International Journal of Foundations of Computer Science Vol. 12 No. 2 (2001) 171–211
© World Scientific Publishing Company

FROM STANDARD TO NON-STANDARD SEMANTICS BY SEMANTICS MODIFIERS

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

and

ROBERT GLÜCK*
Department of Information and Computer Science
School of Science and Engineering, Waseda University
Shinjuku-ku, Tokyo 169-8555, Japan
E-mail: glueck@acm.org

Received 21 January 1999
Revised 29 May 1999
Communicated by M. Sato and Y. Toyama

ABSTRACT

An approach for systematically modifying the semantics of programming languages by *semantics modifiers* is described. Semantics modifiers are a class of programs that allow the development of general and reusable "semantics components". Language independence is achieved through the interpretive approach: an interpreter serves as a mediator between the new language and the language for which the non-standard semantics was implemented. Inverse computation, equivalence transformation and neighborhood analysis are shown to be semantics modifiers. Experiments with these modifiers show the computational feasibility of this approach. Seven modifier projections are given which allow the efficient implementation of non-standard interpreters and non-standard compilers by program specialization or other powerful program transformation methods.

Keywords: programming languages, program transformation, non-standard semantics, partial evaluation, supercompilation, interpreters.

1. Introduction

Interpreters are a convenient way to implement the *standard semantics* of programming languages. But interpreters can be used for more than just running programs. Combined with a semantics modifier, a standard interpreter can be used to port a *non-standard semantics* from one programming language to another.

*Supported by an Invitation Fellowship of the Japan Society for the Promotion of Science (JSPS) under grant no. L99530 On leave from the Department of Computer Science (DIKU), University of Copenhagen, Denmark.

Programs which are, together with a standard interpreter, capable of this computational feat are called *semantics modifiers*. This is different from other forms of program reuse: algorithms are reused by means of interpreters. Language independence is achieved through the interpretive approach: an interpreter serves as a mediator between the new language and the language for which the semantics modifier was implemented. This allows the development of general and reusable "semantics components". Efficient implementations of new programming tools, non-standard interpreters and non-standard compilers, can be obtained automatically by program specialization or other powerful program transformation methods.

For example, *inverse computation* is a non-standard semantics which allows to run programs backwards. Examining the input and output behavior of programs is a powerful analytical tool (e.g., find all states leading to a given critical state). Assume an algorithm for inverse computation of L-programs exists. Given the task to perform inverse computation of N-programs, one will usually attempt to construct a new algorithm for N (e.g., after examining the existing algorithm for L). This will be far from trivial, likely to be error-prone and time-consuming.

We show that the problem of performing inverse computation in N can be reduced to the problem of writing a standard interpreter for N in L, and then porting the existing algorithm from L to N using the standard interpreter. This is possible because inverse computation has the property of being a semantics modifier.

A possible application of semantics modifiers are domain-specific languages. These languages make it easy to express non-trivial computational tasks in a particular application domain (e.g., networks, graphics), but they are expensive to implement when each language requires the development of a new set of programming tools. Semantics modifiers are an inexpensive alternative to the individual construction of such tools (e.g., inverse interpreters, program transformers).

What we have in mind is to establish a set of semantics modifiers, to show their correctness once, and then to reuse them for different programming languages. This promises language independence that could drastically increase the applicability of non-standard methods.

Our results show that semantics modifiers exist for a remarkably wide class of computational problems, including inverse computation, program transformation, and program analysis. In fact, the well-known tower of interpreters, which is often used for porting programming languages from one platform to another, is just a special case in our framework (an *identity modifier*). From the results described in this paper we can see that meta-programming by interpreters is far more powerful than previously thought.

1.1. This paper

The aim of this paper is to develop the underlying theory of semantics modifiers, to identify several non-trivial semantics that have the modifier property, and to give experimental evidence for the viability of our approach.

Specifically, our contributions in this paper are as follows:

- First, we introduce *semantics modifiers* and develop the underlying theory (Sec. 3).
- Second, we present several *non-standard semantics* and prove that each semantics has the modifiers property (Sec. 4–7).
- Third, we put the theoretical ideas on trial by implementing the semantics modifiers using *non-trivial algorithms* and examining several *computer experiments* that modify the semantics of a small imperative language (Sec. 8–12).
- Fourth, we give seven *modifier projections* that show how efficient implementations of non-standard interpreters and non-standard compilers can be obtained by *program specialization* (Sec. 13).

This paper extends our earlier work [4] and draws upon results of [3] and [22, 26]. We assume familiarity with the basic notions of partial evaluation (a good source is the book [31]).

2. Preliminaries

This section defines the basic notions used in this paper, in particular, data domains and their set representation, programming languages and interpreters.

2.1. A Universal Data Domain

We assume a *universal data domain*, *Dat*, for all programming languages and to represent programs in different languages. Letting programs and data have the same form is convenient when dealing with programs as data objects.

A suitable choice of *Dat* is the set of S-expressions known from Lisp. S-expressions are defined by the grammar

$$Dat = Atom \mid (\text{CONS } Dat\ Dat)$$

where *Atom* is a possibly infinite set of symbols, and CONS is a constructor. The definition says that an element of *Dat* is either an atom or has the form (CONS d_1 d_2) for $d_1, d_2 \in Dat$. The data and program domains of any programming language can be mapped to *Dat*. For example, a list $[d_1, ..., d_n]$ can be represented by (CONS d_1 (CONS d_2...(CONS d_n 'NIL)...)) for $n \geq 0$ and all $d_i \in Dat$. These mappings are usually straightforward; we shall not discuss them further.

2.2. Representing Sets of Data

To represent subsets of domain *Dat*, we use *expressions with variables*, or short *classes*. A class is a constructive representation of a possibly infinite set of data. Several of the semantics modifiers in this paper can be defined using this set representation. This has the advantage that we can avoid introducing different notations.

For the experiments in this paper, we use expressions that contain two types of variables: $\mathcal{A}_i$ and $\mathcal{E}_i$ variables, where $\mathcal{A}_i$ ranges over *Atom* and $\mathcal{E}_i$ ranges over *Dat* (the indices are used to distinguish between different variables of the same type, $i \geq 0$). Expressions are defined by the grammar

$$Exp = Atom \mid \mathcal{A}_i \mid \mathcal{E}_i \mid (\text{CONS } Exp\ Exp)$$

This definition shows that *Exp* is an extension of *Dat*, and to represent elements of *Exp* in *Dat* we need, again, a mapping from *Exp* to *Dat* (not shown in this paper). We use $\text{var}(exp)$ to denote the set of all variables occurring in $exp \in Exp$, and $\text{varlist}(exp)$ to denote a list of these variables. An expression without variables is called *ground*.

The meaning of an expression *exp* is the set of data represented by that expression. We define this more formally using substitutions.

Definition 1 (substitution) *A* substitution $s = [x_1 \mapsto exp_1, \ldots, x_n \mapsto exp_n]$ *is a list of variable-expression pairs such that variables x_i are pairwise distinct and exp_i are expressions. The result of applying s to expression exp, exp/s, is an expression obtained by replacing every occurrence of x_i by exp_i in exp, for every $x_i \mapsto exp_i$ in s. Domain* $\text{dom}(s)$ *denotes set $\{x_1, ..., x_n\}$, and s is* ground *if all exp_i are ground.*

Definition 2 (full valid substitution) *A* full valid substitution $\text{fvs}(exp)$ *is the set of all ground substitutions for expression exp such that for all* $s \in \text{fvs}(exp)$, $\text{var}(exp) \subseteq \text{dom}(s)$ *and s does not violate the domains of the variables in* $\text{var}(exp)$.

Definition 3 (class, concretization) *A* class *cls is an expression with variables. The set of data represented by class cls, denoted $\lceil cls \rceil$, is defined by*

$$\lceil cls \rceil \stackrel{\text{def}}{=} \{dat \mid s \in \text{fvs}(cls), dat = cls/s\} \ .$$

Example 1 Consider a class $cls = $ (CONS 'Z (CONS $\mathcal{A}_3$ $\mathcal{E}_7$)). Class *cls* is non-ground because it contains variables $\mathcal{A}_3$ and $\mathcal{E}_7$, while class *cls/s* obtained by applying substitution $s = [\mathcal{A}_3 \mapsto$ 'A, $\mathcal{E}_7 \mapsto$ 'B] to *cls* is ground. Finally, *cls* represents the set of data $\lceil cls \rceil = \{$(CONS 'Z (CONS a d)) $\mid a \in Atom, d \in Dat\}$. □

2.3. Programming Languages

We use the following definitions of programming languages, computation and computational equivalence.

Definition 4 (programming language) [19] 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 (set of input/output data), and SemL is the semantics of L (a partial function $SemL : [PgmL, DatL] \to DatL$).*

Definition 5 (computation) *Let $pgm \in PgmL$ be an L-program, $dat \in DatL$ be data, and $out \in DatL$ be the output of applying pgm to dat in language L, $out = SemL\ [pgm, dat]$, then we denote* computation in *L by* $\underset{L}{\overset{*}{\Rightarrow}}$*, and write* $(pgm\ dat \underset{L}{\overset{*}{\Rightarrow}} out)$.

Definition 6 (computational equivalence) *Two computations $\mathcal{A}, \mathcal{B}$ are* computationally equivalent*, $\mathcal{A} \Longleftrightarrow \mathcal{B}$, if they compute the same output, or both computations are undefined.*

As customary when dealing with programs as data objects [18, 31], we avoid extra mappings between different data domains by using the same universal data domain for all programming languages *L* and for representing all *L*-programs (i.e., for all programming languages *L*, $DatL \subseteq Dat$ and $PgmL \subseteq Dat$). These mappings are not essential for our discussion. We say, *SemL* is the *standard semantics* of *L*. We consider only deterministic programming languages in this paper.

Definition 7 (interpreter) *Let L, M be programming languages, then an M-program intL is an L/M-*interpreter *iff for all pgm* $\in$ *PgmL, and for all dat* $\in$ *DatL,*

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

An interpreter is an implementation of a programming language. It follows from this definition that a programming language can be implemented by two (or more) layers of interpretation. Let *intL* be an *L*/*M*-interpreter and let *intN* be an *N*/*L*-interpreter. Then any *N*-program *pgm* can be computed by a tower of two interpreters:

$$(intL\ [intN, [pgm, dat]] \underset{M}{\overset{*}{\Rightarrow}} out) \Longleftrightarrow (pgm\ dat \underset{N}{\overset{*}{\Rightarrow}} out)\ .$$

3. Semantics Modifiers

In this section we introduce the notion of a *semantics modifier.* We show how, combined with a standard interpreter, a semantics modifier can be used to change the semantics of a programming language. This is the key to the portability of non-standard semantics.

