SEMANTICS MODIFIERS: AN APPROACH TO NON-STANDARD SEMANTICS OF PROGRAMMING LANGUAGES

SERGEI ABRAMOV
Program Systems Institute, Russian Academy of Sciences,
RU-152140 Pereslavl-Zalessky, Russia
E-mail: abram@botik.ru

ROBERT GLÜCK
Department of Computer Science, University of Copenhagen,
Universitetsparken 1,
DK-2100 Copenhagen, Denmark
E-mail: glueck@diku.dk

An approach for systematically modifying the semantics of programming languages by *semantics modifiers* is described. It allows the design of general and reusable "semantics components" without the penalty of being too inefficient. Inverse computation and neighborhood analysis are used to demonstrate the computational feasibility of the approach. Equivalence transformers are shown to fulfill the requirements of semantics modifiers. Finally, projections are given that allow the efficient implementation of non-standard interpreters and non-standard compilers using program specialization.

1 Introduction

We introduce a novel approach for systematically modifying the standard semantics of programming languages by *semantic modifiers*. Our approach allows the design of general and reusable "semantics components" for a wide class of computational problems without the penalty of being too inefficient. Language independence is achieved through the *interpretive approach* [22,9]: an interpreter serves as mediator between the new language and the language for which the non-standard semantics has been implemented. Efficient implementations of programming tools, non-standard interpreters and non-standard compilers, can be obtained automatically using program specialization [6,20,14].

It can be quite tedious to develop a *non-standard semantics* for a programming language. For example, it is considerably more difficult to implement the *inverse semantics* of a language, say a functional language, than to implement its *standard semantics*. It would be ideal to have the possibility to build a set of generic algorithms for non-standard semantics, to establish their correctness once, and to reuse the algorithms for a large class of different programming languages. Semantics modifiers promise language independence

and could drastically increase the applicability of non-standard methods.

As shown in this paper, semantics modifiers exist for a wide class of computational problems, including *non-standard computation*, *program analysis* and *program transformation*. Well-known applications, such as *interpreter hierarchies*, are shown to be instances of our approach. These results are not only interesting from a methodological viewpoint, they allow one to study a large class of different problems within a common framework, but may also lead to new insights regarding the combination and reuse of "semantics components", and possibly of other generic software components.

Programs that manipulate programs as data objects are meta-programs. Meta-programs are often used to implement non-standard semantics for programming languages (e.g., inverse computation). Let `metaL` be a meta-program for language `L` and let `pgmL` be an `L`-program. The application of the meta-program can be illustrated as follows.

```
metaL _________          – meta-program for L
      pgmL data          – program written in L
```

The main idea of the interpretive approach is to insert an `N`-interpreter `intN` written in `L` between the meta-program and an `N`-program `pgmN` to port the non-standard semantics implemented by `metaL` to the new language `N`:

```
metaL ________________________   – meta-program for L
      intN ___________________   – N-interpreter written in L
           [pgmN, data]          – program written in N
```

Our use of the interpretive approach is different from previous work[22,10,18]: our aim is not to expose more information to a meta-program by instrumenting an interpreter, but rather to combine a standard interpreter with different meta-programs. The interpretive overhead can be removed by program specialization or other powerful program transformations.

Note that we use the term non-standard semantics in a broader sense than in abstract interpretation [4] where emphasis is on approximative analysis for program optimization rather than alternate ways of simulating, analyzing and transforming programs.

To summarize, the approach outlined here allows to

1. *develop generic modifiers* for a language `L`,
2. *prototype non-standard semantics* for new languages `N` given their standard semantics and the corresponding `L`-modifier, and
3. *produce efficient implementations* of non-standard interpreters and non-standard compilers for `N` by program transformation.

This paper presents a new class of programs, namely semantics modifiers (Section 3). It shows the correctness and computational feasibility of modifiers for inverse computation (Section 4) and neighborhood analysis (Section 5), establishes that all equivalence transformers fulfill the requirements of semantics modifiers (Section 6), and shows that interpreter hierarchies are covered by our framework (Section 7). Finally, seven projections are given that allow the efficient implementation of non-standard interpreters and non-standard compilers by program specialization (Section 8). Related work is cited in the respective sections. The paper draws upon earlier results[3]. The language `Gofer`, a dialect of `Haskell`[12], is used as presentation language.

2 Basic Notions

2.1 Computation and Interpretation

Definition 1 (programming language)[6] A programming language `L` *is a triple* `L = (PgmL,DatL,SemL)` *where* `PgmL` *is the syntax of* `L` *(the set of syntactically correct* `L`*-programs),* `DatL` *is the data domain of* `L`*, and* `SemL` *is the semantics of* `L`*, a partially recursive function:* `SemL :: [PgmL, DatL] → DatL`.

Definition 2 (computation) *Let* `pgm∈PgmL` *be an* `L`*-program,* `dat∈DatL` *be an input for* `pgm`*, and* `out∈DatL` *be the output of applying* `pgm` *to* `dat` *in language* `L`*:* `out = SemL [pgm,dat]`*, then we denote* computation in `L` *by*

$$\texttt{pgm dat} \overset{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out} \ .$$

In this paper we assume a fixed data domain, `D`, for *all* programming languages, and for representing programs written in different languages. This is convenient when dealing with programs that take programs as input, e.g. interpreters.

Definition 3 (interpreter) *Let* `L`, `M` *be programming languages. An* `M`*-program* `intL` *is an* `L`-interpreter *iff for all* `pgm∈PgmL` *and for all* `dat∈D`*:*

$$(\texttt{intL [pgm,dat]} \overset{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{out}) \iff (\texttt{pgm dat} \overset{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out}) \ .$$

2.2 Data Domains and their Representation

To represent subsets of a data domain `D` (e.g., strings, lists) we use expressions with variables. Variables may range over any subset of `D`. An expression without variables is *ground*.

Definition 4 (substitution) *A* substitution $\texttt{s} = [\texttt{x}_1 \mapsto \texttt{exp}_1, \ldots, \texttt{x}_n \mapsto \texttt{exp}_n]$ *is a list of variable/expression pairs such that variables* $\texttt{x}_i$ *are pairwise distinct and* $\texttt{exp}_i$ *are expressions. The result of applying a substitution* `s` *to an expression* `exp`*, denoted by* `exp/s`*, is an expression obtained by replacing every occurence of* $\texttt{x}_i$ *by* $\texttt{exp}_i$ *in* `exp`*, for every pair* $\texttt{x}_i \mapsto \texttt{exp}_i$ *in* `s`.

Definition 5 (full valid substitution) *A* full valid substitution `FVS(exp)` *is the set of all substitutions for expression* `exp` *such that every substitution defines a value (data from* `D`*) for every variable occurring in* `exp` *and these values do not violate the domains of the variables.*

Definition 6 (class) *A* class `cls` *is an expression with variables where every variable ranges over some subset of* `D`*. A class* `cls` *represents the set of data*

$$|\texttt{cls}| \stackrel{\text{def}}{=} \{\ \texttt{d} \mid \texttt{s} \in \texttt{FVS(cls)},\ \texttt{d = cls/s}\ \}\ .$$

For the experiments shown in this paper let `D` be the set of non-circular S-expressions (known from Lisp) defined by the grammar:

```
D = Atoms | (CONS D D)
```

where `Atoms` is a possibly infinite set of symbols. Two types of variables are used for representing sets of S-expressions: $\mathcal{A}_i$, $\mathcal{E}_i$ where variables $\mathcal{A}_i$ range over `Atoms` and variables $\mathcal{E}_i$ range over `D`.

Example 1 *Consider class* `cls = (CONS 'Z (CONS` $\mathcal{A}_3$ $\mathcal{E}_7$`))`*. Class* `cls` *represents the set* $|\texttt{cls}| = \{$ `(CONS 'Z (CONS` a d`))` $\mid a \in$ `Atoms,` $d \in$ `D`$\}$.

3 Semantics Modifiers

We now define a new class of programs, namely *semantics modifiers*, and discuss their properties.

