Transforming Interpreters into Inverse Interpreters by Partial Evaluation

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

Youhei Kawada
Waseda University, School of
Science and Engineering
Tokyo 169-8555, Japan
kawada@futamura.info.
waseda.ac.jp

Takuya Hashimoto
Waseda University, School of
Science and Engineering
Tokyo 169-8555, Japan
hasimoto@futamura.info.
waseda.ac.jp

ABSTRACT

The experiments in this paper apply the idea of prototyping programming language tools from robust semantics: we used a partial evaluator (Similix) to turn interpreters into inverse interpreters. This way we generated inverse interpreters for several small languages including interpreters for Turing machines, an applied lambda calculus, a flowchart language, and a subset of Java bytecode. Limiting factors of offline partial evaluation were the polyvariant specialization scheme with its lack of generalization; advantages were the availability of higher-order values to specialize a breadth-first tree traversal. This application of self-applicable partial evaluation is different from the classical Futamura projections that tell us how to translate a program by specialization of an interpreter.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*partial evaluation*; D.3.4 [**Programming Languages**]: Processors—*interpreters, preprocessors*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms

Languages, Experimentation

Keywords

binding-time improvements, inverse interpreter, program inversion, self-application, semantics modifier

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

PEPM'03, June 7, 2003, San Diego, California, USA.

1. INTRODUCTION

Robust non-standard semantics can be applied to new languages via standard interpreters [1]. This paper studies this property for the case of inversion semantics and examines experimentally how partial evaluation may serve as an enabling optimization technique. The ultimate goal is to find ways to achieve a higher degree of language independence and to prototype programming language tools from generic implementations of non-standard semantics. We do not claim that this is the best approach in all cases – others exist – but we believe it worthwhile exploring this possibility.

The partial evaluator Similix [8], is one of the reference systems for offline partial evaluation [17]. We use this partial evaluator to transform interpreters into inverse interpreters for several small languages including the call-by-value lambda calculus and a subset of Java bytecode. While our results show that not all interpretive overhead could be removed, they indicate the potential of partial evaluation for porting non-standard semantics. Our experiments also show some of the limiting factors of offline partial evaluation, namely the polyvariant specialization regime lacks generalization and the use of regular partially static structures.

We use inverse computation as an example of robust non-standard semantics. The Universal Resolving Algorithm (URA) [4, 3] used in this paper is an algorithm for inverse computation in a first-order functional language. For the experiments, we implemented the algorithm in Scheme. We studied the topic of porting inverse computation by program specialization for the sake of its own consistency, and occupied ourselves only with the generation of inverse interpreters. Another approach is to write translators instead of interpreters, or simply to build inverse interpreters by hand. For program inversion, the reader is referred to the literature, *e.g.* [12], or recent work [22]. The idea of inverse interpretation can be traced back to [21].

The paper is organized as follows. We review fundamental properties of inverse computation (Sect. 2), outline the Universal Resolving Algorithm (Sect. 3), and describe the binding-time improvements that were necessary for successful specialization (Sect. 4). We report our findings (Sect. 5) and make a conclusion (Sect. 6, Sect. 7). We assume that the reader is familiar with the principles of partial evaluation.

2. FUNDAMENTAL CONCEPTS

We review the properties of inverse computation which we will use later in this paper. We define an inverse interpreter, explain an important property of inverse computation and how the construction of an inverse interpreter may be optimized by a program specializer.

In this section we limit our discussion of inverse computation to source programs that are injective (one-to-one). At the expense of extra definitions the results can be generalized to arbitrary source programs. The Universal Resolving Algorithm used for the experiments in this paper deals with the general case. A detailed exposition of the concepts presented here can be found in the publications [1, 2, 3]. The notation is adapted from [17].

Notation: For any program text p, written in language L, we let $[\![p]\!]_L\, d$ denote the application of L-program $p \in P_L$ to its input $d \in D$. Equality shall always mean strong equivalence: either both sides of an equation are defined and equal, or both sides are undefined. As is customary, we use the same universal data domain D for all languages and for representing all programs (*e.g.*, a suitable choice are lists known from Lisp). Mappings between different data domains are straightforward and not essential for our discussion.

Interpreters and Specializers.

What follows are standard definitions of interpreters and specializers.

Definition 1. (interpreter) An L-program $int \in Int_{N/L}$ is an *N/L-interpreter* iff $\forall p \in P_N, \forall x, y \in D$:

$$[\![int]\!]_L\,[\,p, x\,] = y \iff [\![p]\!]_N\, x = y\ .$$

Definition 2. (self-interpreter) An L-program $sint \in Sint_L$ is a *self-interpreter for L* iff *sint* is an L/L-interpreter.

Definition 3. (specializer) An L-program $spec \in Spec_L$ is a *specializer for L* iff $\forall m, n \geq 0$, $\forall p \in P_L$ with argument list of length $m+n$, and $\forall x_1 ... x_m, y_1 ... y_n, z \in D$:

$$[\![\,[\![spec]\!]_L\,[p, m, x_1 ... x_m]\,]\!]_L\,[y_1 ... y_n] = z$$
$$\iff [\![p]\!]_L\,[x_1 ... x_m, y_1 ... y_n] = z\ .$$

Note: The index m of a specializer *spec* tells how many static arguments are available for specializing a program p. This notation is equivalent to the traditional SD-notation. It is convenient when specializing programs with respect to a different number of parameters.

Inverse Interpreter.

Let p be an N-program that computes output y from input x. For simplicity, let us assume that p is injective. Computation of y by applying p to x is described by:

$$[\![p]\!]_N\, x = y\ . \tag{1}$$

The determination, for a given N-program p and output y, of an input x of p such that $[\![p]\!]_N\, x = y$ is *inverse computation*. A program that performs inverse computation is an *inverse interpreter* and can be described by

$$[\![invint]\!]_L\,[p, y] = x\ . \tag{2}$$

Definition 4. (inverse interpreter for injective programs) An L-program $invint \in Invint_{N/L}$ is an *inverse N/L-interpreter for injective programs* iff $\forall$injective $p \in P_N, \forall x, y \in D$:

$$[\![invint]\!]_L\,[\,p, y\,] = x \iff [\![p]\!]_N\, x = y\ .$$

Note: In general, when p is not injective, inverse computation can either compute all possible inputs or select just one. The Universal Resolving Algorithm used later in this paper is an inverse interpreter for non-injective, partial functions and computes the universal solution of an inversion problem.