3.1. Non-Standard Semantics

When we speak of a *non-standard S-semantics* of a language *L*, then the relation of input/output of all *L*-programs executed under the given non-standard *S*-semantics must satisfy a certain *property S.* For example, we require that the result of inverse computation of an *L*-program represents the possible input of that program under standard computation. Any semantics which satisfies this property is then called an inversion semantics for *L*. When we talk about input and output of non-standard computation, we usually use the terms *request* and *answer* to distinguish them from standard computation.

Definition 8 (non-standard dialect, non-standard semantics) *Let L, L′ be programming languages and S be a property, then L′ is a* non-standard *S*-dialect of *L iff* $L = (PgmL, DatL, SemL)$, $L' = (PgmL, DatL, SemL')$ *and for all pgm* $\in$ *PgmL and for all req* $\in$ *DatL, result ans of L′-computation* $pgm\ req \underset{L'}{\overset{*}{\Rightarrow}} ans$ *satisfies property S. Semantics SemL′ is a* non-standard *S*-semantics of *L.*

Definition 9 (non-standard interpreter) *Let L, M be programming languages and S be a property, then an M-program intL′ is a* non-standard *S*-interpreter for *L* in *M iff there exists an S-dialect L′ of L such that intL′ is an L′/M-interpreter.*

A non-standard *S*-dialect *L′* of *L* is a programming language whose semantics is defined by a non-standard *S*-semantics *SemL′*. An *S*-dialect *L′* is not always uniquely determined by a property *S.* In fact, there may exist infinitely many different *S*-dialects for a language *L.* For example, given a request for inverse computation of an *L*-program, there may be infinitely many answers that satisfy the inversion property. Each *S*-dialect may return a different answer for the same request, and each answer will be correct wrt property *S.*

We say, an interpreter for a language L is an implementation of the *standard semantics* of L, while a non-standard S-interpreter is an implementation of a *non-standard semantics* of L. There is one standard semantics of L, but there may be infinitely many non-standard S-semantics of L. A non-standard interpreter is the implementation of one of them.

We use the term non-standard semantics in a broad sense including any simulation, analysis or transformation of L-programs that can be specified by a property S.

3.2. Semantics Modifiers

Semantics modifiers take non-standard semantics one step further. Combined with a standard interpreter, a semantics modifier can be used to port a non-standard semantics from one programming language to another. The effect of non-standard computation in a new language N can be achieved by applying an L/M-semantics modifier *sem-modL* to a standard N/L-interpreter *intN* (illustrated in Fig. 1).

Definition 10 (semantics modifier) *Let L, M be programming languages, S be a property, and $\mathcal{N}$ be a set of programming languages, then an M-program sem-modL is an L/M-*semantics modifier for S in $\mathcal{N}$ *iff for all $N \in \mathcal{N}$, for all N/L-interpreters intN, for all pgm $\in$ PgmN and for all req $\in$ DatN, program intN$'$ is a non-standard S-interpreter for N in M, where*

$$intN' \ [pgm, req] \overset{\text{def}}{=} sem\text{-}modL \ [intN, pgm, req] \ .$$

Lemma 1 (modifier condition) *Let L, M be programming languages, S be a property, $\mathcal{N}$ be a set of programming languages, and sem-modL be an M-program. If for all $N \in \mathcal{N}$, for all N/L-interpreters intN, for all pgm $\in$ PgmN and for all req $\in$ DatN, result ans $\in$ DatM of computation*

$$sem\text{-}modL \ [intN, pgm, req] \underset{M}{\overset{*}{\Rightarrow}} ans$$

satisfies property S, then sem-modL is an L/M-semantics modifier for S in $\mathcal{N}$.

Proof. Consider any $N \in \mathcal{N}$ and any N/L-interpreter *intN*. We prove that *intN$'$* is a non-standard S-interpreter for N in M where

$$intN' \ [pgm, req] \overset{\text{def}}{=} sem\text{-}modL \ [intN, pgm, req] \ .$$

Define language N' by $N' = (PgmN, DatN, SemN')$ where for all $pgm \in PgmN$ and for all $req \in DatN$ semantics $SemN'$ is defined such that

$$(SemN' \ [pgm, req] = out) \Longleftrightarrow (sem\text{-}modL \ [intN, pgm, req] \underset{M}{\overset{*}{\Rightarrow}} out) \ .$$

We have (i) N' is an non-standard S-dialect of N (according to the definition of $SemN'$, the conditions of the lemma, and Def. 8), and (ii) *intN$'$* is a non-standard S-interpreter of N (according to the definitions of $SemN'$ and *intN$'$*, and Def. 9). According to Def. 10, *sem-modL is an L/M-semantics modifier for S in $\mathcal{N}$.*

□

Lemma 1 is convenient when proving that a program is a semantics modifier. It will be used later in several proofs.

$$\begin{tikzcd}
\mathit{sem\text{-}modL} & {[\mathit{intN}, \mathit{pgm}, \mathit{req}]} \overset{*}{\underset{M}{\Rightarrow}} & \mathit{ans} \\
\textit{semantics modifier} \arrow[u] & \textit{standard semantics} \arrow[u] & \textit{non-standard result} \arrow[u]
\end{tikzcd}$$

Fig. 1. From standard to non-standard semantics by semantics modifiers.

3.3. Building Non-Standard Interpreters

One of the striking features of semantics modifiers is the possibility to port non-standard semantics from one language to another at the price of writing an interpreter. It is straightforward to define a non-standard interpreter for any language given a semantics modifier and a standard interpreter for that language.

Let *sem-modL* be an L/M-modifier for S and let *intN* be an N/L-interpreter, then a non-standard N'/M-interpreter *nsintN* for N can be defined by

$$\mathit{nsintN}\ [\mathit{pgm}, \mathit{req}] \overset{\text{def}}{=} \mathit{sem\text{-}modL}\ [\mathit{intN}, \mathit{pgm}, \mathit{req}]$$

which performs non-standard computation according to S:

$$(\mathit{nsintN}\ [\mathit{pgm}, \mathit{req}] \overset{*}{\underset{M}{\Rightarrow}} \mathit{ans}) \Longleftrightarrow (\mathit{pgm}\ \mathit{req} \overset{*}{\underset{N'}{\Rightarrow}} \mathit{ans}) \ .$$

The correctness of the non-standard interpreter *nsintN* is guaranteed (i) by the correctness of the semantics modifier *sem-modL* (which has to be established only once) and (ii) by the correctness of the standard interpreter *intN* (which is usually easier to show than the correctness of a non-standard interpreter). For example, given a semantics modifier for inverse computation, we can define an inverse interpreter for N using a standard interpreter *intN* for N (see Fig. 2 for more examples).

The definition of *nsintN* shows that we can build different non-standard interpreters for N by combining the semantics modifier with different N/L-interpreters. Regardless of which interpreter we use, the semantics modification we get is precisely what is intended (e.g., inverse computation).

3.4. Two Levels of Interpretation

A non-standard interpreter obtained from a semantics modifier is the result of two distinct metasystem transitions [51].

(0) Direct computation of N-programs:

$$\mathit{pgm}\ \mathit{dat} \overset{*}{\underset{N}{\Rightarrow}} \mathit{out}.$$

(1) First transition from direct computation of N-programs to their interpretation:

$$\mathit{intN}\ [\mathit{pgm}, \mathit{dat}] \overset{*}{\underset{L}{\Rightarrow}} \mathit{out}$$

(2) Second transition from direct computation of N's interpretation to non-standard computation of N's interpretation in M:

$$\mathit{sem\text{-}modL}\ [\mathit{intN}, \mathit{pgm}, \mathit{req}] \overset{*}{\underset{M}{\Rightarrow}} \mathit{ans}$$

Let *intNL* be an N/L-interpreter,

$$\begin{array}{ll} \mathit{intNL}\ [\mathit{pgm}, \mathit{dat}] \overset{*}{\underset{L}{\Rightarrow}} \mathit{out} & \text{– \textit{standard interpreter for N in L}} \end{array}$$

define

$$\begin{array}{rll}
\mathit{intNM}\ [\mathit{pgm}, \mathit{dat}] & \overset{\text{def}}{=} \mathit{id\text{-}modL}\ [\mathit{intNL}, \mathit{pgm}, \mathit{dat}] & \text{– \textit{ID-modifier for L (Sec. 4)}} \\
\mathit{invNM}\ [\mathit{pgm}, \mathit{req}] & \overset{\text{def}}{=} \mathit{inv\text{-}modL}\ [\mathit{intNL}, \mathit{pgm}, \mathit{req}] & \text{– \textit{INV-modifier for L (Sec. 5)}} \\
\mathit{eqtNM}\ [\mathit{pgm}, \mathit{cls}] & \overset{\text{def}}{=} \mathit{eqt\text{-}modL}\ [\mathit{intNL}, \mathit{pgm}, \mathit{cls}] & \text{– \textit{EQT-modifier for L (Sec. 6)}} \\
\mathit{nanNM}\ [\mathit{pgm}, \mathit{dat}] & \overset{\text{def}}{=} \mathit{nan\text{-}modL}\ [\mathit{intNL}, \mathit{pgm}, \mathit{dat}] & \text{– \textit{NAN-modifier for L (Sec. 7)}}
\end{array}$$

then

$$\begin{array}{rll}
\mathit{intNM}\ [\mathit{pgm}, \mathit{dat}] & \overset{*}{\underset{M}{\Rightarrow}} \mathit{out} & \text{– \textit{standard interpreter for N}} \\
\mathit{invNM}\ [\mathit{pgm}, \mathit{req}] & \overset{*}{\underset{M}{\Rightarrow}} \mathit{ans} & \text{– \textit{inverse interpreter for N}} \\
\mathit{eqtNM}\ [\mathit{pgm}, \mathit{cls}] & \overset{*}{\underset{M}{\Rightarrow}} \mathit{pgm}' & \text{– \textit{equivalence transformer for N}} \\
\mathit{nanNM}\ [\mathit{pgm}, \mathit{dat}] & \overset{*}{\underset{M}{\Rightarrow}} (\mathit{cls}^{[\mathit{pgm}, \mathit{dat}]}, \mathit{out}) & \text{– \textit{neighborhood analyzer for N}}
\end{array}$$

Fig. 2. Semantics modifier + standard interpreter = non-standard interpreter.

It is clear that each transition ($N \to L \to M$) adds extra computational overhead. To improve the efficiency of non-standard computation we shall apply program specialization. Such an optimization is very important for building practical non-standard tools and we examine different ways of optimization in Section 13.

3.5. Observations about the Source Language

In general, we are interested in establishing semantics modifiers for the set $\mathcal{P}$ of *all* programming languages. In fact, all modifiers we are going to present in this paper are valid for $\mathcal{P}$. We take this as starting point for some considerations about the source language L of a semantics modifier for $\mathcal{P}$.

