The Universal Resolving Algorithm: Inverse Computation in a Functional Language

Sergei Abramov[1] and Robert Glück[2]*

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

[2] Department of Information and Computer Science
School of Science and Engineering, Waseda University
Shinjuku-ku, Tokyo 169-8555, Japan
glueck@acm.org

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 (URA) introduced in this paper is sound and complete, and computes each solution, if it exists, 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 pattern matching and the inverse interpretation of While-programs.

1 Introduction

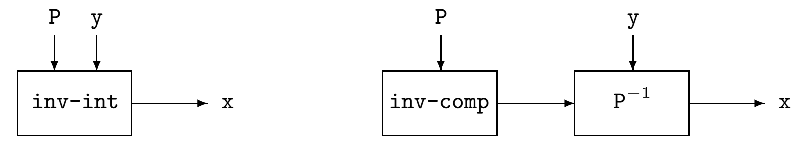
While standard computation is the calculation of the output of a program for a given input ('forward execution'), inverse computation is the calculation of the possible input of a program for a given output ('backward execution'). Inverse computation is an important and useful concept in many different 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 and decoding). Interestingly, inversion has spanned relatively little interest in the area of functional programming (exceptions are [5, 9, 18, 20, 21, 25]), 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 compiler* that performs program inversion. Determining for a given program `P` and output `y` an input `x` of `P` such that $[\![\mathtt{P}]\!]\mathtt{x} = \mathtt{y}$ is inverse computation. A program that produces $\mathtt{P}^{-1}$, is an inverse compiler (also called program inverter). Using $\mathtt{P}^{-1}$ will then determine input `x` of `P`.

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

R. Backhouse and J. N. Oliveira (Eds.): Mathematics of Program Construction. Proceedings.
LNCS 1837, pp. 187–212, 2000.



(1) Inverse interpreter (2) Inverse compiler

As shown in [3, 4], 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 strings as input, `pat` and `str`, and returns `SUCCESS` if `pat` is a substring of `str`; `FAILURE` otherwise. For instance, computation with pattern `"BC"` and string `"ABCD"` returns `SUCCESS`, and the same string with pattern `"CB"` returns `FAILURE`.

$$\begin{array}{ll} \texttt{match ["BC", "ABCD"]} \overset{*}{\Rightarrow} \texttt{'SUCCESS} & \\ & \textit{standard computation} \\ \texttt{match ["CB", "ABCD"]} \overset{*}{\Rightarrow} \texttt{'FAILURE} & \end{array}$$

Given string `str`, we may want to ask inverse questions such as: Which patterns are substrings of `str`, or which patterns are *not* substrings of `str`? To compute the answer, we can either implement new programs, in general a time consuming and error prone task, or we can use an inverse interpreter `ura` to extract the answer from the original program. We do so by fixing the output to `SUCCESS` (or `FAILURE`) and the string to `str`, while leaving the pattern unspecified (placeholders X_1, X_2).

$$\begin{array}{ll} \texttt{ura [match, [}X_1\texttt{, "ABCD"], 'SUCCESS]} \overset{*}{\Rightarrow} ans_1 & \\ & \textit{inverse computation} \\ \texttt{ura [match, [}X_2\texttt{, "ABCD"], 'FAILURE]} \overset{*}{\Rightarrow} ans_2 & \end{array}$$

The answer tells us which values the placeholders 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.*, [5–7, 16, 17]) 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 [4] is that to do this, it suffices, in principle, to stage an inverse interpreter: via the Futamura projections this will give an inverse compiler. 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* [12] to systematically trace the space of possible execution paths by *standard computation*, in order to find the inverse computation.

The *Universal Resolving Algorithm* (URA) introduced in this paper is sound and complete, and computes each solution, if it exists, in finite time. The algorithm has been designed for a first-order functional language with S-expressions as data structures. However, the principles and methods 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 Universal Resolving Algorithm 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. In Section 2 we formalize a set representation of S-expressions and in Section 3 we define our source language. A program-related extension of the set representation is introduced in Section 4. Sections 5–7 present the three steps to inverse computation. Implementation and experiments are discussed in Section 8 and 9. We conclude with a discussion of related work in Section 10 and future work in Section 11.

2 A Set Representation of S-Expressions

This section introduces the basic notions needed for inverse computation using a source language with S-expressions. In particular, we define a set representation of S-expressions and related operations such as substitution and concretization, contraction and splitting.

A simple and elegant way to represent subsets of a value domain is to use *variables*, *expressions with variables* and *restrictions on variables*. Let us consider an example from mathematics. The definition of a set of 3D-points

$$P = \{\, (x, y, x + y) \mid x > 0, y > x \,\}$$

is expressed by means of (i) variables x and y (typed variables, in fact: it is assumed that x and y range over the set of reals), (ii) expression $(x, y, x + y)$ with variables, and (iii) restrictions $x > 0$ and $y > x$ on variables. We will use the same approach for representing sets of S-expressions and introduce similar notions: c-variables, c-expressions and restrictions.

2.1 S-Expressions

We use S-expressions known from Lisp as value domain for our programs. The syntax of S-expressions is given by the grammar in Fig. 1. Values are build recursively from an infinite set of symbols using **atom** and **cons** as constructors. A value $d \in \mathrm{Dval}$ is *ground*. We will use 'z as shorthand for (**atom** z).

S-Expressions

$$d ::= (\textbf{cons}\ d\ d) \mid da$$
$$da ::= (\textbf{atom}\ z)$$

C-Expressions

$$\widehat{d} ::= (\textbf{cons}\ \widehat{d}\ \widehat{d}) \mid Xe \mid \widehat{da}$$
$$\widehat{da} ::= (\textbf{atom}\ z) \mid Xa$$
$$X ::= Xe \mid Xa$$

Value Domains

$$d \in \text{Dval} \qquad \widehat{d} \in \text{Cexp}$$
$$da \in \text{DAval} \qquad \widehat{da} \in \text{CAexp}$$
$$Xe \in \text{CEvar} \qquad X \in \text{Cvar}$$
$$Xa \in \text{CAvar} \qquad z \in \text{Symb}$$

Fig. 1. S-expressions and c-expressions

2.2 Representing Sets of S-Expressions

Expressions with variables, called *c-expressions* (Fig. 1), 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. 2). A *restriction* is a set of inequalities defining a set of values a ca-variable Xa must not be equal to. An *inequality* can be expressed between ca-variables and atoms.

Finally, we form pairs of c-expressions and restrictions, short *cr-pairs* (Fig. 2). This will be our main method for representing and manipulating sets of S-expressions in a constructive way. These structures may contain c-variables and for notational convenience we indicate this by notation $\widehat{\cdot}$.

Definition 1 (c-expression). *A* c-expression *is an expression* $\widehat{d} \in \text{Cexp}$ *as defined in Fig. 1. By* $\text{var}(\widehat{d})$ *we denote the set of all c-variables occurring in* $\widehat{d}$.

Definition 2 (c-construction). *A c-expression is a* c-construction $\widehat{cc} \in \text{Ccon}$. *We define* $\text{Ccon} = \text{Cexp}$.[1]

Definition 3 (inequality, restriction). *An* inequality $ineq \in \text{Ineq}$ *is an unordered pair* $(\widehat{da}_1 \# \widehat{da}_2)$ *with* $\widehat{da}_1, \widehat{da}_2 \in \text{CAexp}$, *or the symbol* contra. *A* restriction $\widehat{r} \in \text{Restr}$ *is a finite set of inequalities. By* $\text{var}(\widehat{r})$ *we denote the set of all ca-variables occurring in* $\widehat{r}$.