Definition 7 (non-standard dialect, non-standard semantics) *Let* L *be a set of programming languages,* S' *be a map (function),* S'`(L) = L'` *where* `L = (PgmL, DatL, SemL)`$\in L$, `L' = (PgmL, DatL, SemL')`*, then* `L'` *is a* non-standard S'-dialect *of* `L`*,* `SemL'` *is a* non-standard S'-semantics of `L`*, and* S' *is a* non-standard semantics for L.

Definition 8 (semantics modifier) *Let* N *be a set of programming languages and* S' *be a non-standard semantics for* N*. An* `M`*-program* `sem-mod` *is an* `L/M`-semantics modifier for S' *if for all languages* `N` $\in N$*, all* `N/L`*-interpreters* `intN`*, all* `N`*-programs* `pgm`*, all input* `req`*, and* S'*-dialect* `N'` *of* `N`

$$(\texttt{sem-mod [intN, pgm, req]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{answer}) \iff (\texttt{pgm req} \stackrel{*}{\underset{\texttt{N'}}{\Rightarrow}} \texttt{answer})\ .$$

Definition 8 states that the answer returned by applying a semantics modifier `sem-mod` to any standard interpreter `IntN` for language `N` is the same as the answer returned by the non-standard semantics of language `N'`. Once we have a semantics modifier for S', we have a straightforward method to obtain an S'-semantics for any language in N.

Semantics modifiers are powerful because they allow the definition of *non-standard interpreters* for programming languages given their *standard interpreter*. A semantics modifier can be regarded as a generic implementation of

the non-standard semantics S'. In other words, we have a constructive method to realize map S'.

Given an N/L-interpreter intN and an L/M-modifier sem-mod for S', we can immediately define a non-standard N'/M-interpreter intN' for the non-standard S'-dialect N' of N by

$$\texttt{intN' [pgm, req]} \stackrel{\text{def}}{=} \texttt{sem-mod [intN, pgm, req]}$$

which performs non-standard computation of N'-programs according to S':

$$(\texttt{intN' [pgm, req]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{answer}) \iff (\texttt{pgm req} \stackrel{*}{\underset{\texttt{N'}}{\Rightarrow}} \texttt{answer}) \ .$$

Assuming that the standard interpreter intN is correct, the correctness of the non-standard interpreter intN' is guaranteed by the correctness of the semantics modifier sem-mod (which has to be established only once). This guarantees that the semantics modification of N is what was intended. For example, using a semantics modifier for inverse computation we get the inverse semantics $\texttt{N}_{\texttt{inv}}$ of N (Sec. 4).

One can imagine that several different non-standard semantics are ported to a new language N just by providing a single standard interpreter for N written in L. This idea is illustrated in Fig. 1 where inv-mod-L is a modifier for inverse computation, nan-mod-L a modifier for neighborhood analysis, eqt-mod-L a modifier for equivalence transformation, and id-mod-L a modifier for identity semantics. Experiments with inverse computation and neighborhood analysis are shown in Sec. 4.4 and 5.4. This way new language dialects can be obtained in an easy and systematic way.

All programs that fulfill the requirements of Def. 8 are semantics modifiers. The question remains: are there any 'interesting' modifiers? It is clear that the concept would be too restricted, if semantics modifiers had only limited use, or if they could not be implemented efficiently. These issues have to be addressed in order to justify the further development of this approach. First answers to these questions are presented in the remainder of this paper.

We conjecture that all non-standard semantics that can be specified extensionally, independently of particular intensional properties of interpreters, are potential candidates for semantics modifiers. We show that semantics modifiers exist for a rather diverse class of problems:

- Non-standard computation: inverse computation (Sec. 4).
- Program analysis: neighborhood analysis (Sec. 5).
- Program transformation: equivalence transformation (Sec. 6).
- Interpretation: meta-interpreters (Sec. 7).

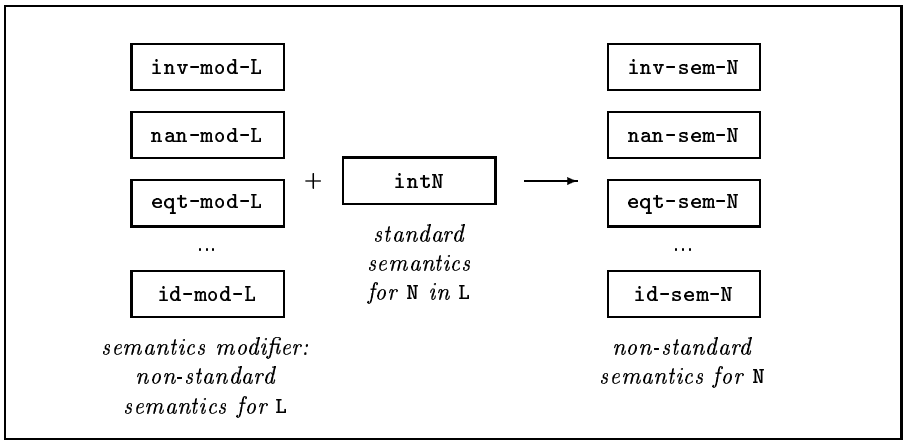

Figure 1: Semantics modifier + standard semantics = non-standard semantics.

4 Non-Standard Computation: Inversion Modifier

Inverse computation is a non-standard semantics. This section defines a semantics modifier for an inverse semantics. The practical feasibility is illustrated using a universal resolving algorithm for inverse computation in a functional language and porting it to a simple imperative language. The relation between inversion modifiers and logic programming is discussed.

4.1 Inverse Semantics

Inverse computation is an important and useful concept in different areas, e.g. logic programming and program verification. While conventional computation is the calculation of the output of a program for a given input, inverse computation is the calculation of the possible input of a program for a given output.[a]

Definition 9 (inverse semantics) *Let* L *be a programming language,* pgm *be an* L*-program,* cls *be a class, and* out *be data, then the* inverse semantics *is the set*

$$\texttt{INV(L,pgm,cls,out)} \stackrel{\text{def}}{=} \{\ \texttt{dat} \mid \texttt{dat} \in |\texttt{cls}|,\ \texttt{pgm dat} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out}\ \} .$$

[a] The solution of an inversion problem can either be *universal* (all possible input) or *existential* (one possible input). Here we consider only universal solutions.

Definition 10 (universal resolving algorithm) *An* M*-program* uraL *is a* universal resolving algorithm *for a programming language* L *if for all* L*-programs* pgm*, all classes* cls*, and all data* out*:*

$$\texttt{uraL [pgm, [cls, out]]} \overset{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{answer}$$

where answer = [s_1,...] *is a (possibly infinite) enumeration of substitutions* s_i *such that*

$$\texttt{INV(L,pgm,cls,out)} = \bigcup_i |\texttt{cls}/s_i| \ .$$

The pair [cls, out] is a *request* for inverse computation where class cls represents the set of admissible input and out is the desired output data. Set INV(L,pgm,cls,out) is the *solution* of the given inversion problem. It is the largest subset of |cls| such that pgm dat $\overset{*}{\underset{\texttt{L}}{\Rightarrow}}$ out for all elements of the solution.

The computation of a constructive representation of INV(L,pgm,cls,out) is called *inverse computation*. Given request [cls, out], a universal resolving algorithm, if it terminates, performs inverse computation of a program pgm. The enumeration [cls/s_1,...] is a representation of set INV(L,pgm,cls,out). Computability of the answer is not always guaranteed, even with sophisticated strategies. Some inversions are too resource consuming, while others are undecidable. In general, it is undecidable whether all substitutions have been found.

4.2 A Semantics Modifier for Inverse Computation

Let uraL perform inverse computation in language L, then inverse computation in a new language N can be performed by inverting N's interpretation.