In particular, $\mathcal{P}$ includes the set of universal languages. A programming language is *universal* if it is sufficiently powerful to specify all computable functions. Consequently, a universal language permits to simulate any computation in any programming language. Among others, we can write an interpreter for the language in itself (called a self-interpreter).

What can we conclude about L, the source language of a semantics modifier from the fact that the modifier is valid for $\mathcal{P}$? First, in order to write an interpreter for every programming language in $\mathcal{P}$, source language L must be universal. Second, it does not matter which universal language we choose because every universal language is powerful enough to write an interpreter for every languages in $\mathcal{P}$. Third, it is possible to write a self-interpreter for L in L.

The latter observation implies that we can always provide a non-standard semantics to L-programs by means of a self-interpreter. Let *sem-modL* be an L/M-semantics modifier for S in $\mathcal{P}$, *sint* be a self-interpreter for L, and *pgm* be an L-program, then we have

$$\mathit{sem\text{-}modL}\ [\mathit{sint}, \mathit{pgm}, \mathit{req}] \overset{*}{\underset{M}{\Rightarrow}} \mathit{ans}\ .$$

3.6. Existence of Semantics Modifiers

So far we only specified the class of semantics modifiers, but we have not touched upon the question whether semantics modifiers exist, and if so, whether there are any 'interesting' semantics modifiers?

The first question can be answered directly. Semantics modifiers exist, at least for 'nonsensical' semantics. Let us illustrate this with an example.

A language L' is an *S10*-dialect of L iff for all L-programs *pgm* and for all data *req*, the result of L'-computation is always 10, and we call this property *S10*:

$$\mathit{pgm}\ \mathit{req} \underset{L'}{\overset{*}{\Rightarrow}} 10 \ . \qquad (\textit{Property S10})$$

The definition of a semantics modifier for *S10* in $\mathcal{P}$ is straightforward. Regardless of interpreter *intN*, program *pgm*, and input *req*, the result is a fixed value:

$$\textit{S10-modL}\ [\mathit{intN}, \mathit{pgm}, \mathit{req}] \overset{\text{def}}{=} 10 \ .$$

A non-standard interpreter *S10-intN* for an *S10*-dialect of N can be defined by

$$\textit{S10-intN}\ [\mathit{pgm}, \mathit{req}] \overset{\text{def}}{=} \textit{S10-modL}\ [\mathit{intN}, \mathit{pgm}, \mathit{req}] \ .$$

An answer to the second question is presented in the next sections. More specifically, we show that several non-trivial semantics satisfy the requirements for semantics modifiers.

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

In each case, we specify a non-standard semantics and then prove that the non-standard semantics, if a constructive definition exists, satisfies the requirements for semantics modifiers. Later we show that constructive definitions exist and demonstrate the algorithms by several computer experiments.

4. Standard Computation: Identity Modifier

Using interpreters to easily port programming languages from one machine to another is a familiar approach to programming language development [18]. The role of the interpreters is not to change, but to preserve the semantics of a programming language across different platforms.

The essence of this method, which allows to port N-programs to any M-machine that can execute L-programs, is to combine two language definitions, namely the definition of N in L and the definition of L in M. The big advantage is that once the semantics of N is defined by an N/L-interpreter, N-programs can be executed on any M-machine that can execute L-programs. In other words, given the standard semantics of L, we can give standard semantics to N. This can be regarded as the identity operation in the algebra of semantics modifiers (an *identity modifier*).

4.1. A Modifier for Identity Semantics

To show that a tower of two interpreters falls into the class of semantics modifiers, we introduce the notion of an identity dialect of a language, and then define a semantics modifier for identity semantics.

Definition 11 (*ID*-dialect) *Let L be a programming language, then L' is an ID-dialect of L iff for all $pgm \in PgmL$ and for all $dat \in DatL$, the result of computation $(pgm\ dat \overset{*}{\underset{L'}{\Rightarrow}} out)$ is $out \in DatL$ such that*

$$(pgm\ dat \overset{*}{\underset{L'}{\Rightarrow}} out) \Longleftrightarrow (pgm\ dat \overset{*}{\underset{L}{\Rightarrow}} out)\ . \qquad (Property\ ID)$$

Definition 12 (*ID*-modifier) *Given an L/M-interpreter intL, let M-program id-modL be defined as*

$$id\text{-}modL\ [intN, pgm, dat] \overset{\text{def}}{=} intL\ [intN, [pgm, dat]]\ .$$

Theorem 1 (modifier property of *id-modL*) *Let $\mathcal{P}$ be the set of all programming languages, then id-modL is an L/M-semantics modifier for ID in $\mathcal{P}$.*

Proof. Program *id-modL* is an L/M-semantics modifier for *ID*-semantics (according to Lemma 1) because for all $N \in \mathcal{P}$, for all N/L-interpreters *intN*, for all N-programs *pgm*, and for all data *dat*, the result *out* produced by computation

$$id\text{-}modL\ [intN, pgm, dat] \overset{*}{\underset{M}{\Rightarrow}} out$$

satisfies Property *ID* (Def. 11):

$$\begin{aligned} & (id\text{-}modL\ [intN, pgm, dat] \overset{*}{\underset{M}{\Rightarrow}} out) \\ \overset{\text{(Def. 12)}}{\Longleftrightarrow}\ & (intL\ [intN, [pgm, dat]] \overset{*}{\underset{M}{\Rightarrow}} out) \\ \overset{\text{(Def. 7)}}{\Longleftrightarrow}\ & (intN\ [pgm, dat] \overset{*}{\underset{L}{\Rightarrow}} out) \\ \overset{\text{(Def. 7)}}{\Longleftrightarrow}\ & (pgm\ dat \overset{*}{\underset{N}{\Rightarrow}} out) \end{aligned}$$

□

The identity-modifier is a program that takes three inputs: an N/L-interpreter *intN*, an N-program *pgm* and data *dat*. It computes the output of *pgm* by interpreting *intN* using L-interpreter *intL*. In other words, the familiar approach of porting programming languages by means of interpreters is nothing but an instance of semantics modification (the identity modification).

This covers also the case of meta-interpreters which are popular for executing operational language definitions and prototyping languages. Such a meta-interpreter is nothing but an M-program that takes the definition of a language N given in a definition language L, an N-program and the corresponding data as input, and gives meaning to the N-program. Again, the role of the meta-interpreter is not to change, but to preserve the semantics of the defined language.

This means that all what applies to semantics modifiers in general, applies to interpreter towers and meta-interpreters in particular.

5. Non-Standard Computation: Inversion Modifier

While standard computation is the calculation of the output of a program for a given input ('forward execution'), inverse computation is the calculation of the possible input of a program for a given output ('backward execution'). Advances in this direction have been achieved in the area of logic programming (e.g., [36, 47]), based on solutions emerging from logic and proof theory. Inverse computation is an important and useful concept in many areas. Typical applications range from expert systems and artificial intelligence to model checking and program verification (cf. [6, 47]).

In this section we show that the problem of performing inverse computation in a language N can be reduced to the problem of constructing a *standard interpreter* for N written in L provided an algorithm for inverse computation in L is given.

5.1. A Modifier for Inverse Computation

We introduce the notion of an inverse dialect of a language and then, based on that notion, define an inverse interpreter for that language. Finally, we show that inverse computation is indeed a semantics modifier for any programming language.

Definition 13 (*INV*-dialect) *Let L be a programming language, then L' is a* non-standard *INV*-dialect of L *iff for all $pgm \in PgmL$, for all $out \in DatL$, and for all classes cls, the result of computation ($pgm\ [cls, out] \underset{L'}{\overset{*}{\Rightarrow}} ans$) is a (possibly infinite) list $ans = [s_1, \ldots]$ of substitutions that satisfies the following condition:*

$$\bigcup_i \lceil cls/s_i \rceil = INV(L, pgm, cls, out) \qquad (Property\ INV)$$

$$where\ INV(L, pgm, cls, out) \stackrel{\text{def}}{=} \{dat \mid dat \in \lceil cls \rceil,\ pgm\ dat \underset{L}{\overset{*}{\Rightarrow}} out\}\ .$$

Definition 14 (*INV*-interpreter) *Let L, M be programming languages, then an M-program inv-intL is an* inverse interpreter *for L in M iff there exists an INV-dialect L' of L such that inv-intL is an L'/M-interpreter. For inverse computation using inv-intL, we write ($inv\text{-}intL\ [pgm, [cls, out]] \underset{M}{\overset{*}{\Rightarrow}} ans$).*

We say list $[cls, out]$ is a *request* for inverse computation where class *cls* specifies the set of admissible input (the search space), and *out* is the given output. The set $INV(L, pgm, cls, out)$ is the *solution* of the given inversion problem. It is the largest subset of $\lceil cls \rceil$ such that $pgm\ dat \underset{L}{\overset{*}{\Rightarrow}} out$ for all elements *dat* of the solution.[a]

The computation of a constructive representation of $INV(L, pgm, cls, out)$ is called *inverse computation*. Given request $[cls, out]$, an inverse interpreter performs inverse computation of a program *pgm*. The enumeration $[cls/s_1, \ldots]$ is a representation of set $INV(L, pgm, cls, out)$. Each substitution, if it exists, can be computed in finite time, but termination of the inverse interpreter cannot be guaranteed, even with sophisticated strategies (in general, it is undecidable whether all substitutions have been found).

Let *inv-intL* be an inverse interpreter for language L, then inverse computation in a new language N can be performed by inverting an N-interpreter.

[a]Here we defined the universal solution (all possible input) as answer to the given inversion problem. Existential solutions (one possible solution) can be obtained by picking an arbitrary element from the universal solution.