Example: Let *encode* be a program to encode a piece of text. It is injective so that we can recover the text from its code.

$$[\![encode]\!]_N\, text = code\ .$$

Instead of writing a decoder to reproduce the original text,

$$[\![decode]\!]_N\, code = text\ ,$$

we can use an inverse interpreter *invint* to obtain the same text, even though this may be slower than using a decoder:

$$[\![invint]\!]_L\,[\,encode, code\,] = text\ .$$

Inverse Computation via an Interpreter.

We now explain an important property of inverse computation, namely that inverse computation can be performed in any language via an interpreter for that language [1]. This property is the theoretical basis for our experiments.

Suppose we have two functionally equivalent programs p and q written in languages P and Q, respectively. For the sake of simplicity, let p and q be injective. Let *invintP* and *invintQ* be two inverse interpreters for P and Q, respectively. Since p and q are functionally equivalent, inverse computation of p and q returns the same result:

$$(\forall x :\ [\![p]\!]_P\, x = [\![q]\!]_Q\, x) \implies$$
$$(\forall y :\ [\![invintP]\!]_L\,[p, y] = [\![invintQ]\!]_M\,[q, y])\ . \tag{3}$$

This is the key observation: regardless of how two functionally equivalent programs are implemented, the solution to the given inversion problem is always the same. In the terminology of [1], the 'inverse semantics' is *robust*.

Two programs which are functionally equivalent under the standard semantics are also functionally equivalent under a non-standard semantics provided that semantics modification is robust (such as the inverse semantics of L and Q). This means, it does not matter how a function is written, the result of inverse computation is the same for all possible implementations. Robustness and a theory for combining non-standard semantics was formally developed in [1]. In this paper we make use of these theoretical results.[1]

How can we use Property (3)? Suppose we have an inverse interpreter for P, but none for Q. Then we can perform inverse computation of a Q-program by writing an interpreter *intQ* for Q in P. Given a Q-program q we define a new program q' and apply to it the inverse interpreter *invintP*:

$$[\![invintP]\!]_L\,[q', y] = x$$
$$\text{where } q' = \lambda x.[\![intQ]\!]_P\,[q, x]\ . \tag{4}$$

The result x is a correct solution for the inversion problem $(q,\ y)$ because programs q and q' are functionally equivalent.

Let us examine this in more detail because this construction forms the basis for producing inverse interpreters by

[1] A simple example of a non-standard semantics that is not robust is one that, given a program and some data, returns the number of lines in that program. Two functionally equivalent programs may now return two different results.

specialization. Using Def. 1 and the definition of q' in (4), we have the functional equivalence between q and q':

$$\forall x : [\![q']\!]_P\, x = [\![intQ]\!]_P\, [q, x] = [\![q]\!]_Q\, x\ . \tag{5}$$

Recalling (3), we can conclude that the inversion problem $(q,\ y)$ is indeed solved by the construction in (4):

$$\forall y : [\![invintP]\!]_L\, [q', y] = [\![invintQ]\!]_M\, [q, y]\ . \tag{6}$$

This statement holds for all inverse interpreters *invintP* and *invintQ*, for all Q-interpreters *intQ*, and for all Q-programs q. That is, we need not write an inverse interpreter for Q if we have an inverse interpreter for P and a Q/P-interpreter.

Semantics Modifier. To explain how program specialization may be useful in this context, we will first define a semantics modifier for inverse computation. Let us abstract the interpreter *intQ* from the construction in (4) and define a new program *invmod* which we call a *semantics modifier for inverse computation* [2] (short: inversion modifier). The modifier takes a Q-interpreter *intQ* written in P, a Q-program q, and y as input, and computes the solution for the inversion problem (q, y) given in Q:

$$[\![invmod]\!]_L[intQ, q, y] = x\ . \tag{7}$$

Here is how the semantics modifier can be constructed:

$$\begin{aligned} invmod \ &\stackrel{\text{def}}{=}\ \lambda(intQ, q, y).[\![invintP]\!]_L\, [q', y] \\ &\quad \text{where } q' = \lambda x.[\![intQ]\!]_P\, [q, x]\ . \end{aligned} \tag{8}$$

Other constructions exist. Formally, a semantics modifier for inverse computation is defined as follows.

Definition 5. (inversion modifier for injective programs) An L-program *invmod* $\in$ $Invmod_{P/L}$ is a *P/L-inversion modifier for injective programs* iff $\forall intQ \in Int_{Q/P}$, $\forall$injective $q \in P_Q$, and $\forall x, y \in D$:

$$[\![invmod]\!]_L\, [\, intQ, q, y\,] = x \iff [\![q]\!]_Q\, x = y\ .$$

The program is called *semantics modifier* because, given the standard semantics of a language in form of an interpreter, it 'magically' ports inverse computation to that language. There exists a class of semantics modifiers for robust semantics [1], not only for inverse computation. It is interesting to observe that the language paradigm of Q is quite irrelevant for a semantics modifier. Thus, we should be able to port inverse computation to a variety of languages without having to develop a separate tool for each language.