Definition 4 (tautology, contradiction). *A* tautology *is an inequality of the form* $(\widehat{da}_1 \# \widehat{da}_2) \in \text{Ineq}$ *where* $\widehat{da}_1, \widehat{da}_2$ *are ground and* $\widehat{da}_1 \neq \widehat{da}_2$. *A* contradiction *is either an inequality of the form* $(\widehat{da} \# \widehat{da}) \in \text{Ineq}$ *or the symbol* contra. *By* Tauto *and* Contra *we denote the set of tautologies and the set of contradictions, respectively.*

[1] In Sect. 4 we will extend the definition of domain Ccon with program-related constructions: c-state $\widehat{s}$, c-environment $\widehat{\sigma}$, *etc.*

CR-Pairs	*Restrictions*
$\widehat{cr}$ ::= $\langle \widehat{cc}, \widehat{r} \rangle$	$\widehat{r}$ ::= $ineq^*$
$\widehat{cc}$ ::= $\widehat{d}$ (see Fig. 1)	$ineq$ ::= $(\widehat{da} \# \widehat{da})$ | contra

Value Domains

$$\begin{array}{lll} \widehat{cr} \in \text{CRpair} & \quad & \widehat{r} \in \text{Restr} \\ \widehat{cc} \in \text{Ccon} & & ineq \in \text{Ineq} \end{array}$$

Fig. 2. CR-pairs and restrictions

Definition 5 (cr-pair). *A* cr-pair $\widehat{cr} \in$ CRpair *is a pair* $\langle \widehat{cc}, \widehat{r} \rangle$ *where* $\widehat{cc} \in$ Ccon *is a c-construction and* $\widehat{r} \in$ Restr *is a restriction. By* $\mathrm{var}(\widehat{cr})$ *we denote the set of c-variables occurring in* $\widehat{cr}$*:* $\mathrm{var}(\widehat{cr}) = \mathrm{var}(\widehat{cc}) \cup \mathrm{var}(\widehat{r})$ *.*

Example 1. The following expressions are cr-pairs:

$$\begin{array}{l}
\widehat{cr}_1 = \langle (\mathbf{cons}\ Xa\ (\mathbf{cons}\ Xe\ \text{'}Z)), \emptyset \rangle \\
\widehat{cr}_2 = \langle (\mathbf{cons}\ Xa\ (\mathbf{cons}\ Xa\ \text{'}Z)), \emptyset \rangle \\
\widehat{cr}_3 = \langle (\mathbf{cons}\ Xa\ (\mathbf{cons}\ Xa\ \text{'}Z)), \{\, (Xa \# \text{'}A) \,\} \rangle \\
\widehat{cr}_4 = \langle (\mathbf{cons}\ Xa_1\ (\mathbf{cons}\ Xa_2\ \text{'}Z)), \{\, (Xa_1 \# Xa_2) \,\} \rangle \ .
\end{array}$$

2.3 Substitution and Concretization

We now define substitution and concretization. The semantics of applying a *substitution* θ to a cr-pair $\widehat{cr}$ is defined in Fig. 3. Substitution will be used to define *concretization* $\lceil \widehat{cr} \rceil$, namely the set of S-expressions represented by $\widehat{cr}$.

Definition 6 (substitution). *A* substitution $\theta = [X_1 \mapsto \widehat{d}_1, \ldots, X_n \mapsto \widehat{d}_n]$ *is a sequence of typed bindings such that c-variables* X_i *are pairwise distinct,* $\widehat{d}_i$ *are c-expressions, and* $X_i \in$ CAvar *implies* $\widehat{d}_i \in$ CAexp*,* $i = 1 \ldots n$*. Substitution* θ *is* ground *if all* $\widehat{d}_i$ *are ground. By* $\mathrm{dom}(\theta)$ *we denote the set* $\{X_1, ..., X_n\}$ *.*

Definition 7 (substitution on c-construction). *Let* $\widehat{cc} \in$ Ccon *be a c-construction, and let* $\theta = [X_1 \mapsto \widehat{d}_1, \ldots, X_n \mapsto \widehat{d}_n] \in$ CCsub *be a substitution, then the result of* applying θ to $\widehat{cc}$*, denoted* $\widehat{cc}/\theta$*, 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*.*

Definition 8 (full substitution). *Let* $\widehat{cc}$ *be a c-construction (or restriction, or cr-pair), let* θ *be a substitution. Then* θ *is a* full substitution for $\widehat{cc}$ *iff* θ *is ground and* $\mathrm{var}(\widehat{cc}) \subseteq \mathrm{dom}(\theta)$*. By* $FS(\widehat{cc})$ *we denote the set of all full substitutions for* $\widehat{cc}$*.*

Definition 9 (substitution on restriction). *Let* $\theta \in$ CCsub*, let* $\widehat{r} \in$ Restr*, then the result of applying* θ *to* $\widehat{r}$*, denoted* $\widehat{r}/\theta$*, is defined by*

$$\widehat{r}/\theta \stackrel{\text{def}}{=} \begin{cases} \{\mathsf{contra}\} & \textit{if } \widehat{r}' \cap \text{Contra} \neq \emptyset \\ \widehat{r}' \setminus \text{Tauto} & \textit{otherwise,} \end{cases}$$
$$\textit{where } \widehat{r}' = \{\ ineq/\theta \ \mid\ ineq \in \widehat{r}\ \}\ .$$

$$\begin{array}{llcl}
\textit{CR-Pair:} & \langle \widehat{cc}, \widehat{r} \rangle/\theta & = & \langle \widehat{cc}/\theta, \widehat{r}/\theta \rangle \\
\textit{C-Expression:} & & & \\
& X/\theta & = & \begin{cases} \theta(X) & \textit{if } X \in \mathrm{dom}(\theta) \\ X & \textit{otherwise} \end{cases} \\
& (\textbf{atom } z)/\theta & = & (\textbf{atom } z) \\
& (\textbf{cons } \widehat{d}_1\ \widehat{d}_2)/\theta & = & (\textbf{cons } \widehat{d}_1/\theta\ \widehat{d}_2/\theta) \\
\textit{Inequality:} & & & \\
& \mathsf{contra}/\theta & = & \mathsf{contra} \\
& (\widehat{da}_1 \# \widehat{da}_2)/\theta & = & (\widehat{da}_1/\theta \# \widehat{da}_2/\theta) \\
\textit{Restriction:} & & & \\
& \widehat{r}/\theta & = & \begin{cases} \{\mathsf{contra}\} & \textit{if } \widehat{r}' \cap \mathrm{Contra} \neq \emptyset \\ \widehat{r}' \setminus \mathrm{Tauto} & \textit{otherwise,} \end{cases} \\
& & & \qquad \textit{where } \widehat{r}' = \{\, \textit{ineq}/\theta \mid \textit{ineq} \in \widehat{r} \,\}
\end{array}$$

Fig. 3. Definition of substitutions $\widehat{cr}/\theta$, $\widehat{d}/\theta$, $\textit{ineq}/\theta$ and $\widehat{r}/\theta$

The definition says that the result of applying a substitution θ to a restriction $\widehat{r}$ is either a contradiction, which means it is impossible to satisfy the new restriction, or a new set of inequalities from which all tautologies have been removed.[2]

Let *ineq* be an inequality such that $\mathit{var}(\mathit{ineq}) = \emptyset$. According to Def. 4 we have: *ineq* is either a tautology or a contradiction. This fact allows us to prove the following proposition.

Proposition 1. *Let* $\widehat{r} \in \mathrm{Restr}$ *be a restriction and let* $\theta \in \mathit{FS}(\widehat{r})$ *be a full substitution for* $\widehat{r}$*, then either* $\widehat{r}/\theta = \emptyset$ *or* $\widehat{r}/\theta = \{\, \mathsf{contra} \,\}$.

Definition 10 (substitution on cr-pair). *Let* $\widehat{cr} = \langle \widehat{cc}, \widehat{r} \rangle \in \mathrm{CRpair}$ *be a cr-pair and* $\theta \in \mathrm{CCsub}$ *be a substitution, then the result of applying* θ *to* $\widehat{cr}$*, denoted* $\widehat{cr}/\theta$*, is defined by*

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

Definition 11 (cr-concretization). *The set of data represented by cr-pair* $\langle \widehat{cc}, \widehat{r} \rangle \in \mathrm{CRpair}$*, denoted* $\lceil \langle \widehat{cc}, \widehat{r} \rangle \rceil$*, is defined by*

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

Example 2. The cr-pairs from Example 1 represent the following sets of values:

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

[2] 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.* Prop. 1).

2.4 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.

Definition 12 (contraction). *A contraction* $\kappa \in \mathrm{Contr}$ *is either a substitution* $\theta \in \mathrm{CCsub}$ *or a restriction* $\widehat{r} \in \mathrm{Restr}$.

