbrace{/ \widehat{r}_{\text{id}}}_R \quad (4)$$

We define the intersection of two *io*-classes cls_1, cls_2 as an operation (\star) which produces a pair (θ, \widehat{r}) such that $\lceil cls_1 / \theta / \widehat{r} \rceil = \lceil cls_2 / \theta / \widehat{r} \rceil = \lceil cls_1 \rceil \cap \lceil cls_2 \rceil$.

Definition 36 (Intersection of *io*-classes). Let $cls_1, cls_2 \in \text{IOClass}$ be two *io*-classes where $cls_1 = \langle \widehat{dd}_1, \widehat{r}_1 \rangle$ and $cls_2 = \langle \widehat{dd}_2, \widehat{r}_2 \rangle$ such that $\text{var}(cls_1) \cap \text{var}(cls_2) = \emptyset$, and let $\text{mgu}(\widehat{dd}_1, \widehat{dd}_2)$ denote the most general unifier of $\widehat{dd}_1, \widehat{dd}_2$ if it exists, then define *io-class intersection* \star by

$$cls_1 \star cls_2 \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if no unifier exists for } \widehat{dd}_1, \widehat{dd}_2 \\ \emptyset & \text{if } (\widehat{r}_1 + \widehat{r}_2) / \theta = \{\text{contra}\} \text{ where } \theta = \text{mgu}(\widehat{dd}_1, \widehat{dd}_2) \\ \{(\theta, \widehat{r})\} & \text{otherwise, where } \theta = \text{mgu}(\widehat{dd}_1, \widehat{dd}_2), \widehat{r} = (\widehat{r}_1 + \widehat{r}_2) / \theta. \end{cases}$$

Theorem 37 (Correctness of (\star)). Let $cls_1, cls_2 \in \text{IOClass}$ be *io*-classes, then:

$$\begin{aligned} \lceil cls_1 \rceil \cap \lceil cls_2 \rceil = \emptyset &\Leftrightarrow cls_1 \star cls_2 = \emptyset \\ \lceil cls_1 \rceil \cap \lceil cls_2 \rceil \neq \emptyset &\Leftrightarrow \lceil cls_1 \rceil \cap \lceil cls_2 \rceil = \lceil cls_1 / \theta / \widehat{r} \rceil = \lceil cls_2 / \theta / \widehat{r} \rceil \\ &\text{where } \{(\theta, \widehat{r})\} = cls_1 \star cls_2. \end{aligned}$$

Definition 38 (Configuration). A cr-pair $\langle \widehat{s}, \widehat{r} \rangle$ where $\widehat{s} \in \text{PCstate}$ is a *configuration*. We denote the set of configurations by Conf .

Proposition 39 (Element of configuration). *Let $c = \langle (t, \hat{\sigma}), \hat{r} \rangle \in \text{Conf}$ be a configuration and let $s = (t', \sigma) \in \text{PDstate}$ be a state, then*

$$(s \in [c]) \Leftrightarrow (t' = t \wedge \sigma \in [\langle \hat{\sigma}, \hat{r} \rangle]).$$

Definition 40 (Initial class, initial configuration). Let p be a well-formed TSG-program with main function $q = (\text{define } f \ x_1 \dots x_n \ t)$, let $cls = \langle [\hat{d}_1, \dots, \hat{d}_n], \hat{r} \rangle \in \text{Class}$. We say that cls is an *initial class* for p if $[cls] \neq \emptyset$ and variable $x_i \in \text{PAvar}$ implies $\hat{d}_i \in \text{CAexp}$ ($i = 1 \dots n$). We say that $cls_{io} \in \text{IOClass}$ is an *initial io-class* for p if $\text{in}(cls_{io})$ is an initial class for p . We define *initial configuration* $c^\circ(p, cls) \stackrel{\text{def}}{=} \langle (t_0, \hat{\sigma}_0), \hat{r} \rangle$ where cls is an initial class for p , $t_0 = (\text{call } f \ x_1 \dots x_n)$ and $\hat{\sigma}_0 = [x_1 \mapsto \hat{d}_1, \dots, x_n \mapsto \hat{d}_n]$.

6. Driving, tabulation, and inversion

This section presents the three steps of inverse computation which we outlined in Section 2; see Fig. 2. First, we formalize the construction of a perfect process tree and introduce the notion of perfect driving, then we define tabulation and inversion of the table. Each of the three steps is presented in its own subsection. The correspondence of key terms can be summarized as follows.

Standard computation	Inverse computation
value d	c-expression \hat{d}
state s	configuration c
input ds	class cls

6.1. Trace semantics

A computation process is a possibly infinite sequence of states and transitions. Each state and transition in a deterministic computation are fully defined. The set of computation processes captures the semantics of a program as a whole. A *process tree* represents the set of computation processes when the computation is non-deterministic (the input is only partly specified). Each node in a process tree then represents a set of states. A node which branches to two or more configurations corresponds to a conditional transition from one set of program states to two or more sets of program states. The construction of a process tree is called *driving* in supercompilation [42]; a variant is *positive driving* [39].

The transition relation in Fig. 10 defines walks through a process tree constructed by *perfect driving* [14]. Starting from a partially specified input (cls_{in}), the goal is to follow all possible walks a standard evaluation may take under this partially specified input. This will be the basis for inverse computation where the input of a program is not fully specified.

Condition Eqt?

$$\begin{array}{c}
\frac{ea_1/\widehat{\sigma} = ea_2/\widehat{\sigma}}{\widehat{\sigma} \vdash_{if} (\mathbf{eqa?} \ e a_1 \ e a_2) \ t_1 \ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}), \kappa_{id} \rangle} \\
\frac{ea_1/\widehat{\sigma} \neq ea_2/\widehat{\sigma} \quad (ea_1/\widehat{\sigma} \ \# \ ea_2/\widehat{\sigma}) \notin \text{Tauto} \quad \kappa = [mkBind(ea_1/\widehat{\sigma}, ea_2/\widehat{\sigma})]}{\widehat{\sigma} \vdash_{if} (\mathbf{eqa?} \ e a_1 \ e a_2) \ t_1 \ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}), \kappa \rangle} \\
\frac{ea_1/\widehat{\sigma} \neq ea_2/\widehat{\sigma} \quad \kappa = \{(ea_1/\widehat{\sigma} \ \# \ ea_2/\widehat{\sigma})\}}{\widehat{\sigma} \vdash_{if} (\mathbf{eqa?} \ e a_1 \ e a_2) \ t_1 \ t_2 \Rightarrow \langle (t_2, \widehat{\sigma}), \kappa \rangle}
\end{array}$$

Condition Cons?

$$\begin{array}{c}
\frac{e/\widehat{\sigma} = (\mathbf{cons} \ \widehat{d}_1 \ \widehat{d}_2) \quad \widehat{\sigma}' = \widehat{\sigma}[x_1 \mapsto \widehat{d}_1, x_2 \mapsto \widehat{d}_2]}{\widehat{\sigma} \vdash_{if} (\mathbf{cons?} \ e \ x_1 \ x_2 \ x_3) \ t_1 \ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}'), \kappa_{id} \rangle} \\
\frac{e/\widehat{\sigma} = \widehat{da} \quad \widehat{\sigma}' = \widehat{\sigma}[x_3 \mapsto \widehat{da}]}{\widehat{\sigma} \vdash_{if} (\mathbf{cons?} \ e \ x_1 \ x_2 \ x_3) \ t_1 \ t_2 \Rightarrow \langle (t_2, \widehat{\sigma}'), \kappa_{id} \rangle} \\
\frac{e/\widehat{\sigma} = Xe \quad \widehat{\sigma}' = \widehat{\sigma}[x_1 \mapsto Xe_1^\diamond, x_2 \mapsto Xe_2^\diamond] \quad \kappa = [Xe \mapsto (\mathbf{cons} \ Xe_1^\diamond \ Xe_2^\diamond)]}{\widehat{\sigma} \vdash_{if} (\mathbf{cons?} \ e \ x_1 \ x_2 \ x_3) \ t_1 \ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}'), \kappa \rangle} \\
\frac{e/\widehat{\sigma} = Xe \quad \widehat{\sigma}' = \widehat{\sigma}[x_3 \mapsto Xa^\diamond] \quad \kappa = [Xe \mapsto Xa^\diamond]}{\widehat{\sigma} \vdash_{if} (\mathbf{cons?} \ e \ x_1 \ x_2 \ x_3) \ t_1 \ t_2 \Rightarrow \langle (t_2, \widehat{\sigma}'), \kappa \rangle}
\end{array}$$