Example: Suppose we have an encoder written in a language *New* and we want to decode an encoded text:

$$[\![encode]\!]_{New}\, text = code\ .$$

Instead of implementing a decoder in *New*, we can write an interpreter *intNew* for *New* in P, and use the semantics modifier *invmod* to obtain the original text:

$$[\![invmod]\!]_L[intNew, encode, code] = text\ .$$

Interpreter Transformation. Efficiency problems become more serious due to the insertion of an interpreter *intQ* between the inverse interpreter *invintP* and the program q. The key idea is to specialize the semantics modifier *invmod* with respect to a Q-interpreter and thereby obtain a more efficient implementation of inverse computation in Q. When we specialize the inversion modifier with respect to *intQ*, we obtain a new program *invintQ'*:

$$[\![spec]\!]_L\, [invmod, 1, intQ] \ = \ invintQ' \tag{9}$$

$$[\![invintQ']\!]_L\, [q, y] \ = \ x \tag{10}$$

Program *invintQ'* has the functionality of an inverse interpreter for Q. Eq. 9 can be viewed as transforming an interpreter for Q into an inverse interpreter for Q. This usage of program specialization is new. It is different from the Futamura projections which tell us how to *translate* a source program into a target program by specializing an interpreter. Instead of specializing an interpreter, we specialize an inversion modifier to *invert* a program (here interpreter *intQ*).

While the 1st Futamura projection can be viewed as the implementation of a *translation modifier* [2, Sect. 6.2] (and allows us to port a translation semantics via an interpreter to another language); an inversion modifier can be implemented by an inverse interpreter as in Eq. 8. The specialization of a translation modifier with respect to an interpreter corresponds to the 2nd Futamura projection (and produces a translator); the projection in Eq. 9 is the specialization of an inversion modifier with respect to an interpreter (and produces an inverse interpreter, not a translator). Both are examples of the same fundamental principle [1]: applying a robust non-standard semantics to a program via an identity semantics (implemented by a standard interpreter). Fig. 1 illustrates the conversion of several interpreters into inverse interpreters by specializing an inversion modifier.

The idea of specialization can be carried further. When we specialize *spec* in Eq. 9 with respect to *invmod* and 1, we obtain a *program inverter* (instead of a translator as in the Futamura projections). The program inverter obtained by this self-application turns an interpreter *intQ* into an inverse interpreter *invintQ'* (the same as in Eq. 9):

$$[\![spec]\!]_L\, [spec, 2, invmod, 1] \ = \ inverter \tag{11}$$

$$[\![inverter]\!]_L\, intQ \ = \ invintQ' \tag{12}$$

In our experiments with the self-applicable partial evaluator Similix the transformation in Eq. 11 was performed by Similix's generator generator, and for efficiency reasons we prefer to use the program inverter (instead of using Eq. 9).

The correctness of the equations above can be verified using Def. 1, 3 and 5. The transformations say nothing about the efficiency of the generated programs. This depends on the strength of the specializer and the algorithmic structure of the inversion modifier.

Example: Suppose we have an interpreter *intNew* for *New* written in P. Then we can use the program inverter from Eq. 12 to turn it into an inverse interpreter for *New*:

$$[\![inverter]\!]_L\, intNew = invintNew\ .$$

Instead of using the semantics modifier *invmod*, we can now use the new inverse interpreter:

$$[\![invintNew]\!]_L[encode, code] = text\ .$$

We will see what can be achieved by existing offline partial evaluation specializing an algorithm for inverse computation in a first-order functional language. Other schemes for optimizing semantics modifiers exist [2, 3]. For example, we may also produce inverse programs by these techniques. This paper focuses on the transformations described above.

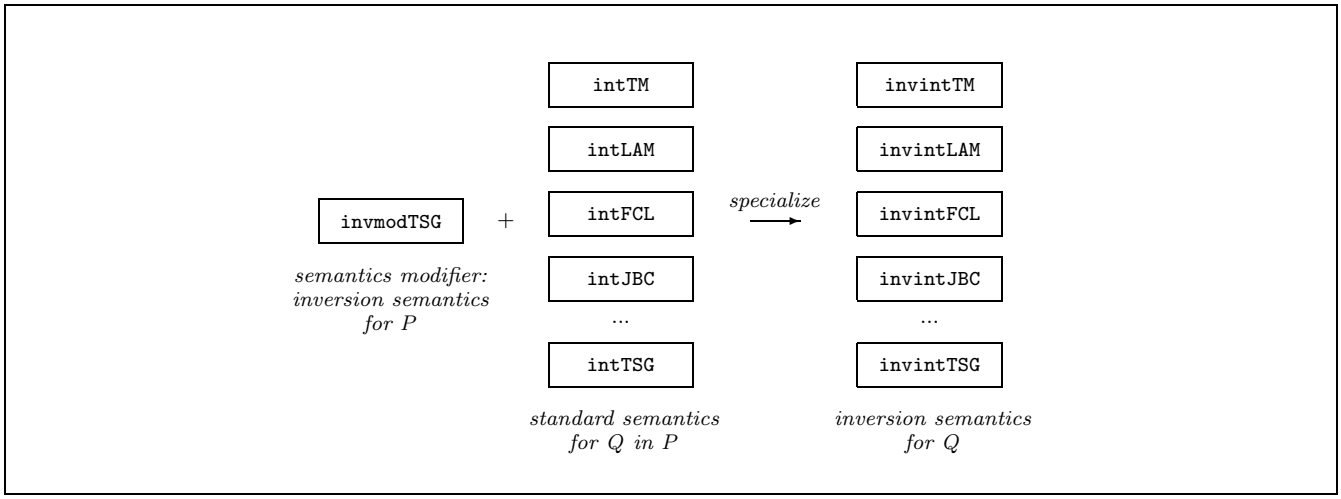

Figure 1: Semantics modifier + standard semantics = non-standard semantics.

3. AN INVERSE INTERPRETER

In general, when a program p is not injective, inverse computation using an inverse interpreter *invint* takes the form

$$[\![invint]\!]_L\ [p, cls_{io}] \ =\ ans \tag{13}$$

where p is a P-program and cls_{io} is an input-output class. We specify the input and output domains using an *input-output class* cls_{io} [4]. 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 given by

$$\begin{aligned} &Inv(P, p, cls_{io}) \\ &\quad = \{\ (ds_{in}, d_{out})\ \mid\ (ds_{in}, d_{out}) \in \lceil cls_{io} \rceil,\ [\![p]\!]_P\ ds_{in} = d_{out}\ \} \end{aligned} \tag{14}$$

where P is a programming language, p is a P-program, and cls_{io} is an input-output class. The solution $Inv(P, p, cls_{io})$ is the largest subset of $\lceil cls_{io} \rceil$ such that $[\![p]\!]_P\ ds_{in} = d_{out}$ for all elements (ds_{in}, d_{out}) of this subset.

