

# An Universal Resolving Algorithm for Inverse Computation of Lazy Languages

Sergei Abramov<sup>1</sup>, Robert Glück<sup>2</sup>, and Yuri Klimov<sup>3</sup>

<sup>1</sup> Program Systems Institute, Russian Academy of Sciences  
RU-152140 Pereslavl-Zalessky, Russia, [abram@botik.ru](mailto:abram@botik.ru)

<sup>2</sup> DIKU, Dept. of Computer Science, University of Copenhagen  
DK-2100 Copenhagen, Denmark, [glueck@acm.org](mailto:glueck@acm.org)

<sup>3</sup> M.V. Keldysh Institute for Applied Mathematics, Russian Academy of Sciences  
RU-125047 Moscow, Russia, [yuri@klimov.net](mailto:yuri@klimov.net)

**Abstract.** The Universal Resolving Algorithm [3] was formulated for inverse computation of tail-recursive programs. We present an extension of the algorithm to deal with recursive programming languages. This extension improves the efficiency and termination behavior of inverse computation because partially produced output is used to reduce the search space. We explain the extension and present a new technique which we designed and implemented for a first-order, lazy functional programming language. Several examples demonstrate the advantages of the new technique. We expect that similar methods can be used when performing inverse computation in other programming languages.

**Keywords:** Inverse computation, perfect process tree, non-standard semantics, functional programming, unification.

## 1 Introduction

Many problems in computation can be specified in terms of computing the inverse of an easily constructed function [5]. Inverse computation is the calculation of the possible input of a program for a given output. The *Universal Resolving Algorithm* (URA) [2, 3] is an algorithm for inverse computation in a functional language with tail-recursion. The algorithm is sound and complete with respect to the solutions defined by a given program, but not always terminating. Termination and efficiency depends directly on the search space traversed when performing inverse computation. The original definition of the algorithm relies on perfect splits to reduce the search space.

The original algorithm was formulated for a tail-recursive programming language. Since our algorithm is sound and complete [3], other algorithms can not improve on this property, but they can be more efficient. In this paper we present an extension of the original algorithm to programming languages with general recursion. This allows us to reduce the search space drastically when partially defined output becomes available. We show how termination and efficiency of inverse computation can be improved by some simple techniques. We demonstrate

the gains of our method with several examples. We should hasten to stress that a tail-recursive language is computationally complete, which follows from the fact that the Universal Turing Machine can be programmed in it. Thus, the method has full generality and can be applied to any computable function. This paper concerns the question on how to make inverse computation faster and more terminating. Another proposal [21] which approximates functional programs by grammars is complete, but sacrifices soundness for termination.

The paper is organized as follows. After presenting the principles of URA (Sect. 2), we describe the source language (Sect. 3) and explain the principles of reducing the search space during inverse computation (Sect. 4). Then we define a surprisingly simple equivalence transformation for reducing the search space (Sect. 5) and demonstrate the advantages of the new technique with several examples (Sect. 6). We conclude with related work (Sect. 7) and discuss future work (Sect. 8).

## 2 Background: An Approach to Inverse Computation

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

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

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

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.

In general, inverse computation using an algorithm for inverse computation *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 the input-output class  $cls_{io}$  and of the solution set  $ans$ . In this paper we use S-expressions known from Lisp [15] as the value domain and represent the search space  $cls_{io}$  by *expressions with variables* and *restrictions* that constrain the domains of the

variables [2, 3, 18, 19]. (Other algorithms for inverse computation may choose other representations.)

The *Universal Resolving Algorithm* (URA) [3, 2] is an algorithm for inverse computation in a first-order functional language with tail-recursion. The answer produced by URA is a set of substitution-restriction pairs  $ans = \{(\theta_1, \hat{r}_1), \dots\}$  which represents set  $Inv$  for the given inversion problem. More formally, the correctness of the answer produced by URA is given by

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

where  $(cls_{io}/\theta_i)/\hat{r}_i$  is narrowing of given input-output class  $cls_{io}$  by applying substitution  $\theta_i$  to  $cls_{io}$  and adding restriction  $\hat{r}_i$ . Our algorithm produces a universal solution, hence the first word of its name.

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).

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.

Our approach is based on the notion of a *perfect process tree* [6] which represents the computation of a program with *partially specified input* (class  $cls_{in}$  taken from  $cls_{io}$ ) by a tree of all possible computation traces. Each fork in a perfect tree partitions the input class  $cls_{in}$  into disjoint and exhaustive subclasses. The algorithm then constructs, breadth-first and lazily, a perfect process tree for a given program  $p$  and input class  $cls_{in}$ . Note that we first construct a forward trace of the computation given  $p$  and  $cls_{in}$ , and then use  $cls_{io}$  to extract the solution to the backward problem. The construction of a process tree is similar to unfolding in partial evaluation where a computation is traced under partially specified input (*cf.* [9]).

Criterion of termination and correctness of the original algorithm are proven, and several examples are shown in the literature [3]. Since the algorithm is sound and complete, other algorithms can not improve on this property, but they can have better efficiency and termination behavior. In this paper we present an extension of the original algorithm to programming languages with general recursion.

### 3 Source Language

In this paper a first-order, lazy functional language NTSG is used. The language extends S-Graph [6] with nested function calls and non-atomic equality. The

<i>Grammar</i>		
$p ::= q^+$		Program
$q ::= (\text{define } f \ x^* \ e)$		Definition
$e ::= (\text{call } f \ e^*) \mid (\text{if } k \ e \ e) \mid (e : e) \mid 'z \mid x$		Expression
$k ::= (\text{equ? } e \ e) \mid (\text{cons? } e \ xe \ xe \ xa)$		Condition
$x ::= xe \mid xa$		Typed Variable
<i>Syntax Domains</i>		
$p \in \text{Program}$	$k \in \text{Cond}$	$x \in \text{Pvar}$
$q \in \text{Definition}$	$f \in \text{Fname}$	$xe \in \text{PEvar}$
$e \in \text{Pexp}$	$z \in \text{Symb}$	$xa \in \text{PAvar}$

**Fig. 1.** Abstract syntax of NTSG

body of a function is a expression which is either a function call, a conditional, a cons-pair  $(e : e)$ , an atom  $('z)$  or a variable, Fig. 1. The semantics of NTSG (Fig. 2) is similar to the semantics of TSG that has been given elsewhere [2, 3]. Values can be tested and/or decomposed in two ways. Test **equ?** checks the equality of S-expressions (discussed below), and **(cons?  $e \ xe' \ xe'' \ xa$ )** works in the following way: if  $e$  has a form  $(e' : e'')$ , then variable  $xe'$  is bound to head  $e'$  and variable  $xe''$  to tail  $e''$ ; if  $e$  is an atom, then this atom is bound to variable  $xa$ . We simply write  $'_$  when a variable is not used (*e.g.*, in the first conditional of function a2b where the else-branch returns an empty list, Fig. 5).