Definition 15 (*INV*-modifier) *Given an inverse interpreter inv-intL for L in M, let M-program inv-modL be defined as*

$$\textit{inv-modL}\ [\textit{intN}, \textit{pgm}, [\textit{cls}, \textit{out}]] \overset{\text{def}}{=} \textit{inv-intL}\ [\textit{intN}, [[\textit{pgm}, \textit{cls}], \textit{out}]].$$

Theorem 2 (modifier property of *inv-modL*) *Let $\mathcal{P}$ be the set of all programming languages, then inv-modL is an L/M-semantics modifier for INV in $\mathcal{P}$.*

Proof. According to Lemma 1, *inv-modL* is an L/M-semantics modifier for *INV* in $\mathcal{P}$ because for all $N \in \mathcal{P}$, for all N/L-interpreters *intN*, for all $\textit{pgm} \in \textit{PgmN}$, for all classes $\textit{cls} \in \textit{DatN}$, and for all $\textit{out} \in \textit{DatN}$, the result $\textit{ans} = [s_1, \ldots]$ produced by computation

$$\textit{inv-modL}\ [\textit{intN}, \textit{pgm}, [\textit{cls}, \textit{out}]] \underset{M}{\overset{*}{\Rightarrow}} [s_1, \ldots]$$

satisfies Property *INV* (Def. 13):

$$\begin{array}{cl}
 & \textit{inv-modL}\ [\textit{intN}, \textit{pgm}, [\textit{cls}, \textit{out}]] \underset{M}{\overset{*}{\Rightarrow}} \textit{ans} \\
\overset{\text{(Def. 15)}}{\Longrightarrow} & \textit{inv-intL}\ [\textit{intN}, [[\textit{pgm}, \textit{cls}], \textit{out}] \underset{M}{\overset{*}{\Rightarrow}} \textit{ans} \\
\overset{\text{(Def. 14)}}{\Longrightarrow} & \{\textit{dat} \mid \textit{dat} \in \lceil [\textit{pgm}, \textit{cls}] \rceil, \textit{intN}\ \textit{dat} \underset{L}{\overset{*}{\Rightarrow}} \textit{out}\} = \bigcup_i \lceil [\textit{pgm}, \textit{cls}]/s_i \rceil \\
\overset{\text{(format)}}{\Longrightarrow} & \{[\textit{pgm}', \textit{dat}'] \mid [\textit{pgm}', \textit{dat}'] \in \lceil [\textit{pgm}, \textit{cls}] \rceil, \textit{intN}\ [\textit{pgm}', \textit{dat}'] \underset{L}{\overset{*}{\Rightarrow}} \textit{out}\} = \\
 & \qquad \bigcup_i \lceil [\textit{pgm}, \textit{cls}]/s_i \rceil \\
\overset{(\textit{pgm}\ \text{gnd})}{\Longrightarrow} & \{[\textit{pgm}, \textit{dat}'] \mid \textit{dat}' \in \lceil \textit{cls} \rceil, \textit{intN}\ [\textit{pgm}, \textit{dat}'] \underset{L}{\overset{*}{\Rightarrow}} \textit{out}\} = \bigcup_i \lceil [\textit{pgm}, \textit{cls}/s_i] \rceil \\
\overset{\text{(format)}}{\Longrightarrow} & \{\textit{dat}' \mid \textit{dat}' \in \lceil \textit{cls} \rceil, \textit{intN}\ [\textit{pgm}, \textit{dat}'] \underset{L}{\overset{*}{\Rightarrow}} \textit{out}\} = \bigcup_i \lceil \textit{cls}/s_i \rceil \\
\overset{\text{(Def. 7)}}{\Longrightarrow} & \{\textit{dat}' \mid \textit{dat}' \in \lceil \textit{cls} \rceil, \textit{pgm}\ \textit{dat}' \underset{N}{\overset{*}{\Rightarrow}} \textit{out}\} = \bigcup_i \lceil \textit{cls}/s_i \rceil \\
\overset{\text{(Def. 13)}}{\Longrightarrow} & \textit{INV}(N, \textit{pgm}, \textit{cls}, \textit{out}) = \bigcup_i \lceil \textit{cls}/s_i \rceil
\end{array}$$

□

Theorem 2 states that inverse computation can be performed in any programming language N given an inverse interpreter for L and a standard interpreter for N written in L. The result obtained by inverse computation of N's interpreter is a result for inverse computation in N. The theorem guarantees that the result is *correct* for all N-program regardless of *intN*'s algorithmic properties. It is straightforward to define an inverse interpreter for any language N in M given a standard interpreter for N in L (Fig. 2).

The results above are interesting for several reasons.

We believe the observation that a semantics modifier can encapsulate the essence of inverse computation can be useful for a variety of verification problems. For example, reasoning about the correctness of a hardware or software specification, one may need to verify whether and how a critical state can be reached from any earlier state. As observed earlier, the language paradigm of N is quite irrelevant for a semantics modifier. Thus, we should be able to apply inverse computation directly to a variety of language paradigms, such as Petri-nets, without having to develop separate tools for each language. By exploiting these connections, one may be able to significantly extend the capabilities of such tools.

From a programming language perspective, *inverse programming* [1] is a style of programming where one writes programs so that their inverse computation produces the desired result. This gives rise to a declarative style of programming [36] where one specifies the result rather than describes how it is computed. Inversion problems are not restricted to the context of logic programming and proof theory. Inversion is useful in any programming language, e.g., if one direction of an algorithm is easier to define than the other, or if both directions are needed. Given a semantics modifier, inversion problems can be solved in other languages without writing an inverse interpreter for that language. Experiments demonstrating the computational feasibility of these ideas are presented in Section 10.

6. Program Transformation: Equivalence Transformation Modifier

We are now going to examine another important class of programs, namely the class of *equivalence transformers*. We show that all equivalence transformers are semantics modifiers with respect to an equivalence transformation semantics. This can be a substantial advantage when developing new program transformers.

An equivalence transformer modifies the structure of a program with the purpose of optimizing some aspects of the program's performance, e.g. its time or space consumption, while preserving the program's functionality. A classical example are *translators* which convert programs from one language into another without changing their functionality. Two more recent examples are *program specializers* which construct efficient specialized versions of programs, and *program composers* which transform the composition of two programs into a single, more efficient program. We refer to these programs collectively as equivalence transformers.

After examining the modifier property for equivalence transformation, we present two modifiers, one for specialization and one for translation. In this section we shall not assume a specific transformation method. Later, for our experiments we will choose a concrete algorithm.

6.1. A Modifier for Equivalence Transformation

We introduce the notion of an EQT_R-dialect of a language L and then, based on that notion, define an EQT_R-transformer for that language. Finally, we show that all EQT_R-transformers have the property of being semantics modifiers.

Definition 16 (EQT_R-dialect) *Let L, R be programming languages, then L' is a* non-standard EQT_R-dialect of L *iff for all $pgm \in PgmL$, and for all classes cls, the result of computation ($pgm\ cls \overset{*}{\underset{L'}{\Rightarrow}} pgm'$) is an R-program pgm' such that*

$$pgm' \in EQT_R(L, pgm, cls) \qquad\qquad (Property\ EQT_R)$$

$$\textit{where } EQT_R(L, pgm, cls) \stackrel{\text{def}}{=}$$
$$\{pgm' \mid pgm' \in PgmR,\ pgm' \textit{ has argument list } vars = \text{varlist}(cls),$$
$$\forall s \in \text{fvs}(cls) : (pgm\ (cls/s) \overset{*}{\underset{L}{\Rightarrow}} out) \Longleftrightarrow (pgm'\ (vars/s) \overset{*}{\underset{R}{\Rightarrow}} out)\}\ .$$

Definition 17 (EQT_R-transformer) *Let L, M, R be programming languages, then an M-program eqt is an* $L{\rightarrow}R/M$-equivalence transformer *iff there exists an*

EQT_R-dialect L' of L such that eqt is an L'/M-interpreter. For equivalence transformation using eqt, we write $(eqt\ [pgm, cls] \underset{M}{\overset{*}{\Rightarrow}} pgm')$.

The equivalence transformation semantics states that an L-program *pgm* and a class *cls* is mapped into a functionally equivalent R-program pgm' that takes values for the free variables of *cls* as input. An equivalence transformer is a program that produces pgm' given *pgm* and *cls* as *input*.

Definition 18 (EQT_R-modifier) *Given an $L{\rightarrow}R/M$-equivalence transformer eqt, let M-program eqt-modL be defined as*

$$eqt\text{-}modL\ [intN, pgm, cls] \overset{\text{def}}{=} eqt\ [intN, [pgm, cls]]\ .$$

Theorem 3 (modifier property of *eqt-modL***)** *Let $\mathcal{P}$ be the set of all programming languages, then eqt-modL is an L/M-semantics modifier for EQT_R in $\mathcal{P}$.*

Proof. According to Lemma 1, *eqt-modL* is an L/M-semantics modifier for EQT_R-semantics because for all $N \in \mathcal{P}$, for all N/L-interpreters *intN*, for all N-programs *pgm*, and for all classes *cls*, the R-program pgm' produced by computation

$$eqt\text{-}modL\ [intN, pgm, cls] \underset{M}{\overset{*}{\Rightarrow}} pgm'$$

satisfies Property EQT_R (Def. 16).

Since *pgm* is ground, we have varlist([*pgm*, *cls*]) = varlist(*cls*), and fvs([*pgm*, *cls*]) = fvs(*cls*). Thus we have:

$$\begin{array}{ll}
 & eqt\text{-}modL\ [intN, pgm, cls] \underset{M}{\overset{*}{\Rightarrow}} pgm' \\
\overset{\text{(Def. 18)}}{\Longrightarrow} & eqt\ [intN, [pgm, cls]] \underset{M}{\overset{*}{\Rightarrow}} pgm' \\
\overset{\text{(Def. 17)}}{\Longrightarrow} & pgm' \in EQT_R(L, intN, [pgm, cls]) \\
\overset{\text{(Def. 16)}}{\Longrightarrow} & pgm' \in \{pgm' \mid pgm' \in PgmR, pgm'\ \textit{has argument list vars} = \text{varlist}(cls), \\
 & \forall s \in \text{fvs}(cls) : (intN\ ([pgm, cls]/s) \underset{L}{\overset{*}{\Rightarrow}} out) \iff (pgm'\ (vars/s) \underset{R}{\overset{*}{\Rightarrow}} out)\} \\
\overset{(pgm\ \text{gnd})}{\Longrightarrow} & pgm' \in \{pgm' \mid pgm' \in PgmR, pgm'\ \textit{has argument list vars} = \text{varlist}(cls), \\
 & \forall s \in \text{fvs}(cls) : (intN\ ([pgm, cls/s]) \underset{L}{\overset{*}{\Rightarrow}} out) \iff (pgm'\ (vars/s) \underset{R}{\overset{*}{\Rightarrow}} out)\} \\
\overset{\text{(Def. 7)}}{\Longrightarrow} & pgm' \in \{pgm' \mid pgm' \in PgmR, pgm'\ \textit{has argument list vars} = \text{varlist}(cls), \\
 & \forall s \in \text{fvs}(cls) : (pgm\ (cls/s) \underset{N}{\overset{*}{\Rightarrow}} out) \iff (pgm'\ (vars/s) \underset{R}{\overset{*}{\Rightarrow}} out)\} \\
\overset{\text{(Def. 16)}}{\Longrightarrow} & pgm' \in EQT_R(N, pgm, cls)
\end{array}$$

□

Theorem 3 asserts the possibility to design generic algorithms for equivalence transformation and, in principle, to apply them to programs written in any language via an interpretive definition. It is straightforward to define an equivalence transformer for any language N in M given a standard interpreter for N in L (Fig. 2).

The experiments in Section 11 illustrate these ideas using supercompilation [51], a powerful transformation method capable of program specialization and program composition (see [45]), and the projections in Section 13 show how an $N{\rightarrow}R/M$-transformer can be obtained from an $L{\rightarrow}R/M$-transformer and an N/L-interpreter.