Universal Resolving Algorithm. The *Universal Resolving Algorithm* (URA) [4, 3] is an algorithm for inverse computation in a first-order functional language (TSG, a typed version of S-Graph [13]). The answer produced by URA is a set of substitution-restriction pairs $ans = \{(\theta_1, r_1), \ldots\}$ which represents set *Inv* for the given inversion problem. The correctness of the answer produced by URA is given by

$$\bigcup_i \lceil (cls_{io}/\theta_i)/r_i \rceil \ =\ Inv(P, p, cls_{io}) \tag{15}$$

where $(cls_{io}/\theta_i)/r_i$ narrows the pairs of values represented by cls_{io} by applying substitution θ_i to cls_{io} and adding restriction r_i to the domains of the free variables.

The algorithm is based on the notion of a *perfect process tree* [13] 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. A process tree is perfect when it contains only branches that can be reached by at least one input value. Each fork in a perfect tree partitions the input class cls_{in} into disjoint and exhaustive subclasses. The algorithm then constructs, breadth-first, a perfect process tree for a given program p and input class cls_{in}. First, it constructs a forward trace of the computation given p and cls_{in}, and then extracts the solution to the backward problem using cls_{io}. The construction of a process tree is similar to unfolding in partial evaluation where a computation is traced under partially specified input.

Inverse computation can be organized into three steps: (1) walking through a perfect process tree (PPT), (2) tabulating the input and output, and (3) extracting the answer to the inversion problem from the table. The three steps can be carried out in a single phase. A Haskell implementation exists [4] that implements all three steps separately and exploits the underlying lazy semantics of Haskell to construct the PPT lazily. Our Scheme implementation performs the three steps interleaved due to its call-by-value semantics. The algorithm is sound and complete [4], and since the source language of the algorithm is a universal programming language, it allows us to perform inverse computation of any computable function.

Example (from [3]): Consider a program *inorder* that traverses a binary tree and returns a list of its nodes. The program is not injective. Given a list of nodes, inverse computation then produces all binary trees that lead to that list. The implementation in TSG performs an inorder traversal and, for simplicity, returns the list of nodes in reversed order. Binary trees are represented by cons-structures. For example, **1:2:3** represents a binary tree with left child 1, root 2 and right child 3. For example, five binary trees t_i can be built from seven nodes and we have for all $t_i, i = 1...5$:

$$[\![inorder]\!]_{TSG}\ \ t_i = [7, 6, 5, 4, 3, 2, 1]\ .$$

Given this node list, inverse computation of *inorder* by URA returns all five binary trees. The input tree is unknown. It is represented by a variable (Xe_1) where the restriction on the domain of the variable is empty $(\emptyset)$. The output list is given $([7, 6, 5, 4, 3, 2, 1])$. This information forms a class:

$$cls_{io} = \langle ([Xe_1], [7, 6, 5, 4, 3, 2, 1]), \emptyset \rangle\ .$$

Applying URA to *inorder* and cls_{io} finds all five binary trees. The result are five substitution-restriction pairs (θ_i, r_i) where

r_i is empty ($\emptyset$). URA does not terminate since it continues the search for more solutions (indicated by '...').

$[\![ura]\!]_{SCM}\,[inorder, cls_{io}] = [$
$([Xe_1 \mapsto ((1{:}2{:}3){:}4{:}5){:}6{:}7],\ \emptyset),$
$([Xe_1 \mapsto (1{:}2{:}3{:}4{:}5){:}6{:}7],\ \emptyset),$
$([Xe_1 \mapsto (1{:}2{:}3){:}4{:}5{:}6{:}7],\ \emptyset),$
$([Xe_1 \mapsto 1{:}2{:}(3{:}4{:}5){:}6{:}7],\ \emptyset),$
$([Xe_1 \mapsto 1{:}2{:}3{:}4{:}5{:}6{:}7],\ \emptyset),\ \ldots$

4. SPECIALIZATION

The section introduces the partial evaluator Similix and describes the binding-time improvements that were necessary to obtain good residual programs.

4.1 The Partial Evaluator Similix

Similix is a self-applicable offline partial evaluator for the programming language Scheme. The polyvariant specialization algorithm performs a continuation-based reduction; the monovariant binding-time analysis handles higher-order values and partially static structures. Similix is self-applicable and can perform all three Futamura projections. It is a reference system for offline partial evaluation and is well documented in the literature [6, 8, 9]. It has been used for a number of studies on partial evaluation (see [19]).[2]

1. An example of a call to the partial evaluator is

   ```
   (similix 'ura (list int '***) "uraTSG.sim")
   ```

   Here, `ura` is the name of the main function in the program `"uraTSG.sim"`, the Scheme implementation of URA for TSG, and `(list int '***)` is a specification of the input to `ura`. The stars `***` represent the dynamic part of the input; the variable `int` is the static part (the text of an interpreter written in TSG).

2. Similix is self-applicable and we can use its compiler generator `cogen` to perform the same specialization as in (1): first convert `ura` into a program inverter `comp`, then use it to produce the desired inverse interpreter from `int` (the name `comp` is a default name assigned by Similix):

   ```
   (cogen 'ura '(static dynamic) "uraTSG.sim")
   (comp (list int '***))
   ```

The residual program obtained by (1) or (2) can then be applied to the input-output class `clsio` of our inversion problem (the name `ura-0` is the default name of the residual program). The advantage of (2) is the transformation speed.

```
(ura-0 clsio)
```

When using an algorithm for inverse computation, we need a concrete representation of the input-output class cls_{io} and the solution set *ans*. URA uses lists known from Lisp and represents the search space cls_{io} by expressions with variables and restrictions [4, 3]. (Other algorithms for inverse computation may choose other representations.)