Theorem 11 (interpretive inversion) *Let* intN *be an* N*-interpreter written in* L *and* uraL *be a universal resolving algorithm for* L *written in* M*. For all* N*-programs* pgm *and all requests* [cls, out]*,* M*-program* uraN *is a universal resolving algorithm for* N *where*

$$\texttt{uraN [pgm, [cls, out]]} \overset{\text{def}}{=} \texttt{uraL [intN, [[pgm, cls], out]]} \ .$$

Proof. *Program* uraN *performs inverse computation in language* N *because for all* N/L*-interpreters* intN*, all* N*-programs* pgm*, all classes* cls*, and all data* out*, the result* answer = [s_1,...] *produced by*

$$\texttt{uraN [pgm, [cls, out]]} \overset{*}{\Rightarrow} \texttt{answer}$$

satisfies the condition for inverse computation in N*:*

$$\texttt{INV(N,pgm,cls,out)} = \bigcup_i |\texttt{cls}/s_i|.$$

Since $\texttt{uraN [pgm, [cls, out]]} \stackrel{\text{def}}{=} \texttt{uraL [intN, [[pgm, cls], out]]}$*:*

$$\texttt{INV(L,intN,[pgm,cls],out)} = \bigcup_i |\texttt{[pgm, cls]}/\texttt{s}_i|$$

$$\stackrel{\text{(Def. 9)}}{\Longleftrightarrow} \quad \{\ \texttt{dat} \mid \texttt{dat} \in |\texttt{[pgm, cls]}|,\ \texttt{intN dat} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out}\ \} = \bigcup_i |\texttt{[pgm, cls]}/\texttt{s}_i|$$

$$\stackrel{\textit{(format)}}{\Longleftrightarrow} \quad \{\ \texttt{[pgm', dat']} \mid \texttt{[pgm', dat']} \in |\texttt{[pgm, cls]}|,\ \texttt{intN [pgm', dat']} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out}\ \} = \bigcup_i |\texttt{[pgm, cls]}/\texttt{s}_i|$$

$$\stackrel{(\texttt{pgm } \textit{gnd})}{\Longleftrightarrow} \quad \{\ \texttt{[pgm, dat']} \mid \texttt{dat'} \in |\texttt{cls}|,\ \texttt{intN [pgm, dat']} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out}\ \} = \bigcup_i |\texttt{[pgm, cls/s}_i\texttt{]}|$$

$$\stackrel{\textit{(format)}}{\Longleftrightarrow} \quad \{\ \texttt{dat'} \mid \texttt{dat'} \in |\texttt{cls}|,\ \texttt{intN [pgm, dat']} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out}\} = \bigcup_i |\texttt{cls/s}_i|$$

$$\stackrel{\text{(Def. 3)}}{\Longleftrightarrow} \quad \{\ \texttt{dat'} \mid \texttt{dat'} \in |\texttt{cls}|,\ \texttt{pgm dat'} \stackrel{*}{\Rightarrow}_{\texttt{N}} \texttt{out}\ \} = \bigcup_i |\texttt{cls/s}_i|$$

$$\stackrel{\text{(Def. 9)}}{\Longleftrightarrow} \quad \texttt{INV(N,pgm,cls,out)} = \bigcup_i |\texttt{cls/s}_i|$$

□

Theorem 11 states that the answer obtained by inverse computation of N's interpreter is an answer for inverse computation in N. Inverse computation can be performed in any programming language N given a universal resolving algorithm uraL for L and an N-interpreter intN written in L. The theorem guarantees that the answers are *correct* for all N-program regardless of intN's algorithmic properties. Note that the language paradigm of N (imperative, declarative, object-oriented, etc.) is quite irrelevant. It is straightforward to define a program inv-mod, a semantics modifier for inverse computation.

Definition 12 (inversion modifier) *Given a universal resolving algorithm* uraL *for* L *written in* M*, let* M*-program* inv-mod *be defined as*

$$\texttt{inv-mod [intN, pgm, [cls, out]]} \stackrel{\text{def}}{=} \texttt{uraL [intN, [[pgm, cls], out]]}.$$

Theorem 13 (modifier property of inv-mod**)** *Program* inv-mod *is an* L/M*-semantics modifier for inverse semantics.*

Proof. *That* inv-mod *performs inverse computation in any language* N *given an* N*-interpreter* intN *written in* L *follows directly from Theorem 11.* □

A *non-standard semantics* invN for an inverse dialect $\texttt{N}_{\texttt{inv}}$ of N:

$$\texttt{intN [pgm, dat]} \stackrel{*}{\Rightarrow} \texttt{out} \qquad \text{– standard computation in N}$$

$$\texttt{invN [pgm, request]} \stackrel{*}{\Rightarrow} \texttt{answer} \qquad \text{– inverse computation in N}$$

with $\texttt{invN [pgm, [cls, out]]} \stackrel{\text{def}}{=} \texttt{inv-mod [intN, pgm, [cls, out]]}$

4.3 A Universal Resolving Algorithm

We are now going to illustrate the practical feasibility of the inversion modifier. First we give an algorithm for inverse computation and then port it to a simple imperative language.

There exist different methods for inverse computation[19,5,11]. The universal resolving algorithm outlined in this section uses methods from supercompilation, in particular driving[20]. The idea for this algorithm appeared in the early seventies[19] and since then several variations were implemented for functional languages[15,16,2]. We use an algorithm[b] for `TSG`, a typed dialect of the first-order, functional language `S-Graph`[8]. The algorithm is implemented in `Gofer` (321 lines of pretty-printed source text).

Algorithm URA (outline). Given `TSG`-program `pgm`, class `cls` and output `out`, the algorithm starts by driving the initial configuration `pgm cls` and builds a potentially infinite process tree[8] using a *breadth-first strategy*. In each step of driving a configuration $\texttt{pgm cls}_i$, the algorithm may encounter two basic cases:

1. *Driving finishes.* If the result of $\texttt{pgm cls}_i$ is $\texttt{out}_i$ and the unification $\texttt{out}_i\texttt{:out}$ succeeds with substitution `s`, where $\texttt{out}_i\texttt{/s = out}$, then substitution `s'` resulting from unification $\texttt{cls:(cls}_i\texttt{/s)}$ is added to the answer (and printed); otherwise nothing is added.

2. *A contraction is encountered.* Configuration $\texttt{pgm cls}_i$ is split into disjoint configurations $\{\texttt{pgm cls}_1, \ldots, \texttt{pgm cls}_n\}$ such that $|\texttt{pgm cls}_1| \cup \ldots \cup |\texttt{pgm cls}_n| = |\texttt{pgm cls}_i|$ and $|\texttt{pgm cls}_1| \cap \ldots \cap |\texttt{pgm cls}_n| = \emptyset \quad (n \geq 1)$.[c]

The algorithm stops if all configurations in the process tree are completed; otherwise it continues driving the next unfinished configuration $\texttt{pgm cls}_j$.

We shall not be concerned with technical details, only with the fact that the answer produced by the algorithm is indeed a correct solution and that each solution, if it exists, is computed in finite time. Note that the algorithm does not always terminate because the search for solutions may continue infinitely (even though all substitutions have been enumerated).

Theorem 14 (correctness of URA)[3] *Given* `TSG`*-program* `pgm`*, class* `cls` *and output* `out`*, algorithm* URA *computes a representation of the set* `INV(TSG, pgm,cls,out)`.

[b] All run-times are given in cpu-seconds (including garbage collection, if any) using WinGofer 2.30b and a PC/Intel Pentium 133Mhz, Windows 95. Programs available from: `ftp://ftp.botik.ru/pub/local/Sergei.Abramov/book.appndx/mc_and_appls.zip`

[c] In the actual implementation restrictions on variable domains are used to make the classes disjoint (e.g., $\mathcal{A}_1 \neq$ `'B`). An empty restriction list is shown as `[]` in the figures.

```
ura [ match,
      [ ([E1, (CONS 'A (CONS 'A (CONS 'A 'NIL)))],[]),
        'FAILURE
      ]
    ] =   -- Inverse computation returns the solution for E1:
  [ ([E1 ↦ (CONS (CONS E13 E14) E11)],[]),
    ([E1 ↦ (CONS 'A (CONS (CONS E19 E20) E17))],[]),
    ([E1 ↦ (CONS 'A (CONS 'A (CONS (CONS E25 E26) E23)))],[]),
    ([E1 ↦ (CONS 'A (CONS 'A (CONS 'A (CONS (CONS E31 E32)E29))))],[]),
    ([E1 ↦ (CONS 'A (CONS 'A (CONS 'A (CONS A33 E29))))],[]),
    ([E1 ↦ (CONS A15 E11)],[A15 ≠ 'A]),
    ([E1 ↦ (CONS 'A (CONS 'A (CONS A27 E23)))], [A27 ≠ 'A]),
    ([E1 ↦ (CONS 'A (CONS A21 E17))], [A21 ≠ 'A])
  ]
```

Figure 2: Inverse computation: the set of strings which are *not* substrings of "AAA".