**MGU-based Term Equality** (RG051022: new section intro needed: focus on mgu-based equality. The good effect for computation (test fails if any component in known expressions disagree; fair to all subexpressions and independent of order in expressions) and URA (new 'self-application' trick with transformation of source program; gentle extension of source semantics).)

The languages also include an equality predicate for expressions in the languages. For simplicity, this is sometimes replaced by exhaustive pattern matching, but as we shall see in the next section, an equality predicate in the source language allows us to achieve the desired effects in an elegant way. In particular, we want to ensure the preservation of the operational semantics of program transformers based on this language. We choose to use the *most general unifier* (*mgu*) as the semantics of equality in the source languages. The *mgu* gives preference to none of the expressions involved in a comparison, and allows us to obtain 'fail' as soon as two constructors in a partially computed expression mismatch. It also allows us to use the same machinery in the source language and in driving, which allows us to preserve the operational semantics and to eliminate branches during driving as early as possible.

The semantics of the test **(equ?  $e' \ e''$ )** is the following (Fig. 2). The test chooses the false branch ( $e_2$ ) as soon as the skeletons of  $e'$  and  $e''$  are known to have different constructors (operator *skel* replaces in a expression all function

*Condition Equ?*

$$\frac{mgu(skel(e'), skel(e'')) \text{ fails}}{\vdash_{if} (\mathbf{equ?} \ e' \ e'') \ e_1 \ e_2 \Rightarrow e_2}$$

$$\frac{mgu(skel(e'), skel(e'')) = \kappa_{id} \quad (e' = skel(e') \wedge e'' = skel(e''))}{\vdash_{if} (\mathbf{equ?} \ e' \ e'') \ e_1 \ e_2 \Rightarrow e_1}$$

*Condition Cons?*

$$\frac{e = (e' : e'')}{\vdash_{if} (\mathbf{cons?} \ e \ xe' \ xe'' \ xa) \ e_1 \ e_2 \Rightarrow e_1 / [xe' \mapsto e', xe'' \mapsto e'']}$$

$$\frac{e = 'z}{\vdash_{if} (\mathbf{cons?} \ e \ xe' \ xe'' \ xa) \ e_1 \ e_2 \Rightarrow e_2 / [xa \mapsto 'z]}$$

*Expressions*

$$\frac{\Gamma(f) = (\mathbf{define} \ f \ x_1 \dots x_n \ e)}{\vdash_{\Gamma} (\mathbf{call} \ f \ e_1 \dots e_n) \Rightarrow e / [x_1 \mapsto e_1, \dots, x_n \mapsto e_n]} \quad \frac{\vdash_{if} \ k \ e_1 \ e_2 \Rightarrow e}{\vdash_{\Gamma} (\mathbf{if} \ k \ e_1 \ e_2) \Rightarrow e}$$

*Transition*

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

*Semantic Values*

$$s \in \text{PDstate} = \text{Pexp} \quad \Gamma \in \text{ProgMap} = \text{Fname} \rightarrow \text{Definition}$$

**Fig. 2.** Operational semantics of NTSG-programs

calls and **if**-subexpressions by fresh variables). The true branch ( $e_1$ ) is chosen if  $e'$  and  $e''$  are identical and *passive*. We say that an expression  $e$  is passive iff it contains no function calls and **if**-subexpressions:  $e = skel(e)$ .<sup>4</sup>

## 4 Reducing the Search Space

When performing inverse computation in languages with general recursion, we can take advantage of partially computed output and thereby reduce the search space drastically. Before we present the technique which we used for a lazy functional language, we show how this goal can be achieved in languages with nested function calls. We explain the main ideas with two examples, and then define our method.

<sup>4</sup> Remark: In the operational semantics, when both operands are passive,  $mgu$  reduces to pattern matching; only during driving, the mechanism is fully used. For formal reasons, we prefer  $mgu$ .

<i>Grammar</i>	
$\widehat{e} ::= (\text{call } f \widehat{e}^*) \mid (\text{if } \widehat{k} \widehat{e} \widehat{e}) \mid (\widehat{e} : \widehat{e}) \mid 'z \mid x \mid Xe \mid Xa$	C-Expression
$\widehat{k} ::= (\text{equ? } \widehat{e} \widehat{e}) \mid (\text{cons? } \widehat{e} xe xe xa)$	C-Condition
<i>Syntax Domains</i>	
$\widehat{e} \in \text{Cexp}$	$\widehat{k} \in \text{Ccond}$

**Fig. 3.** Program-related constructions with c-variables

The process of tracing a program (shown in Fig. 3 and 4, discussed in details in [2,3]) with partially specified input and constructing a perfect process tree can be optimized by two operations:

1. examine partially computed output to cut branches, and
2. backpropagate new bindings to reduce the search space.

Both operations can make inverse computation *more efficient* and *more terminating*. We illustrate this with two example NTSG-programs (Fig. 5). Program a2b replaces each 'A by 'B in a list of symbols; leaving all other symbols unchanged. Function f tuples the input and the output of function a2b. For instance, applying a2b to a symbol list ['A, 'B, 'C] returns the symbol list ['B, 'B, 'C]:

$$\llbracket \text{a2b} \rrbracket \llbracket ['A, 'B, 'C] \rrbracket = \llbracket ['B, 'B, 'C] \rrbracket .$$

**Cutting branches** The first example is inverse computation of program a2b. Suppose we have output ['B] and want to find all possible inputs which produce this output. We specify the input and output domain for inverse computation by the input-output class

$$cls_{io} = \langle \underbrace{([Xe_1])}_{\widehat{ds}_{in}}, \underbrace{['B]}_{\widehat{d}_{out}}, \underbrace{\emptyset}_{\widehat{r}_{io}} \rangle$$

where  $\widehat{ds}_{in}$  is the partially specified input,  $\widehat{d}_{out}$  is the desired output, and  $\widehat{r}_{io} = \emptyset$  is empty restriction (does not constrain the domains of c-variables). Placeholders  $Xe_i$  are called *configuration variables* (c-variables); they range over the set of S-expressions. Inverse computation with URA then takes the form:

$$\llbracket \text{ura} \rrbracket \llbracket \text{a2b}, cls_{io} \rrbracket = ans .$$

As the answer *ans* of inverse computation, we expect a (possibly infinite) sequence of substitution-restriction pairs for the c-variables occurring in  $cls_{io}$  (cf. Eq. 3). In our example, there is one c-variable,  $Xe_1$ , and we expect two substitutions as answer:  $[Xe_1 \mapsto ['A]]$  and  $[Xe_1 \mapsto ['B]]$ .