[2]Similix can be obtained from `http://www.diku.dk/forskning/topps/activities/similix.html`.

4.2 Binding-Time Improvements

The first attempt to specialize URA did not produce good residual programs. Most parts of the algorithm were annotated as dynamic by Similix's binding-time analysis (BTA). Several binding-time improvements were necessary to improve the flow of static information at specialization time. Binding-time improvements are semantics-preserving transformations that are applied to a source program before specialization. The Similix manual [7, Sect. 7] describes several binding-time improvements.

We summarize the main modifications. Some of the modifications are "classical" binding-time improvements, such as the introduction of partially static structures for environments, others were not obvious to us at first, such as the specialization of the breadth-first traversal of the perfect process tree. The modifications required about a dozen passes over the algorithm, and the analysis of cause and effect required insights into the properties of URA and the partial evaluation algorithms. It increased the number of procedures and the size of the program.[3]

URA	*cons cells*	*procedures*
original	3098	137
bti-improved	4474	167

Monovariant BTA. The binding-time analysis of Similix is monovariant. If the same procedure is used with different binding times for the arguments, the "most dynamic" annotated version will be used in all cases and possible reductions will be lost. This can be avoided by *making a copy* of a function for each of the different binding-time contexts (thereby achieving the effect of a polyvariant BTA). Examples are auxiliary functions used in different binding-time contexts:

```
(define (mkEnvBinds_s xs ds) ...)  ; xs:s
(define (mkEnvBinds_d xs ds) ...)  ; xs:d
```

Partially Static Structures. Structures consisting of static-dynamic pairs need to be represented as lists generated by user-defined constructors, otherwise the structure would become completely dynamic with bad residual programs. It is only through user-defined constructors that Similix offers partially static data structures. Bindings of static-dynamic name-value pairs are an example:

```
(defconstr (mynil1) (mycons1 mycar1 mycdr1))
(defconstr (mynil2) (mycons2 mycar2 mycdr2))
...
(define (mkPCBinds_s cxs ces)  ; <cxs:s ces:ps>
  (if (null? cxs) (mynil2)
      (mycons2 (mkPCBind (car cxs) (mycar1 ces))
               (mkPCBinds_s (cdr cxs) (mycdr1 ces)))))
```

Higher-Order Nodes. While partially static structures improve the flow of static information, they can lead to *infinite specialization* when the size of a partially static structure is not bound statically. The breadth-first traversal of the perfect process tree by URA is such a case: the width of the tree is not bound statically. Non-termination of Similix is

[3]*cons cells* gives the number of cons cells needed to represent a program; it is proportional to the size of its abstract syntax tree; *procedures* is the number of Scheme procedures.

Before:

```
  ... (bforder (node term env ...) '()) ...

(define (bforder tree* new-tree*)
  (if (null? tree*)           ; curr level empty?
      (if (null? new-tree*)   ; next level empty?
          'Done               ; done (finite PPT)
          (bforder (cdr new-tree*)
                   (apply node (car new-tree*))))
      (bforder (cdr tree*)
               (append (apply node (car tree*))
                       new-tree*))))

(define (node term env ...)
  (cond
    ((isCall? term) ...)
    ((isIf?   term) ...
      (list (list (if->then term) env1 ...))  ; child1
            (list (if->else term) env2 ...))) ; child2
    ((isExp?  term) ... '())                  ; no child
    ...))
```

After:

```
  ... (bforder (node term env ...) '()) ...

(define (bforder tree* new-tree*)
  (if (null? tree*)           ; curr level empty?
      (if (null? new-tree*)   ; next level empty?
          'Done               ; done (finite PPT)
          (bforder (cdr new-tree*)
                   ((car new-tree*))))
      (bforder (cdr tree*)
               (append ((car tree*))
                       new-tree*))))

(define (node term env ...)
  (cond
    ((isCall? term) ...)
    ((isIf?   term) ...
      (list (lambda () (node (if->then term) env1 ...))
            (lambda () (node (if->else term) env2 ...)))
    ((isExp?  term) ... '())
    ...))
```

Figure 2: Breadth-first tree traversal: binding-time improvement using higher-order values

due to the polyvariant specialization scheme that folds only to program points with identical static values; there is no generalization (for a discussion of this issue as a defining difference of online and offline partial evaluation see [11]).

Fig. 2 shows the main procedure managing the breadth-first traversal. Procedure `bforder` has two arguments: list `tree*` contains the nodes of the current level in the tree, and list `new-tree*` contains the children of those nodes. Each node at the current level may give rise to zero or more children. They are collected in `new-tree*`. When all nodes of the current level `tree*` are processed by procedure `node`, procedure `bforder` switches to the new level `new-tree*`. Procedure `bforder` maintains only the nodes at the frontier of the tree; it does not need to construct an actual tree. Enforcing a breadth-first traversal is essential in order to find all solutions for the given inversion problem.

Each node in `tree*` and `new-tree*` represents a computation state including a program term `term` (static) and an environment `env` (partially static). To preserve this vital static information at specialization time, both node lists can be made partially static, but this causes infinite specialization. When we keep both lists dynamic, Similix terminates, but with bad residual programs as a consequence.

To avoid the loss of static information *and* to make Similix terminate, both lists are made dynamic, but each element added to the list is enclosed in a `lambda` whose body applies procedure `node`. This does not change the order of the breadth-first traversal, but allows Similix to specialize the body of the lambda-expression thereby producing the desired specialized versions of procedure `node` (Similix specializes higher-order values at the point of definition). Since the number of program terms and the number of variables in the environment is finite, Similix terminates and produces the desired specialization of the breadth-first traversal.

Note: This binding-time improvement is applicable to all programs traversing trees or maintaining work lists, including the work lists of a polyvariant specializer under self-application. It requires a higher-order program specializer.