Definition 13 (contracting). *The result of contracting cr-pair* $\langle \widehat{cc}, \widehat{r} \rangle \in \mathrm{CRpair}$ *by contraction* $\kappa \in \mathrm{Contr}$*, denoted* $\langle \widehat{cc}, \widehat{r} \rangle / \kappa$*, is a cr-pair defined by*

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

For notational convenience we also define

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

It is easy to show that $\lceil \widehat{cr}/\kappa \rceil \subseteq \lceil \widehat{cr} \rceil$ for all $\widehat{cr} \in \mathrm{CRpair}$ and for all $\kappa \in \mathrm{Contr}$. That is, a contraction κ does never enlarge the set represented by a cr-pair.

Definition 14. *Define two special contractions: identity* $\kappa_{\mathsf{id}} \stackrel{\mathrm{def}}{=} [\,] \in \mathrm{CCsub}$ *and contradiction* $\kappa_{\mathsf{contra}} \stackrel{\mathrm{def}}{=} \{\mathsf{contra}\} \in \mathrm{Restr}$.

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

$$\lceil \widehat{cr}/\kappa_{\mathsf{id}} \rceil = \lceil \widehat{cr} \rceil \quad \text{and} \quad \lceil \widehat{cr}/\kappa_{\mathsf{contra}} \rceil = \emptyset\ .$$

Definition 15 (split). *A split* $sp \in \mathrm{Split}$ *is a pair* (κ_1, κ_2) *where* $\kappa_1, \kappa_2 \in \mathrm{Contr}$.

Definition 16 (perfect splitting). *A split* $(\kappa_1, \kappa_2) \in \mathrm{Split}$ *is* perfect *for* $\widehat{cr} \in \mathrm{CRpair}$ *if* (κ_1, κ_2) *divides* $\lceil \widehat{cr} \rceil$ *into two disjoint sets* $\lceil \widehat{cr}/\kappa_1 \rceil$ *and* $\lceil \widehat{cr}/\kappa_2 \rceil$ *such that*

$$\lceil \widehat{cr}/\kappa_1 \rceil \cup \lceil \widehat{cr}/\kappa_2 \rceil = \lceil \widehat{cr} \rceil \quad \textit{and} \quad \lceil \widehat{cr}/\kappa_1 \rceil \cap \lceil \widehat{cr}/\kappa_2 \rceil = \emptyset\ .$$

Theorem 1 (perfect splits). *For all cr-pairs* $\langle \widehat{cc}, \widehat{r} \rangle \in \mathrm{CRpair}$ *the following four splits are perfect:*

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

where $Xa_1, Xa_2, Xe_3 \in \mathrm{var}(\widehat{cc})$*,* $Xa^{\diamond}, Xe_h^{\diamond}, Xe_t^{\diamond} \notin \mathrm{var}(\widehat{cc}) \cup \mathrm{var}(\widehat{r})$*,* $da \in \mathrm{DAval}$*. Remark: we use notation* $^{\diamond}$ *to denote fresh c-variables for* $\langle \widehat{cc}, \widehat{r} \rangle$*.*

Proof: Omitted. ∎

Grammar

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

Syntax Domains

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

Fig. 4. Abstract syntax of typed S-Graph (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 [12]. The syntax of TSG is given by the grammar in Fig. 4; the operational semantics is defined in Fig. 5. An example program in concrete syntax is shown in Fig. 13. This family of languages has been used earlier for work on program transformation [2, 11, 12].

Syntax. A TSG-program is a sequence of function definitions where each definition contains the name, the parameters and the body of the function. The body of a function is a term which is either a function call **call**, a conditional **if**, or an expression e. Values can be constructed by **atom**, **cons**, and tested and/or decomposed by **eqa?**, **cons?**. Variables xa range over atoms, variables xe range over arbitrary values. The language is restricted to tail-recursion.

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, and that every variable x used in the body of a definition q is a parameter of q or defined in an enclosing conditional. 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.

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 then a *state transformation* computing the effect of the term on the state. We consider the *input* of a program to be the arguments of a call to the program's main

Condition Eqa?

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

Condition Cons?

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

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

Terms

$$\frac{\sigma \vdash_{\mathit{if}} k\ t_1\ t_2 \Rightarrow (t_i, \sigma') \qquad i \in \{1,2\}}{\vdash_{\Gamma} ((\textbf{if}\ k\ t_1\ t_2), \sigma) \Rightarrow (t_i, \sigma')}$$

$$\frac{\Gamma(f) = (\textbf{define}\ f\ x_1 \ldots x_n\ t) \qquad \sigma' = [x_1 \mapsto e_1/\sigma, \ldots, x_n \mapsto e_n/\sigma]}{\vdash_{\Gamma} ((\textbf{call}\ f\ e_1 \ldots e_n), \sigma) \Rightarrow (t, \sigma')}$$

Transition

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

Semantic Values

$$\begin{array}{lll} s \in \text{PDstate} & = & \text{Term} \times \text{PDenv} \\ \sigma \in \text{PDenv} & = & (\text{Pvar} \times \text{Dval})^* \\ \Gamma \in \text{ProgMap} & = & \text{Fname} \rightharpoonup \text{Definition} \end{array}$$

Fig. 5. Operational semantics of TSG-programs

function, and the *output* of a program (if it exists) to be the value returned by evaluating this call. The semantics of TSG relies on the following definitions.

Values and variables. Values $d \in \text{Dval}$ are defined by the grammar in Fig. 1. In addition, we use tuples of values $ds = [d_1, \ldots, d_n]$ as input for programs $(0 \leq n)$. The set of all value tuples will be denoted by Dvals. A program contains two types of variables $x \in \text{Pvar}$. Variables $xa \in \text{PAvar}$ range over DAval, variables $xe \in \text{PEvar}$ range over Dval. Recall that $\text{DAval} \subseteq \text{Dval}$.

Environment. An *environment* $\sigma = [x_1 \mapsto d_1, \ldots, x_n \mapsto d_n] \in \text{PDenv}$ 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 $d_i \in \text{DAval}$ $(i = 1 \ldots n)$. An environment σ holds the values of the program variables.

We write $\sigma[x \mapsto d]$ to denote the environment that is just like σ except that variable x is bound to value d, and we write $\sigma(x)$ to denote the value of x in σ. Let $e \in \text{Pexp}$ and $\sigma \in \text{PDenv}$, then e/σ denotes *the value of* e *on* σ defined as the result of replacing every variable x occurring in e by value $\sigma(x)$. If a program is well-formed, then σ in the rules of Fig. 5 defines a value for every x in e.

State. A *state* $s = (t, \sigma) \in$ PDstate is a term-environment pair that represents the current state of computation. A state of the form $s = (e, \sigma)$ with $e \in$ Pexp is a *terminal* state; otherwise s is a *non-terminal* state.

Evaluation. Figure 5 defines a transition relation $\rightarrow$ between states. 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.

The rule for **if** states that, depending on the evaluation of condition k under environment σ, a new state (t_i, σ') is returned that contains one of the two branches t_1 or t_2, and an updated environment σ'.

The two rules for **eqa?** state 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?** state that, depending on value e/σ, a new state is formed containing term t_1 or t_2, and an updated environment σ'. If value e/σ has outermost constructor **cons**, environment σ is extended with variables xe_1, xe_2 bound to head and tail component of the value, respectively. Otherwise, environment σ is extended with variable xa_3 bound to atom e/σ.

Finally, the Γ-indexed transition relation $\rightarrow_\Gamma \subseteq$ (PDstate $\times$ PDstate) defines a transition from a state s to a state s' in a program represented by program map Γ. Even though the rule's formulation in Fig. 5 is trivial, we keep it for later extension. We write $\rightarrow_\Gamma$ in infix notation and drop the Γ-index when it is clear from the context. For example, we write $s \rightarrow s'$ when $(s, s') \in \rightarrow_\Gamma$.

Definition 17 (program evaluation). *Let p be a well-formed* TSG*-program with main function $q =$ (**define** $f\ x_1 \ldots x_n\ t$), and let $ds = [d_1, \ldots, d_n] \in$* Dvals. *We define* initial state $s^\circ(p, ds) \stackrel{\text{def}}{=} (t_0, \sigma_0)$ *where* $t_0 =$ (**call** $f\ x_1 \ldots x_n$) *and* $\sigma_0 = [x_1 \mapsto d_1, \ldots, x_n \mapsto d_n]$. *We define* program evaluation $[\![\cdot]\!]$ *as follows:*

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

4 Program-Related Extension of the Set Representation

We extend the set representation introduced in Sect. 2 to program-related constructions needed for inverse computation of TSG-programs, such as state, environment, and input. These notions are language dependent and relate to the operational semantics of TSG.