Condition Equ?

$$\begin{array}{c}
\frac{mgu(skel(\hat{e}'), skel(\hat{e}'')) \text{ fails}}{\vdash_{if} (\mathbf{equ?} \hat{e}' \hat{e}'') \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_2, \kappa_{id})} \quad \frac{mgu(skel(\hat{e}'), skel(\hat{e}'')) = \kappa}{\vdash_{if} (\mathbf{equ?} \hat{e}' \hat{e}'') \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_2, \neg \kappa)} \\
\\
\frac{mgu(skel(\hat{e}'), skel(\hat{e}'')) = \kappa \quad (\hat{e}' = skel(\hat{e}') \wedge \hat{e}'' = skel(\hat{e}''))}{\vdash_{if} (\mathbf{equ?} \hat{e}' \hat{e}'') \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_1, \kappa)} \\
\\
\frac{mgu(skel(\hat{e}'), skel(\hat{e}'')) = \kappa \quad \neg(\hat{e}' = skel(\hat{e}') \wedge \hat{e}'' = skel(\hat{e}''))}{\vdash_{if} (\mathbf{equ?} \hat{e}' \hat{e}'') \hat{e}_1 \hat{e}_2 \Rightarrow ((\mathbf{if} (\mathbf{equ?} \hat{e}' \hat{e}'') \hat{e}_1 \hat{e}_2), \kappa)}
\end{array}$$

Condition Cons?

$$\begin{array}{c}
\frac{\hat{e} = (\hat{e}' : \hat{e}'')}{\vdash_{if} (\mathbf{cons?} \hat{e} xe' xe'' xa) \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_1/[xe' \mapsto \hat{e}', xe'' \mapsto \hat{e}''], \kappa_{id})} \\
\\
\frac{\hat{e} = 'z}{\vdash_{if} (\mathbf{cons?} \hat{e} xe' xe'' xa) \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_2/[xa \mapsto 'z], \kappa_{id})} \\
\\
\frac{\hat{e} = Xe \quad \kappa = [Xe \mapsto (Xe'^{\circ} : Xe''^{\circ})]}{\vdash_{if} (\mathbf{cons?} \hat{e} xe' xe'' xa) \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_1/[xe' \mapsto Xe'^{\circ}, xe'' \mapsto Xe''^{\circ}], \kappa)} \\
\\
\frac{\hat{e} = Xe \quad \kappa = [Xe \mapsto Xa^{\circ}]}{\vdash_{if} (\mathbf{cons?} \hat{e} xe' xe'' xa) \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_2/[xa \mapsto Xa^{\circ}], \kappa)} \\
\\
\frac{\hat{e} = Xa}{\vdash_{if} (\mathbf{cons?} \hat{e} xe' xe'' xa) \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}_2/[xa \mapsto Xa], \kappa_{id})}
\end{array}$$

Expressions

$$\frac{\Gamma(f) = (\mathbf{define} f x_1 \dots x_n e)}{\vdash_{\Gamma} (\mathbf{call} f \hat{e}_1 \dots \hat{e}_n) \Rightarrow (e/[x_1 \mapsto \hat{e}_1, \dots, x_n \mapsto \hat{e}_n], \kappa_{id})} \quad \frac{\vdash_{if} k \hat{e}_1 \hat{e}_2 \Rightarrow (\hat{e}, \kappa)}{\vdash_{\Gamma} (\mathbf{if} k \hat{e}_1 \hat{e}_2) \Rightarrow (\hat{e}, \kappa)}$$

Transition

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

Semantic Values

$$\hat{s} \in \text{PCstate} = \text{Cexp} \quad \Gamma \in \text{ProgMap} = \text{Fname} \rightarrow \text{Definition}$$

**Fig. 4.** Trace semantics for perfect process trees of NTSG-programs

Tracing a program with partially specified input,  $cls_{in} = \langle \hat{ds}_{in}, \hat{r}_{io} \rangle$ , may confront us with tests that depend on unspecified values (represented by c-variables), and we have to consider the possibility that either branch is entered with some input value. This leads to traces that branch at conditionals that depend on un-

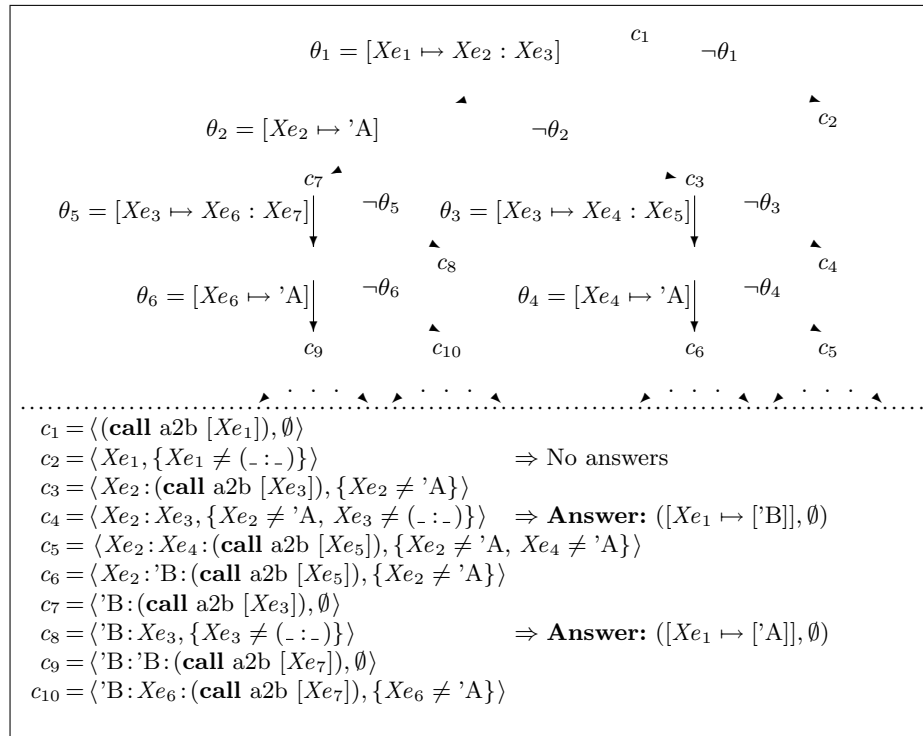
```

(define f [x]
  ([x, (call a2b [x])]))

(define a2b [x]
  (if (cons? x h t _)
      (if (equ? h 'A)
          ('B:(call a2b [t]))
          (h:(call a2b [t])))
      []))