Dead Code. Similix gives certain guarantees concerning its residual programs: computations are never discarded (partial evaluation thus preserves termination properties). This can lead to the undesired effect that computations are preserved in a residual program whose results are not needed and do not harm termination either. For example, when substitutions on an environment are performed by URA and some of the values are not accessed later. A substitution operation for each value appears in the residual program. To avoid this, each conditional in a TSG-program is annotated with the set of variables that are used in a branch. Unused variables are then stripped from the environment before a substitution is performed. This binding-time improvement is similar to the removal of bindings from environments in [18] to reduce the size of target programs.

```
(updEnv (/env (strip-env env freevars) k) b))
```

Annotation of TSG-program. When specializing URA with respect to an interpreter, variables of that interpreter are kept in the URA environment. Some of those variables are bound to ground values (which means they are actually static). Similix can only keep track of partially static structures that have a regular shape, while some values in the environment may actually be known at specialization time. In case of conditionals in the TSG-program, knowing that a value which is tested by a conditional is ground allows URA to proceed into one of the branches without forking (transient configuration). When this information is lost, dead code is generated. To avoid this, conditionals in the TSG-program were annotated as ground ('IFS') and non-ground ('IFD'). This halved the code size of the generated residual programs, even though it may not be that elegant.

5. EXPERIMENTS

This section reports the results of specializing the binding-time improved version of URA with respect to five different interpreters. This transforms the interpreters into inverse

interpreters. We implemented interpreters for several small languages, including imperative and functional languages. We implemented the interpreters in TSG, the source language of URA. Fig. 1 illustrates the conversion of the interpreters into inverse interpreters.[4]

5.1 Interpreters

Turing Machine. Program `intTM` is an interpreter for Turing programs; it is similar to the one in [17]. The tape is a list of symbols where `B` stands for 'blank'. A Turing program is a list of labeled instructions where each instruction is of the form (where a is a symbol of the tape alphabet): `LEFT`, `RIGHT`, `(WRITE` a`)`, `(GOTO` *label*`)`, `(IF` a `GOTO` *label*`)`.

Lambda Interpreter. Program `intLAM` is an interpreter for an applied call-by-value lambda calculus. It includes constants (lists, unary numbers), a conditional `IF`, operations on lists (`CAR`, `CDR`, `CONS`, `EQUAL?`) and on unary numbers (`+`, `-`, `*`, `/`, `=`). It is an extension of the language treated by the lambda-interpreter in [17].

Flowchart Language. Program `intFCL` is an interpreter for the Flowchart language in [16]. A Flowchart program consists of a list of labeled basic blocks each consisting of a sequence of assignments terminated by a conditional or unconditional jump. Program operates over a global store. Flowchart programs have a Pascal-like semantics.

Java Bytecode. Program `intJBC` is an interpreter for a subset of Java bytecode. The subset is similar to the one used in [20]. The subset contains operations to transfer values between the stack and the local store (`istore`, `iload`), jump instructions (`ifzero`, `goto`, `ireturn`), stack operations (`swap`, `pop`, `dup`), and several arithmetic and relational operators. Programs operate on natural numbers. This is the largest program which we used in the experiments. The instructions are as follows:

```
iload v | istore v | iconst n |
ifzero l | goto l | ireturn |
iadd | isub | |imul | idiv | imod |
iequal | ibigger | dup | pop |swap
```

Self-interpreter. Program `intTSG` is a self-interpreter for the language TSG defined in [4]: the same syntax and semantics.

interpreter	*cons cells*	*functions*
`intTM`	1315	10
`intLAM`	3279	18
`intFCL`	9745	57
`intJBC`	12439	85
`intTSG`	6224	32

5.2 Generation of Inverse Interpreters

Table 1 shows the results of specializing URA with respect to the five interpreters (all timings in seconds). This was done (1) by a direct application of `similix` to URA, and (2) by the application of Similix's generator generator `cogen` to URA and then using the generated program inverter `comp` to transform the interpreters into inverse interpreters (these two ways of specializing URA are described in Sect. 4.1). The residual program produced by (1) or (2) are identical. Program `comp` is the program inverter in Eq. 12.

The conversion of URA by `cogen` into a program inverter `comp` took 8.69 secs. Using `comp` produces the inverse interpreters up to 40 times faster than the direct specialization of URA by `similix` (which in one case led to a segmentation fault). This speedup is somewhat higher than what we expected from Similix [6, 8].

The table also shows the number of cons cells and procedures of the generated inverse interpreters.

5.3 Reduction of Metalevels

Tables 2 to 5 compare the running times of the generated inverse interpreters (`invintTM`, `invintLAM`, `invintFCL`, `invintJBC`) with running times using URA as a semantics modifier (`invmodTSG`) (these two applications are described in Eqs. 7 and 10).

Table 2 shows the inverse computation of a Turing program implementing a pattern matcher. The input to the matcher is a pattern and a text separated by a blank on the input tape. They are replaced by Success or Fail depending on whether the pattern occurs in the text. Inverse computation can determine all such patterns [4, Sect. 10]. We provided also the length of the patterns we looked for, the text and the output Success. The running times are shown for patterns of length=2...5 and a text of symbols (length 42). This is a typical speedup for the pattern matcher with similar input-output classes. Program balance checks whether the parentheses in a text are balanced and depending on this returns Success or Fail. The task was to find all lists with balanced parentheses. Program sort returns the sorted sequence of a sequence of 0's and 1's. The task was to find all possible input sequences given a sorted output sequence (0...01...1) and length=4,6,8,10.

Table 3 shows inverse computation of lambda-programs implementing the list length function, the computation of x^2 and of a cubic equation. In the case of length, the output is fixed and inverse computation returns a list of corresponding lengths, but unspecified elements. Given the value x^2, inverse computation returns the corresponding base x. The running time is the time to find the first (and only) solution; the inverse interpreter does not terminate the search. Given a lambda-program computing a cubic equation, inverse computation with output 0 finds all three solutions that exist ($x = 1, 2, 3$). The running time is the time to compute each of the three solutions. After that, inverse computation continues the search. This non-termination behavior is familiar from logic programming (which is a tool for inverse computation of relational programs). The speedups in these experiments were smaller than those for the Turing interpreter.