Terms

$$\begin{array}{c}
\frac{\widehat{\sigma} \vdash_{if} k \ t_1 \ t_2 \Rightarrow \langle (t_i, \widehat{\sigma}'), \kappa \rangle \quad i \in \{1, 2\}}{\vdash_T ((\mathbf{if} \ k \ t_1 \ t_2), \widehat{\sigma}) \Rightarrow \langle (t_i, \widehat{\sigma}'), \kappa \rangle} \\
\frac{\Gamma(f) = (\mathbf{define} \ f \ x_1 \dots x_n \ t) \quad \widehat{\sigma}' = [x_1 \mapsto e_1/\widehat{\sigma}, \dots, x_n \mapsto e_n/\widehat{\sigma}]}{\vdash_T ((\mathbf{call} \ f \ e_1 \dots e_n), \widehat{\sigma}) \Rightarrow \langle (t, \widehat{\sigma}'), \kappa_{id} \rangle}
\end{array}$$

Transition

$$\frac{\vdash_T \widehat{s} \Rightarrow \langle \widehat{s}', \kappa \rangle \quad \widehat{r}/\kappa \neq \{\mathbf{contra}\}}{\Vdash_\Gamma \langle \widehat{s}, \widehat{r} \rangle \mapsto \langle \widehat{s}', \widehat{r} \rangle / \kappa}$$

Semantic values

$$\begin{array}{l}
\widehat{s} \in \text{PCstate} = \text{Term} \times \text{PCenv} \\
\widehat{\sigma} \in \text{PCenv} = (\text{Pvar} \times \text{Cexp})^* \\
\Gamma \in \text{ProgMap} = \text{Fname} \rightarrow \text{Definition}
\end{array}$$

Fig. 10. Trace semantics for perfect process trees of TSG-programs.

As defined in [14], a walk w in a process tree g is *feasible* if at least one initial state exists whose trace passes along w . A node n in a process tree g is *feasible* if it belongs to at least one feasible walk w in g . A process tree g is *perfect* if all walks in g are feasible.

6.1.1. Perfect splits and infeasible branches

The two most important operations when developing 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 is either non-terminating, or terminating in an unreachable node. The risk of entering non-terminating branches 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 in the solution (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 inputs cannot be detected, neither during the development of the tree nor in the solution. Thus, we believe there exists an input which reaches the terminal node, even though this is not the case.

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. The formulation of our transition relation includes both operations.

6.1.2. Walking a process tree

The rules in Fig. 10 define a transition relation \mapsto between configurations in a program represented by program map Γ . The transition relation does not construct a tree, but allows us to perform all walks in a perfect process tree. If a condition (**eqa?**, **cons?**) depends on an unspecified value, the rules permit us to follow any of the two possible branches.

The rules for conditional and term are similar to the rules in Fig. 4 except that they take a c-state to a new c-state and an associated contraction κ . In case of **call**, identity contraction κ_{id} is returned (no split), in case of **if**, contraction κ produced by evaluating condition k is returned.

The three rules for **eqa?** state that, depending on the equality of ca-expressions $ea_1/\hat{\sigma}$ and $ea_2/\hat{\sigma}$, a new c-state is formed which is associated with a contraction κ . The first equality rule applies if ca-expressions $ea_1/\hat{\sigma}$ and $ea_2/\hat{\sigma}$ are equal, which means they represent the same set of atoms. The second and third rule apply at the same time when $ea_1/\hat{\sigma}$ and $ea_2/\hat{\sigma}$ are not equal and at least one of the two ca-expressions is a c-variable (i.e., non-equality $(ea_1/\hat{\sigma} \# ea_2/\hat{\sigma})$ is not a tautology). Then c-states $(t_1, \hat{\sigma})$ and $(t_2, \hat{\sigma})$ are associated with the corresponding contraction of the *perfect split* (Theorem 31, split 2, 3). Auxiliary function *mkBind* makes a binding of its arguments ensuring that a ca-variable appears on the left-hand side of that binding.

The four rules for **cons?** associate a new c-state with a contraction κ . The first two rules correspond to the two cons rules in Fig. 4 except that $e/\hat{\sigma}$ is a c-expression. If $e/\hat{\sigma}$ has outermost constructor **cons** then the true-branch is entered, otherwise, the false-branch is entered. In case $e/\hat{\sigma}$ is a ce-variable Xe , the third and fourth rule apply

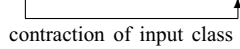
and c-states $(t_1, \hat{\sigma}_1)$ and $(t_2, \hat{\sigma}_2)$ are equipped with the corresponding contraction of the perfect split (Theorem 31, split 4).

The transition rule states that a configuration $\langle \hat{s}, \hat{r} \rangle$ is transformed into a new configuration which is obtained by evaluating c-state \hat{s} to a new c-state \hat{s}' , and applying contraction κ of the associated perfect split to $\langle \hat{s}', \hat{r} \rangle$ provided this does not lead to a contradiction (which would mean the transition is not feasible). The rule ensures perfect splitting and cutting of infeasible branches. Applying it repeatedly allows us to construct a perfect process tree.

6.2. Tabulation

Tabulation is collecting *io*-classes in a set which we call $Tab(p, cls_{in})$. For this we divide input class cls_{in} into disjoint classes each of which is associated with a terminal node (output) in the process tree. The partitioning can be carried out while tracing a path in the perfect process tree. For this we define an extended transition relation \mapsto_{tab} which carries, in addition to a configuration $\langle \hat{s}, \hat{r} \rangle$, a class cls and applies to it every contraction κ encountered along the path.

$$\frac{\vdash_{\Gamma} \hat{s} \Rightarrow \langle \hat{s}', \kappa \rangle \quad \hat{r}/\kappa \neq \{\mathbf{contra}\}}{\Vdash_{\Gamma} (cls, \langle \hat{s}, \hat{r} \rangle) \mapsto_{tab} (cls/\kappa, \langle \hat{s}', \hat{r} \rangle/\kappa)}$$



contraction of input class

Table $Tab(p, cls_{in})$ then contains an *io*-class $io(cls, e/\hat{\sigma})$ for each class cls and the corresponding output $e/\hat{\sigma}$ which we obtain from program p and input class cls_{in} by repeatedly applying transition relation \mapsto_{tab} until we reach a terminal configuration $\langle (e, \hat{\sigma}), \hat{r} \rangle$. Let us note that the restrictions in cls and $\langle (e, \hat{\sigma}), \hat{r} \rangle$ are identical because this is initially the case for cls_{in} and $c^{\circ}(p, cls_{in})$ and relation \mapsto_{tab} applies the same contractions to both of them.

Definition 41 (*Tabulation*). Let p be a well-formed TSG-program and let cls_{in} be an initial class for p . Define tabulation of p on cls_{in} as follows:

$$Tab(p, cls_{in}) \stackrel{\text{def}}{=} \{io(cls, e/\hat{\sigma}) \mid (cls_{in}, c^{\circ}(p, cls_{in})) \mapsto_{tab}^* (cls, \langle (e, \hat{\sigma}), \hat{r} \rangle)\}.$$

6.3. Inversion

Finally, we extract the solution to the inversion problem from the table by intersecting each cls'_{io} in Tab with request cls_{io} . Formally, the solution of inverse computation of program p and request cls_{io} is defined as the set $Ans(p, cls_{io})$.

Definition 42 (*Inverse computation*). Let p be a well-formed TSG-program and let cls_{io} be an initial *io*-class for p . Define inverse computation of p on cls_{io} as follows:

$$Ans(p, cls_{io}) \stackrel{\text{def}}{=} \bigcup_{cls'_{io} \in T} (cls_{io} \star cls'_{io}) \quad \text{where } T = Tab(p, in(cls_{io})).$$

<pre> t₁ (define match [p, t] (call check [p, t, p, t])) (define next [p, t] t₁₃ (if (cons? t - tt -) t₁₄ (call match [p, tt]) t₁₅ 'Failure)) </pre>	<pre> (define check [p, t, ps, ts] t₂ (if (cons? p ph pt -) t₃ (if (cons? t th tt -) t₄ (if (cons? ph - - pa) t₅ (cons 'Error '1st_arg) t₆ (if (cons? th - - ta) t₇ (cons 'Error '2nd_arg) t₈ (if (eqa? pa ta) t₉ (call check [pt, tt, ps, ts]) t₁₀ (call next [ps, ts]))) t₁₁ 'Failure) t₁₂ 'Success)) </pre>
---	---

Fig. 11. Naive pattern matcher written in TSG.