```

**Fig. 5.** Program ‘f’ and program ‘a2b’



**Fig. 6.** Cutting branches desired

specified values. Tracing a computation is called *driving* in supercompilation [24]; the method used below is *perfect driving* [6]. (A variant is *positive driving* [22].)

The perfect process tree of our example is shown in Fig. 6. Tracing starts in the root, and then proceeds with one of the successor configurations depending on the shape of the values in the c-environment. For example, the first test we encounter in our program is **cons?**. It tests whether the value of program variable



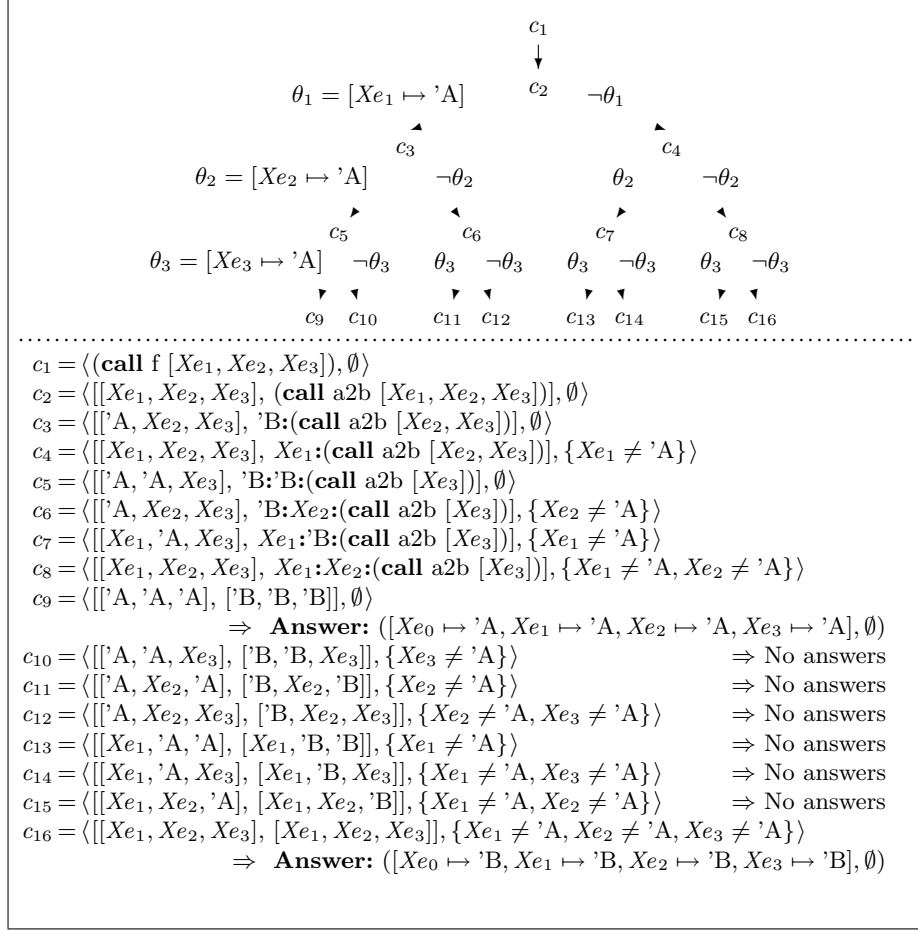


Fig. 7. Backpropagation desired

$x$  is a pair. Since  $x$  is bound to  $Xe_1$ , we have to consider two possibilities (*perfect split*): is a pair of the form  $Xe_2: Xe_3$  or not a pair. These two assumptions lead to two new configurations in Fig. 6.

Let us emphasize that in the driving *perfect splits*  $(\theta, \neg\theta)$  — pair of contructions, — are used, where  $\theta$  is substitution, *e.g.* conjunction of equalities (presented as bindings),  $\neg\theta$  is negation of  $\theta$ , presented by disjunction of corresponding inequalities. Thus, in NTSG the restriction system is more complex than was used in TSG [3]. In general case a restriction is a conjunction of disjunctions of inequalities. Such restriction system, operations with restrictions and its proven properties are described in details in [18, 19].