Theorem 15 (termination of URA)[3] *Given* TSG*-program* pgm*, class* cls *and output* out*, algorithm* URA *computes each substitution* s_i *and each prefix* $s_1,\ldots,s_i$ *of the answer* $[s_1,\ldots]$ *in finite time. For each solution* d∈INV(TSG, pgm,cls,out)*, if it exists, the corresponding substitution* s_j*:* d∈(cls/s_j)*, is computed in finite time.*

Example 2 *Consider a naive pattern matcher* match[8] *which tests whether a string* pat *is a substring of string* str*:*

match [pat, str] $\stackrel{*}{\underset{\text{TSG}}{\Rightarrow}}$ 'SUCCESS *or* 'FAILURE .

For instance, computation match ["B","AAA"] $\stackrel{*}{\underset{\text{TSG}}{\Rightarrow}}$ 'FAILURE*. Assume we want to find the set of strings* pat *which are* not *substrings of* str*. Inverse computation can solve this problem. Figure 2 shows the result of applying the universal resolving algorithm to* match *where* pat *is unknown (*$\mathcal{E}_1$*),* str = "AAA"*, and the output is fixed to* 'FAILURE*. The result is a finite representation of the set of strings which are not substrings of* "AAA"*.*

In case we want to find the set of strings pat *which* are *substrings of* "AAA"*, we set the desired output to* 'SUCCESS*. The answer is a finite representation of all possible substrings of string* "AAA" *(not shown). The universal resolving algorithm terminates after 1.2 sec. in both cases.*

4.4 Inverse Computation in an Imperative Language

Consider the imperative language Norma[13]. A Norma-program works on two variables (x, y) holding natural numbers. It is a sequence of instructions including conditional (zeroX, zeroY) and unconditional jumps (goto), as well as

```
inv-mod [intNorma, pgmNorma, cls, y] =
      [ ([ℰ1 ↦ (CONS ℰ2 (CONS ℰ5 (CONS ℰ8 𝒜13)))], [])
      -- Inverse computation returns the solution for ℰ1
      -- in equation 10 = 2*((ℰ1)+1)+2.  The result is the
      -- most general unary representation of 3.

intNorma = ...                -- Norma interpreter written in TSG.

pgmNorma ::  PgmNorma         -- Norma program computes y = 2*x+2.
pgmNorma = list[ incY,        -- :0
                 incY,        -- :1
                 (zeroX 7),   -- :2 -- jump if x=0
                 incY,        -- :3
                 incY,        -- :4
                 decX,        -- :5
                 (goto 2)     -- :6
               ]              -- :7 -- stop, return y

cls = ( [(CONS '1 ℰ1), []] )
      -- Unary representation of expression (ℰ1)+1.

y =   (CONS '1 (CONS '1 (CONS '1 (CONS '1 (CONS '1
      (CONS '1 (CONS '1 (CONS '1 (CONS '1 (CONS '1 'NIL )))))))))
      -- Unary representation of 10.
```

Figure 3: Inversion modifier: inverse computation of a Norma program.

operations to increment and decrement the variables (`incX`, `incY`). Standard computation of a `Norma`-programs takes the value of `x` as input, initializes `y` with zero, and returns the value of `y` as output (if the program terminates). Numbers are represented by lists of length n. An example program to compute `y = 2*x + 2` is shown in Fig. 3.

Instead of designing a universal resolving algorithm for inverse computation in `Norma`, we can use the semantics-modifier `inv-mod` using the universal resolving algorithm URA for `TSG` together with a `Norma`-interpreter written in `TSG` (95 lines of pretty-printed source text). A number of non-trivial imperative programs has been inverted this way.

Example 3 *Assume we have a Norma program to compute* `y` *in equation* `y = 2*x + 2`. *The inversion modifier can be used to compute a solution for* `x` *in equation*

```
y = 2*(x + z) + 2
```

where `y` *and* `z` *are given. The result of inverse computation in Norma is shown in Fig. 3 where* `y=10`, `z=1`. *The result for* `x` *is the most general unary represen-*

tation of `3`. *This solves the inversion problem without implementing a universal resolving algorithm for* `Norma`. *The first (and only) solution was found in 2.3 secs.; the search continues further.*

Example 3 illustrates the power of Theorem 11: inverse computation can be performed in any programming language given its standard semantics in the form of an interpreter and an algorithm for inverse computation. We ported inverse computation to several languages, including a language for describing finite state automata and other imperative languages.

Note that Theorem 11 guarantees the correctness of all answers, but it does not specify their quality (e.g., any enumeration representing the correct solution is a correct answer). In other words, although the answer is correct, a particular representation may be less useful in practice (e.g., the order in which substitutions are listed may be different, or a substitution may be represented by several substitutions). More advanced inversion strategies may be useful in this regard[17].

4.5 Inversion Modifiers and Logic Programming

Inverse programming is a style of programming where one writes programs so that their inverse computation produces the desired result. This gives rise to a declarative, non-procedural style of programming where one *specifies* the properties of the result rather than describes how it is computed. Instead of computing `y` by `result[x]` $\stackrel{*}{\Rightarrow}$ `y`, one solves the inversion problem for `y` given values for `x` and `out`: `isResult[x,y]` $\stackrel{*}{\Rightarrow}$ `out`.

In logic programming one writes a relational program `isResult` and solves the inversion problem for `out = 'TRUE`. Inverse programming does not depend on a particular type of language, but can be performed in any programming language. An inversion modifier is a tool that allows to port inverse computation to other languages. Indeed, we have seen that inverse computation can be performed in an imperative language given an interpreter written in a functional language and the corresponding universal resolving algorithm (Sec. 4.3, 4.4).

A logic programming system, say Prolog, can be regarded as a semantics modifier for inverse computation. Work in this direction includes the inversion of imperative programs by treating their relational semantics as logic programs.[17] Alternatively, one can perform inverse computation by providing an interpreter written in the corresponding relational language (e.g., a Norma interpreter written in Prolog). The success of these methods depends on the underlying logic programming control strategies.

While these approaches are technically feasible, they do not always allow us to address the main issues discussed above (generality, efficiency). First, a logic

programming system is a specific semantics modifier, while we are interested in a more general framework of semantics modifications (Sec. 3). Second, there is usually no access to the underlying implementation of the system that would allow us to remove the interpretive overhead between the new language N and the implementation language M. Finally, one cannot transform a program pgm into the inverse program $\texttt{pgm}^{-1}$. The modifier approach advocated here allows these and other transformation (Sec. 8).

5 Program Analysis: Neighborhood-Analysis Modifier

Similar to inverse computation, the semantics of *neighborhood analysis* [3] can be ported from one programming language to another using semantics modifiers. We show the correctness of the semantics modifier for neighborhood analysis and illustrate the computational feasibility of the modifier by porting the analysis from the functional language TSG to the imperative language Norma (using the same interpreter already used for porting inverse computation to Norma).

5.1 Semantics of Neighborhood Analysis

Given a program pgm and data dat, a neighborhood analysis determines which part of pgm and dat is actually used in the computation $\texttt{pgm dat} \stackrel{*}{\Rightarrow} \texttt{out}$. The result of the analysis is a class $\texttt{cls}^{\texttt{[pgm,dat]}}$, the *neighborhood of* [pgm, dat], that describes how one can change pgm and dat without changing the output of the computation. Neighborhood analysis has, among others, applications to program testing [2] and termination of supercompilation [21].