First, we extend the definition of c-construction $\widehat{cc}$ to include *c-state* $\widehat{s}$, *c-binding* $\widehat{b}$, *c-environment* $\widehat{\sigma}$, and *c-input* $\widehat{ds}$ (Fig. 6). That is, domain Ccon (Def. 2) is extended to include all of these sets. Second, we extend the application of substitution to all program-related c-constructions (Fig. 7). Beside these

Extended C-Constructions

$$
\begin{array}{lll}
\widehat{cc} & ::= & \widehat{s} \mid \widehat{b} \mid \widehat{\sigma} \mid \widehat{ds} \mid \widehat{d} \\
\widehat{s} & ::= & (t, \widehat{\sigma}) \\
\widehat{b} & ::= & xa \mapsto \widehat{da} \mid xe \mapsto \widehat{d} \\
\widehat{\sigma} & ::= & [\widehat{b}^*] \\
\widehat{ds} & ::= & [\widehat{d}^*] \\
\widehat{d} & ::= & \text{(defined in Fig. 1)}
\end{array}
\qquad
\begin{array}{l}
\text{C-Construction} \\
\text{C-State} \\
\text{C-Binding} \\
\text{C-Environment} \\
\text{C-Input} \\
\text{C-Expression}
\end{array}
$$

Value Domains

$$
\begin{array}{lll}
\widehat{s} & \in & \text{PCstate} \\
\widehat{b} & \in & \text{PCbind} \\
\widehat{\sigma} & \in & \text{PCenv}
\end{array}
\qquad
\begin{array}{lll}
\widehat{cc} & \in & \text{Ccon} \\
\widehat{ds} & \in & \text{Cexps}
\end{array}
$$

Fig. 6. A program-related extension of c-constructions

C-State:	$(t, \widehat{\sigma})/\theta = (t, \widehat{\sigma}/\theta)$
C-Binding:	$(x \mapsto \widehat{d})/\theta = (x \mapsto \widehat{d}/\theta)$
C-Environment:	$[\widehat{b}_1, \ldots, \widehat{b}_n]/\theta = [\widehat{b}_1/\theta, \ldots, \widehat{b}_n/\theta]$
C-Input:	$[\widehat{d}_1, \ldots, \widehat{d}_n]/\theta = [\widehat{d}_1/\theta, \ldots, \widehat{d}_n/\theta]$

Fig. 7. Substitution on program-related c-constructions

extensions, all definitions and results from Sect. 2 remain valid. In particular, Thm. 1 (perfect splits) holds for the extended set of c-constructions.

The extension of domain Ccon leads to new cr-pairs. A cr-pair containing a c-state $\widehat{s}$ is called *configuration*. A cr-pair containing a c-input $\widehat{ds}$ is called *class*. Each of them represents a set of states and a set of value tuples, respectively.

Definition 18 (class, configuration). *A cr-pair* $\langle \widehat{ds}, \widehat{r} \rangle$ *where* $\widehat{ds} \in$ Cexps *is a* class. *A cr-pair* $\langle \widehat{s}, \widehat{r} \rangle$ *where* $\widehat{s} \in$ PCstate *is a* configuration. *By* Class *and* Conf *we denote the set of classes and the set of configurations, respectively.*

Definition 19 (well-formed input class, initial configuration). *Let p be a well-formed* TSG*-program with main function $q =$ (**define** $f\ x_1 \ldots x_n\ t$), and let $cls = \langle [\widehat{d}_1, \ldots, \widehat{d}_n], \widehat{r} \rangle \in$ Class. We say that cls is* a well-formed input class for p *if* $\lceil cls \rceil \neq \emptyset$ *and variable* $x_i \in$ PAvar *implies* $\widehat{d}_i \in$ CAexp $(i = 1 \ldots n)$. *We define* initial configuration $c^\circ(p, cls) \stackrel{\text{def}}{=} \langle (t_0, \widehat{\sigma}_0), \widehat{r} \rangle$ *where cls is a well-formed input class for p, $t_0 =$ (**call** $f\ x_1 \ldots x_n$) and* $\widehat{\sigma}_0 = [x_1 \mapsto \widehat{d}_1, \ldots, x_n \mapsto \widehat{d}_n]$.

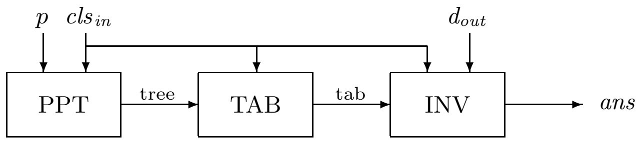


Fig. 8. Conceptual approach: three steps to inverse computation

5 Three Steps to Inverse Computation

Inverse computation can be organized into three steps: walking through a perfect process tree, then tabulating the input-output pairs, and finally extracting the answer to the inversion problem from the input-output pairs.

The key idea used in our approach is based on the notion of a *perfect process tree* which represents the computation of a program with missing input by a tree of all possible computation traces. Each fork in the tree partitions the input class into disjoint classes. Our algorithm then constructs, breadth-first and lazily, a perfect process tree for a given program p and input class cls_{in}. We shall not be concerned with different implementation techniques, but with a rigorous development of the principles and foundations of inverse computation.

In general, inverse computation using *ura* takes the form

$$[\![ura]\!][p, [cls_{in}, d_{out}]] \; = \; ans$$

where p is a program, cls_{in} is an input class, and d_{out} the output. We say, tuple $[cls_{in}, d_{out}]$ is a *request* for inverse computation where class cls_{in} specifies the set of admissible input (the search space), and d_{out} is the fixed output. The set ans is a *solution* of the given inversion problem. It is a set of substitution-restriction pairs $ans = \{(\theta_1, \widehat{r}_1), \ldots\}$ which represents the largest subset of $\lceil cls_{in} \rceil$ such that $[\![p]\!] ds_{in} = d_{out}$ for all elements $(\theta_i, \widehat{r}_i)$ of the solution and $ds_{in} \in \lceil cls_{in} \rceil$. More formally, a correct solution to an inversion problem is specified by

$$\bigcup_i \lceil cls_{in}/\theta_i/\widehat{r}_i \rceil \; = \; \{ \; ds_{in} \;\mid\; ds_{in} \in \lceil cls_{in} \rceil, [\![p]\!] ds_{in} = d_{out} \; \} \; .$$

In the following sections we present each of the three steps:

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

The structure of the algorithm is illustrated in Fig. 5. Since our method is sound and complete, and since TSG is a universal programming language, which follows from the fact that the Universal Turing Machine can be defined in it, we can apply inverse computation, in principle, to any computable function. Thus our method of inverse computation has full generality.

The organization of inverse computation given here can be used for virtually any programming language. TSG is only a means to develop and fully formalize an algorithm for inverse computation. In fact, the set representation introduced in Sect. 2 can be used for any programming language with S-expressions, for example, for a subset of Lisp, or a simple flowchart language with S-expressions. Only the notions of state and configuration may change depending on the language. Changing the source language affects the construction of the perfect process tree, while the tabulation and inversion steps are not affected.

6 Walking the Perfect Process Tree

The transition relation in Fig. 9 defines walks through a *perfect process tree* [12]. Starting from a partially specified input, the goal is to follow all possible walks a standard evaluation may take under this generalized input. This will be the basis for inverse computation where the input of a program is only partially specified.

Process tree. A computation process is a potentially infinite sequence of states and transitions. Each state and transition in a deterministic computation is fully defined. The set of computation processes captures the semantics of a program as a whole. A *process tree* is used to examine the set of computation processes when the computation is not deterministic (because the input is only partly specified). Each node in a process tree contains a set of states represented by a *configuration*. A configuration which branches to two or more configurations in a process tree corresponds to a conditional transition from one set of program states to two or more sets of program states.

As defined in [12], a walk w in a process tree g is *feasible* if at least one initial state exists which selects w. A node n in a process tree g is feasible if it belongs at least to one feasible walk w in g. A process tree g is *perfect* if all walks in g are feasible.

Role of perfectness. 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.