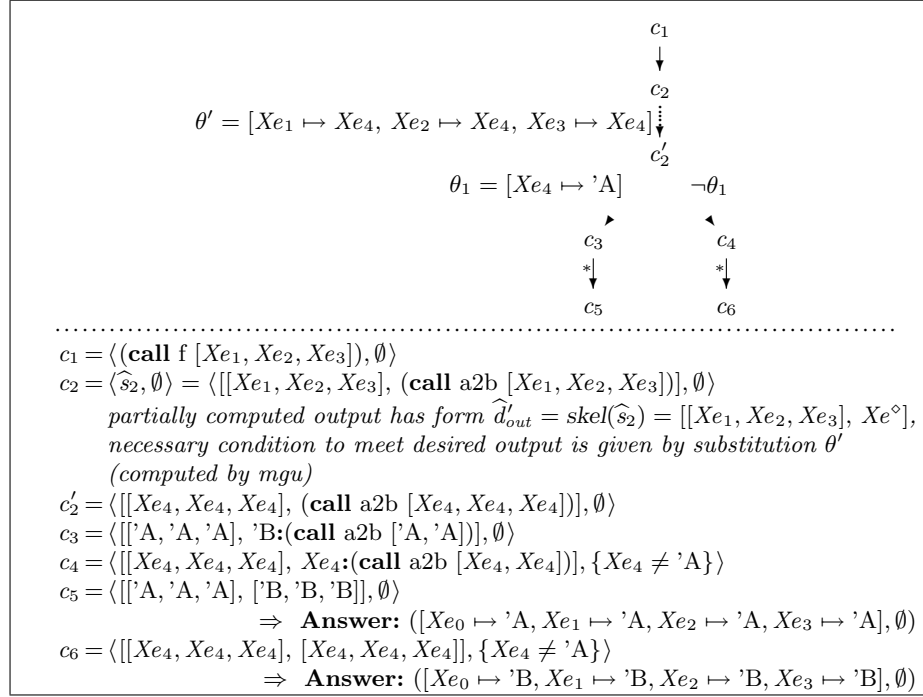


Fig. 8. Backpropagation achieved

Repeating driving steps leads to an infinite tree. Three terminal nodes are found two of which represent an answer for our inversion problem ( $c_4$ ,  $c_8$ ). To produce  $['B]$  as output, the input must either be  $['A]$  or  $['B]$ .

Even though these are the only two answers to the given inversion problem, the search continues and an infinite tree is constructed. Configurations  $c_5$ ,  $c_6$ ,  $c_9$ ,  $c_{10}$  search input lists that produce output lists of length greater than one. Clearly, they will never lead to an answer. Comparing the desired output  $\widehat{d}_{out}$  with the *partially computed answers* in each of those four configurations makes it clear that they will never produce a list of length one (such as  $['B]$ ). Thus, instead of waiting until a terminal node is reached, we can check the partially computed answers and stop tracing in all four configurations. Similar methods are used in narrowing in functional-logic languages (*e.g.*, [8, 4]).

(RG051022: 'similar methods in FL': this is too vague; we must be concrete, say what is same/similar and add reference.)

**Backpropagation** The second example is inverse computation of program  $f$ . Suppose we have the partially known output  $[[Xe_0, Xe_0, Xe_0], ['B, 'B, 'B]]$  and want to find all symbol lists of length three that can produce such an output. The three identical  $c$ -variables  $Xe_0$  stand for three identical values. For inverse computation, we specify the input as an arbitrary list of length three and give

the partially known output by the input-output class

$$cls_{io} = \langle (\underbrace{[[Xe_1, Xe_2, Xe_3]]}_{\widehat{ds}_{in}}, \underbrace{[[Xe_0, Xe_0, Xe_0], ['B, 'B, 'B]]}_{\widehat{d}_{out}}), \underbrace{\emptyset}_{\widehat{r}_{io}} \rangle .$$

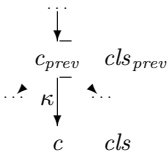
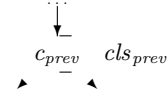
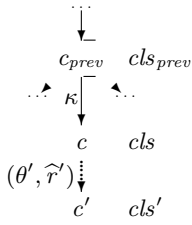
The process tree is shown in Fig. 7. The process tree is finite because the length of the input list is given by  $\widehat{ds}_{in} = [Xe_1, Xe_2, Xe_3]$ . The search stops when all possibilities are exhausted. The two answers to our inversion problem can be found by examining the terminal configurations  $c_9$  and  $c_{16}$ : the lists  $['A, 'A, 'A]$  and  $['B, 'B, 'B]$ . None of the configurations  $c_{10}$  to  $c_{15}$  leads to a valid answer because all of their first components violate the requirement that we desire as first output component: a list containing three identical elements. This constraint was specified by  $\widehat{d}_{out} = [[Xe_0, Xe_0, Xe_0], [...]]$ . It would be desirable to avoid constructing the branches leading to such configurations as early as possible. The lazy semantics of the source language allows us to do just this. The method described below gives a dramatic speed-up as demonstrated by our later examples.

(RG051024: Check thate statement is compatible with decompilation example, and whether we can write “It reduces the time to built the process tree from (exponential in...) to (linear in...)”. Here, or later, a comparision with needed narrowing is in place.)(SA+YK20060122: “decompilation” is commented, comparision with narrowing is good enough to say “dramatic speed-up” and “from (exponential in...) to (linear in...)”.)

**Approach** The method described in Fig. 9 achieves the desired effects:

1. The desired *cutting of subtrees* (exemplified in Fig. 6) is achieved by detecting when the partially computed output and the given output will *not* unify. In this case the branch and all its entire subtree can be eliminated from the search space.
2. The desired *backpropagation* of bindings (exemplified in Fig. 7) is achieved by applying the obtained substitution to the node. In this case the search space may be reduced by the additional information. Since this operation does not enlarge the set of states represented by a configuration, it never enlarges the process tree. The process tree reduced by backpropagation is shown in Fig. 8

This approach can be used for any language in which partially computed output can be identified already at inner nodes of the process tree. As soon as some constructors become available, they can be checked against the desired output. This is usually the case for functional languages with general recursion, such as Lisp or Haskell, when traversing a data structure by recursive decent. More precisely, during the construction of a process tree, we approximate the io-class of the current configuration  $c = \langle \widehat{s}, \widehat{r} \rangle$  using the known constructor skeleton:  $\widehat{d}'_{out} = skel(\widehat{s})$ . For instance, in Fig. 8 for  $c_2 = \langle \widehat{s}_2, \emptyset \rangle$  we have  $\widehat{d}'_{out} = skel(\widehat{s}_2) =$

Given a request with io-class $cls_{io} = \langle (ds_{in}, \widehat{d}_{out}), \widehat{r}_{io} \rangle$ and program $p$ :	
<p>Current process tree:</p> 	<p>Current node:  <math>c = \langle \widehat{s}, \widehat{r} \rangle</math> configuration, and  <math>cls = \langle \widehat{ds}, \widehat{r} \rangle</math> corresponding input class.</p> <p>Approximate the output of <math>c</math> by io-class  <math>cls'_{io} = \langle (\widehat{ds}, \widehat{d}'_{out}), \widehat{r} \rangle</math>          where the partially computed output <math>\widehat{d}'_{out}</math> is obtained          from c-state <math>\widehat{s}</math>: <math>\widehat{d}'_{out} = skel(\widehat{s})</math>.</p>
<p>(1) Cutting:</p> 	<p><b>if</b> <math>cls'_{io} \star cls_{io} = \emptyset</math> (intersection of io-classes)  <b>then</b></p> <ol style="list-style-type: none"> <li>1. Cut node <math>c</math> and its subtree.</li> <li>2. Continue driving.</li> </ol>
<p>(2) Backpropagation:</p> 	<p><b>else let</b> <math>cls'_{io} \star cls_{io} = \{(\theta, \widehat{r})\}</math></p> <ol style="list-style-type: none"> <li>1. Define <math>(\theta', \widehat{r}')</math> by removing from <math>(\theta, \widehat{r})</math> all bindings and restrictions on c-variables which do not occur in input class <math>cls</math>.</li> <li>2. Perform contractions  <math>c' = c / \theta' / \widehat{r}'</math>,  <math>cls' = cls / \theta' / \widehat{r}'</math>.</li> <li>3. Add a new branch labeled <math>(\theta', \widehat{r}')</math> and a new node with configuration <math>c'</math> and corresponding <math>cls'</math>.</li> <li>4. Continue driving.</li> </ol>