Table 4 shows inverse computation of Flowchart programs implementing the factorial, the inorder traversal of a binary tree, and the computation of x^2 with unary numbers. Given the corresponding output, inverse computation computes the inverse of these functions.

Table 5 shows inverse computation of Java bytecode programs implementing the computation of the nth prime num-

[4]Hardware: CPU Pentium3 Mobile 1.2GHz, Memory 1024MB; Software: SCM version 5d2-3, SLIB version 2d2-1, Similix 5.1; OS: Linux, kernel 2.4.18, Debian woody.

Table 1: Specialization by Similix

	intTM	intLAM	intFCL	intJBC	intTSG
Generation Time:					
similix	6.99	55.48	521.2	seg. fault	346.4
cogen	8.69				
comp	1.21	3.65	12.17	16.01	12.09
similix/comp	**5.8**	**15.2**	**42.8**	–	**28.6**
Residual Program:					
name	invintTM	invintLAM	invintFCL	invintJBC	invintTSG
cons cells	7289	21751	51602	60289	46367
procedures	54	130	301	381	218

Table 2: Inverse Interpretation of Turing Programs

	matcher(p,t)				balanced(l)				sort(l)			
input	$\|p\|=2,3,4,5$; $t=$"0...1",$\|t\|=42$				$\|l\|=3,4,5,6$				$\|s\|=4,6,8,10$			
output	'Success				'Success				sorted sequence s=0...01...1			
invmodTSG	8.14	27.51	145.2	844.3	2.22	10.46	180.9	180.5	1.36	11.21	82.87	594.7
invintTM	0.69	1.62	10.39	65.53	0.22	0.92	18.1	18.32	0.15	1.2	9.28	63.41
ratio	**11.8**	**16.9**	**14**	**12.9**	**10.1**	**11.4**	**10**	**9.9**	**9.1**	**9.3**	**8.9**	**9.4**

Table 3: Inverse Interpretation of Lambda Programs

	length(l)				square(x)				cubic$(x)\triangleq x^3-6x^2+11x-6$		
output	20	25	30	35	121	144	169	196	0 (finds solutions $x=1,2,3$)		
invmodTSG	7.49	13.44	21.75	32.21	3.97	5.25	6.91	9.08	21.23	32.61	59.57
invintLAM	1.62	2.84	4.62	6.84	1.65	2.17	2.75	3.6	7.86	11.84	21.29
ratio	**4.6**	**4.7**	**4.7**	**4.7**	**2.4**	**2.4**	**2.5**	**2.5**	**2.7**	**2.7**	**2.8**

Table 4: Inverse Interpretation of Flowchart Programs

	factorial(n)				inorder(t)				square(n)			
output	6	24	120	720	node list l, $\|l\|=5,7,9,11$				25	36	49	64
invmodTSG	13.07	30.7	80.46	450.1	2.17	12.92	82.26	502.3	5.64	6.15	6.6	7.26
invintFCL	1.79	3.44	8.77	39.9	0.31	2.09	14.19	87.65	0.89	0.92	0.98	1.09
ratio	**7.3**	**8.9**	**9.2**	**11.3**	**7**	**6.2**	**5.8**	**5.7**	**6.3**	**6.7**	**6.7**	**6.7**

Table 5: Inverse Interpretation of Java Bytecode Programs

	prime-nth(n)				is-perfect(n)				hailstorm(n)			
output	17	19	23	29	0 (finds $n=...7,9,8,11,...$)				1 (finds $n=...3,20,64,6,...$)			
invmodTSG	27.38	35.97	50.75	81.25	246.9	657.5	736.3	916.1	940.8	971.2	995.8	1211
invintJBC	2.04	2.65	3.71	5.76	27.8	70.18	77.92	96.61	101.2	103.7	107.1	128.9
ratio	**13.4**	**13.6**	**13.7**	**14.1**	**8.9**	**9.4**	**9.4**	**9.5**	**9.3**	**9.4**	**9.3**	**9.4**

Table 6: Inverse Interpretation for Self-Interpreter for TSG

	matcher(p, t)				is-walk(w, g)			
input	$t=$"AB...Z", $\|t\|=26,52,78,104$				g with 9, 13 nodes			
output	'Success				'True		'False	
invmodTSG (t_1)	123.2	642.7	1465	2684	97.93	54.17	108.3	56.52
invintTSG (t_2)	9.2	57.6	105.9	197.1	14.27	7.73	15.21	8.23
ura (t_3)	0.78	3.72	8.14	14.36	1.53	0.62	2.36	1.04
ratio (t_1/t_2)	**13.4**	**11.2**	**13.8**	**13.6**	**6.9**	**7**	**7.1**	**6.9**
ratio (t_2/t_3)	**11.8**	**15.5**	**13**	**13.7**	**9.3**	**12.5**	**6.4**	**7.9**

ber, a predicate that checks whether a number is perfect[5], and the Hailstorm function. Given a prime number, inverse computation of the nth-prime number program computes the number of that prime. Given output 0, which represents false, the inverse computation of program is-perfect produces an (infinite) sequence of numbers that are not perfect. Given output 1, the inverse computation of the Hailstorm function produces a (infinite) sequence of numbers that produce that value. The running times are improved about 10-fold.

5.4 Self-Generation

We have seen the speedup that can be obtained by specializing URA with respect to the interpreters `intTM`, `intLAM`, `intFCL`, and `intJBC`. We would like to know the efficiency of the inverse interpreters compared with their *hand-written* counterparts. However, there is no way to know this because no inverse interpreters have been built for these languages nor are there theories or algorithms for inverse computation for these languages that could serve as a reference point.

To get an idea how far partial evaluation brings us, we implemented a self-interpreter `intTSG` for TSG to *self-generate* an inverse interpreter `invintTSG` and to compare it to the inverse interpreter `ura` which we implemented by hand in Scheme (described in Sect. 3). This indicates how much inverse interpretation overhead was actually removed by the partial evaluator, and how much there is still to go. Program match is a pattern matcher (described above) and program is-walk checks whether a given walk can be realized without cycles in a directed graph. The task was to find all cycle-free walks in two given graphs (True) and to find all values which do not represented cycle-free walks (False) in the same graphs. URA enumerates the answers and terminates because the process tree is finite for a finite graph.