6.2. A Program Specializer as Eqt-Modifier

A *program specializer* is an equivalence transformer that specializes a program with respect to some part of its input. The specialized program is faithful to the original program, but is often significantly faster. The specializer projections [22] assert the possibility of specializing programs in any language using an interpreter for that language, a property that we identified above more generally as semantics modifier property.

We use the familiar definition [31] of a program specializer and show how a program specializer can serve as a semantics modifier for specialization *and* translation.

Definition 19 (specializer) *An M-program pe is an $L\!\rightarrow\!R/M$-*program specializer *if for all $m, n \geq 0$, for all L-programs pgm with argument list of length $m+n$, for all input $x_1, ..., x_m, y_1, ..., y_n$, and SD-classification $S_1...S_m,D_1...D_n$:*

$$(\mathit{pgm}\ [x_1, ..., x_m, y_1, ..., y_n] \overset{*}{\underset{L}{\Rightarrow}} \mathit{out}) \Longleftrightarrow$$
$$((\mathit{pe}\ [\mathit{pgm}^{S_1...S_m,D_1...D_n}, x_1, ..., x_m])\ \ [y_1, ..., y_n] \overset{*}{\underset{R}{\Rightarrow}} \mathit{out})\ .$$

Specializer *pe* takes a source program *pgm*, an SD-classification $S_1...S_m,D_1...D_n$, and the known values $x_1,...,x_m$ as input. The result is a residual program that returns the same output when applied to the remaining input $y_1, ..., y_n$ as *pgm* applied to $x_1, ..., x_m, y_1, ..., y_n$. The SD-classification tells the specializer which part of the input is known ('*S*', static) and which part is unknown ('*D*', dynamic). For clarity we ordered the arguments into two groups, $x_1...x_m$ and $y_1...y_n$, but it is easy to see that the arguments can be listed in any order, provided an appropriate SD-classification is given. The SD-classification[b], added as superscript on programs for readability, is just another representation of a class.

Example 2 Let *pe* be a program specializer and let *pgm* be a program with a two-element argument list. Then we can specialize *pgm* with respect to different combinations of static/dynamic input and corresponding SD-classifications.

$$\begin{array}{llll}
(\mathit{pe}\ [\mathit{pgm}^{SS}, x, y])\ [\,] & = \mathit{pgm}\ [x, y] \qquad & (\mathit{pe}\ [\mathit{pgm}^{DS}, y])\ [x] & = \mathit{pgm}\ [x, y] \\
(\mathit{pe}\ [\mathit{pgm}^{SD}, x])\ \ [y] & = \mathit{pgm}\ [x, y] \qquad & (\mathit{pe}\ [\mathit{pgm}^{DD}])\ \ [x, y] & = \mathit{pgm}\ [x, y]
\end{array}$$

□

We now define two semantics modifiers using specializer *pe*, one for specialization and one for translation.

Let *pe* be an $L\!\rightarrow\!R/M$-program specializer, let *intN* be an N/L-interpreter, and let *pgm* be an *N*-program with two arguments. According to Def. 7 we have the following computational equivalence for *intN* and *pgm*:

$$(\mathit{intN}\ [\mathit{pgm}, [x, y]] \overset{*}{\underset{L}{\Rightarrow}} \mathit{out}) \Longleftrightarrow (\mathit{pgm}\ [x, y] \overset{*}{\underset{N}{\Rightarrow}} \mathit{out})\ .$$

Specialization modifier A *semantics modifier for specialization*, *pe-mod*, can be defined as follows:

$$\mathit{pe\text{-}mod}\ [\mathit{intN}, \mathit{pgm}, x] \overset{\text{def}}{=} \mathit{pe}\ [\mathit{intN}^{S[SD]}, \mathit{pgm}, x]$$
$$\text{such that } \mathit{pe\text{-}mod}\ [\mathit{intN}, \mathit{pgm}, x] \overset{*}{\underset{M}{\Rightarrow}} \mathit{pgm}'\ .$$

[b]This does not imply a program specializer based on offline partial evaluation.

The interpreter *intN* is annotated with *S[SD]* meaning that *pgm* and x are static, while y is dynamic. The result of applying *pe-mod* to *intN*, *pgm* and x is a new program *pgm*′. It is easy to verify that R-program *pgm*′ is a specialized version of N-program *pgm*:

$$(\mathit{pgm}'\ [y] \overset{*}{\underset{R}{\Rightarrow}} \mathit{out}) \Longleftrightarrow (\mathit{pgm}\ [x, y] \overset{*}{\underset{N}{\Rightarrow}} \mathit{out})\ .$$

This asserts the possibility of specializing N-programs without writing a specializer for N. The application of *pe-mod* corresponds to the 1st specializer projection [22]. Practical results for a specialization modifier were first reported in [23, 24]. The efficiency of the specialization process can be improved by specializing the specialization modifier *pe-mod*. This transformation corresponds to the 2nd and 3rd specializer projection.

Translation modifier A *semantics modifier for translation*, *trans-mod*, can be defined as follows:

$$\mathit{trans\text{-}mod}\ [\mathit{intN}, \mathit{pgm}, \mathit{nil}] \overset{\text{def}}{=} \mathit{pe}\ [\mathit{intN}^{S[DD]}, \mathit{pgm}]$$
$$\text{such that } \mathit{trans\text{-}mod}\ [\mathit{intN}, \mathit{pgm}, \text{'NIL}] \overset{*}{\underset{M}{\Rightarrow}} \mathit{pgm}''\ .$$

In this case, the interpreter *intN* is annotated with *S[DD]* so that *pgm* is static, while both x and y are dynamic. Note that variable *nil* is not used on the right hand side of *trans-mod*'s definition. The request is empty. The result of applying *trans-mod* to *intN* and *pgm* is a new program *pgm*″. It is easy to verify that R-program *pgm*″ is a translated version of N-program *pgm*:

$$(\mathit{pgm}''\ [x, y] \overset{*}{\underset{R}{\Rightarrow}} \mathit{out}) \Longleftrightarrow (\mathit{pgm}\ [x, y] \overset{*}{\underset{N}{\Rightarrow}} \mathit{out})\ .$$

This asserts the possibility of translating programs from N to R without writing an $N{\rightarrow}R$-translator. The application of *trans-mod* corresponds to the 1st Futamura projection [20]. Practical results for a translation modifier were first reported in [32, 33]. Similar to the specializer projections, the efficiency of the translation process can be improved by specializing the translation modifier *trans-mod*. This transformation corresponds to the 2nd and 3rd Futamura projection. We will discuss this relation in more detail in Sec. 13.

Even though a conventional translator can be used to define a translation modifier, this modifier will be rather trivial in its transformational power. The modifier only embeds *pgm* as constant in *intN*, and then translates the new program. As experience shows, *pgm* itself will not be translated, only *intN*. Clearly, this is not the translation of *pgm* we expect. On the other hand, we know that a translation modifier based on a program specializer (in the form of the 1st Futamura projection) is a practical translation modifier, as it has been shown in a series of impressive experiments (see [31]). Again, this is practical evidence that the theoretical concept of semantics modifiers can be put to work.

7. Program Analysis: Neighborhood Analysis Modifier

Similar to inverse computation, the semantics of *neighborhood analysis* [3] can be ported from one programming language to another. We show the correctness of a semantics modifier for neighborhood analysis.

7.1. A Modifier for Neighborhood Analysis

Given an L-program *pgm* and data *dat*, the semantics of neighborhood analysis is the set of all program-data pairs that compute the same result as *pgm* applied to *dat*. Neighborhood analysis has, among others, applications to program testing [2] and termination of supercompilation [52], because it tells us how much we can change *pgm* and *dat* without affecting the output of a computation.

Definition 20 (*NAN*-dialect) *Let L be a programming language, then L' is a* non-standard *NAN*-dialect of *L iff for all pgm* $\in$ *PgmL, and for all dat* $\in$ *DatL, the result of computation* $(pgm\ dat \underset{L'}{\overset{*}{\Rightarrow}} ans)$ *is a pair* $ans = (cls^{[pgm,dat]}, out)$ *such that*

$$\lceil (cls^{[pgm,dat]}, out) \rceil \subseteq NAN(L, pgm, dat) \qquad\qquad (Property\ NAN)$$

$$\textit{where } NAN(L, pgm, dat) \overset{\text{def}}{=} \{([pgm', dat'], out) \mid pgm' \in PgmL, dat' \in Dat, (pgm'\ dat' \underset{L}{\overset{*}{\Rightarrow}} out) \Longleftrightarrow (pgm\ dat \underset{L}{\overset{*}{\Rightarrow}} out)\}\ .$$

Definition 21 (nan-analyzer) *Let L, M be programming languages, then an M-program nanL is an L/M*-neighborhood analyzer *iff there exists an NAN-dialect L' of L such that nanL is an L'/M-interpreter. For neighborhood analysis using nanL, we write* $(nanL\ [pgm, dat] \underset{M}{\overset{*}{\Rightarrow}} (cls^{[pgm,dat]}, out))$.

Set $NAN(L, pgm, dat)$ is the semantics of neighborhood analysis of L-program *pgm* and data *dat*. It is the largest set such that for all its elements $([pgm', dat'], out)$, pgm' applied to dat' returns the same result as *pgm* applied to *dat*.

A neighborhood analysis computes an approximation of $NAN(L, pgm, dat)$ (the question whether all functionally equivalent program-data pairs have been found is undecidable in general). The answer, if it exists, is a pair $(cls^{[pgm,dat]}, out)$ that represents a subset of $NAN(L, pgm, dat)$. We say class $cls^{[pgm,dat]}$ is the *neighborhood* of *pgm* and *dat* (we use $[pgm, dat]$ to index *cls*). The class tells us how much we can change *pgm* and *dat* without changing the output of the computation. A safe, but trivial approximation, provided *pgm* applied to *dat* terminates, is a singleton set containing only $([pgm, dat], out)$ as element. Clearly, this is not what we expect from a practical neighborhood analysis. As we shall see later in Sec. 12, there exists algorithms that produce non-trivial approximations.

For our purposes, namely to define a semantics modifier for neighborhood analysis that allows us to port the analysis to all programming languages, we introduce the notion of a *program-trivial* neighborhood analyzer. The analyzer computes an approximation of the set of data only (a class cls^{dat}). Class cls^{dat} produced by a program-trivial nan-analyzer tells us how much the input of *pgm* can be changed without changing the output.

Definition 22 (program-trivial nan-analyzer) *Let L, M be programming languages, then an M-program nanL is a* program-trivial L/M-neighborhood analyzer *iff for all* $pgm \in PgmL$*, and for all* $dat \in DatL$*: if* $(nanL\ [pgm, dat] \underset{M}{\overset{*}{\Rightarrow}} (cls^{dat} out))$ *is defined then* $\lceil([pgm, cls^{dat}], out)\rceil \subseteq NAN(L, pgm, dat)$.