**Fig. 9.** Reduction of search space by cutting and backpropagation

$skel([Xe_1, Xe_2, Xe_3], (a2b \dots)) = [[Xe_1, Xe_2, Xe_3], Xe^\diamond]$  where  $Xe^\diamond$  is a fresh c-variable.

The central operation is the intersection ( $\star$ ) of the approximated io-class  $cls'_{io}$  with the desired io-class  $cls_{io}$ . If the intersection is empty, then the current node can never lead to a valid answer; otherwise, the intersection returns a contraction  $(\theta, \widehat{r})$  containing a substitution  $\theta$  and restriction  $\widehat{r}$  which may further constrain the current configuration. The *intersection operation* ( $\star$ ) is based on the *most general unifier* (*mgu*) and checks that the substitution  $\theta = mgu(\dots)$  does not lead to a contradiction when applied to the added restrictions  $(\widehat{r}_1 + \widehat{r}_2)$  [3]. Thus, there are three cases in the definition of the intersection operation: (i) the *mgu* fails, (ii) the *mgu* succeeds, but the substitution  $\theta = mgu(\dots)$  leads to a contradiction in the restrictions, and (iii) the *mgu* succeeds and the substitution is compatible with the restrictions (which means that the intersection is not empty).

**Definition 1 (intersection of io-classes).** Let  $cls_1, cls_2$  be two io-classes,  $cls_1 = \langle \widehat{dd}_1, \widehat{r}_1 \rangle$  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  $\widehat{dd}_1$  and  $\widehat{dd}_2$ , if it exists, then define io-class intersection  $(\star)$  by

$$cls_1 \star cls_2 \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \text{mgu}(\widehat{dd}_1, \widehat{dd}_2) \text{ fails} \\ \emptyset & \text{if } (\widehat{r}_1 + \widehat{r}_2)/\theta = \{\text{contra}\} \text{ where } \theta = \text{mgu}(\widehat{dd}_1, \widehat{dd}_2) \\ \{(\theta, \widehat{r})\} & \text{otherwise, where } \theta = \text{mgu}(\widehat{dd}_1, \widehat{dd}_2), \widehat{r} = (\widehat{r}_1 + \widehat{r}_2)/\theta. \end{cases}$$

(RG051022: mention good effect of lazy semantics here or elsewhere. The good effect: check partially computed results during inverse computation as early as possible while conforming to source language semantics. “We choose a lazy semantics for our source language because, among others, it allows us to perform only needed computations in order to obtain an output. In particular, this will be useful since we want to check partially computed results as early as possible.”)

(RG051022: one might mention that the request  $[Xe_1, [[Xe_0, Xe_0, Xe_0], ['B, 'B, 'B]]]$  which originally led to an infinite tree since the length of the input list is unspecified (much as the cutting example), now terminates and has essentially the same slim tree as in backpropagation.)

(RG051024: show and example how a substitution can lead to a contradiction in the restrictions.) (SA+YK20060122: Done:  $cls_{io} = \langle ([Xa], Xa), (Xa \# 'A) \rangle$ ,  $cls'_{io} = \langle ([A], 'A), \emptyset \rangle$ .)

## 5 Equivalence Transformation of Requests

The solution presented in Sect. 4 is general in that it can be applied to many languages using a similar set representation. In this section we show another, surprisingly simple, solution that can be used in NTSG. Instead of extending URA according to the method shown in Fig. 9, we perform an equivalence transformation of the given request for inverse computation.

There is an implementation of URA for NTSG according to [3] for the given source language. Instead of modifying the algorithm to implement the concepts described in Sect. 4, we perform an equivalence transformation of the given request. The transformation is shown in Fig. 10. Given a program  $p$  and input-output class  $cls_{in}$ , we transform them into a new program  $p'$  and a new input-output class  $cls'_{in}$ . The new program  $p'$  is constructed by adding two new functions to the original program.<sup>5</sup> The new main function is defined as a  $p$ 's main function call nested to call of function test. Function test compares the result computed by  $p$  with the desired output out, and returns the corresponding Boolean value.

The new input-output class  $cls'_{in}$  is a reformatted version of  $cls_{in}$  where the desired output is fixed to 'True and the user desired output is now the last argument for the new main function of  $p'$ . The restriction  $\widehat{r}$  remains unchanged.

<sup>5</sup> We assume that the names of the new functions (“main” and “test”) do not occur in  $p$ ; otherwise they are renamed.

For all NTSG-programs  $p$  and for all input-output classes  $cls_{in}$ :

$$Inv(NTSG, p, cls_{io}) = Inv(NTSG, p', cls'_{io}) \quad (4)$$

where

$$cls_{io} = \langle ([\hat{d}_1, \dots, \hat{d}_n], \hat{d}_{out}), \hat{r} \rangle$$

$$cls'_{io} = \langle ([\hat{d}_1, \dots, \hat{d}_n], \hat{d}_{out}], 'True), \hat{r} \rangle$$

$$p' = [(\text{define main } [in_1, \dots, in_n, out]$$

$$(\text{call test } [(\text{call mainfct}(p) [in_1, \dots, in_n]), out])),$$

$$(\text{define test } [res, out]$$

$$(\text{if (equ? res out) 'True 'False}))] \# p$$

**Fig. 10.** Answer equality of a transformed request

**Theorem 1 (answer equivalence of transformed request).** *Given language NTSG, for all programs  $p$  and for all input-output classes  $cls_{in}$ , the equation (4) holds.*

If a source program terminates on a given input, and produces the desired output, URA will find that input given the desired output [3]. Thus, the solution of  $Inv$  is unchanged for the transformation of the program and the reformatting of the io-class.