The results in Table 6 show that partial evaluation brought us half-way: we gained an order of magnitude for inverse computation in TSG, but we are still an order of magnitude away from our hand-written implementation of URA in Scheme (`ura`). This can be seen from the ratios t_1/t_2 and t_2/t_3. The proportions change depending on the programs and the test runs. Whether this can serve as an indication of how far we are away from hand-written inverse interpreters for other languages (Turing machines, lambda calculus, flowchart language, Java bytecode, and so on) is an open question.

6. RELATED WORK

The transformation of interpreters into translators is a prime application of partial evaluation. Numerous studies have shown that partial evaluation can drastically reduce interpretive overheads (*e.g.*, [17, 18, 10, 24]). We took advantage of this capability for another application. None of these works applies partial evaluation to the conversion of interpreter into non-standard interpreters (in [23] an inverse interpreter was specialized with respect to a program, but not an interpreter.)

This paper reports first results using a partial evaluator for the reduction of the interpretive overhead introduced by an inversion modifier. A theory for combining non-standard interpreters was developed in [1]. It forms the theoretical basis for semantics-modifiers [2]. URA was applied interpretively to interpreters for several imperative languages [3, 4].

The Universal Resolving Algorithm used in this paper is derived from perfect driving [13] combined with a mechanical extraction of the answers which makes the algorithm as powerful as SLD-resolution, but for a first-order, functional language (*cf.* [14]). Logic programming inherently supports inverse computation for a relational language. To our knowledge, no logic programming system has been subjected to a specializer in order to invert programs.

Similar to ordinary programming, there exists no single programming paradigm that would satisfy all needs of inverse programming. Recently, work has been done on the integration of the functional and logic programming paradigm using narrowing, a unification-based goal-solving mechanism [15]; for a good survey see [5]. The specialization of these techniques with respect to interpreters has not been considered.

7. CONCLUSION

The experiments in this paper apply the idea of prototyping programming language tools from robust semantics: we have produced automatically inverse interpreters for languages for which no inverse interpreter existed before (*e.g.*, Turing machines, lambda calculus, flowchart language, a subset of Java bytecode). To the best of our knowledge, these are the first results regarding this use of partial evaluation. The inverse interpreters generated by a specializer from an interpreter are correct by construction (if the specializer, the semantics modifier and the interpreter are correct).

Our results show that a speedup of an order of magnitude can be achieved for some of the interpreters. Then generation times are very fast when using Similix's generator generator (between 1–16 seconds). Limiting factors of offline partial evaluation are the polyvariant specialization scheme that folds only to program points with identical static arguments, that is the lack of generalization, and the necessity that partial static structures are regular static/dynamic patterns. Several binding-time improvements were necessary to improve the flow of static information at specialization time. To achieve a deeper specialization partially static structures built in the value component during the development of the perfect process tree should be exploited.

We believe there is still more to gain by partial evaluation. Loss of efficiency can be attributed to the familiar encoding problem when running an interpreter on top of an (inverse) interpreter, and the limitation of exploiting static information in the node environments.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for valuable feedback. Special thanks are due to Yoshihiko Futamura and Masahiko Kawabe for useful comments on an earlier version of this paper, to Yusuke Ichikawa for implementing the initial version of `intFCL` and `intJBC`, and to Takeshi Takeuchi for testing some of the programs. The first author would like to thank Sergei Abramov for stimulating joint work leading to the results presented in Sect. 2 and 3.

[5]A perfect number is an integer n that is equal to the sum of all its exact divisors incl. 1 but excl. n (*e.g.* 6=1+2+3).

9. REFERENCES

[1] S. M. Abramov and R. Glück. Combining semantics with non-standard interpreter hierarchies. In S. Kapoor and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, LNCS 1974, pages 201–213. Springer-Verlag, 2000.

[2] S. M. Abramov and R. Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12(2):171–211, 2001.

[3] S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, pages 269–295. Springer-Verlag, 2002.

[4] S. M. Abramov and 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.

[5] E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Computing*, 20(1):3–26, 2002.

[6] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, 1991.

[7] A. Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, Denmark, 1993. Included in the Similix distribution, 82 pages.

[8] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[9] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 11:315–346, 1993.

[10] A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

[11] N. H. Christensen and R. Glück. On the equivalence of online and offline partial evaluation. Manuscript, DIKU, Department of Computer Science, University of Copenhagen, 2001.

[12] E. W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction: International Summer School*, LNCS 69, pages 54–57. Springer-Verlag, 1978.

[13] R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis. Proceedings*, LNCS 724, pages 112–123. Springer-Verlag, 1993.

[14] R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, pages 165–181. Springer-Verlag, 1994.

[15] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[16] J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation. Practice and Theory*, LNCS 1706, pages 20–82. Springer-Verlag, 1999.

[17] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[18] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Conference Record of the Nighteenth Symposium on Principles of Programming Languages*, pages 258–268. ACM Press, 1992.

[19] J. Jørgensen. Similix: a self-applicable partial evaluator for Scheme. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation. Practice and Theory*, LNCS 1706, pages 83–107. Springer-Verlag, 1999.

[20] S. Katsumata and A. Ohori. Proof-directed de-compilation of Java bytecode. In D. Sands, editor, *Programming Languages and Systems. Proceedings*, LNCS 2028, pages 352–366. Springer-Verlag, 2001.

[21] J. McCarthy. The inversion of functions defined by Turing machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 177–181. Princeton University Press, 1956.

[22] S.-C. Mu and R. Bird. Inverting functions as folds. In E. A. Boiten and B. Möller, editors, *Mathematics of Program Construction. Proceedings*, LNCS 2386, pages 209–232. Springer-Verlag, 2002.

[23] A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, LNCS 1181, pages 249–260. Springer-Verlag, 1996.

[24] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.