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 terminating branch leads to a terminal state that can only be associated with an empty set of input in the solution (but soundness of the solution is preserved). Short, the correctness of the solution can be guaranteed, but an algorithm for inverse computation becomes less terminating and less efficient. The correctness of the solution cannot

Condition Eqt?

$$\frac{ea_1/\widehat{\sigma} = ea_2/\widehat{\sigma}}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{eqa?}\ ea_1\ ea_2)\ t_1\ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}), \kappa_{\mathsf{id}} \rangle}$$

$$\frac{ea_1/\widehat{\sigma} \neq ea_2/\widehat{\sigma} \quad (ea_1/\widehat{\sigma} \mathrel{\#} ea_2/\widehat{\sigma}) \notin \text{Tauto} \quad \kappa = [\, \mathit{mkBind}(ea_1/\widehat{\sigma}, ea_2/\widehat{\sigma})\,]}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{eqa?}\ ea_1\ ea_2)\ t_1\ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}), \kappa \rangle}$$

$$\frac{ea_1/\widehat{\sigma} \neq ea_2/\widehat{\sigma} \quad \kappa = \{(ea_1/\widehat{\sigma} \mathrel{\#} ea_2/\widehat{\sigma})\}}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{eqa?}\ ea_1\ ea_2)\ t_1\ t_2 \Rightarrow \langle (t_2, \widehat{\sigma}), \kappa \rangle}$$

Condition Cons?

$$\frac{e/\widehat{\sigma} = (\textbf{cons}\ \widehat{d}_1\ \widehat{d}_2) \quad \widehat{\sigma}' = \widehat{\sigma}[x_1 \mapsto \widehat{d}_1, x_2 \mapsto \widehat{d}_2]}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{cons?}\ e\ x_1\ x_2\ x_3)\ t_1\ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}'), \kappa_{\mathsf{id}} \rangle}$$

$$\frac{e/\widehat{\sigma} = \widehat{da} \quad \widehat{\sigma}' = \widehat{\sigma}[x_3 \mapsto \widehat{da}]}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{cons?}\ e\ x_1\ x_2\ x_3)\ t_1\ t_2 \Rightarrow \langle (t_2, \widehat{\sigma}'), \kappa_{\mathsf{id}} \rangle}$$

$$\frac{e/\widehat{\sigma} = Xe \quad \widehat{\sigma}' = \widehat{\sigma}[x_1 \mapsto Xe_1^{\diamond}, x_2 \mapsto Xe_2^{\diamond}] \quad \kappa = [Xe \mapsto (\textbf{cons}\ Xe_1^{\diamond}\ Xe_2^{\diamond})]}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{cons?}\ e\ x_1\ x_2\ x_3)\ t_1\ t_2 \Rightarrow \langle (t_1, \widehat{\sigma}'), \kappa \rangle}$$

$$\frac{e/\widehat{\sigma} = Xe \quad \widehat{\sigma}' = \widehat{\sigma}[x_3 \mapsto Xa^{\diamond}] \quad \kappa = [Xe \mapsto Xa^{\diamond}]}{\widehat{\sigma} \vdash_{\mathit{if}} (\textbf{cons?}\ e\ x_1\ x_2\ x_3)\ t_1\ t_2 \Rightarrow \langle (t_2, \widehat{\sigma}'), \kappa \rangle}$$

Terms

$$\frac{\widehat{\sigma} \vdash_{\mathit{if}} k\ t_1\ t_2 \Rightarrow \langle (t_i, \widehat{\sigma}'), \kappa \rangle \quad i \in \{1, 2\}}{\vdash_{\Gamma} ((\textbf{if}\ k\ t_1\ t_2), \widehat{\sigma}) \Rightarrow \langle (t_i, \widehat{\sigma}'), \kappa \rangle}$$

$$\frac{\Gamma(f) = (\textbf{define}\ f\ x_1 \ldots x_n\ t) \quad \widehat{\sigma}' = [x_1 \mapsto e_1/\widehat{\sigma}, \ldots, x_n \mapsto e_n/\widehat{\sigma}]}{\vdash_{\Gamma} ((\textbf{call}\ f\ e_1 \ldots e_n), \widehat{\sigma}) \Rightarrow \langle (t, \widehat{\sigma}'), \kappa_{\mathsf{id}} \rangle}$$

Transition

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

Semantic Values

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

Fig. 9. Trace semantics for perfect process trees of TSG-programs

be guaranteed without *applying perfect splits* because in this case empty sets of input cannot be detected neither during the development of the tree nor in the solution. Our formulation of the transition relation includes both operations.

Walking a process tree. Fig. 9 defines a transition relation $\mapsto$ between configurations. The transition relation does not actually construct a tree, but allows to perform all walks in a perfect process tree. The transition relation is non-deterministic when a condition (**eqa?**, **cons?**) cannot be decided. In this case the rules permit us to follow any of the two possible branches.

The transition rule states that a configuration $\langle \widehat{s}, \widehat{r} \rangle$ is transformed into a new configuration which is obtained by evaluating c-state $\widehat{s}$ to a new c-state $\widehat{s}'$, and applying contraction κ of the associated perfect split to configuration $\langle \widehat{s}', \widehat{r} \rangle$ if 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.

The rules for **if** and **call** are similar to the rules for the operational semantics in Fig. 5 except that they take a c-state to a new c-state and an associated contraction κ. In case of a call, identity contraction κ_{id} is returned (no split), in case of a conditional, contraction κ produced by evaluating condition k is returned.

We now describe the rules for conditions in more detail. The three rules for **eqa?** state that, depending on the equality of ca-expressions $ea_1/\widehat{\sigma}$ and $ea_2/\widehat{\sigma}$, a new c-state is formed which is associated with a contraction κ. The first equality rule applies if ca-expressions $ea_1/\widehat{\sigma}$ and $ea_2/\widehat{\sigma}$ are equal, which means they represent the same set of values. The second and third rule may apply at the same time. This is the case when $ea_1/\widehat{\sigma}$ and $ea_2/\widehat{\sigma}$ are not equal and at least one of the two ca-expressions is a c-variable (*i.e.*, inequality $(ea_1/\widehat{\sigma} \mathrel{\#} ea_2/\widehat{\sigma})$ is not a tautology). Then c-states $(t_1, \widehat{\sigma})$ and $(t_2, \widehat{\sigma})$ are associated with the corresponding contraction of the *perfect split* (Thm. 1, split 2, 3): $(t_1, \widehat{\sigma})$ is equipped with a substitution binding the ca-variable to the other ca-expression, and $(t_2, \widehat{\sigma})$ is equipped with an inequality between $ea_1/\widehat{\sigma}$ and $ea_2/\widehat{\sigma}$. 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 rule correspond to the two cons rules in Fig. 5 except that $e/\widehat{\sigma}$ is a c-expression. If $e/\widehat{\sigma}$ has outermost constructor **cons** then the true-branch is entered, otherwise, the false-branch is entered. In case $e/\widehat{\sigma}$ is a ce-variable Xe, the third and fourth rule apply and c-states $(t_1, \widehat{\sigma}_1)$ and $(t_2, \widehat{\sigma}_2)$ are equipped with the corresponding contraction of the perfect split (Thm. 1, split 4): $(t_1, \widehat{\sigma}_1)$ is equipped with a substitution instantiating Xe to a new cons-expression (where $Xe_1^\diamond$ and $Xe_2^\diamond$ are fresh ce-variables), and $(t_2, \widehat{\sigma}_2)$ is equipped with a substitution binding ce-variable Xe to a fresh ca-variable $Xa^\diamond$.

Correctness. Proving the trace semantics for perfect process trees (Fig. 9) correct *wrt* 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. We shall not be concerned with the technical details of the proofs in this paper, only with the fact [2] that the trace semantics is correct *wrt* the operational semantics.

Theorem 2 (correctness of initial configuration). *Let p be a well-formed* TSG*-program, let cls be well-formed input class for p, then*
Completeness and Soundness: $\lceil c^\circ(p, cls) \rceil = \{\ s^\circ(p, ds) \ \mid\ ds \in \lceil cls \rceil\ \}$.