Definition 16 (nan-semantics) *Let* L *be a programming language,* pgm *be an* L*-program, and* dat *be data, then the* neighborhood-analysis semantics *is the set*

$$\begin{aligned}&\texttt{NAN(L,pgm,dat)} \stackrel{\text{def}}{=}\\ &\{\ (\texttt{out},\texttt{cls}^{\texttt{[pgm,dat]}})\ \mid\ \texttt{pgm dat} \stackrel{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out},\\ &\quad \texttt{cls}^{\texttt{[pgm,dat]}}\ \textit{is a class},\ \texttt{[pgm,dat]} \in |\texttt{cls}^{\texttt{[pgm,dat]}}|,\\ &\quad (\forall \texttt{[pgm',dat']} \in |\texttt{cls}^{\texttt{[pgm,dat]}}|\texttt{: pgm' dat'} \stackrel{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out})\ \}\ .\end{aligned}$$

Definition 17 (nan-analyzer) *An* M*-program* nanL *is an* L/M-neighborhood analyzer *if for all* L*-programs* pgm*, all data* dat*, the analyzer produces the result:*

$$\begin{aligned}&\texttt{nanL [pgm, dat]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} (\texttt{out},\texttt{cls}^{\texttt{dat}})\ \textit{and}\\ &(\texttt{out},\texttt{[pgm},\texttt{cls}^{\texttt{dat}}\texttt{]}) \in \texttt{NAN(L,pgm,dat)}\ .\end{aligned}$$

5.2 A Semantics Modifier for Neighborhood Analysis

Given a neighborhood analyzer nanL for language L, the analysis can be performed in any programming language N given an N-interpreter written in L. The following theorem guarantees that the result of the analysis is *correct* for all N-program regardless of intN's algorithmic properties. The definition of the semantics modifier nan-mod is straightforward.

Definition 18 (nan-modifier) *Given an* L/M*-neighborhood analyzer* nanL*, let* M*-program* nan-mod *be defined as*

$$\texttt{nan-mod [intN, pgm, dat]} \stackrel{\text{def}}{=} \texttt{nanL [intN, [pgm,dat]]} .$$

Theorem 19 (modifier property of nan-mod**)** *Program* nan-mod *is an* L/M*-semantics modifier for nan-semantics.*

Proof. *Program* nan-mod *is an* L/M*-semantics modifier for nan-semantics because for all* N/L*-interpreters* intN*, all* N*-programs* pgm*, and all data* dat*, the result of*

$$\texttt{nan-mod [intN, pgm, dat]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{(out,cls}^{\texttt{[pgm,dat]}}\texttt{)}$$

satisfies the condition for nan-semantics in N*:*

$$\texttt{(out,cls}^{\texttt{[pgm,dat]}}\texttt{)} \in \texttt{NAN(N,pgm,dat)}.$$

Let [intN,cls] *be a class. Since* intN *is ground, we have:*

cls *is a class and*
[intN',x] $\in$ |[intN,cls]| $\iff$ (intN' = intN *and* x $\in$ |cls|).

Thus we have:

$$\begin{array}{ll}
 & \texttt{nan-mod [intN, pgm, dat]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{(out,cls}^{\texttt{[pgm,dat]}}\texttt{)} \\
\overset{\text{(Def. 18)}}{\iff} & \texttt{nanL [intN, [pgm,dat]]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{(out,cls}^{\texttt{[pgm,dat]}}\texttt{)} \\
\overset{\text{(Def. 17)}}{\iff} & \texttt{(out,[intN,cls}^{\texttt{[pgm,dat]}}\texttt{])} \in \texttt{NAN(L,intN,[pgm,dat])} \\
\overset{\text{(Def. 16)}}{\iff} & \texttt{intN [pgm,dat]} \stackrel{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out, [intN,cls}^{\texttt{[pgm,dat]}}\texttt{]} \textit{ is a class,} \\
 & \texttt{[intN,[pgm,dat]]} \in |\texttt{[intN,cls}^{\texttt{[pgm,dat]}}\texttt{]}|, \\
 & (\forall \texttt{[intN',[pgm',dat']]} \in |\texttt{[intN,cls}^{\texttt{[pgm,dat]}}\texttt{]}|: \\
 & \texttt{intN' [pgm',dat']} \stackrel{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out}) \\
\overset{\text{(Def. 3)}}{\iff} & \texttt{pgm dat} \stackrel{*}{\underset{\texttt{N}}{\Rightarrow}} \texttt{out, cls}^{\texttt{[pgm,dat]}} \textit{ is a class,} \\
 & \texttt{[pgm,dat]} \in |\texttt{cls}^{\texttt{[pgm,dat]}}|, \\
 & (\forall \texttt{[pgm',dat']} \in |\texttt{cls}^{\texttt{[pgm,dat]}}|\texttt{: intN [pgm',dat']} \stackrel{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out}) \\
\overset{\text{(Def. 3)}}{\iff} & \texttt{pgm dat} \stackrel{*}{\underset{\texttt{N}}{\Rightarrow}} \texttt{out, cls}^{\texttt{[pgm,dat]}} \textit{ is a class,} \\
 & \texttt{[pgm,dat]} \in |\texttt{cls}^{\texttt{[pgm,dat]}}|, \\
 & (\forall \texttt{[pgm',dat']} \in |\texttt{cls}^{\texttt{[pgm,dat]}}|\texttt{: pgm' dat'} \stackrel{*}{\underset{\texttt{N}}{\Rightarrow}} \texttt{out}) \\
\overset{\text{(Def. 16)}}{\iff} & \texttt{(out,cls}^{\texttt{[pgm,dat]}}\texttt{)} \in \texttt{NAN(N,pgm,dat)}
\end{array}$$

□

5.3 A Neighborhood Analyzer

Although Definition 17 states nothing about the quality of the neighborhood analysis, we expect that the analysis used in the nan-modifier is non-trivial. In the section we outline a non-trivial algorithm for neighborhood analysis of TSG-programs. [2] The neighborhood returned by this algorithm describes the set of all input which has the same computation process ('trace') and the same output. The analysis is implemented in Gofer (310 lines of pretty-printed source text).

Algorithm NAN (outline). Given TSG-program pgm, data dat, let cls be the most general class representing all possible data for pgm. The algorithm performs two processes: computation of pgm dat and driving of pgm cls. It builds a potentially infinite process tree by driving, but selects only those branches for driving that are chosen by the computation (interpretation) of program pgm dat. In each step of computing a state pgm $\mathtt{dat}_i$ and driving a configuration pgm $\mathtt{cls}_i$, the algorithm may encounter two basic cases:

1. *Computation and driving finish.* If the result of computing pgm $\mathtt{dat}_i$ is out and the result of driving pgm $\mathtt{cls}_i$ is $\mathtt{out}_i$, then unification $\mathtt{out}_i$:out succeeds with substitution s, where $\mathtt{out}_i$/s = out. The algorithm stops and returns (out, $\mathtt{cls}_i$/s).

2. *A test and contraction is encountered.* Computation performs the test and does one computation step. Driving splits configuration pgm $\mathtt{cls}_i$ into disjoint configurations $\{\mathtt{pgm}\,\mathtt{cls}_1, \ldots, \mathtt{pgm}\,\mathtt{cls}_n\}$ such that $|\mathtt{pgm}\,\mathtt{cls}_1| \cup \ldots \cup |\mathtt{pgm}\,\mathtt{cls}_n| = |\mathtt{pgm}\,\mathtt{cls}_i|$ and $|\mathtt{pgm}\,\mathtt{cls}_1| \cap \ldots \cap |\mathtt{pgm}\,\mathtt{cls}_n| = \emptyset$ ($n \geq 1$). Configuration pgm $\mathtt{cls}_j$, where $\mathtt{dat}_i \in |\mathtt{cls}_j|$, is selected and driven one step; all other configurations are ignored.

We shall not describe the details of the algorithm, only state two essential properties and illustrate the algorithm with an example.