6.4. Example: pattern matcher

We now illustrate the three steps described above with an example. Consider the naive pattern matcher (Fig. 11) which takes a pattern p and a text t as input. We assume both strings are represented as lists of atoms. The matcher returns 'Success if p is found in t , 'Failure if not, or an error message if an element is found in the input lists which is not an atom.

Suppose we are given a text, and need to find all patterns which are *not* contained in the text. Let us illustrate this inverse problem for a simple text $t = ['A]$.⁶ For this task we have: the partially specified input $\hat{ds}_{in} = [Xe_1, ['A]]$, the desired output $\hat{d}_{out} = 'Failure$, and no restriction on the domain of c-variables. The initial class is $cls_{in} = \langle \hat{ds}_{in}, \emptyset \rangle$ and the io-class $cls_{io} = \langle (\hat{ds}_{in}, \hat{d}_{out}), \emptyset \rangle$.

6.4.1. Perfect process tree

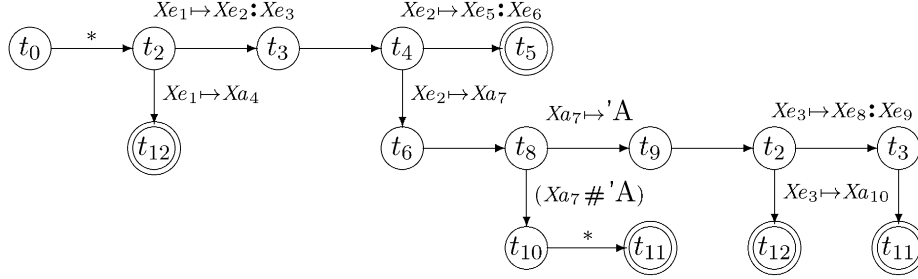
We begin with a perfect process tree whose single node is the initial configuration: the program term is a call to match, the c-environment binds p and t to the corresponding c-expression, and the restriction is empty.

$$c^\circ(\text{match}, cls_{in}) = \underbrace{\langle ((\text{call match}[p, t]), [p \mapsto Xe_1, t \mapsto ['A]]) \rangle}_{\text{term } t_0} \underbrace{\langle \rangle}_{\text{c-environment}} \underbrace{\langle \emptyset \rangle}_{\text{restr.}}$$

Tracing starts in the root, and then proceeds using the rules of the trace semantics in Fig. 10. The first test we encounter after unfolding the calls to match and check is $(\text{cons? } p \text{ ph pt } -)$ in term t_2 which tests whether the value of p is a pair. Since p is bound to c-variable Xe_1 , two transition rules apply, and we have to consider two possibilities: Xe_1 is a pair of the form $Xe_2:Xe_3$ or an atom Xa_4 . In the tree below, these assumptions are expressed by attaching substitutions $Xe_1 \mapsto Xe_2:Xe_3$ and $Xe_1 \mapsto Xa_4$ to the corresponding edges (the pair is a perfect split). The branching leads to two new terms, t_3 and t_{12} . Term $t_{12} = 'Success$ is a terminal node and we proceed with t_3 . The next test is $(\text{cons? } t \text{ th tt } -)$ in term t_3 . Since the value of t is the list $['A]$, only one

⁶ We use two shorthands in the example: we write $(d_1:d_2)$ for $(\text{cons } d_1 \text{ } d_2)$ and $[d_1, d_2, \dots, d_n]$ for a proper list $d_1:(d_2:(\dots(d_n:\text{nil})\dots))$.

rule applies and the then-branch is entered which is term t_4 . 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 match with cls_{in} , where each branching corresponds to an assumption about a c-variable. Each node in the tree refers to a term in the source program in Fig. 11.

6.4.2. Tabulation

To build table *Tab*, we follow each path from the root to a terminal node. All contractions encountered on such a path are applied to cls_{in} , and the subclass cls_i we get is associated with the corresponding output expression \hat{d}_i . To the table we add entry $io(cls_i, \hat{d}_i)$. Each class cls_i represents the set of input values which lead to the corresponding output \hat{d}_i . Since all splits in the tree are perfect, set $[cls_{in}]$ is divided into disjoint sets: $[cls_i] \cap [cls_j] = \emptyset$, $0 < i < j$.

Class cls_i	Output \hat{d}_i
$\langle [Xa_4, ['A]], \emptyset \rangle$	'Success
$\langle [((Xe_5 : Xe_6) : Xe_3), ['A]], \emptyset \rangle$	'Error: '1st_arg
$\langle [(Xa_7 : Xe_3), ['A]], \{(Xa_7 \# 'A)\} \rangle$	'Failure
$\langle [('A : Xa_{10}), ['A]], \emptyset \rangle$	'Success
$\langle [('A : Xe_8 : Xe_9), ['A]], \emptyset \rangle$	'Failure

6.4.3. Inversion

By intersecting each *io*-class $io(cls_i, \hat{d}_i)$ in $Tab(match, cls_{in})$ with the initial *io*-class cls_{io} we obtain the answer to our inverse problem:

$$Ans(match, cls_{io}) = \{([Xe_1 \mapsto 'A : Xe_8 : Xe_9], \emptyset), ([Xe_1 \mapsto Xa_7 : Xe_3], \{(Xa_7 \# 'A)\})\}.$$

The result represents the set of patterns which are not contained in text $['A]$: all patterns with length greater than one where the first element is 'A, and all patterns where the first element is not an 'A. Given *Tab*, we can solve other inverse problems. For example, with $cls'_{io} = \langle ([Xe_1, ['A]], 'Error : Xe_2), \emptyset \rangle$ we get

$$Ans(match, cls'_{io}) = \{([Xe_1 \mapsto (Xe_5 : Xe_6) : Xe_3, Xe_2 \mapsto '1st_arg], \emptyset)\}.$$

The answer describes all solutions (only one) of an equation in which both sides are partially specified: $\llbracket \text{match} \rrbracket [Xe_1, [A]] = \text{'Error: } Xe_2$. Binding $Xe_2 \mapsto \text{'1st_arg}$ tells us that the second argument cannot cause an error, only the first one. Other inverse problems can be answered in a similar way.

7. Correctness

Proving the trace semantics for perfect process trees (Fig. 10) correct with respect to the operational semantics of TSG must consist of a soundness and completeness argument. First, we state the correctness of an initial configuration and a transition step, and then state the main correctness result.

Theorem 43 (Correctness of initial configuration). *Let p be a well-formed TSG-program and let cls be an initial class for p , then*

$$\text{Completeness and Soundness: } [c^\circ(p, cls)] = \{s^\circ(p, ds) \mid ds \in [cls]\}.$$

Theorem 44 (Correctness of ppt-transition). *Let p be a well-formed TSG-program and let c be an initial configuration for p , then*

$$\begin{aligned} \text{Completeness: } & \forall s \in [c] . \forall s' . (\Vdash_\Gamma s \rightarrow s') \Rightarrow (\exists c' . (\Vdash_\Gamma c \mapsto c' \wedge s' \in [c'])) \\ \text{Soundness: } & \forall c' . (\Vdash_\Gamma c \mapsto c') \Rightarrow (\forall s' \in [c'] . \exists s \in [c] . \Vdash_\Gamma s \rightarrow s'). \end{aligned}$$

Theorem 45 (Correctness of ppt). *Let p be a well-formed TSG-program and let cls be an initial class for p , then*

Completeness :

$$\begin{aligned} \forall ds \in [cls] . \forall s_0 \dots s_n . s_0 = s^\circ(p, ds) \wedge \left(\bigwedge_{i=0}^{n-1} \Vdash_\Gamma s_i \rightarrow s_{i+1} \right) \Rightarrow \\ \exists c_0 \dots c_n . c_0 = c^\circ(p, cls) \wedge \left(\bigwedge_{i=0}^{n-1} \Vdash_\Gamma c_i \mapsto c_{i+1} \right) \wedge \left(\bigwedge_{i=0}^n s_i \in [c_i] \right) \end{aligned}$$

Soundness :

$$\begin{aligned} \forall c_0 \dots c_n . c_0 = c^\circ(p, cls) \wedge \left(\bigwedge_{i=0}^{n-1} \Vdash_\Gamma c_i \mapsto c_{i+1} \right) \Rightarrow \\ \exists ds \in [cls] . \exists s_0 \dots s_n . s_0 = s^\circ(p, ds) \wedge \left(\bigwedge_{i=0}^{n-1} \Vdash_\Gamma s_i \rightarrow s_{i+1} \right) \wedge \left(\bigwedge_{i=0}^n s_i \in [c_i] \right). \end{aligned}$$

Correctness. Proving the correctness of tabulation $Tab(p, cls_{in})$ must consist of a soundness and completeness argument. For completeness we must prove that for each evaluation $\llbracket p \rrbracket ds_{in} = d_{out}$ where $ds_{in} \in [cls_{io}]$, there is an *io*-class $cls'_{io} \in Tab(p, cls_{in})$ such that $(ds_{in}, d_{out}) \in [cls'_{io}]$. For soundness we must prove that each $cls'_{io} \in Tab(p, cls_{in})$ and each $(ds_{in}, d_{out}) \in [cls'_{io}]$ implies $\llbracket p \rrbracket ds_{in} = d_{out}$. The corresponding argument for set $Ans(p, cls_{io})$ is based on the correctness of the tabulation.