Due to this transformation we can achieve the effects described above. Instead of adding the steps shown in Fig. 10 to URA itself, we encode the problem in the source language taking advantage of the equality test. This is possible because the semantics of the equality test coincides with the desired *mg*-based method in Fig. 9.

The transformation of the original request to the new request makes the determination of test **equ?** the root of the perfect process tree. This has two effects. First, the test demands the calculation of the components of  $p$ 's output  $res$  until a mismatch with the desired output  $out$ , which establishes 'False'. This will stop any further development in the corresponding branch of the perfect process that has been developed until this point. This is the cutting operation. Second, any new contractions on c-variables obtained by **equ?** are applied to the current configuration. This achieves backpropagation. The effectiveness of this approach will be demonstrated in the next section. Another advantage is that the driving of  $p$  is guided by the equality test. This avoids the unnecessary computations which do not contribute to the goal of establishing an answer to the inversion problem.

## 6 Demonstration

This section demonstrates the improved termination and efficiency of inverse computation using the equivalence transformation from Fig. 10. The examples

<pre> (define bftTr [t]   (call bftTr1 [(t:[])]) (define app [x, y]   (if (cons? x h t _)     (h:(call app [t, y]))     y)) </pre>	<pre> (define bftTr1 [tfs]   (if (cons? tfs t fs _)     (if (cons? t t1 cr _)       (if (cons? cr c t2 _)         (c:(call bftTr1           (call app [fs, (t1:(t2:[])])           'Error:bad_tree_structure)           (t:(call bftTr1 [fs]))           []))       (call bftTr1 [fs])     )   ) </pre>
--	---

**Fig. 11.** Program ‘bftTr’: breadth-first traversal of binary trees

include inverse computation of tree traversal functions and experiments for comparison the URA implemented for NTSG vs. the CURRY functional-logic programming system.<sup>6</sup>

### 6.1 Breadth-First Labeling

A breadth-first labeling of a tree with respect to a given list of values is a labeling of nodes in the tree with values in the list in breadth-first order. The example is taken from [16]. We define the programs as the inverse of the function we want. We implemented a program in NTSG which, given a binary tree, collects the values of the nodes by a breadth-first traversal. Inverse computation of the program then performs the desired breadth-first labeling.

We performed two experiments. We used URA before and after the equivalence transformation of the request. Given a list with 13 values, the time to find the 132 trees labeled in breadth-first order is 216.05 secs; the search does not terminate. After the equivalence transformation of the request (Sect. 5), the time to find the 132 trees is 6.90 secs (the solution is found 31.31 times faster than before); the search terminates after 15.43 secs telling us that all trees have been found.

$$\llbracket ura \rrbracket [\text{bftTr}, \langle ([Xe_1], [1, 2, \dots, 13]), \emptyset \rangle] = [\dots \text{omitted } 132 \text{ solutions } \dots]$$

### 6.2 Comparison with the Curry functional-logic programming system

According their main ideas [8, ?] functional-logic programming languages use techniques that are very similar to the approach described in this paper. Thus we can assume that morden functional-logic programming system supports some

<sup>6</sup> All running times for CPU AMD Athlon 64 3500+ (2.2GHz), RAM 2GB, OS Debian Linux, The Glorious Glasgow Haskell Compilation System, version 6.4 (with -H1536m run-time option, *e.g.* 1.5 GB heap size).

```

-- [[ura]] [bpt, clsio] — was run for all  $n \in [1..100]$ 
-- where  $cls_{io} = \langle \langle [(\text{unary } n), Xe, Xa], \text{'True'} \rangle, \emptyset \rangle$ 
--       unary 0 = 'Nil
--       unary n = ('1 : (unary (n - 1)))

(define main [n, x, y]           -- main function
  (if (equ? ((call a2b [x]) : x)
        ((call replicate [n, 'B]) : (call replicate [n, y])))
      'True
      'False))

(define a2b [x]
  ... not shown, see Fig. 5
)

(define replicate [n, x]
  (if (cons? n - n' -)
      (x : (call replicate [n', x]))
      'Nil))

```

**Fig. 12.** Program “bpt”: backpropagation test written in NTSG

**Fig. 13.** Run time for *ura* computation

techniques of search space reducing that is similar to the techniques described above, in particular backpropagation and cutting subtrees.

In this section we describe an experiment that shows that Curry functional-logic programming system<sup>7</sup> has not perfectly support backpropagation and cutting subtrees.

**Backpropagation experiments** The NTSG-program “bpt” (Fig. 12) demonstrates acceleration effect of backpropagation: it easy to show that without backpropagation techniques the inverse computation:

$$[[ura]] [bpt, \langle \langle [(\text{unary } n), Xe, Xa], \text{'True'} \rangle, \emptyset \rangle]$$

need exponential in  $n$  time to find all solution; using backpropagation this time becomes polynomial in  $n$  (more precise  $O(n^2)$ ).

We performed this inverse computations for all  $n \in [1..100]$ , the run time for  $n = 1$  was 0.042 sec, for  $n = 22$  – 0.134 sec, for  $n = 100$  – 4.068 sec. The

<sup>7</sup> Was used compiler from Curry into Prolog from Portland Aachen Kiel Curry System (PACKS) 1.6.1-5 (<http://www.informatik.uni-kiel.de/pakcs/>) and SICStus Prolog 3.12.3 (<http://www.sics.se/isl/sicstuswww/site/index.html>).



```

import System
data Data = A | B | C

-- main function, it was run for all n ∈ [1..22]
main = findall (f n) -- n was edited by hand

f n (x, y) = (a2b x, x) :=
              (replicate n B, replicate n y)

a2b []      = []
a2b (x:xs) | equ x A   = B:(a2b xs)
           | otherwise = x:(a2b xs)

-- for all x,y<-Data we define ‘‘equ x y’’:
equ A A = True
equ A B = False
           ... 6 cases not shown
equ C C = True

```

**Fig. 14.** Program “bpt-c”: backpropagation test written in Curry



**Fig. 15.** Run time for computation of Curry-program

correspondent chart is shown in Fig. 13, it is easy to see that the time is  $O(n^2)$  — thus backpropagation supported perfectly in this case (*ura*).