Transition

$$\frac{\vdash_\Gamma \widehat{s} \Rightarrow \langle \widehat{s}', \kappa \rangle \qquad \widehat{r}/\kappa \neq \{\mathsf{contra}\}}{\Vdash_\Gamma (cls, \langle \widehat{s}, \widehat{r} \rangle) \rightarrow_{tab} (cls/\kappa, \langle \widehat{s}', \widehat{r} \rangle/\kappa)}$$

Semantic Values

$$tab \in \text{Tab} = \text{Class} \times \text{Cexp}$$

Fig. 10. Tabulation of TSG-programs

Theorem 3 (correctness of ppt-transition). *Let p be a well-formed* TSG-*program, and let c be a well-formed configuration for p, then*
Completeness: $\forall s \in \lceil c \rceil \,.\, \forall s' \,.\, (\Vdash_\Gamma s \rightarrow s') \Rightarrow (\exists c' \,.\, (\Vdash_\Gamma c \mapsto c' \wedge s' \in \lceil c' \rceil))$
Soundness: $\forall c' \,.\, (\Vdash_\Gamma c \mapsto c') \Rightarrow (\forall s' \in \lceil c' \rceil \,.\, \exists s \in \lceil c \rceil \,.\, \Vdash_\Gamma s \rightarrow s')$.

Theorem 4 (correctness of ppt). *Let p be a well-formed* TSG-*program, let cls be well-formed input class for p, then*
Completeness:

$$\forall ds \in \lceil cls \rceil \,.\, \forall s_0 \ldots s_n \,.\, s_0 = s^\circ(p, ds) \wedge (\wedge_{i=0}^{n-1} \Vdash_\Gamma s_i \rightarrow s_{i+1}) \quad \Rightarrow$$
$$\exists c_0 \ldots c_n \,.\, c_0 = c^\circ(p, cls) \wedge (\wedge_{i=0}^{n-1} \Vdash_\Gamma c_i \mapsto c_{i+1}) \wedge (\wedge_{i=0}^{n} s_i \in \lceil c_i \rceil)$$

Soundness:

$$\forall c_0 \ldots c_n \,.\, c_0 = c^\circ(p, cls) \wedge (\wedge_{i=0}^{n-1} \Vdash_\Gamma c_i \mapsto c_{i+1}) \quad \Rightarrow$$
$$\exists ds \in \lceil cls \rceil \,.\, \exists s_0 \ldots s_n \,.\, s_0 = s^\circ(p, ds) \wedge (\wedge_{i=0}^{n-1} \Vdash_\Gamma s_i \rightarrow s_{i+1}) \wedge (\wedge_{i=0}^{n} s_i \in \lceil c_i \rceil) \,.$$

Proof: Omitted (base case Thm. 2, induction step Thm. 3). ∎

7 Tabulation and Inversion

Before defining the solution of inverse computation, we define the tabulation of a program p for a given input class cls_{in}. Tabulation divides input class cls_{in} into disjoint input classes each of which is associated with a leave (output) in the process tree. All input-output pairs are collected in a set $TAB(p, cls_{in})$. For this we define a transition relation $\rightarrow_{tab}$ (Fig. 10) that carries an input class and applies to it every contraction κ encountered while following a path in the process tree. Finally, we define the solution of inverse computation as the set $ANS(p, cls_{in}, d_{out})$.

Definition 20 (tabulation). *Let p be a well-formed* TSG-*program, let cls_{in} be a well-formed input class for p. Define tabulation of p on cls_{in} as follows:*

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

Definition 21 (inverse computation). *Let p be a well-formed* TSG-*program, let cls_{in} be a well-formed input class for p, and let $d_{out} \in$* Dval. *Define inverse computation of p on cls_{in} and d_{out} as follows:*

$$ANS(p, cls_{in}, d_{out}) \stackrel{\text{def}}{=} \{ (\theta, \widehat{r}) \mid (cls, \widehat{d}) \in TAB(p, cls_{in}), \ \theta, \theta' \in \text{CCsub}, \ \widehat{r} \in \text{Restr}, \ \widehat{d}/\theta' = d_{out}, \ cls_{in}/\theta/\widehat{r} = cls/\theta' \} \,.$$

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 $[\![p]\!][d_1, \ldots, d_n] = d$, there is a input-output pair $(cls, \widehat{d}) \in TAB(p, cls_{in})$ such that $[d_1, \ldots, d_n] \in \lceil cls \rceil$ and $d \in \lceil \widehat{d} \rceil$. For soundness we must prove that each $(cls, \widehat{d}) \in TAB(p, cls_{in})$ and each $[d_1, \ldots, d_n] \in \lceil cls \rceil$ implies $[\![p]\!][d_1, \ldots, d_n] = d$ and $d \in \lceil \widehat{d} \rceil$. The corresponding argument for set $ANS(p, cls_{in}, d_{out})$ is based on the correctness of the tabulation. We shall not be concerned with the technical details of the proofs, only with the fact [2] that tabulation and inversion are correct *wrt* the operational semantics.

Theorem 5 (correctness of ***TAB*****).** *Let p be a well-formed* TSG*-program, let cls_{in} be a well-formed input class for p, and let $T = TAB(p, cls_{in})$, then completeness and soundness:*

$$\begin{aligned}\{\ (ds_{in}, d)\ \mid\ ds_{in} \in \lceil cls_{in} \rceil, [\![p]\!] ds_{in} = d\ \}\ =& \\ \{\ (\widehat{ds}/\theta, \widehat{d}/\theta)\ \mid\ (\langle \widehat{ds}, \widehat{r} \rangle, \widehat{d}) \in T, \theta \in FS(\langle \widehat{ds}, \widehat{r} \rangle), \widehat{r}/\theta = \emptyset\ \}\ .&\end{aligned}$$

Theorem 6 (correctness of ***ANS*****).** *Let p be a well-formed* TSG*-program, let cls_{in} be a well-formed input class for p, let $d_{out} \in$* Dval *and $A = ANS(p, cls_{in}, d_{out})$, then completeness and soundness:*

$$\{\ ds_{in}\ \mid\ ds_{in} \in \lceil cls_{in} \rceil, [\![p]\!] ds_{in} = d_{out}\ \}\ =\ \bigcup_{(\theta, \widehat{r}) \in A} \lceil cls_{in}/\theta/\widehat{r} \rceil\ .$$

The most important property of set $TAB(p, cls_{in})$ is *the perfectness property*—this allows us to inverse all input-output pairs independently and in any order.

Theorem 7 (perfectness of ***TAB*****).** *Let p be a well-formed* TSG*-program, let cls_{in} be a well-formed input class for p, and let $(cls_1, \widehat{d}_1)$ and $(cls_2, \widehat{d}_2)$ be two different input-output pairs from $TAB(p, cls_{in})$, then $\lceil cls_1 \rceil \cap \lceil cls_2 \rceil = \emptyset$.*

8 Algorithmic Aspects

In this section we discuss algorithmic aspects related to the Universal Resolving Algorithm and present our Haskell implementation. While Def. 21 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.

The algorithm is fully implemented in Hugs, a dialect of Haskell, a lazy functional language (321 lines of pretty-printed source text).[3] The algorithm is structured into 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 $ANS(p, cls_{in}, d_{out})$.

The main function `ura` which performs inverse computation is defined by

[3] Hugs-script available by `http://www.botik.ru/AbrGlu/URA/MPC2000`

```
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
                               (split,i') = splitE ce i
                               (S[_:->(CONS cxh cxt)],S[_:->cxa])=split
```

Fig. 11. Function ppt for constructing perfect process trees (written in Haskell)

```
ura  :: ProgTSG -> Class -> Dval -> [(CCsub,Restr)]
ura  p cls out = inv (tab (ppt p cls) cls) cls out
```

Given source program `p`, class `cls` and output `out`, 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

```
type Tab = [(Class, Cexp)]

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

inv  :: Tab -> Class -> Dval -> [(CCsub,Restr)]
inv  tab cls out = concat (map ans tab)
     where ans (cls_i, ce_i) =
            case (clash [ce_i] [out]) of
             (False, _)  -> []
             (True,sub') -> case cls_i' of
                             (_, [CONTRA]) -> []
                             _             -> [(sub, r)]
                            where cls_i' = cls_i/.sub'
                                  (sub, r) = subClassCntr cls cls_i'
```

Fig. 12. Functions `tab` and `inv` for tabulation and inversion (written in Haskell)

function `ura`. The type definitions `Class`, `Dval`, `CCsub` and `Restr` correspond to the domains Class, Dval, CCsub, and Restr; the source program is typed `ProgTSG`. The implementation of the functions `ppt`, `tab`, `inv` is shown in Figs. 11 and 12.