Theorem 46 (Correctness of Tab). *Let p be a well-formed TSG-program, let cls_{in} be an initial class for p , and let $T = Tab(p, cls_{in})$, then completeness and soundness are*

captured as follows:

$$\{(ds_{in}, d_{out}) \mid ds_{in} \in \lceil cls_{in} \rceil, \llbracket p \rrbracket ds_{in} = d_{out}\} = \bigcup_{cls'_{io} \in T} \lceil cls'_{io} \rceil.$$

Theorem 47 (Correctness of *Ans*). *Let p be a well-formed TSG-program, let cls_{io} be an initial io-class for p , and let $A = \text{Ans}(p, cls_{io})$, then completeness and soundness are captured as follows:*

$$\{(ds_{in}, d_{out}) \mid (ds_{in}, d_{out}) \in \lceil cls_{io} \rceil, \llbracket p \rrbracket ds_{in} = d_{out}\} = \bigcup_{(\theta, \hat{r}) \in A} \lceil (cls_{io}/\theta)/\hat{r} \rceil.$$

The most important property of set $\text{Tab}(p, cls_{in})$ is the *perfectness property*—this allows us to invert all *io*-classes in the table independently and in any order.

Theorem 48 (Perfectness of *Tab*). *Let p be a well-formed TSG-program, let cls_{in} be an initial class for p , and let cls'_{io} and cls''_{io} be two different io-classes from $\text{Tab}(p, cls_{in})$, then $\lceil in(cls'_{io}) \rceil \cap \lceil in(cls''_{io}) \rceil = \emptyset$.*

8. Algorithmic aspects

We discuss algorithmic aspects related to the URA and our Haskell implementation. While Definition 42 specifies the solution obtained from the tabulation of the perfect process tree, an algorithm for inverse computation must actually traverse the process tree according to some algorithmic strategy and extract the solution from the leaves. We are interested in presenting an implementation that reflects our approach in a clear and understandable way.

The algorithm is fully implemented in Haskell, a lazy functional language (about 300 lines of pretty-printed source text).⁷ The notions used in the program are similar to those introduced in the previous sections. The type definitions `Class`, `IOClass`, `Conf`, `CCsub` and `Restr` correspond to the domains `Class`, `IOClass`, `Conf`, `CCsub`, and `Restr`; the TSG-program is typed `ProgTSG`. Infix operators $(/.), (*.),$ and $(+.)$ implement substitution $(/)$, intersection (\star) , and update $\widehat{\sigma}[x_1 \mapsto \widehat{d}_1, \dots, x_n \mapsto \widehat{d}_n]$, and functions `in_` and `io` implement operations *in* and *io*, respectively.

The organization of the program corresponds exactly to the structure shown in Fig. 2. The algorithm has three separate functions: (1) function `ppt` that builds a potentially infinite process tree, (2) function `tab` that consumes the tree to perform the tabulation, and (3) function `inv` that enumerates set $\text{Ans}(p, cls_{io})$. The main function `ura` which performs inverse computation is defined by a composition of these three

⁷ Haskell-scripts available by <http://www.botik.ru/AbrGlu/URA/UraJ>.

functions:

```
ura :: ProgTSG -> IOClass -> [(CCsub, Restr)]
ura p clsio = inv (tab (ppt p clsin) clsin) clsio
               where clsin = in_ clsio
```

Given source program p and io -class $clsio$, function ura returns a list of substitution–restriction pairs $(CCsub, Restr)$. Due to the lazy evaluation strategy of Haskell, the process tree and the tabulation are only developed on demand by function ura . The implementation of the functions ppt , tab , inv is shown in Figs. 12 and 13. Function ppt in Fig. 12 implements the trace semantics from Fig. 10 such that all applicable rules are fired at the same time. The function makes use of a tree structure to record all walks:

```
data Tree    = LEAF Conf | NODE Conf [Branch]
type Branch = (Contr, Tree)
```

For each rule that applies a branch is added (one branch if the transition is deterministic, two branches if the transition is non-deterministic). In fact, every parent has at most two children in our case. Each node is labeled with the current configuration c , and each branch with the contraction κ used to split c (the contraction κ is needed for tabulation). Function ppt is the initial function, function $evalT$ constructs the tree, and function $ccond$ evaluates a condition. The reader may notice the format returned by function $ccond$: a tuple that contains the split to be performed on the current configuration, possibly updated bindings for the true- and false-branch, and a free index i for generating fresh variables.

Auxiliary functions $splitA$ and $splitE$ return the perfect splits for ca - and ce -variables, respectively (as defined in Theorem 31, perfect splits):

```
splitA :: CAvar -> CAexp -> Split           -- Thm.2: split 2,3
splitA cxa cea = (S[cxa:->cea], R[cxa:#:cea])

splitE :: CAvar -> FreeInd -> (Split, FreeInd) -- Thm.2: split 4
splitE cxe i = ((S[cxe:->(CONS cxe'h cxe't)], S[cxe:->cxa]), i')
               where cxe'h = newCEvar(i);   cxa = newCAvar(i+2)
                   cxe't = newCEvar(i+1);   i' = i+3
```

Function tab in Fig. 13 consumes the process tree produced by ppt using a *breadth-first strategy*⁸ in order to ensure that all leaves on finite branches will eventually be visited. This is important because a depth-first strategy may fall into an infinite branch, never visiting other branches. Function inv in Fig. 13 enumerates the set $Ans(p, clsio)$ according to Definition 42.

⁸ The breadth-first strategy is implemented in the last line of function tab by appending the list of next-level-nodes produced by map to the end of list cts .

```

ppt :: ProgTSG -> Class -> Tree
ppt p cls@(ces, r) = evalT c p i
    where (DEFINE f xs _): _ = p
          env = mkEnv xs ces
          c   = ((CALL f xs, env), r)
          i   = freeind 0 cls

evalT :: Conf -> ProgTSG -> FreeInd -> Tree
evalT c@((CALL f es, env), r) p i = NODE c [(kId, evalT c' p i)]
    where DEFINE _ xs t = getDef f p
          env' = mkEnv xs (es/.env)
          c'   = ((t, env'), r)

evalT c@((IF cond t1 t2, env), r) p i = NODE c (brT++brF)
    where ((kT, kF), bindsT, bindsF, i') = ccond cond env i
          brT = mkBr t1 kT bindsT
          brF = mkBr t2 kF bindsF
          mkBr t k binds = case r' of
              [CONTRA] -> []
              _         -> [(k, evalT c' p i')]
              where ((_, env'), r') = c/.k
                    c' = ((t, env'+.binds), r')

evalT c@((e, env), r) p i = LEAF c

ccond :: Cond -> PCEnv -> FreeInd -> (Split, PCEnv, PCEnv, FreeInd)
ccond (EQA? ea1 ea2) env i =
    let cea1 = ea1/.env; cea2 = ea2/.env in case (cea1, cea2) of
        (a, b) | a==b -> ((kId, kContra), [], [], i)
        (ATOM _, ATOM _) -> ((kContra, kId), [], [], i)
        (XA _, cea) -> (splitA cea1 cea, [], [], i)
        (cea, XA _) -> (splitA cea2 cea, [], [], i)

ccond (CONS? e xh xt xa) env i =
    let ce = e/.env in case ce of
        CONS ceh cet -> ((kId, kContra), [xh:=ceh, xt:=cet], [], i)
        ATOM a -> ((kContra, kId), [], [xa:=ce], i)
        XA _ -> ((kContra, kId), [], [xa:=ce], i)
        XE _ -> (split, [xh:=cxh, xt:=cxt], [xa:=cxa], i')
            where ((S [_ :-> (CONS cxh cxt)],
                  S [_ :-> cxa]), i') = splitE ce i

```

Fig. 12. Function ppt for inverse computation (written in Haskell).