Given a program-trivial neighborhood analyzer for L, neighborhood analysis can be performed in any programming language N given an interpreter for that language written in L. The following theorem guarantees the correctness of the results. The definition of the semantics modifier *nan-mod* is straightforward.

Definition 23 (*NAN*-modifier) *Given a program-trivial L/M-neighborhood analyzer nanL, let M-program nan-modL be defined as*

$$nan\text{-}modL\ [intN, pgm, dat] \overset{\text{def}}{=} nanL\ [intN, [pgm, dat]]\ .$$

Theorem 4 (modifier property of *nan-modL***)** *Let $\mathcal{P}$ be the set of all programming languages, then nan-modL is an L/M-semantics modifier for NAN in $\mathcal{P}$.*

Proof. According to Lemma 1, *nan-modL* is an L/M-semantics modifier for *NAN*-semantics because for all $N \in \mathcal{P}$, for all N/L-interpreters *intN*, for all N-programs *pgm*, and for all data *dat*, the result of computation

$$nan\text{-}modL\ [intN, pgm, dat] \underset{M}{\overset{*}{\Rightarrow}} (cls^{[pgm,dat]}, out)$$

satisfies Property *NAN* (Def. 20).

Since *intN* is ground, we have for any class $[intN, cls]$:

(i) *cls is a class*, and
(ii) $[intN', x] \in \lceil[intN, cls]\rceil \iff (intN' = intN\ \wedge\ x \in \lceil cls\rceil)$.

Thus we have:

$$\begin{aligned}
& nan\text{-}mod\ [intN, pgm, dat] \underset{M}{\overset{*}{\Rightarrow}} (cls^{[pgm,dat]}, out) \\
\overset{\text{(Def. 23)}}{\Longrightarrow}\ & nanL\ [intN, [pgm, dat]] \underset{M}{\overset{*}{\Rightarrow}} (cls^{[pgm,dat]}, out) \\
\overset{\text{(Def. 22)}}{\Longrightarrow}\ & \lceil([intN, cls^{[pgm,dat]}], out)\rceil \subseteq NAN(L, intN, [pgm, dat]) \\
\overset{\text{(Def. 20)}}{\Longrightarrow}\ & intN\ [pgm, dat] \underset{L}{\overset{*}{\Rightarrow}} out, \\
& (\forall[intN', [pgm', dat']] \in \lceil[intN, cls^{[pgm,dat]}]\rceil : intN'\ [pgm', dat'] \underset{L}{\overset{*}{\Rightarrow}} out) \\
\overset{\text{(Def. 7)}}{\Longrightarrow}\ & pgm\ dat \underset{N}{\overset{*}{\Rightarrow}} out, \\
& (\forall[pgm', dat'] \in \lceil cls^{[pgm,dat]}\rceil : intN\ [pgm', dat'] \underset{L}{\overset{*}{\Rightarrow}} out) \\
\overset{\text{(Def. 7)}}{\Longrightarrow}\ & pgm\ dat \underset{N}{\overset{*}{\Rightarrow}} out, \\
& (\forall[pgm', dat'] \in \lceil cls^{[pgm,dat]}\rceil : pgm'\ dat' \underset{N}{\overset{*}{\Rightarrow}} out) \\
\overset{\text{(Def. 20)}}{\Longrightarrow}\ & \lceil(cls^{[pgm,dat]}, out)\rceil \subseteq NAN(N, pgm, dat)
\end{aligned}$$

□

Given Theorem 4 it is straightforward to define a neighborhood analyzer for any language N in M given an interpreter for N in L (Fig. 2). Experiments demonstrating a semantics modifier for neighborhood analysis are presented in Section 12.

8. Case Study: Porting Non-Standard Semantics

We now put the ideas developed in the previous sections on trial. We do so by implementing the semantics modifiers for a first-order functional language, and then porting the respective non-standard semantics from the functional language to an imperative language. The emphasis of this case study is on demonstrating the existence of non-trivial semantics modifiers; it is beyond the scope of this paper to give a definitive account of different algorithms and their practical ramifications. In this section we present the programming languages we used and give an overview of the experiments we performed.

8.1. Programming Languages TSG and MP

The choice of the source languages is important for studying foundational problems. On the one hand the languages must be small enough to develop and fully implement all algorithms, on the other hand the languages must be large enough to substantiate theoretical claims by computer experiments. We choose two language that have been used earlier in the context of program transformation: TSG, a first-order functional language, and MP, a small imperative language. Their syntax is shown in Fig. 3. The data domain is in both cases the set of S-expressions (Sec. 2.1).

TSG [2, 3, 5] is a typed dialect of the *first-order, functional language* S-Graph [26]. A TSG-program is a list of function definitions. Variables range over atoms (`a`*name*) or S-expressions (*name*). Data structures can be constructed (`CONS`), tested and decomposed (`EQA?`, `CONS?`). The language is restricted to tail-recursion. The semantics is formally defined elsewhere [3, 5, 26]. This family of languages has been used for clarifying the essence of supercompilation and related issues of meta-programming [2, 26, 29].

MP [44] is a small *imperative language* with assignments (`<==`), conditionals (`oIF`) and loops (`oWHILE`). An MP-program consists of a parameter list, a declaration of local variables, and a sequence of statements. An MP-program operates over a store consisting of parameters and local variables. The semantics is conventional Pascal-style semantics. The language is popular for experiments in partial evaluation [12, 39, 44].

8.2. Experiments Overview

We implemented each semantics modifier for TSG, and ported the corresponding semantics from TSG to MP using an interpreter for MP written in TSG. We used the *same* MP/TSG-interpreter for all experiments; there were no changes made to the interpreter when used with different semantics modifiers. All semantics modifiers were implemented in Gofer, a dialect of Haskell [30]. To compare the results for TSG and MP, we used the same example program, a naive pattern matcher, for all experiments.[c]

[c]All run times in this paper are given in CPU-seconds (including garbage collection, if any) using WinGofer 2.30b and a PC/Intel Pentium II-266MHz, MS Windows 95. All programs available from: `http://www.botik.ru/AbrGlu/SemMod/IJFCS`

pgm	::= [fd_1, ... fd_n]	— **TSG-program**
fd	::= (DEFINE fn[v_1, ... v_n] t)	— function definition
v	::= a*name* | *name*	— a- and e-variables (range over Atoms and D)
t	::= *exp* | (ALT *cond* t_1 t_2)	— term, alternative
	| (CALL fn[exp_1, ... exp_n])	— function call
cond	::= (CONS? *exp* v_{car} v_{cdr} v_{atom})	— test and decomposition of pair
	| (EQA? exp_{atom1} exp_{atom2})	— test for equality
exp	::= *atom* | *v* | (CONS exp_1 exp_2)	— expression

pgm	::= oPROGRAM	— **MP-program**
	(oPARS [var_1, ... var_p])	— parameters (take initial value from input)
	(oVARS [var_{p+1}, ... var_n])	— additional variables (initial value is 'NIL)
	stmt	— statement—body of the program
stmt	::= (*var* <== *exp*)	— assignment statement;
	| (oIF(exp_{cond}) $stmt_1$ $stmt_2$)	— conditional statement;
	| (oWHILE(exp_{cond}) *stmt*)	— loop statement
	| oNOP	— no operation;
	| (oDO[$stmt_1$, ... $stmt_n$])	— compound statement.
	| (oRETURN *exp*)	— stops the program, defines result of computation as value of *exp*;
exp	::= *var* | (oQUOTE *S-expression*)	
	| (oCAR *exp*) | (oCDR *exp*)	— head and tail of a list
	| (oCONS exp_1 exp_2)	— pair of exp_1 and exp_2
	| (oIsAtom *exp*)	— test for atom
	| (oIsEqual exp_1 exp_2)	— test for equality

Fig. 3. Syntax of programming languages TSG and MP.

All algorithms we used are outlined in the respective sections. For a more detailed discussion of each algorithm, the interested reader is referred to the original literature. We use a graph of configurations [51, 26] for describing the algorithms.

More specifically, in the next sections we present results for four modifiers:

- Standard interpretation (Sec. 9).
- Inverse computation (Sec. 10).
- Equivalence transformation (Sec. 11).
- Neighborhood analysis (Sec. 12).

9. Identity Modifier Applied to MP

To illustrate the identity modifier, we implemented two interpreters for TSG and MP: a TSG/Gofer-interpreter and an MP/TSG-interpreter. We shortly describe the interpreters and assess their interpretive overhead using a naive pattern matcher. The interpreters will also be used in the experiments in the following sections.

Given these two interpreters, MP-programs can be run in Gofer by interpreting the MP/TSG-interpreter on the TSG/Gofer-interpreter. Here the TSG/Gofer-

```
match =  [ (DEFINE "Match"[pattern, string]
              (CALL "CheckPos"[pattern, string, pattern, string]) ),
            (DEFINE "CheckPos" [pat, str, pattern, string]
              (ALT (CONS? pat patHead patTail a_ )
                (ALT (CONS? patHead e_ e_ a_patHead)
                  ('ERROR:Atom_expected)
                  (ALT (CONS? str strHead strTail a_)
                    (ALT (CONS? strHead e_ e_ a_strHead)
                      ('ERROR:Atom_expected)
                      (ALT (EQA? a_patHead a_strHead)
                        (CALL "CheckPos"[patTail, strTail, pattern, string])
                        (CALL "NextPos" [pattern, string]) ) )
                    ('FAILURE) ) )
                ('SUCCESS) ) ),
            (DEFINE "NextPos" [pattern,string]
              (ALT (CONS? string e_ stringTail a_)
                (CALL "Match" [pattern, stringTail])
                ('FAILURE) ) ) ]
```

— *String representation:*

`str"ABC" = CONS 'A (CONS 'B (CONS 'C 'NIL))`

— *Examples of computation:*

`match [ str"AB", str"AAAAAAAAB" ]` $\overset{*}{\underset{\text{TSG}}{\Rightarrow}}$ `'SUCCESS`

`match [ str"AB", str"AAAAAAAAA" ]` $\overset{*}{\underset{\text{TSG}}{\Rightarrow}}$ `'FAILURE`

`match [ str"AB", (CONS (CONS 'A 'B) 'NIL) ]` $\overset{*}{\underset{\text{TSG}}{\Rightarrow}}$ `'ERROR:Atom_expected`

Fig. 4. Naive pattern matcher written in TSG.

interpreter plays the role of an identity-modifier (Sec. 4), and the MP/TSG-interpreter defines the standard semantics of MP in TSG.