Theorem 20 (correctness of NAN) [3] *Given* TSG*-program* pgm*, data* dat*, algorithm* NAN *computes an element of set* NAN(TSG,pgm,dat).

Theorem 21 (termination of NAN) [3] *For all* TSG*-programs* pgm*, all data* dat*, algorithm* NAN *terminates with* pgm, dat *iff* pgm dat $\stackrel{*}{\underset{\mathtt{TSG}}{\Rightarrow}}$ out *terminates.*

Example 4 *The result of the neighborhood analysis applied to the naive pattern matcher (Example 2) is shown in Fig. 4: the output* 'SUCCESS *of computation* match ["AB","XYZAB"] *and a class describing which part of input* ["AB","XYZAB"] *was used in the computation process. All fragments that are not relevant are overlined with variables. We can replace them by arbitrary atoms ($\mathcal{A}_i$) or arbitrary lists ($\mathcal{E}_i$) without changing the computation process or*

```
nan [match,
      [ (CONS 'A (CONS 'B 'NIL)),
        (CONS 'X (CONS 'Y (CONS 'Z (CONS 'A (CONS 'B 'NIL)))))
      ]
    ] =
    ( 'SUCCESS,                              -- Output of computation.
      ( [                                    -- Neighborhood of input.
            (CONS A1/'A (CONS A2/'B A3/'NIL)),
            (CONS A4/'X (CONS A5/'Y (CONS A6/'Z (CONS A1/'A (CONS A2/'B E7/'NIL)))))
        ],
        [A1 ≠ A4, A1 ≠ A5, A1 ≠ A6]
      )
    )
```

Figure 4: Neighborhood analysis: the input used in computation `match ["AB","XYZAB"]`.

the output. They pass through the 'machinery' in the same way. The run-time of the analysis was 1.4 secs.

5.4 Neighborhood Analysis in an Imperative Language

Instead of writing a new neighborhood analysis for the imperative language `Norma`, we show that the existing analysis can be ported to `Norma` using the same `Norma`-interpreter as in Sec. 4.4. Similar to inverse computation, a number of non-trivial imperative programs has been analyzed this way. The analysis has been applied to several other languages using the same interpreters as with `inv-mod`. Again, Theorem 19 guarantees the correctness all analysis results, but does not specify their quality. The following example shows that the nan-modifier produces non-trivial results for `Norma`.

Example 5 *The result of the neighborhood analysis for the Norma-program* `y = 2*x + 2` *(Fig. 3) and data* `x=0`*, represented by* `'NIL`*, is shown in Fig. 5: the output* `y` *of computation* `pgmNorma x` $\overset{*}{\underset{N}{\Rightarrow}}$ `y` *and neighborhood* $\texttt{cls}^{\texttt{[pgmNorma,x]}}$ *of* `[pgmNorma,x]`*. The neighborhood describes which part of program text* `pgmNorma` *and data* `x` *was used by the interpreter* `intNorma` *in the computation* `pgmNorma x` $\overset{*}{\underset{N}{\Rightarrow}}$ `y`*. We can replace the overlined fragments by arbitrary atoms or lists, respectively, without changing the interpretation process of the* `Norma`*-program or the output. Note that the analysis shows that a considerable part of the* `Norma`*-program was not used at all. The run-time of the analysis was 1.8 secs.*

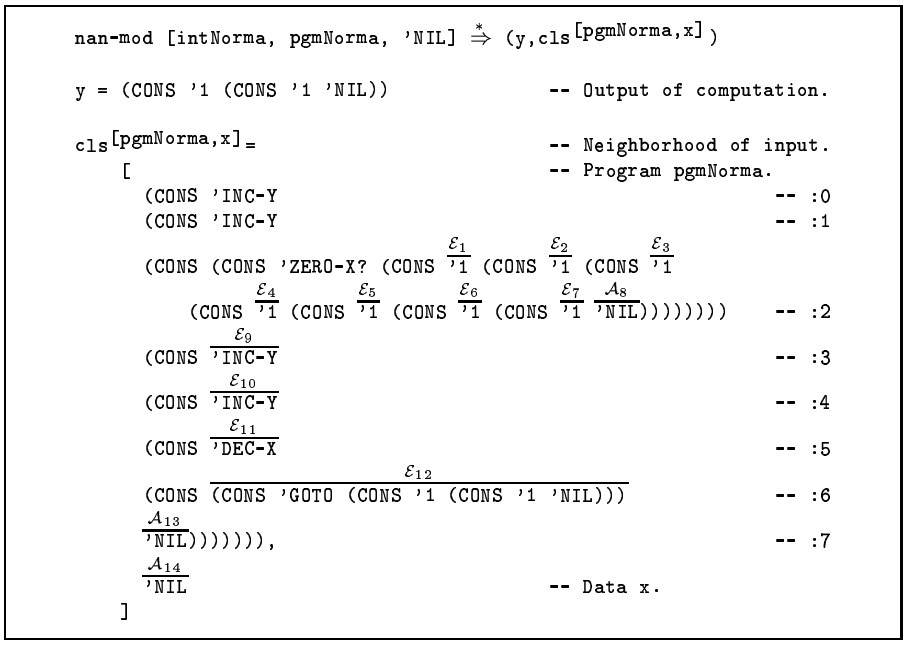

Figure 5: Nan-modifier: neighborhood analysis of a Norma program.

6 Program Transformation: Equivalence-Transformation Modifier

An equivalence transformer modifies the structure of a program with the purpose of optimizing some aspect of the programs performance while preserving the programs functionality. This covers a large class of transformers, including partial evaluation, partial deduction, deforestation and supercompilation.

Definition 22 (eqt-semantics) *Let* L,R *be programming languages,* pgm *be an* L*-program, and* cls *be a class, then the* equivalence-transformation semantics *is the set*

$$\texttt{EQT(L,R,pgm,cls)} \stackrel{\text{def}}{=}$$
$$\{\ \texttt{pgm'}\ \mid\ \texttt{pgm'} \in \texttt{PgmR},\ \textit{where}\ \texttt{pgm'}\ \textit{has argument list}\ \texttt{vars = vlist(cls)},$$
$$\forall \texttt{s} \in \texttt{FVS(cls)} : (\texttt{pgm (cls/s)} \stackrel{*}{\underset{\texttt{L}}{\Rightarrow}} \texttt{out}) \iff (\texttt{pgm' (vars/s)} \stackrel{*}{\underset{\texttt{R}}{\Rightarrow}} \texttt{out})\ \}.$$

Definition 23 (eqt-transformer) *An* M*-program* trafo *is an* L→R/M-equivalence transformer *if for all* L*-programs* pgm*, all classes* cls*, the transformer produces* R*-programs* pgm'*:*

$$\texttt{trafo [pgm, cls]} \stackrel{*}{\underset{\texttt{M}}{\Rightarrow}} \texttt{pgm'}\ \textit{and}\ \texttt{pgm'} \in \texttt{EQT(L,R,pgm,cls,out)}\ .$$

Definition 24 (eqt-modifier) *Given an* L→R/M*-equivalence transformer* trafo*, let* M*-program* eqt-mod *be defined as*

$$\texttt{eqt-mod [intN, pgm, cls]} \stackrel{\text{def}}{=} \texttt{trafo [intN, [pgm, cls]]} .$$

Theorem 25 (modifier property of eqt-mod**)** *Given language* R*,* eqt-mod *is an* L/M*-semantics modifier for* N/R*-eqt-semantics.*

Proof. *Program* eqt-mod *is an* L/M*-semantics modifier for* N/R*-eqt-semantics because for all* N/L*-interpreters, all* N*-programs* pgm*, and all classes* cls*, the* R*-program* pgm' *produced by*

$$\texttt{eqt-mod [intN, pgm, cls]} \stackrel{*}{\underset{M}{\Rightarrow}} \texttt{pgm'}$$

satisfies the condition for N/R*-eqt-semantics.* ∀s∈FVS(cls)*:*

$$\texttt{(pgm (cls/s)} \stackrel{*}{\underset{N}{\Rightarrow}} \texttt{out)} \iff \texttt{(pgm' (vars/s)} \stackrel{*}{\underset{R}{\Rightarrow}} \texttt{out)} .$$