9. 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.

```

tab :: Tree -> Class -> [IOClass]
tab tree cls = tb [(cls, tree)]
  where
    tb [] = []
    tb ((cls, LEAF ((e, env), _)) : cts) = (io cls (e/.env)) : (tb cts)
    tb ((cls, NODE _ brs) : cts) =
      tb (cts++(map \ (k, tree) -> (cls/.k, tree)) brs))

inv :: [IOClass] -> IOClass -> [(CCsub, Restr)]
inv tab clsio = concat (map ((*.) clsio) tab)

```

Fig. 13. Functions `tab` and `inv` for inverse computation (written in Haskell).

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 continues along an infinite branch in the process tree). Since termination of inverse computation is undecidable, we can only hope to design “more” terminating algorithms, for example by detecting certain finite solution sets or cutting some of the infinite branches, but we will never be able to decide termination in general. Our algorithm is sound and complete, and other algorithms cannot improve on this property, but they may be more efficient.

Our algorithm terminates *iff* the process tree is finite. This criterion can be rephrased as follows: the algorithm terminates iff for a given program p and a class cls_{in} there exists a number n such that for all $ds \in [cls_{in}]$ the application $\llbracket p \rrbracket ds$ terminates in at most n steps.⁹ In this case inverse computation of p with request cls_{io} where $cls_{in} = in(cls_{io})$ terminates regardless of the desired output. For example, application $\llbracket match \rrbracket [p, t]$ terminates in at most the square of the length of t steps regardless of pattern p . Therefore, our algorithm terminates on any request for inverse computation of `match` with given text t (even though there may be an infinite number of patterns that produce the desired output).

The analysis above is summarized by the following theorem.

Theorem 49 (Criteria of termination of `ura`). *Let p be a well-formed TSG-program, let cls_{io} be an initial io-class for p , and let $cls_{in} = in(cls_{io})$, then:*

- I. *The following three conditions are equivalent:*
 - (a) *The computation $\llbracket ura \rrbracket p cls_{io} = ans$ terminates in finite time.*
 - (b) *The perfect process tree for p on cls_{in} is finite.*
 - (c) *There exists a number $n \geq 0$ such that for all $ds_{in} \in [cls_{in}]$ the computation $\llbracket p \rrbracket ds_{in}$ terminates in at most n steps.*
- II. *The question whether for given p , cls_{io} program `ura` terminates, is undecidable in general.*

⁹ This rephrases the previous sentence because a process tree represents all possible walks of a standard evaluation on input class cls_{in} and n is the depth of that tree.

10. Experiments and results

This section illustrates the URA by means of some examples. The first example illustrates inverse computation of a pattern matcher, the second example shows the inverse interpretation of While-programs.¹⁰

10.1. Pattern matcher

We performed the two inversion tasks from Section 1 using a naive pattern matcher written in TSG (Fig. 11).

Task 1: Find the set of patterns which *are* substrings of text “ABC”. To perform this task we leave argument p unknown (Xe_1), set argument t to “ABC” and set the desired output to ‘Success’.

Task 2: Find the set of patterns which *are not* substrings of text “AAA”. To perform this task we use a setting similar to Task 1 ($p = Xe_1, t = \text{“AAA”}$), but set the desired output to ‘Failure’.

Fig. 14 shows the results of using URA. The answer for Task 1 is a finite representation of all substrings of text “ABC”, Fig. 14(i). The answer for Task 2 is a finite representation of all patterns which are not substrings of text “AAA”, Fig. 14(ii). URA terminates after 0.01 s in both cases.

10.2. Interpreter for an imperative language

Consider the small imperative programming language MP 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 are S-expressions. An MP-program operates over a global store. The semantics is conventional Pascal-style semantics.

```
(i) ura match (([Xe1, ['A, 'B, 'C]], 'Success), ∅) = [
  ([Xe1 ↦ Xa4], ∅),           -- ε
  ([Xe1 ↦ 'A:Xa10], ∅),       -- A
  ([Xe1 ↦ 'A:'B:Xa16], ∅),    -- AB
  ([Xe1 ↦ 'B:Xa10], ∅),       -- B
  ([Xe1 ↦ 'A:'B:'C:Xa22], ∅), -- ABC
  ([Xe1 ↦ 'B:'C:Xa16], ∅),    -- BC
  ([Xe1 ↦ 'C:Xa10], ∅) ]     -- C

(ii) ura match (([Xe1, ['A, 'A, 'A]], 'Failure), ∅) = [
  ([Xe1 ↦ Xa7:Xe3], {(Xa7 # 'A)}), -- [^A].*
  ([Xe1 ↦ 'A:Xa13:Xe9], {(Xa13 # 'A)}), -- A[^A].*
  ([Xe1 ↦ 'A:'A:Xa19:Xe15], {(Xa19 # 'A)}), -- AA[^A].*
  ([Xe1 ↦ 'A:'A:'A:Xa20:Xe21], ∅) ] -- AAA.+
```

Fig. 14. Inverse computation of pattern matcher.

¹⁰ Run times for PC/Intel Pentium III-600 MHz, OS Linux, GHC v.5.0, excl. gc-time.

We implemented an MP-interpreter `intMP` in TSG (309 lines of pretty-printed program text; 30 functions in TSG) and rewrote the pattern matcher in MP. The MP-interpreter is too big to be shown. In fact, the experiments with inverse computation of the MP-interpreter described below are the biggest examples of inverse computation in this paper.

In order to compare the result of inverse computation of the MP-matcher via the MP/TSG-interpreter with the application of URA to the TSG-matcher, we repeated the tasks from above. URA terminates after 0.58 s (Task 1) and after 0.47 s (Task 2). Inverse computation in MP (implemented by `ura` and `intMP`) produced the same results as inverse computation in TSG.

`ura intMP (([matchMP, [Xe1, ['A, 'B, 'C]]], ∅), 'Success) = ... (i) Results in`
`ura intMP (([matchMP, [Xe1, ['A, 'A, 'A]]], ∅), 'Failure) = ... (ii) Fig. 14`

This result is noteworthy because it shows that inverse computation in MP can be achieved through an interpreter for MP (without writing an inverse interpreter for MP). Inverse computation in MP via the MP/TSG-interpreter takes longer than inverse computation in TSG. This is what can be expected: an extra level of interpretation increases the run time (in our example about 50 times). Naturally, our approach extends to multiple levels of interpretation and we repeated the experiment above via two interpreters (MP/FCL- and FCL/TSG-interpreter where FCL is a flowchart language [24]) giving the same answers. The run times for Tasks 1 and 2 via two interpreters were 113 and 121 min, respectively.

Earlier work [3], ported inverse computation from TSG to a small assembler-like programming language (called Norma [8]). The only other experimental work we are aware of, inverts imperative programs by treating their relational semantics as logic program [36]. Our example showed inverse computation of an operational semantics defined in a functional language. This gives further practical evidence for the idea of *semantics modifiers* [4,6], namely that semantics that specify extensional properties can be ported from one language to another by means of interpreters. This underlines our thesis that, in such cases, the programming language *per se* is secondary, and that the essence of these semantics can be realized in a generic way (as shown above for inverse computation).

11. Related work

An early result [40] regarding *inverse computation* in a functional language was obtained in 1972 when it was shown that *driving*, a unification-based program transformation technique [42], can be used to perform subtraction given an algorithm for binary addition (see [1,19]). The URA presented in this paper is derived from *perfect driving* [14] and combined with a mechanical extraction of answers (cf. [1,34]) giving our algorithm the power comparable to SLD-resolution, but for a first order, functional language (cf. [17]). The use of driving for theorem proving is discussed in [41] and the relation to partial evaluation in [25]. Another technique for inverse interpretation uses

walk grammars for a restricted form of functional programs [33,43]. The first work on *program inversion* appears to be [31], suggesting a “generate and test approach” for Turing machines. With the exception of [3,36], we know of no paper addressing inverse computation in imperative languages.

Logic programming [28] inherently supports inverse computation. The use of an appropriate inference procedure permits to determine any computable answer [29]. It is not surprising that the capabilities of logic programming provided the foundation for many applications in artificial intelligence, program verification and logical reasoning. Connections between logic programming and inverse computation are discussed in [1,3]. Driving and partial deduction, a technique for program specialization in logic programming, were related in [17].

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 outside the domain of logic programming. Recently, work in this direction has been done regarding the integration of the functional and logic programming paradigm using narrowing, a unification-based goal-solving mechanism [22]; for a survey see [7].

The first efforts on *program inversion* have gone into *imperative programs* [10,11,20,21] but use non-automatic (sometimes heuristic) methods for deriving the inverse program. For example, the technique suggested in [11] provides for inverting programs symbolically, but requires that the programmer provide inductive assertions on conditionals and loop statements. The relation of program inversion and inverse computation is discussed in [3,6]; see also [18].