Function `ppt` in Fig. 11 implements the trace semantics from Fig. 9 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                  type Branch = (Contr, Tree)
          | NODE Conf [Branch]
```

For each rule that applies a branch is added (one branch if the transition is deterministic, two branches if the transition is non-deterministic). 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` (used for generating fresh variables). Infix operator `/.` implements substitution /, and infix operator `+.` implements update $\widehat{\sigma}[x_1 \mapsto \widehat{d}_1, \ldots, x_n \mapsto \widehat{d}_n]$.

Auxiliary functions `splitA` and `splitE` return the perfect splits for ca- and ce-variables, respectively (as defined in Thm. 1, perfect splits):

```
splitA :: CAvar -> CAexp -> Split              -- Thm.1: split 2,3
splitA cxa cea = (S[cxa:->cea], R[cxa:#:cea])
```

```
splitE :: CAvar -> FreeInd -> (Split,FreeInd)    -- Thm.1: 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. 12 consumes the process tree produced by `ppt` using a *breadth-first strategy*[4] 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. 12 enumerates the set $ANS(p, cls_{in}, d_{out})$ according to Def. 21. Two auxiliary functions `clash` and `subClassCntr` are used. Given $\widehat{ds}_1, \widehat{ds}_2 \in$ Cexps, the auxiliary function `clash` returns `(True,`θ`)` if a substitution $\theta \in$ CCsub exists such that $\widehat{ds}_1/\theta = \widehat{ds}_2$ and $dom(\theta) = var(\widehat{ds}_1)$; otherwise `(False,[])`. The requirement for the domain of θ ensures that no redundant bindings are added and that, if a solution exists, we produce a unique θ.

Given $cls, cls' \in$ Class where cls' can be obtained from cls by several contractions, the auxiliary function `subClassCntr` returns `(`θ`,` $\widehat{r}$`)` where $\theta \in$ CCsub, $\widehat{r} \in$ Restr such that $cls' = (cls/\theta)/\widehat{r}$ and $dom(\theta) = var(cls)$.

Termination. Of course, inverse computation is undecidable, so an algorithm cannot be sound, complete, and terminating. Our algorithm is sound and complete, but not always terminating. Each solution, if it exists, is computed in finite time due to the breadth-first strategy. The algorithm does not always terminate because the search for solutions in a process tree may continue infinitely (even though all elements of the solution were found). The algorithm terminates if all branches in a process tree are finite.

9 Experiments and Results

This section illustrates the Universal Resolving Algorithm by means of examples. The first example illustrates inverse computation of a pattern matcher, the second example demonstrates the inverse interpretation of While-programs.[5]

Pattern matching. We performed the two inversion tasks from Sect. 1 using a naive pattern matcher written in TSG (Fig. 13).

- **Task 1**: Find the set of strings `pattern` which *are* substrings of `"ABC"`. To perform this task we leave input `pattern` unknown (Xe_1), set input `string = "ABC"` and the desired output to `'SUCCESS`.
- **Task 2**: Find the set of strings `pattern` which *are not* substrings of `"AAA"`. To perform this task we use a setting similar to Task 1 (`pattern =` Xe_1, `string = "AAA"`), but the desired output is set to `'FAILURE`.

[4] 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`.

[5] Run times given for Hugs 98, PC/Intel Pentium MMX-233MHz, MS Windows 95.

```
match =
 [(DEFINE "Match"[p,s]                  (DEFINE "NextPos" [p,s]
    (CALL"CheckPos"[p,s,p,s]) ),          (IF (CONS? s sh st a)
  (DEFINE "CheckPos" [p,s,pp,ss]            (CALL "Match" [p,st])
    (IF (CONS? p ph pt a)                   'FAILURE ) ) ]
      (IF (CONS? ph _ _ a'ph)
        'ERROR:Atom_expected
        (IF (CONS? s sh st a)
          (IF (CONS? sh _ _ a'sh)
            'ERROR:Atom_expected
            (IF (EQA? a'ph a'sh)
              (CALL "CheckPos"[pt,st,pp,ss])
              (CALL "NextPos" [pp,ss]) ) )
          'FAILURE ) )
      'SUCCESS ) ),
```

Fig. 13. Naive pattern matcher written in concrete TSG syntax

Figure 14 shows the results of applying URA to the matcher. The answer for Task1 is a finite representation of all possible substrings of string `"ABC"`, Fig. 14(i). The answer for Task 2 is a finite representation of all strings which are not substrings of `"AAA"`, Fig. 14(ii). URA terminates after 0.5 seconds (Task 1, Task 2).

Interpreter inversion. As proven in [3, 4], inverse computation can be performed in a programming language N given a standard interpreter *intN* for N written in L, and an inverse interpreter for L. The result obtained by inverse computation of N's interpreter is a solution for inverse computation in N. The theorem guarantees that the solution is *correct* for all N-program regardless of *intN*'s operational properties. Since TSG is a universal programming language we can, in principle, perform inverse computation in any programming language.

According to this result, we should now be able to apply our algorithm to programs written in languages other than TSG. To put this theorem to a practical trial, we implemented an interpreter for an imperative language, called MP, in TSG. MP [27] is a small *imperative language* with assignments (`<==`), conditionals (`oIF`) and loops (`oWHILE`). An MP-program operates over a store consisting of parameters and local variables. The semantics is conventional Pascal-style semantics. The MP-interpreter has 309 lines of pretty-printed source text, 30 functions in TSG, and is the largest TSG-program we implemented.

To compare the results with inverse computation in TSG, we rewrote the naive pattern matcher in MP. Figure 14 shows the results for the inversion of the MP-matcher. The answer for Task 1 is a finite representation of all possible substrings of string `"ABC"`, Fig. 14(iii). The answer for Task 2 is a finite representation of all strings which are not substrings of `"AAA"`, Fig. 14(iv). URA terminates after 36 sec (Task 1) and after 34 sec (Task 2).

```
(i)   ura [ match, [ ([Xe1, str"ABC"],[]), 'SUCCESS ]] =*=>
        [ ([Xe1 |-> Xa4], []),                                  --str""
          ([Xe1 |-> (CONS 'A Xa10)], []),                       --str"A"
          ([Xe1 |-> (CONS 'A (CONS 'B Xa16))], []),             --str"AB"
          ([Xe1 |-> (CONS 'B Xa10)], []),                       --str "B"
          ([Xe1 |-> (CONS 'A (CONS 'B (CONS 'C Xa22)))], []),   --str"ABC"
          ([Xe1 |-> (CONS 'B (CONS 'C Xa16))], []),             --str "BC"
          ([Xe1 |-> (CONS 'C Xa10)], []) ]                      --str  "C"
(ii)  ura [ match, [ ([Xe1, str"AAA"],[]), 'FAILURE ]] =*=>
        [ ([Xe1 |-> (CONS 'A (CONS 'A (CONS 'A (CONS Xa25 Xe21))))],[]),
          ([Xe1 |-> (CONS Xa7 Xe3)],[Xa7:#:'A]),
          ([Xe1 |-> (CONS 'A (CONS 'A (CONS Xa19 Xe15)))],[Xa19:#:'A]),
          ([Xe1 |-> (CONS 'A (CONS Xa13 Xe9))],[Xa13:#:'A]) ]
(iii) ura [ intMP, [[matchMP,([Xe1,str"ABC"],[])],'SUCCESS]] =*=>
        [ ([Xe1 |-> Xa4], []),                                  --str""
          ([Xe1 |-> (CONS 'A Xa10)], []),                       --str"A"
          ([Xe1 |-> (CONS 'A (CONS 'B Xa16))], []),             --str"AB"
          ([Xe1 |-> (CONS 'B Xa10)], []),                       --str "B"
          ([Xe1 |-> (CONS 'A (CONS 'B (CONS 'C Xa22)))], []),   --str"ABC"
          ([Xe1 |-> (CONS 'B (CONS 'C Xa16))], []),             --str "BC"
          ([Xe1 |-> (CONS 'C Xa10)], []) ]                      --str  "C"
(iv)  ura [ intMP, [[matchMP,([Xe1,str"AAA"],[])],'FAILURE]] =*=>
        [ ([Xe1 |-> (CONS 'A (CONS 'A (CONS 'A (CONS Xe20 Xe21))))],[]),
          ([Xe1 |-> (CONS Xa7 Xe3)],[Xa7:#:'A]),
          ([Xe1 |-> (CONS 'A (CONS 'A (CONS Xa19 Xe15)))],[Xa19:#:'A]),
          ([Xe1 |-> (CONS 'A (CONS Xa13 Xe9))],[Xa13:#:'A]) ]
```

Fig. 14. Inverse computation of pattern matcher (i, ii) and interpreter (iii, iv)

Inverse computation in MP (implemented by `ura` and `intMP`) produces results very similar[6] to inverse computation in TSG (implemented directly by `ura`). This is noteworthy because inverse computation in MP is done through a *standard interpreter* for MP (and not by an inverse interpreter for MP). It demonstrates that inverse computation can be ported successfully, here, from a functional language to an imperative language. Inverse computation in MP takes longer than in TSG due to the additional interpretive overhead (about 70 times).