Since pgm *is ground,* vars([pgm, cls]) = vars(cls)*,* FVS([pgm, cls]) = FVS(cls)*, and* eqt-mod [intN, pgm, cls] $\stackrel{\text{def}}{=}$ scp [intN, [pgm, cls]] *we have* ∀s∈FVS(cls)*:*

$$\begin{array}{ll} & \texttt{(pgm' (vars/s)} \stackrel{*}{\underset{R}{\Rightarrow}} \texttt{out)} \\ \stackrel{\text{(Def. 24)}}{\iff} & \texttt{(intN ([pgm, cls]/s)} \stackrel{*}{\underset{L}{\Rightarrow}} \texttt{out)} \\ \stackrel{\text{(pgm } gnd\text{)}}{\iff} & \texttt{(intN [pgm, cls/s]} \stackrel{*}{\underset{L}{\Rightarrow}} \texttt{out)} \\ \stackrel{\text{(Def. 3)}}{\iff} & \texttt{(pgm (cls/s)} \stackrel{*}{\underset{N}{\Rightarrow}} \texttt{out)} \end{array}$$

□

Two important types of equivalence transformers are program specializers and programs composers. For example, a program specializer implements the following case of eqt-semantics. The result of specializing a program pgm with respect to input x is a residual program that returns the same result when applied to the remaining input y as pgm applied to x,y.

Definition 26 (specializer) *An* M*-program* pe *is an* L→R/M-program specializer *if for all* L*-programs* pgm*, all input* x,y*, and binding-time classification SD:*

$$\texttt{(pgm [x, y]} \stackrel{*}{\underset{L}{\Rightarrow}} \texttt{out)} \iff \texttt{((pe [pgm}^{SD}\texttt{, x]) [y]} \stackrel{*}{\underset{R}{\Rightarrow}} \texttt{out)} .$$

Like all other equivalence transformers, as shown above, program specializers can be ported to new languages. Indeed, this is what the specializer projections[9] assert and what has been supported by several computer experiments[10]. This application is different from the well-known Futamura projections[7] which describe the compilation of programs, not their specialization via interpreters. We shall not repeat the experiments here. The eqt-semantics generalizes these results to the class of all equivalence transformers.

7 Interpretation: Identity Modifier

This sections shows that a well-known application, namely meta-interpreters, are special case of semantics modifiers.

Definition 27 (id-semantics) *Let* `L` *be a programming language,* `pgm` *be an* `L`*-program, and* `dat` *be input, then the* identity semantics *is the data*

$$\texttt{ID(L,pgm,dat)} \stackrel{\text{def}}{=} \texttt{out} \quad \textit{where} \quad \texttt{pgm dat} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out} \ .$$

Definition 28 (id-modifier) *Given an* `L/M`*-interpreter* `intL`*, let* `M`*-program* `id-mod` *be defined as*

$$\texttt{id-mod [intN, pgm, dat]} \stackrel{\text{def}}{=} \texttt{intL [intN, [pgm, dat]]} \ .$$

Theorem 29 (modifier property of `id-mod`) *Program* `id-mod` *is an* `L/M`*-semantics modifier for id-semantics.*

Proof. *It is easy to prove that* `id-mod` *is an* `L/M`*-semantics modifier for identity semantics because for all* `N/L`*-interpreters* `intN`*, all* `N`*-programs* `pgm`*, and all data* `dat`*:*

$$\begin{array}{ll}
& \texttt{(id-mod [intN, pgm, dat]} \stackrel{*}{\Rightarrow}_{\texttt{M}} \texttt{out)} \\
\stackrel{\text{(Def. 28)}}{\Longleftrightarrow} & \texttt{(intL [intN, [pgm, dat]]} \stackrel{*}{\Rightarrow}_{\texttt{M}} \texttt{out)} \\
\stackrel{\text{(Def. 3)}}{\Longleftrightarrow} & \texttt{(intN [pgm, dat]} \stackrel{*}{\Rightarrow}_{\texttt{L}} \texttt{out)} \\
\stackrel{\text{(Def. 3)}}{\Longleftrightarrow} & \texttt{(pgm dat} \stackrel{*}{\Rightarrow}_{\texttt{N}} \texttt{out)} \\
\stackrel{\text{(Def. 27)}}{\Longleftrightarrow} & \texttt{ID(N,pgm,dat)}
\end{array}$$

□

Although id-modifiers make no change to the semantics of a programming language, which may not seem very useful, they cover an interesting and well-known application: meta-interpreters for executing operational language definitions. A meta-interpreter is a program that takes a language definition (interpreter), a program in the defined language, and its data as input. Their role is not to change, but to preserve the semantics of the defined language. In other words, meta-interpreters port a *standard* semantics instead of a *non-standard* semantics. We have shown that they can be viewed as id-modifiers. Consequently, all what applies to semantics modifiers in general, applies to meta-interpreters in particular.

8 Projections for Non-Standard Semantics

A solution to the efficiency problem of semantics modifiers is to use existing methods for program transformation, in particular *program specialization*. Program specialization, or *partial evaluation*, has been applied successfully to

a wide range of transformation problems.[14] All residual programs produced by a program specializer are faithful to the original program, but are often significantly faster. Program specialization can collapse hierarchies of interpreters and remove their interpretive overhead by *automatic means*. Self-applicable specializers can convert ordinary programs into generating extensions.[7] Two applications of program specialization are discussed in this section: the generation of *non-standard interpreters* and *non-standard compilers*.

8.1 Specialization of Non-Standard Semantics

Let `sem-mod` be an `L/M`-semantics modifier for non-standard semantics S, `intN` an `N/L`-interpreter, `pgm` an `N`-program, and `req` the non-standard input. A specializer can be used in three ways to reduce the interpretive overhead. Let `pe` be an `M`→`M/M`-specializer in the first two cases, and an `L`→`L/L`-specializer in the last case (see Def. 26).

(a) *Non-standard interpreter*:

`pe[sem-mod`^{SDD}`, intN]` $\stackrel{*}{\Rightarrow}_{\mathtt{M}}$ `miN` where `miN[pgm, req]` $\stackrel{*}{\Rightarrow}_{\mathtt{M}}$ `answer`.

Program `miN` is a non-standard interpreter for `N`. The non-standard interpretation of `N`-programs via `intN` is avoided.

(b) *Non-standard compilation*:

`pe[sem-mod`^{SSD}`, intN, pgm]` $\stackrel{*}{\Rightarrow}_{\mathtt{M}}$ `mpN` where `mpN[req]` $\stackrel{*}{\Rightarrow}_{\mathtt{M}}$ `answer`.

Program `mpN` is the result of non-standard compilation from `N` into `M`. All standard/non-standard interpretation levels are removed.

(c) *Standard compilation*:

`pe[intN`^{SD}`, pgm]` $\stackrel{*}{\Rightarrow}_{\mathtt{L}}$ `tar` where `tar[dat]` $\stackrel{*}{\Rightarrow}_{\mathtt{L}}$ `out`.

Program `tar` is the result of standard compilation from `N` to `L`. The interpretive overhead of the semantics modifier is not removed, but `intN`'s interpretation level. The transformation of imperative programs into their relational semantics[17] is standard compilation. Here we shall not purse this approach further. It corresponds to the Futamura projections[7].

8.2 Projections for Non-Standard Semantics

By repeatedly applying the specializer `pe` to the initial equations in (a) and (b), we obtain seven *projections for non-standard semantics* (Fig. 6). We shall only comment the results; the correctness of the generated programs can be verified easily (omitted). The first three programs deal with non-standard interpretation while the last four program deal with non-standard compilation.