Some papers deal with the program inversion of *functional programs*, mostly by hand [9,12,23,26,27,34,35,37]. The work with functional languages focuses usually on program inversion. An automatic system for synthesizing recursive programs from first-order functional programs is InvX [26]. Experiments with program inversion using program transformation are reported in [16,19,33].

12. Conclusion

We presented an algorithm for inverse computation in a first-order functional language based on the notion of a perfect process tree, discussed the organization and structure of inverse computation, stated the main correctness results, and illustrated our Haskell implementation with several examples.

Our work was also motivated by the thesis [15] that program inversion is one of the three fundamental operations for transforming programs (beside program specialization and program composition). We believe that, in order to achieve full generality of program transformation, ultimately all three operations have to be mastered. So far, progress has been achieved mostly on program specialization.

In general, inverse computation using URA will be more efficient than a generate and test approach (which enumerates all possible ground input and tests the corresponding output) since URA explores program traces only once under partially specified input. Inverse computation of a program p using URA will be less efficient than computation

of the corresponding (non-trivial) inverse program p^{-1} . This is the tradeoff known from interpreters and translators.

It is desirable, though not difficult, to extend our algorithm to user-defined constructor domains. This requires an extension of the set representation in Section 4 and an extension of the source language (e.g., case-expressions). In this paper we focused on a rigorous development of the principles and foundations of inverse computation and used data structures known from Lisp. Other extensions may involve the use of constraint systems [30] or theorem proving as in GPC [13].

The question of a more efficient implementation is also left for future work. The 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 perfect process trees and the inversion of the tabulation are conveniently separated). Clearly, more efficient implementations exist. Techniques from program transformation and logic programming may prove to be useful in this context. Methods for detecting finite solution sets and cutting infinite branches can make the algorithm "more" terminating.

Acknowledgements

Discussions with our colleagues from the Refal group and the participants of MPC'2000 are greatly appreciated. The second author would like to thank Michael Leuschel for joint work leading to some of the material in Section 11. Special thanks are due to Yoshihiko Futamura for generous support of this research, and to the anonymous reviewers for many constructive suggestions. Research leading to this paper was also supported by the Japan Society for the Promotion of Science and the Danish Natural Sciences Research Council.

Appendix A. Proofs

Theorem 43 (Correctness of initial configuration).

Proof. We use the following identities:

$cls = \langle \widehat{ds}, \widehat{r} \rangle = \langle [\widehat{d}_1, \dots, \widehat{d}_n], \widehat{r} \rangle$ — the initial class for p ;

$q = (\mathbf{define} \ f \ x_1 \dots x_n \ t)$ — the main function of p ;

$t_0 = (\mathbf{call} \ f \ x_1 \dots x_n)$ — the term in $s^\circ(p, ds)$ and $c^\circ(p, cls)$ (Definitions 1 and 40).

According to Definition 21 we have: $c^\circ(p, cls) = \langle (t_0, [x_1 \mapsto \widehat{d}_1, \dots, x_n \mapsto \widehat{d}_n]), \widehat{r} \rangle$. Thus (Definitions 2 and 13):

$$var(cls) = var(c^\circ(p, cls)), \quad FS(cls) = FS(c^\circ(p, cls)). \quad (\text{A.1})$$

Then we have:

$$\begin{aligned}
& \lceil c^\circ(p, cls) \rceil \\
(\text{Definition 40}) &= \lceil \langle (t_0, [x_1 \mapsto \widehat{d}_1, \dots, x_n \mapsto \widehat{d}_n]), \widehat{r} \rangle \rceil \\
(\text{Definition 21}) &= \{ (t_0, [x_1 \mapsto \widehat{d}_1, \dots, x_n \mapsto \widehat{d}_n]) / \theta \mid \theta \in FS(c^\circ(p, cls)), \widehat{r} / \theta = \emptyset \} \\
(\text{Fig. 7, Eq. (A.1)}) &= \{ (t_0, [x_1 \mapsto \widehat{d}_1 / \theta, \dots, x_n \mapsto \widehat{d}_n / \theta]) \mid \theta \in FS(cls), \widehat{r} / \theta = \emptyset \} \\
(\text{Definition 1}) &= \{ s^\circ(p, [\widehat{d}_1 / \theta, \dots, \widehat{d}_n / \theta]) \mid \theta \in FS(cls), \widehat{r} / \theta = \emptyset \} \\
(\text{Fig. 7}) &= \{ s^\circ(p, \widehat{ds} / \theta) \mid \theta \in FS(cls), \widehat{r} / \theta = \emptyset \} \\
&= \{ s^\circ(p, ds) \mid ds = \widehat{ds} / \theta, \theta \in FS(cls), \widehat{r} / \theta = \emptyset \} \\
(\text{Definition 21}) &= \{ s^\circ(p, ds) \mid ds \in \lceil cls \rceil \}. \quad \square
\end{aligned}$$

Theorem 44 (Correctness of ppt-transition).

Proof. Let us denote $c = \langle (t, \widehat{\sigma}), \widehat{r} \rangle$.

Completeness. Let $s \in \lceil c \rceil$, let $s' = (t', \sigma')$, and let $\Vdash_\Gamma s \rightarrow s'$. According to Proposition 39 and Definition 21 we have:

$$s = (t, \sigma), \quad \sigma \in \lceil \langle \widehat{\sigma}, \widehat{r} \rangle \rceil, \quad \exists \theta \in FS(\langle \widehat{\sigma}, \widehat{r} \rangle). (\sigma = \widehat{\sigma} / \theta \wedge \widehat{r} / \theta = \emptyset). \quad (\text{A.2})$$

We need to prove that $\exists c' . (\Vdash_\Gamma c \mapsto c' \wedge s' \in \lceil c' \rceil)$, or in other words (Proposition 39):

$$\begin{aligned}
\exists c' = \langle (t', \widehat{\sigma}'), \widehat{r}' \rangle . (\Vdash_\Gamma c \mapsto c' \wedge \\
\exists \theta' \in FS(\langle \widehat{\sigma}', \widehat{r}' \rangle). (\sigma' = \widehat{\sigma}' / \theta' \wedge \widehat{r}' / \theta' = \emptyset)). \quad (\text{A.3})
\end{aligned}$$

The proof of Eq. (A.3) is by case analysis of transition $\Vdash_\Gamma s \rightarrow s'$. We examine all cases of the operational semantics (Fig. 4) corresponding to this transition:

	Term t	Condition
Case 1.	(if (eqa? ea_1 ea_2) t_1 t_2)	True
Case 2.	(if (eqa? ea_1 ea_2) t_1 t_2)	False
Case 3.	(if (cons? e xe_1 xe_2 xa_3) t_1 t_2)	True
Case 4.	(if (cons? e xe_1 xe_2 xa_3) t_1 t_2)	False
Case 5.	(call f $e_1 \dots e_n$)	—

We need to show for each case that there are rules in the trace semantics for PPT (Fig. 10) which allow us to make transition $\Vdash_\Gamma c \mapsto c'$ such that Eq. (A.3) holds. We prove Eq. (A.3) for Case 1; Cases 2–5 are proven in a similar way (not shown).

Case 1. We have:

$$t = (\text{if (eqa? } ea_1 \text{ } ea_2) \text{ } t_1 \text{ } t_2) \quad (\text{A.4})$$

$$ea_1 / \sigma = ea_2 / \sigma = da \quad (\text{A.5})$$

$$t' = t_1, \quad \sigma' = \sigma, \quad s' = (t_1, \sigma) \quad (\text{A.6})$$

There are two possible cases:

1. $ea_1/\widehat{\sigma} = ea_2/\widehat{\sigma}$

Using the first rule of the trace semantics (Fig. 10) we define $c' = \langle (t_1, \widehat{\sigma}), \widehat{r} \rangle$.

Case 1 is proven because: $\Vdash_{\Gamma} c \mapsto c'$, $\theta \in FS(\langle \widehat{\sigma}, \widehat{r} \rangle)$, $\sigma = \widehat{\sigma}/\theta$, $\widehat{r}/\theta = \emptyset$.