In earlier work [4], inverse computation was ported from TSG to a small assembler-like programming language (called Norma). The only other experimental work we are aware of that ported inverse computation, inverses imperative programs by treating their relational semantics as logic program [26]. Our

[6] The results differ slightly (Fig. 14: compare (ii) line 2 and (iv) line 2) due to small differences in the implementation of the source programs.

experiment gives further practical evidence for the idea of porting inverse computation from one language to another.

10 Related Work

The first work on program inversion appears to be [22], suggesting a *generate and test approach* for Turing machines; this will correctly find an inverse when it exists, but is computationally infeasible. Several efforts have gone into *imperative programs* [16, 7, 17, 6] but use non-automatic (sometimes heuristic) methods for deriving the inverse program. For example, the technique suggested in [7] provides for inverting programs symbolically, but requires that the programmer provide inductive assertions on conditionals and loop statements.

Few papers have been devoted to inversion of *functional programs* [5, 9, 18, 20, 21, 25] in a similar manner, sometimes automatically. The work in functional languages is usually on program inversion. An automatic system for synthesizing recursive programs from first-order functional programs is InvX [20]. The inverse of functions has been paid attention to, at least conceptually, in program analysis and program verification (*e.g.*, [8, 24]).

An early result [28] for inverse computation in a functional language was obtained in 1972 by a unification-based transformation technique called *driving* [29] which was used to perform subtraction by inverse computation of binary addition. Later, universal resolving algorithms were implemented using methods from supercompilation [29] for first-order functional programs by combining them with a mechanical extraction of answers (*cf.* [1, 25]).

We know of two techniques for inverse computation in functional languages: the universal resolving algorithm (see [1, 4]) and walk grammars for inverse interpretation [30, 23]. The universal resolving algorithm in this paper uses methods from supercompilation [29], in particular driving, and is based on perfect process trees [12]. Connections between inverse computation and logic programming are discussed in [1, 4]; partial deduction and driving were formally related in [14]. An abstract framework for describing partial evaluation and supercompilation is [19]. A comprehensive bibliography on supercompilation can be found in [15].

To conclude, there exists only a small number of papers addressing inverse computation in the context of functional languages. With the exception of [26, 4], we know of no paper addressing inverse computation in imperative languages.

11 Conclusion and Future Work

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

Among others, a motivation for our work was the thesis [13] that program inversion is one of the three fundamental operations on programs (beside program specialization and program composition). We believe that to achieve full generality of program manipulation, ultimately all three operations have to be mastered. So far, progress has been achieved mostly on program specialization.

For future work it is desirable, though not difficult, to extend our algorithm to user-defined constructor domains. This requires an extension of the set representation in Sect. 2 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 S-expressions familiar from Lisp.

In general, cutting all infeasible branches from a process tree cannot be guaranteed, in particular, when the underlying logic of the set representation is undecidable for certain logic formulas (or too time consuming to prove). For example, this is the case when using a tree developer based on generalized partial computation [10]. In this case, the solution of inverse computation may contain elements which represent empty sets of input (the correctness of the solution can be preserved). The set representation we used expresses structural properties of values that can always be resolved. Perfect splits are essential to guarantee the correctness of the solution, cutting infeasible branches improves termination and efficiency of the algorithm.

The question of a more efficient implementation is also left for future work. Our algorithm is fully implemented in Haskell and 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). The design of a more efficient algorithm would require to merge these steps. Compilation techniques and strategies developed for logic programming may be beneficial for a more practical implementation.

Finally, the relation to narrowing used in logic-functional programming and term rewriting should be studied more formally (reference [14] relates driving and partial deduction).

Acknowledgments

The authors would like to thank their colleagues from the Refal group for various discussions related to this work. Special thanks are due to Jeremy Gibbons for suggesting to submit this material to MPC, to the five anonymous referees for valuable comments, and to Kazuhiko Kakehi for careful proofreading. The second author would like to thank Michael Leuschel for joint work leading to some of the material in Section 10. Special thanks are due to Yoshihiko Futamura for generous support of this research. The authors were also supported by the Japan Society for the Promotion of Science and the Danish Natural Sciences Research Council.

References

1. S. M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).

2. S. M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications).* Nauka-Fizmatlit, Moscow 1995. (In Russian).
3. S. M. Abramov, R. Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, to appear.
4. S. M. Abramov, R. Glück. Semantics modifiers: an approach to non-standard semantics of programming languages. In M. Sato, Y. Toyama (eds.), *Third Intern. Symposium on Functional and Logic Programming*, 247–270. World Scientific, 1998.
5. R. Bird, O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.
6. W. Chen, J. T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15:1–13, 1990.
7. E. W. Dijkstra. EWD671: Program inversion. In *Selected Writings on Computing: A Personal Perspective*, 351–354. Springer-Verlag, 1982.
8. P. Dybjer. Inverse image analysis generalises strictness analysis. *Information and Computation*, 90(2):194–216, 1991.
9. D. Eppstein. A heuristic approach to program inversion. In *Intern. Joint Conf. on Artificial Intelligence (IJCAI-85)*, 219–221. William Kaufmann Inc., 1985.
10. Y. Futamura, K. Nogi, A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
11. R. Glück, J. Hatcliff, J. Jørgensen. Generalization in hierarchies of online program specialization systems. In P. Flener (ed.), *Logic-Based Program Synthesis and Transformation. Proceedings*, LNCS 1559, 179–198. Springer-Verlag, 1999.
12. R. Glück, A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot et al. (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.
13. R. Glück, A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl (ed.), *Cybernetics and Systems'94*, 1563–1570. World Scientific, 1994.
14. R. Glück, M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, 165–181. Springer-Verlag, 1994.
15. R. Glück, M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy et al. (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 137–160. Springer-Verlag, 1996.
16. D. Gries. Inverting programs (chapter 21). In *The Science of Programming*, 265–274. Springer-Verlag, 1981.
17. D. Gries, J. L. A. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In E. W. Dijkstra (ed.), *Formal Development of Programs and Proofs*, 37–42. Addison Wesley, 1990.
18. P. G. Harrison, H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.
19. N. D. Jones. The essence of program transformation by partial evaluation and driving. In N. D. Jones et al. (eds.), *Logic, Language and Computation.* LNCS 792, 206–224. Springer-Verlag, 1994.
20. H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications (RTA'89)*, LNCS 355, 564–568. Springer-Verlag, 1989.
21. R. E. Korf. Inversion of applicative programs. In *Proceedings of the Seventh Intern. Joint Conference on Artificial Intelligence (IJCAI-81)*, 1007–1009. William Kaufmann, Inc., 1981.

22. J. McCarthy. The inversion of functions defined by Turing machines. In C. E. Shannon, J. McCarthy (eds.), *Automata Studies*, 177–181. Princeton University Press, 1956.
23. A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner et al. (eds.), *Perspectives of System Informatics*, LNCS 1181, 249–260. Springer-Verlag, 1996.
24. M. Ogawa. Automatic verification based on abstract interpretation. In A. Middeldorp, T. Sato (eds.), *Functional and Logic Programming. Proceedings*, LNCS 1722, 131–146. Springer-Verlag, 1999.
25. A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner et al. (eds.), *Partial Evaluation and Mixed Computation*, 427–444. North-Holland, 1988.
26. B. J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348, 1997.
27. P. Sestoft. The structure of a self-applicable partial evaluator. Technical Report 85/11, DIKU, University of Copenhagen, Denmark, 1985.
28. V. F. Turchin. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal). In *Teorija Jazykov i Metody Programmirovanija (Proceedings of the Symp. on the Theory of Languages and Programming Methods)*, 31–42, 1972. (In Russian).
29. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
30. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.