Non-standard interpretation:	
answer = sem-mod [intN,pgm,req]	
= pe[sem-mod^{SDD},intN] [pgm,req]	(i1)
= pe[pe^{SD},sem-mod^{SDD}] [intN] [pgm,req]	(i2)
= pe[pe^{SD},pe^{SD}] [sem-mod^{SDD}] [intN] [pgm,req]	(i3)
Non-standard compilation:	
answer = sem-mod [intN,pgm,req]	
= pe[sem-mod^{SSD},intN,pgm] [req]	(c1)
= pe[pe^{SSD},sem-mod^{SSD},intN] [pgm] [req]	(c2)
= pe[pe^{SSD},pe^{SSD},sem-mod^{SSD}] [intN] [pgm] [req]	(c3)
= pe[pe^{SSD},pe^{SSD},pe^{SSD}][sem-mod^{SSD}] [intN] [pgm] [req]	(c4)

Figure 6: Seven projections for non-standard semantics.

Non-standard interpretation tools:

(i1) *S-non-standard* N*-interpreter:*
miN = pe[sem-mod^{SDD},intN]

(i2) *Generator of S-non-standard interpreters:*
gmi = pe[pe^{SD},sem-mod^{SDD}]

(i3) *Generator of generators of non-standard interpreters:*
ggnsi = pe[pe^{SD},pe^{SD}]

Non-standard compilation tools:

(c1) *S-non-standard* N→M*-compilation:*
mpM = pe[sem-mod^{SSD},intN,pgm]

(c2) *S-non-standard* N→M*-compiler:*
mcN = pe[pe^{SSD},sem-mod^{SSD},intN]

(c3) *Generator of S-non-standard compilers:*
gmc = pe[pe^{SSD},pe^{SSD},sem-mod^{SSD}]

(c4) *Generator of generators of non-standard compilers:*
ggnsc = pe[pe^{SSD},pe^{SSD},pe^{SSD}]

The projections are different from the Futamura projections which describe compilation via interpreters. The projections for non-standard interpretation describe the generation of a non-standard interpreter by specializing a semantics modifier with respect to an interpreter. The projections for non-standard compilation describe the generation of a non-standard compiler by specializing a partial evaluator with respect to a semantics modifier and an interpreter.

Computer experiments for particular cases of the seven projections have been reported, e.g. specializer generation [10] is a special case of non-standard compilation and collapsing towers of interpreters [14] is a special case of non-standard interpretation. Thus there is evidence that the modifier projections can be put to work, but it is clear that future work will need to conduct more experiments to assess their potential and limitations in practice.

Our approach not only relates several applications of program specialization which previously seemed unrelated, such as the *specializer projections* [9] and the *inversion projections* [1], but may lead to new applications of program specialization. This is illustrated by the following example where program specialization is used to produce an inverted program.

Example 6 *Consider the semantics modifier* `inv-mod` *(Def. 12). According to the non-standard semantics projection (c1) the program* `invpM` *is the result of inverse compilation of* `pgm` *from* `N` *to* `M`*:* `invpM = pe [inv-mod`^{SSD}`, intN, pgm]` $\forall$`req`$\in$`Req: (invpM [req]` $\stackrel{*}{\underset{M}{\Rightarrow}}$ `answer)` $\Longleftrightarrow$ `(inv-mod [intN, pgm, req]` $\stackrel{*}{\underset{M}{\Rightarrow}}$ `answer`). *Thus inverse compilation implies two operations:*

1. *inversion of the function of program* `pgm`*;*
2. *representation of the inverted function by an* `M`*-program:* $\texttt{pgm}^{-1}$.

Non-standard compilation for arbitrary semantics modifiers `sem-mod`*:*

1. `sem-mod`*-modification of the function of source program;*
2. *representation of the* `sem-mod`*-modified function by an* `M`*-program.*

9 Conclusion

We defined a new class of programs, semantics modifiers, and showed that they exist for a rather large class of computational problems. A semantics modifier for inverse computation and neighborhood analysis was implemented, their properties were shown, and the algorithms were ported to other (simple) programming languages. Seven projections were given which promise an efficient and automatic implementation of non-standard tools by program specialization. The ultimate goal is the design of general and reusable "semantics components" for automatically implementing non-standard semantics without

the penalty of being too inefficient. We believe our results justify the further investigation of this approach. Several challenging problems lie ahead regarding the theoretical limitations and practical problems of semantics modifiers.

The seven projections are promising but they have not been implemented on a large scale. They may require further improvements of program specializers and related program transformers. Automatically generated non-standard programs may never be as efficient as handwritten programs, but they may be "good enough" in practice, in particular compared to the effort and investment required for developing individual non-standard program by hand.

Another challenging task is the development of structured methods for constructing semantics modifiers from a combination of existing 'atomic' modifiers. It would be worthwhile to answer the following questions:

- How to define the combination `(×)` of semantics modifiers?
- What does the combination `(eqt-mod×inv-mod)` etc. mean?
- Is it possible to define `(×)` so that it has the properties: `(id×mod=mod)`, `(mod×id=mod)`, etc.?

Acknowledgments

The authors are very grateful to Valentin Turchin for introducing them to metacomputation. The authors are thankful to the colleagues from the Moscow Refal group and PSI RAS – Andrei Klimov, Sergei Romanenko, Andrei Nemytukh, Luba Pozlevich – for stimulating and fruitful discussions. The first author was partially supported by the *United States Civilian Research and Development Foundation for the Former Soviet Union* under CRDF grant #RM1-254. The second author was partially supported by the project "Design, Analysis and Reasoning about Tools" funded by the *Danish Natural Sciences Research Council.*

References

1. S.M. Abramov. Metavychislinija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie* 3:31–44, in Russian (1991).
2. S.M. Abramov. Metacomputation and program testing. In *1st International Workshop on Automated and Algorithmic Debugging*. 121–135 (Linköping, Sweden, 1993).
3. S.M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications)*. 128 pp., in Russian (Nauka-Fizmatlit, Moscow, 1995).
4. S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages* (Ellis Horwood, 1987).
5. J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence* 16(1):1–46 (1981).

6. A.P. Ershov. On the essence of compilation. In E.J. Neuhold, editor. *Formal Description of Programming Concepts*. 391–420 (North-Holland, 1978).
7. Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls* 2(5):45–50 (1971).
8. R. Glück and And. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot *et al.*, editors. *Static Analysis. Proceedings*. LNCS, Vol. 724, 112–123 (Springer-Verlag, 1993).
9. R. Glück. On the generation of specializers. *Journal of Functional Programming* 4(4):499–514 (1994).
10. R. Glück and J. Jørgensen. Generating optimizing specializers. *IEEE International Conference on Computer Languages*. 183–194 (IEEE Computer Society Press, 1994).
11. P.G. Harrison. Function inversion. In D. Bjørner *et al.*, editors. *Partial Evaluation and Mixed Computation*. 153–166 (North-Holland, 1988).
12. P. Hudak *et al.* Report on the Programming Language Haskell. *SIGPLAN Notices* 27(5):R1–R164, 1992.
13. N.D. Jones, P. Sestoft and H, Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor. *Rewriting Techniques and Applications*. LNCS, Vol. 202, 124–140 (Springer-Verlag, 1985).
14. N.D. Jones, C.K. Gomard and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. (Prentice Hall, 1993).
15. A.Yu. Romanenko. The generation of inverse functions in Refal. In D. Bjørner *et al.*, editors. *Partial Evaluation and Mixed Computation*. 427–444 (North-Holland, 1988).
16. A.Yu. Romanenko. Inversion and metacomputation. *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. 12–22 (ACM Press, 1991).
17. B.J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348, 1997.
18. P. Thiemann and M. Sperber. Polyvariant expansion and compiler generators. In D. Bjørner *et al.*, editors. *Perspectives of System Informatics. Proceedings*. LNCS, Vol. 1181, 285–296 (Springer-Verlag, 1996).
19. V.F. Turchin. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal). In *Teorija Jazykov i Metody Programmirovanija*. 31–42, in Russian (Kiev–Alushta, USSR, 1972).
20. V.F. Turchin. The concept of a supercompiler. *ACM TOPLAS* 8(3):292–325 (1986).
21. V.F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner *et al.*, editors. *Partial Evaluation and Mixed Computation*. 341–353 (North-Holland, 1988).
22. V.F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming* 3(3):283–313 (1993).