2. $ea_1/\widehat{\sigma} \neq ea_2/\widehat{\sigma}$

According to Eq. (A.5) at least one of the two ca-expressions $ea_1/\widehat{\sigma}$ and $ea_2/\widehat{\sigma}$ is ca-variable, i.e., $(ea_1/\widehat{\sigma} \# ea_2/\widehat{\sigma})$ is not a tautology. Thus we can use the second rule of the trace semantics for PPT (Fig. 10) to define c' . Using Eq. (A.5) we examine all possible cases for $ea_1/\widehat{\sigma}$ and $ea_2/\widehat{\sigma}$:

$ea_1/\widehat{\sigma} \quad ea_2/\widehat{\sigma} \quad \kappa = [mkBind(ea_1/\widehat{\sigma}, ea_2/\widehat{\sigma})]$		
(a)	$Xa_1 \quad da$	$[Xa_1 \mapsto da]$
(b)	$da \quad Xa_2$	$[Xa_2 \mapsto da]$
(c)	$Xa_1 \quad Xa_2$	$[Xa_1 \mapsto Xa_2]$

We complete the proof for Case (a); Cases (b) and (c) are similar (not shown):

$$ea_1/\widehat{\sigma} = Xa_1, \quad ea_2/\widehat{\sigma} = da, \quad (A.7)$$

$$\kappa = [mkBind(ea_1/\widehat{\sigma}, ea_2/\widehat{\sigma})] = [Xa_1 \mapsto da]. \quad (A.8)$$

According to Eqs. (A.5), (A.7), we have $Xa_1/\theta = da$, i.e., θ binds Xa_1 with da . Thus (Eq. (A.8), Definition 19):

$$\theta = [Xa_1 \mapsto da] \uplus \theta' = \kappa \circ \theta'. \quad (A.9)$$

According to the second rule of the trace semantics for PPT (Fig. 10):

$$\Vdash_{\Gamma} c \mapsto c' \quad \text{where} \quad c' = \langle (t_1, \widehat{\sigma}'), \widehat{r}' \rangle, \quad \widehat{\sigma}' = \widehat{\sigma}/\kappa, \quad \widehat{r}' = \widehat{r}/\kappa. \quad (A.10)$$

Finally, we conclude that Eq. (A.3) holds because (Eqs. (A.9), (A.10), Definition 19):

- $\widehat{\sigma}'/\theta' = \widehat{\sigma}/\kappa/\theta' = \widehat{\sigma}/(\kappa \circ \theta') = \widehat{\sigma}/\theta = \sigma$;
- $\widehat{r}'/\theta' = \widehat{r}/\kappa/\theta' = \widehat{r}/(\kappa \circ \theta') = \widehat{r}/\theta = \emptyset$;
- no c-variable occurs in $\widehat{\sigma}'/\theta' = \sigma$ and $\widehat{r}'/\theta' = \emptyset$, i.e. $\theta' \in FS(\langle \widehat{\sigma}', \widehat{r}' \rangle)$.

Completeness of the PPT-transition is proven (for Case 1).

Soundness. Let $c = \langle (t, \widehat{\sigma}), \widehat{r} \rangle$, $c' = \langle (t', \widehat{\sigma}'), \widehat{r}' \rangle$ be configurations, let $\Vdash_{\Gamma} c \mapsto c'$, and let $s' \in [c']$. Then we have (Proposition 39):

$$\begin{aligned} s' &= (t', \sigma'), \quad \sigma' \in [\langle \widehat{\sigma}', \widehat{r}' \rangle], \\ \exists \theta' \in FS(\langle \widehat{\sigma}', \widehat{r}' \rangle). (\sigma' &= \widehat{\sigma}'/\theta' \wedge \widehat{r}'/\theta' = \emptyset). \end{aligned} \quad (A.11)$$

We need to prove that $\exists s. (\Vdash_{\Gamma} s \rightarrow s' \wedge s \in [c])$, or in other words (Proposition 39):

$$\begin{aligned} \exists s = (t, \sigma). (\Vdash_{\Gamma} s \rightarrow s' \wedge \\ \exists \theta \in FS(\langle \widehat{\sigma}, \widehat{r} \rangle). (\sigma = \widehat{\sigma}/\theta \wedge \widehat{r}/\theta = \emptyset)). \end{aligned} \quad (A.12)$$

As in the completeness proof above, we examine each rule of the trace semantics for PPT (Fig. 10), and show for each case that there are rules in the operational semantics (Fig. 4) which make transition $\Vdash_{\Gamma} s \rightarrow s'$ such that Eq. (A.12) holds. We prove Eq. (A.12) for the case below; the other cases are proven in a similar way (not shown). Let us consider the second rule of the trace semantics for PPT (Fig. 10):

$$t = (\text{if } (\text{eqa? } ea_1 \ ea_2) \ t_1 \ t_2) \quad t' = t_1 \quad (A.13)$$

$$ea_1/\widehat{\sigma} \neq ea_2/\widehat{\sigma} \quad (ea_1/\widehat{\sigma} \ \# \ ea_2/\widehat{\sigma}) \notin \text{Tauto}$$

$$ea_1/\widehat{\sigma} = Xa_1 \quad ea_1/\widehat{\sigma} = da \quad (A.14)$$

$$\kappa = [mkBind(ea_1/\widehat{\sigma}, ea_2/\widehat{\sigma})] = [Xa_1 \mapsto da] \quad (A.15)$$

$$\widehat{\sigma}' = \widehat{\sigma}/\kappa \quad \widehat{r}' = \widehat{r}/\kappa \quad (A.16)$$

Let $\theta = \kappa \circ \theta'$, then we have (Definition 19, Eqs. (A.11), (A.14)–(A.16)):

$$\widehat{\sigma}/\theta = \widehat{\sigma}/(\kappa \circ \theta') = \widehat{\sigma}/\kappa/\theta' = \widehat{\sigma}'/\theta' = \sigma' \quad (A.17)$$

$$\widehat{r}/\theta = \widehat{r}/(\kappa \circ \theta') = \widehat{r}/\kappa/\theta' = \widehat{r}'/\theta' = \emptyset \quad (A.18)$$

$$\begin{aligned} ea_1/\sigma' &= ea_1/(\widehat{\sigma}/(\kappa \circ \theta')) = (ea_1/\widehat{\sigma})/\kappa/\theta' = Xa_1/\kappa/\theta' = da/\theta' = da \\ ea_2/\sigma' &= ea_2/(\widehat{\sigma}/(\kappa \circ \theta')) = (ea_2/\widehat{\sigma})/\kappa/\theta' = da/\kappa/\theta' = da/\theta' = da \\ ea_1/\sigma' &= ea_2/\sigma' = da \end{aligned} \quad (A.19)$$

Let $s = (t, \sigma')$, then Eq. (A.12) holds because:

- no c-variable occurs in $\widehat{\sigma}/\theta = \sigma'$ and $\widehat{r}/\theta = \emptyset$ (Eq. (A.17), (A.18)), i.e. $\theta \in FS(\langle \widehat{\sigma}, \widehat{r} \rangle)$;
- according to Eq. (A.13), (A.19) and the first rule of the op. sem. (Fig. 4): $\Vdash_{\Gamma} s \rightarrow s'$.

Soundness of the PPT-transition is proven (for the case: Eqs. (A.13)–(A.16)). \square

Theorem 45 (Correctness of ppt).

Proof. *Completeness.* Let $ds \in [cls]$ and let $s_0 \dots s_n$ be states such that

$$s_0 = s^{\circ}(p, ds) \quad (A.20)$$

$$\forall i \in [0 .. n - 1]. \Vdash_{\Gamma} s_i \rightarrow s_{i+1} \quad (A.21)$$

Using the results about the correctness of the initial configuration (Theorem 43) and the completeness of the PPT-transition (Theorem 44), we can write

$$(\text{Eq. (A.20), Theorem 43}) \quad \exists c_0 . s_0 \in [c_0] \wedge c_0 = c^{\circ}(p, cls) \quad (A.22)$$

$$\begin{aligned} (\text{Eq. (A.21), ind., Theorem 44}) \quad & \forall i \in [1 .. n]. (\exists c_i . s_i \in [c_i] \\ & \wedge \Vdash_{\Gamma} c_{i-1} \mapsto c_i) \end{aligned} \quad (A.23)$$

Soundness. Let $c_0 \dots c_n$ be configurations such that

$$\forall i \in [0 .. n - 1]. \vdash_{\Gamma} c_i \mapsto c_{i+1} \quad (\text{A.24})$$

$$c_0 = c^{\circ}(p, cls) \quad (\text{A.25})$$

Note that $(\forall i \in [0 .. n]. [c_i] \neq \emptyset)$ because if $i = 0$ then $[c_i] \neq \emptyset$ according to Definition 40; if $i > 0$ then $[c_i] \neq \emptyset$ according to Proposition 22 and the trace semantics (Fig. 10, see “Transition”, requirement $\hat{\tau}/\kappa \neq \{\text{contra}\}$).