The TSG-interpreter `int` was written in WinGofer 2.30b (65 lines of pretty-printed source text); the MP-interpreter `intMP` was written in TSG (309 lines of pretty-printed source text; 30 functions in TSG).

To compare the interpretive overhead, a naive pattern matcher was written in TSG and in MP. The programs are shown in Fig. 4 and Fig. 5, respectively. The matcher takes two strings as input: `pattern` and `string`. The matcher returns `'SUCCESS` if the `pattern` was found in `string`, `'FAILURE` if `pattern` was not found in `string`, and `'ERROR:Atom_expected` if there was an error in the format of the strings. The implementation details of the matchers differ slightly because two different programming languages were used.[d]

As example consider `pattern = str"AAB"` and `string = str"A`...$_{100}$...`AB"` where computation `match[pattern,string]` yields `'SUCCESS`. The run time of the matcher in TSG was 2.6 sec and in MP was 208 sec. The interpretive overhead due to `intMP` is about 80 times.

The run time of the same matcher written directly in Gofer was 0.02 sec. Thus, the interpretive overhead due to `int` is about 130 times. It is not surprising that

[d] In particular, checking the format of the input strings is a separate test for `pattern` and `string` in the TSG-matcher (Fig. 4, lines 5–6 and 8–9). In the MP-matcher this check is combined with testing the equality of the current positions in `pattern` and `string` (Fig. 5, line 10): the construction `(oIsEqual (oCAR pat) (oCAR str))` halts the program with result `'Err:EqArg1:notAtom` (or `'Err:EqArg2:notAtom`) if `(oCAR pat)` (or `(oCAR str)`) is not atom.

```
[oPROGRAM (oPARS [pattern, string])
(oVARS [str, pat, goon])
(oDO[   oWHILE (string) (oDO[
            str  <== string,
            pat  <== pattern,
            goon <== oQUOTE(CONS 'NIL 'NIL),   {- goon := True -}
            oWHILE (goon) (
              oIF (pat) (
                oIF (str) (
                  oIF (oIsEqual (oCAR pat) (oCAR str)) (oDO[
                    pat <== oCDR pat,
                    str <== oCDR str
                  ]){-ELSE-}(
                    goon <== (oQUOTE 'NIL)     {- goon := False -}
                )){-ELSE-}(
                  oRETURN (oQUOTE 'FAILURE)
              )){-ELSE-}(
                oRETURN (oQUOTE 'SUCCESS)
            )),
            string <== oCDR string ]
        ),
        oRETURN (oQUOTE 'FAILURE) ]
]
```

Fig. 5. Naive pattern matcher written in MP.

considerable interpretive overhead occurs due to the two levels of interpretation. Running the matcher in Gofer is about 10,400 times faster than running the matcher in MP. This is what can be expected: each level of interpretation multiplies the run time by a significant factor.

10. Inversion Modifier Applied to MP

In this section we show that inverse computation can be ported from a functional language to an imperative language. After giving an algorithm for inverse computation in TSG, we show how inverse computation can be performed in MP. Inverse computation in MP is achieved via the standard MP/TSG-interpreter (Sec. 9).

10.1. The Universal Resolving Algorithm: an Inversion Modifier

There exist different methods for inverse computation [16, 28, 50]. The universal resolving algorithm outlined in this section uses methods from supercompilation, in particular *driving* [51]. The idea for this algorithm appeared in the early seventies [50] and since then several variants were implemented for functional languages [3, 5, 41, 42]. Our algorithm performs inverse computation in TSG and is implemented in Gofer (321 lines of pretty-printed source text).

Algorithm 24 [**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 [26] using a *breadth-first strategy*. In each step of driving a configuration `pgm` $\texttt{cls}_i$, the algorithm may encounter two basic cases:

1. *Driving finishes*. If the result of `pgm` $\texttt{cls}_i$ is $\texttt{out}_i$ and the unification $\texttt{out}_i$`:out` succeeds with substitution `s`, where $\texttt{out}_i$`/s = out`, then substitution `s'` resulting from

unification `cls:(cls`$_i$`/s)` is added to the answer (and printed); otherwise nothing is added.

2. *A contraction is encountered.* Configuration `pgm cls`$_i$ is split into disjoint configurations {`pgm cls`$_1$, ..., `pgm cls`$_n$} such that $\lceil$`pgm cls`$_1\rceil \cup \ldots \cup \lceil$`pgm cls`$_n\rceil$ = $\lceil$`pgm cls`$_i\rceil$ and $\lceil$`pgm cls`$_j\rceil \cap \lceil$`pgm cls`$_k\rceil = \emptyset$ ($n \geq 1, n\geq j > k\geq 1$).[e]

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

We shall not be concerned with technical details, only with the fact [3, 5] that the answer produced by the algorithm is correct solution (soundness), and that each substitution, if it exists, is computed in finite time (completeness). The algorithm does not always terminate because the search for further solutions may continue infinitely (even though all substitutions were found—the question whether further solutions exist or not is undecidable in general).

10.2. Porting Inverse Computation

URA implements inverse computation in TSG—it is an inverse interpreter for TSG. According to the theoretical results of Sec. 5, inverse computation can be ported to other languages by an *INV*-modifier defined as

$$\texttt{inv-mod[intN,pgmN,[cls,out]]} \overset{\text{def}}{=} \texttt{ura [intN,[[pgmN,cls],out]]} \;.$$

Experiment To compare the power and quality of inverse computation in TSG and in MP, we performed two inversion tasks using the naive matchers written in TSG and MP.

- **Task 1**: Find the set of strings `pattern` which *are* substrings of `"ABC"`. To perform this task we leave input `pattern` unknown ($\mathcal{E}_1$), set input `string = "ABC"` and the desired output to `'SUCCESS`.
- **Task 2**: Find the set of strings `pattern` which *are not* substrings of `"AAA"`. To perform this task we use a setting similar to Task 1 (`pattern =` $\mathcal{E}_1$, `string = "AAA"`), but the desired output is set to `'FAILURE`.

Results. Figure 6 (i, ii) shows the results of applying URA to `match` written in TSG. The answer for Task 1 is a finite representation of all possible substrings of string `"ABC"`. The answer for Task 2 is a finite representation of all strings which are not substrings of `"AAA"`. URA terminates after 0.2 seconds (Task 1, Task 2).

Figure 6 (iii, iv) shows the results for the MP-program `matchMP`. The answer for Task 1 is a finite representation of all possible substrings of string `"ABC"`. The answer for Task 2 is a finite representation of all strings which are not substrings of `"AAA"`. URA terminates after 24 sec (Task 1) and after 23 sec (Task 2).

Discussion. Inverse computation in MP (implemented by `inv-mod` and `intMP`) produces results very similar[f] to inverse computation in TSG (implemented directly by `ura`). This is noteworthy because inverse computation in MP is done through

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

[f] The results differ slightly (Fig. 6: compare (ii) line 2 and (iv) line 2) due to small differences in the implementation of the source programs; for more details see footnote *d*.

(i) *Inverse computation in TSG. The set of strings which are substrings of* "ABC".	ura [match, [([$\mathcal{E}_1$, str"ABC"],[]), 'SUCCESS]] $\overset{*}{\Rightarrow}$ [([$\mathcal{E}_1\mapsto\mathcal{A}_4$], []), --str"" ([$\mathcal{E}_1\mapsto$(CONS 'A $\mathcal{A}_{10}$)], []), --str"A" ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'B $\mathcal{A}_{16}$))], []), --str"AB" ([$\mathcal{E}_1\mapsto$(CONS 'B $\mathcal{A}_{10}$)], []), --str "B" ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'B (CONS 'C $\mathcal{A}_{22}$)))], []),--str"ABC" ([$\mathcal{E}_1\mapsto$(CONS 'B (CONS 'C $\mathcal{A}_{16}$))], []), --str "BC" ([$\mathcal{E}_1\mapsto$(CONS 'C $\mathcal{A}_{10}$)], [])] --str "C"
(ii) *Inverse computation in TSG. The set of strings which are not substrings of* "AAA".	ura [match, [([$\mathcal{E}_1$, str"AAA"],[]), 'FAILURE]] $\overset{*}{\Rightarrow}$ [([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'A (CONS 'A (CONS $\mathcal{A}_{25}$ $\mathcal{E}_{21}$))))],[]), ([$\mathcal{E}_1\mapsto$(CONS $\mathcal{A}_7$ $\mathcal{E}_3$)],[$\mathcal{A}_7\neq$'A]), ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'A (CONS $\mathcal{A}_{19}$ $\mathcal{E}_{15}$)))],[$\mathcal{A}_{19}\neq$'A]), ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS $\mathcal{A}_{13}$ $\mathcal{E}_9$))],[$\mathcal{A}_{13}\neq$'A])]
(iii) *Inverse computation in MP. The set of strings which are substrings of* "ABC".	inv-mod [intMP, matchMP, [([$\mathcal{E}_1$, str"ABC"],[]), 'SUCCESS]] $\overset{*}{\Rightarrow}$ [([$\mathcal{E}_1\mapsto\mathcal{A}_4$], []), --str"" ([$\mathcal{E}_1\mapsto$(CONS 'A $\mathcal{A}_{10}$)], []), --str"A" ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'B $\mathcal{A}_{16}$))], []), --str"AB" ([$\mathcal{E}_1\mapsto$(CONS 'B $\mathcal{A}_{10}$)], []), --str "B" ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'B (CONS 'C $\mathcal{A}_{22}$)))], []),--str"ABC" ([$\mathcal{E}_1\mapsto$(CONS 'B (CONS 'C $\mathcal{A}_{16}$))], []), --str "BC" ([$\mathcal{E}_1\mapsto$(CONS 'C $\mathcal{A}_{10}$)], [])] --str "C"
(iv) *Inverse computation in MP. The set of strings which are not substrings of* "AAA".	inv-mod [intMP, matchMP, [([$\mathcal{E}_1$, str"AAA"],[]), 'FAILURE]] $\overset{*}{\Rightarrow}$ [([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'A (CONS 'A (CONS $\mathcal{E}_{20}$ $\mathcal{E}_{21}$))))],[]), ([$\mathcal{E}_1\mapsto$(CONS $\mathcal{A}_7$ $\mathcal{E}_3$)],[$\mathcal{A}_7\neq$'A]), ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS 'A (CONS $\mathcal{A}_{19}$ $\mathcal{E}_{15}$)))],[$\mathcal{A}_{19}\neq$'A]), ([$\mathcal{E}_1\mapsto$(CONS 'A (CONS $\mathcal{A}_{13}$ $\mathcal{E}_9$))],[$\mathcal{A}_{13}\neq$'A])]

Fig. 6. Inversion modifier: inverse computation of TSG- and MP-matcher.

a standard interpreter for MP (not by an inverse interpreter for MP). It demonstrates that inverse computation can be ported successfully, here, from a functional language to an imperative language. Inverse computation in MP takes longer than in TSG due to the additional interpretive overhead (about 150 times).