The NTSG-program “bpt” was also written in the Curry (Fig. 14) and the series of test for  $n \in [1..22]$  was performed; the run time for  $n = 1$  was 0.034 sec, for  $n = 22$  — 285.1 sec. The experiments for  $n > 22$  was not performed due to the exponential in  $n$  run time. It is absolutely clear illustrated by the correspondent chart (Fig. 15). Please, note that logarithmic scale is used for run time in this chart.

Deviding for all  $n \in [1..22]$  the run time of Curry-experiment by the run time of *ura*-experiment we obtain acceliration *ura* vs. Curry. The correspondent chart is shown in Fig. 16, please, note that logarithmic scale is used for run time in this chart. Easy to see that *ura* loses to Curry (acceleration is less than one) for  $n \in [1..8]$ , but for  $n > 8$  acceleration is greater than one and it is exponential in  $n$  — *ura* wins from Curry.

This is clear argument to say: *the backpropagation (or any similar teqnics) is not perfectly supported in Curry.*



**Fig. 16.** Acceleration: fraction run time for Curry-program to run time for *ura*

<pre>-- [[ura]] [cst-test, &lt;([ ], 'True), &lt;math&gt;\emptyset&lt;/math&gt;&gt;]  (define main [ ] -- main function   (if (equ? ((call undef [ ]) : 'A)         ((call undef [ ]) : 'B))       'True       'False))  (define undef [ ] (call undef [ ]))</pre>	<pre>-- the goal is main  data Data = A   B  main = (undef, A) := (undef, B)  undef = undef</pre>
--	---

**Fig. 17.** NTSG- and Curry-programs “cst-test”: cutting subtrees test

**Cutting subtrees experiments** Two simplest NTSG- and Curry-programs “cst-test” shown in Fig. 17 are written to check cutting subtrees techniques in *ura* and Curry. The result of the experiments is following:

- The computation  $\llbracket ura \rrbracket [\text{cst-test}, \langle ([ ], 'True), \emptyset \rangle]$  stops in short time. This illustrates that cutting subtrees works properly in *ura*.
- The computation of the Curry-program “cst-test” does not stop. This is clear argument to say: *the cutting subtrees (or any similar techniques) is not perfectly supported in Curry.*

## 7 Related Work

The Universal Resolving Algorithm presented in this paper is derived from *perfect driving* [6] and is combined with a mechanical extraction of the answers (cf. [1, 17]) giving our algorithm the power comparable to SLD-resolution, but for a first-order functional language with tail-recursion (cf. [7]). The complete algorithm is given in [3]. A program analysis for inverting programs which makes use of positive driving was proposed in [21]. The use of driving for theorem proving is discussed in [23] and its relation to partial evaluation in [11].

Logic programming inherently supports inverse computation [13]. The use of an appropriate inference procedure permits the determination of any computable answer [14]. 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 [8]; for a survey see [4].

## 8 Conclusion

Further work needs to be done in several directions. First, we need to prove the correctness properties of the Universal Resolving Algorithm with respect to the semantics of our source language. Second, we want to investigate the exact relationship between the mechanisms used in functional-logic languages and the methods used in this paper. Third, we plan to establish more empirical results of the algorithm used in this paper. The algorithm is fully implemented in Haskell which serves our experimental purposes quite well. More efficient implementations certainly exist. Fourth, recent works [10] on term rewrite systems define the notion of fully-collapsed jungles on graphs. We want to investigate the use of these techniques in the context of process tree construction as it was suggested in [20].

*Acknowledgements* The second author would like to thank Yoshihiko Futamura for his kind and generous support of his visit to Waseda University, Japan. Special thanks to the Japan Science and Technology Corporation for providing the financial support leading to this research. Thanks to Takuya Hashimoto for programming the main part of the Flowchart translator and to Anton Orlov for participation in basic algorithms programming for NTSG (the set representation, the PPT, *etc.*) (SA+YK20060122: Flowchart translator and Byte-code translator is not included in this paper.)

## References

1. S. M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).
2. S. M. Abramov, R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse, J. N. Oliveira (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 1837, 187–212. Springer-Verlag, 2000.
3. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
4. E. Albert, G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.
5. R. Bird, O. d. Moor. *Algebra of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
6. R. Glück, A. V. Klimov. Occam’s razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, A. Rauzy (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.
7. 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.
8. M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
9. J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.
10. B. Hoffmann, D. Plump. Implementing term rewriting by jungle evaluation. *Informatique Thorique et Applications/Theoretical Informatics and Applications*, 25(5):445–472, 1991.
11. N. D. Jones. The essence of program transformation by partial evaluation and driving. In N. D. Jones, M. Hagiya, M. Sato (eds.), *Logic, Language and Computation*, LNCS 792, 206–224. Springer-Verlag, 1994.
12. S. Katsumata, A. Ohori. Proof-directed de-compilation of Java bytecode. In D. Sands (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2028, 352–366. Springer-Verlag, 2001.
13. R. Kowalski. Predicate logic as programming language. In J. L. Rosenfeld (ed.), *Information Processing 74*, 569–574. North-Holland, 1974.
14. J. W. Lloyd. *Foundations of Logic Programming. Second, extended edition*. Springer-Verlag, 1987.
15. J. McCarthy. Recursive functions of symbolic expressions. *Communications of the ACM*, 3(4):184–195, 1960.
16. S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 2386, 209–232. Springer-Verlag, 2002.
17. A. Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 427–444. North-Holland, 1988.
18. J. P. Secher. Perfect Supercompilation. M. Sc. thesis, Department of Computer Science, University of Copenhagen, 1998.
19. J. P. Secher, M. H. Sørensen. On perfect supercompilation. In *Proceedings Perspectives of System Informatics, Novosibirsk, Russia, July 1999*, LNCS 1755, 113–127. Springer-Verlag, 2000.

20. J. P. Secher. Driving in the jungle. In O. Danvy, A. Filinsky (eds.), *Programs as Data Objects. Proceedings*, LNCS 2053, 198–217. Springer-Verlag, 2001.
21. J. P. Secher, M. H. Sørensen. From checking to inference via driving and DAG grammars. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 41–51. ACM Press, 2002.
22. M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
23. V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker, J. van Leeuwen (eds.), *Automata, Languages and Programming*, LNCS 85, 645–657. Springer-Verlag, 1980.
24. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.