Thus, using correctness of initial configuration (Theorem 43) and soundness (Theorem 44), we have

$$\begin{aligned} & ([c_n] \neq \emptyset) \quad \exists s_n . s_n \in [c_n] \\ (\text{Eq. (A.24), ind., Theorem 44}) \quad & \forall i \in [0 .. n - 1]. \end{aligned} \quad (\text{A.26})$$

$$\exists s_i . s_i \in [c_i] \wedge \vdash_{\Gamma} s_i \rightarrow s_{i+1} \quad (\text{A.27})$$

$$(\text{Eqs. (A.25), (A.27), Theorem 43}) \quad \exists ds \in [cls] . s_0 = s^{\circ}(p, ds)$$

Correctness of PPT is proven. \square

References

- [1] S.M. Abramov, Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming), Programmirovanie 3 (1991) 31–44 (in Russian).
- [2] S.M. Abramov, Metavychislenija i ikh prilozhenija (Metacomputation and its applications), Nauka-Fizmatlit 1995 (in Russian).
- [3] S.M. Abramov, R. Glück, Semantics modifiers: an approach to non-standard semantics of programming languages, in: M. Sato, Y. Toyama (Eds.), International Symposium on Functional and Logic Programming, World Scientific, Singapore, 1998, pp. 247–270.
- [4] S.M. Abramov, R. Glück, Combining semantics with non-standard interpreter hierarchies, in: S. Kapoor, S. Prasad (Eds.), Proc. Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, Vol. 1974, Springer, Berlin, 2000, pp. 201–213.
- [5] S.M. Abramov, R. Glück, The universal resolving algorithm: inverse computation in a functional language, in: R. Backhouse, J.N. Oliveira (Eds.), Proc. Mathematics of Program Construction, Lecture Notes in Computer Science, Vol. 1837, Springer, Berlin, 2000, pp. 187–212.
- [6] S.M. Abramov, R. Glück, From standard to non-standard semantics by semantics modifiers, Internat. J. Found. Comput. Sci. 12 (2) (2001) 171–211.
- [7] E. Albert, G. Vidal, The narrowing-driven approach to functional logic program specialization, New Generation Comput. 20 (1) (2002) 3–26.
- [8] R. Bird, Programs and Machines, Wiley, New York, 1976.
- [9] R. Bird, O. de Moor, Algebra of Programming, Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- [10] W. Chen, J.T. Udding, Program inversion: more than fun!, Sci. Comput. Programming 15 (1990) 1–13.
- [11] E.W. Dijkstra, EWD671: program inversion, in: Selected Writings on Computing: A Personal Perspective, Springer, Berlin, 1982, pp. 351–354.
- [12] D. Eppstein, A heuristic approach to program inversion, in: A.K. Joshi (Ed.), Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI-85), William Kaufmann, Inc., Los Altos, CA, 1985, pp. 219–221.
- [13] Y. Futamura, K. Nogi, Generalized partial computation, in: D. Bjørner, A.P. Ershov, N.D. Jones (Eds.), Partial Evaluation and Mixed Computation, North-Holland, Amsterdam, 1988, pp. 133–151.

- [14] 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.), *Proc. Static Analysis, Lecture Notes in Computer Science*, Vol. 724, Springer, Berlin, 1993, pp. 112–123.
- [15] R. Glück, A.V. Klimov, Metacomputation as a tool for formal linguistic modeling, in: R. Trappl (Ed.), *Cybernetics and Systems '94*, Vol. 2, World Scientific, Singapore, 1994, pp. 1563–1570.
- [16] 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.), *Proc. Perspectives of System Informatics, Lecture Notes in Computer Science*, Vol. 1755, Springer, Berlin, 2000, pp. 93–100.
- [17] R. Glück, M.H. Sørensen, Partial deduction and driving are equivalent in: M. Hermenegildo, J. Penjam (Eds.), *Proc. Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science*, Vol. 844, Springer, Berlin, 1994, pp. 165–181.
- [18] R. Glück, M.H. Sørensen, A roadmap to metacomputation by supercompilation, in: O. Danvy, R. Glück, P. Thiemann (Eds.), *Proc. Partial Evaluation, Lecture Notes in Computer Science*, Vol. 1110, Springer, Berlin, 1996, pp. 137–160.
- [19] R. Glück, V.F. Turchin, Application of metasystem transition to function inversion and transformation, in: *Proc. Int. Symp. on Symbolic and Algebraic Computation (ISSAC'90)*, Tokyo, Japan, ACM Press, New York, 1990, pp. 286–287.
- [20] D. Gries, *Inverting programs, The Science of Programming*, Springer, Berlin, Chap. 21, pp. 265–274.
- [21] 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*, Addison-Wesley, Reading, MA, 1990, pp. 37–42.
- [22] M. Hanus, The integration of functions into logic programming : from theory to practice, *J. Logic Programming* 19,20 (1994) 583–628.
- [23] P.G. Harrison, H. Khoshnevisan, On the synthesis of function inverses, *Acta Inform.* 29 (1992) 211–239.
- [24] 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, Lecture Notes in Computer Science*, Vol. 1706, Springer, Berlin, 1999, pp. 20–82.
- [25] 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, Lecture Notes in Computer Science*, Vol. 792, Springer, Berlin, 1994, pp. 206–224.
- [26] H. Khoshnevisan, K.M. Sephton, InvX: an automatic function inverter, in: N. Dershowitz (Ed.), *Rewriting Techniques and Applications (RTA'89)*, Lecture Notes in Computer Science, Vol. 355, Springer, Berlin, 1989, pp. 564–568.
- [27] R.E. Korf, Inversion of applicative programs, in: A. Drinan (Ed.), *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI-81)*, William Kaufmann, Inc., Los Altos, CA, 1981, pp. 1007–1009.
- [28] R. Kowalski, Predicate logic as programming language, in: J.L. Rosenfeld (Ed.), *Information Processing*, Vol. 74, North-Holland, Amsterdam, 1974, pp. 569–574.
- [29] J.W. Lloyd, *Foundations of Logic Programming*, 2nd extended edn., Springer, Berlin, 1987.
- [30] K. Marriott, P.J. Stuckey, *Programming with Constraints*, MIT Press, Cambridge, MA, 1998.
- [31] J. McCarthy, The inversion of functions defined by turing machines, in: C.E. Shannon, J. McCarthy (Eds.), *Automata Studies*, Princeton University Press, Princeton, 1956, pp. 177–181.
- [32] J. McCarthy, Recursive functions of symbolic expressions, *Commun. ACM* 3 (4) (1960) 184–195.
- [33] A.P. Nemytykh, V.A. Pinchuk, Program transformation with metasystem transitions: experiments with a supercompiler, in: D. Bjørner, M. Broy, I.V. Pottosin (Eds.), *Proc. Perspectives of System Informatics, Lecture Notes in Computer Science*, Vol. 1181, Springer, Berlin, 1996, pp. 249–260.
- [34] 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*, North-Holland, Amsterdam, 1988, pp. 427–444.
- [35] A.Y. Romanenko, Inversion and metacomputation, in: *Proc. Symposium on Partial Evaluation and Semantics-based Program Manipulation*, Yale University, Connecticut, ACM Press, New York, 1991, pp. 12–22.
- [36] B.J. Ross, Running programs backwards : the logical inversion of imperative computation, *Formal Aspects Comput.* 9 (1997) 331–348.

- [37] B. Schoenmakers, Inorder traversal of a binary heap and its inversion in optimal time and space, in: R.S. Bird, C.C. Morgan, J.C.P. Woodcock (Eds.), *Mathematics of Program Construction, Lecture Notes in Computer Science*, Vol. 669, Springer, Berlin, 1993, pp. 291–301.
- [38] J.P. Secher, M.H. Sørensen, On perfect supercompilation, in: D. Bjørner, M. Broy, A. Zamulin (Eds.), *Proc. Perspectives of System Informatics, Lecture Notes in Computer Science*, Vol. 1755, Springer, Berlin, 2000, pp. 113–127.
- [39] M.H. Sørensen, R. Glück, N.D. Jones, A positive supercompiler, *J. Funct. Programming* 6 (6) (1996) 811–838.
- [40] V.F. Turchin, Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal), in: *Teorija Jazykov i Metody Programmirovaniya* (Proc. Symposium on the Theory of Languages and Programming Methods), 1972, pp. 31–42 (in Russian).
- [41] 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, Lecture Notes in Computer Science*, Vol. 85, Springer, Berlin, 1980, pp. 645–657.
- [42] V.F. Turchin, The concept of a supercompiler, *ACM Trans. Programming Languages Systems* 8 (3) (1986) 292–325.
- [43] V.F. Turchin, Program transformation with metasystem transitions, *J. Funct. Programming* 3 (3) (1993) 283–313.