In earlier work [4], inverse computation was successfully ported from TSG to a small assembler-like programming language (called Norma [8]). The experiments in this paper give further practical evidence for the viability of this approach by porting inverse computation to an imperative programming language (MP). The only other experimental work we are aware of that ports inverse computation, inverses imperative programs by treating their relational semantics as logic program [43].

11. Equivalence Transformation Modifier Applied to MP

In this section we show that an equivalence transformation, namely supercompilation, can be ported from a functional language to an imperative language. After giving an algorithm for supercompilation in TSG, we show how supercompilation can be performed in MP. The transformation is achieved via the standard MP/TSG-interpreter (Sec. 9). The transformation passes the KMP-test in TSG and in MP.

11.1. Supercompiler: an Equivalence Transformation Modifier

We outline an algorithm for supercompilation in TSG which will be our algorithm for equivalence transformation. A more detailed discussion of supercompilation and its methods can be found in [3, 26, 45, 51, 52, 53]). The supercompiler for TSG was implemented in Gofer (330 lines of pretty-printed source text).

Algorithm 25 [**SCP (outline)**] Given TSG-program `pgm` and class `cls`, the algorithm starts by driving the initial configuration `pgm cls` and builds a process tree `gr` (initially one node labeled with `pgm cls`) and accumulates a list of *basic configurations* `basics`. Each node in a rooted process tree is labeled with an active configuration `pgm` $\texttt{cls}_i$ or a passive expression `out`. Each edge is labeled with a contraction or a folding mark `(CALL s)`, where `s` is a substitution. To build a finite graph the construction is controlled by a criterion ('whistle') that decides when to stop and generalize [52].

The overall algorithm for supercompilation proceeds in two passes: in the first pass a graph and a list `basics` is constructed given a root and a list of basics initialized with the initial configuration `pgm cls`; in the second pass a new graph is constructed from a root initialized with `pgm cls` and a list of basics set to `basics`. Only the last graph is *converted* (as described in [26]) into a new TSG-program `pgm'`. The second pass increases sharing in the residual program and, thus, improves the quality of the residual program.

In each step of driving a configuration `pgm` $\texttt{cls}_i$, the algorithm may encounter two basic cases:

(i) *Driving finishes.* If the result of `pgm` $\texttt{cls}_i$ is $\texttt{out}_i$, then the node is passive and labeled with $\texttt{out}_i$.

(ii) *A contraction is encountered.* Configuration `pgm` $\texttt{cls}_i$ is split into disjoint configurations `pgm` $\texttt{cls}_{i_1}$, `pgm` $\texttt{cls}_{i_2}$ such that $\lceil pgm\ cls_{i_1} \rceil \cup \lceil pgm\ cls_{i_2} \rceil = \lceil pgm\ cls_i \rceil$ and $\lceil pgm\ cls_{i_1} \rceil \cap \lceil pgm\ cls_{i_2} \rceil = \emptyset$. To prevent infinite development, two conditions are checked:

(a) *Identical modulo renaming: folding.* If unification `pgm` $\texttt{cls}_b$`:pgm` $\texttt{cls}_i$ succeeds with substitution $\texttt{s}_{b,i}$ and unification `pgm` $\texttt{cls}_i$`:pgm` $\texttt{cls}_b$ succeeds with substitution $\texttt{s}_{i,b}$—then current node `pgm` $\texttt{cls}_i$ is connected with node `pgm` $\texttt{cls}_b$ by a backward edge labeled `(CALL` $\texttt{s}_{b,i}$`)`.

(b) *Whistle.* If current configuration `pgm` $\texttt{cls}_i$ is "dangerously similar" to one of its parent configuration `pgm` $\texttt{cls}_{up}$—i.e. `pgm` $\texttt{cls}_{up} \trianglelefteq$ `pgm` $\texttt{cls}_i$, where $\trianglelefteq$ is homeomorphic embedding order [17]—then:

1. *Instance: folding.* If unification `pgm` $\texttt{cls}_{up}$`:pgm` $\texttt{cls}_i$ succeeds with substitution `s` then current node `pgm` $\texttt{cls}_i$ is connected with node `pgm` $\texttt{cls}_{up}$ by a backward edge labeled `(CALL s)` and `pgm` $\texttt{cls}_{up}$ is added to `basics`;
2. *Otherwise: Generalization.* Configurations `pgm` $\texttt{cls}_i$ and `pgm` $\texttt{cls}_{up}$ are generalized to the *most specific generalization* `pgm` $\texttt{cls}_g$—that can be unified with both configurations by substitutions $\texttt{s}_{g,i}$ and $\texttt{s}_{g,up}$,—current node `pgm` $\texttt{cls}_i$ is connected to new child node `pgm` $\texttt{cls}_g$ with an edge labeled `(CALL` $\texttt{s}_{g,i}$`)`, configurations `pgm` $\texttt{cls}_g$ is added to `basics`, and driving continues for `pgm` $\texttt{cls}_g$. Downwards generalization is used.

(c) *Otherwise: branching.* The child-nodes `pgm` $\texttt{cls}_{i_1}$ and `pgm` $\texttt{cls}_{i_2}$ are connected with current node `pgm` $\texttt{cls}_j$ by two edges labeled with the corresponding contraction. Driving continues for the child nodes.

The algorithm stops when the development of all nodes in the graph is completed. The result is a graph `gr` and a list `basics`. □

We shall not be concerned with technical details, only with the fact that the residual program `pgm'` produced by the algorithm meets the conditions of Defini-

(i) Specialization of `match` by supercompiler (`pattern=str"AAB"` is fixed)

`scp [ match,`
`([str"AAB",` $\mathcal{E}_1$`],[]) ]` $\overset{*}{\underset{\text{Gofer}}{\Rightarrow}}$

—the supercompiler returns residual TSG-program `matchAAB[str]`:

```
[(DEFINE "matchAAB"[str]
 (ALT (CONS? str strH strT a_)
  (ALT (CONS? strH e_ e_ a_strH)
   'ERROR:Atom_expected
   (ALT (EQA? a_strH 'A)
    (ALT (CONS? strT strTH strTT a_)
     (ALT (CONS? strTH e_ e_ a_strTH)
      'ERROR:Atom_expected
      (CALL "chk_AB"[a_strTH, strTT]))
     'FAILURE)
    (CALL "matchAAB"[strT])))
  'FAILURE)),

 (DEFINE "chk_AB"[a_strH, strT]
  (ALT (EQA? a_strH 'A)
   (ALT (CONS? strT strTH strTT a_)
    (ALT (CONS? strTH e_ e_ a_strTH)
     'ERROR:Atom_expected
     (ALT (EQA? a_strTH 'B)
      'SUCCESS
      (CALL "chk_AB"[a_strTH, strTT])))
    'FAILURE)
   (CALL "matchAAB"[strT]))) ]
```

(ii) Specialization of `matchMP` by `eqt-mod` (`pattern=str"AAB"` is fixed)

`eqt-mod [intMP, matchMP,`
`([str"AAB",` $\mathcal{E}_1$`],[]) ]` $\overset{*}{\underset{\text{Gofer}}{\Rightarrow}}$

—`eqt-mod`*-computation returns residual TSG-program* `matchAAB_TSG[str]`:

```
[(DEFINE "matchAAB_TSG"[str]
  (ALT (CONS? str strH strT a_)
   (CALL "find_A"[strH, strT])
    'FAILURE)),
 (DEFINE "find_A"[strH, strT]
  (ALT (CONS? strH e_ e_ a_strH)
   'Err:EqArg2:notAtom
   (ALT (EQA? a_strH 'A)
    (ALT (CONS? strT strTH strTT a_)
     (ALT (CONS? strTH e_ e_ a_strTH)
      'Err:EqArg2:notAtom
      (CALL "chk_AB"[a_strTH, strTT]))
     'FAILURE)
    (ALT (CONS? strT strTH strTT a_)
     (CALL "find_A"[strTH, strTT])
     'FAILURE)))),
 (DEFINE "chk_AB"[a_strH, strT]
  (ALT (EQA? a_strH 'A)
   (ALT (CONS? strT strTH strTT a_)
    (ALT (CONS? strTH e_ e_ a_strTH)
     'Err:EqArg2:notAtom
     (ALT (EQA? a_strTH 'B)
      'SUCCESS
      (CALL "chk_AB"[a_strTH, strTT])))
    'FAILURE)
   (ALT (CONS? strT strTH strTT a_)
    (CALL "find_A"[strTH, strTT])
    'FAILURE))) ]
```

Fig. 7. Eqt-modifier: equivalence transformation of TSG- and MP-matcher.

tion 17, thus algorithm `scp` is an *equivalence transformer* for TSG. In the implemented version of the algorithm, as is common to other transformation systems, transient reductions are performed without testing for similarity of ancestor and, thus, the transformer does not always terminate.

11.2. Porting Supercompilation

The supercompiler (SCP) described above is a powerful equivalence transformer for TSG. We shall use it as a modifier for TSG→TSG/Gofer-equivalence transformation. According to the theoretical results of Sec. 6, supercompilation can be applied to other languages using an *EQT*-modifier defined as

`eqt-mod[intN,pgmN,cls]` $\overset{\text{def}}{=}$ `scp[intN,[pgmN,cls]]` .

Experiment In order to compare the power and quality of supercompilation in TSG and in MP we used the well-known KMP-test [45], a classical example of program transformation [21].

Consider a naive pattern matcher, namely the TSG-program `match` (Fig. 4). Assume we want to transform `match` into a TSG-program `matchAAB` that is equivalent to `match`, but specialized wrt `pattern=str"AAB"`:

$$\text{for all } \texttt{string} \in \texttt{Dat}$$
$$(\texttt{matchAAB[string]} \overset{*}{\underset{\text{TSG}}{\Rightarrow}} \texttt{out}) \iff (\texttt{match[str"AAB",string]} \overset{*}{\underset{\text{TSG}}{\Rightarrow}} \texttt{out})\ .$$

Program `matchAAB` can be obtained by supercompilation:

$$\texttt{scp[match, cls]} \overset{*}{\underset{\text{Gofer}}{\Rightarrow}} \texttt{matchAAB}$$
$$\text{where } \texttt{cls = ([str"AAB", } \mathcal{E}_1\texttt{],[])}\ .$$

Here the first parameter (`pattern`) of `match` is known and fixed to `str"AAB"` in `cls`; the second parameter (`string`) is unknown and represented by variable $\mathcal{E}_1$ in `cls`. The list of restrictions on variable domains is empty (`[]`).

Consider program `matchMP`, the naive pattern matcher written in MP (Fig. 5). The transformation can be realized using the MP/TSG-interpreter `intMP` and *EQT*-modifier: `eqt-mod [intMP, matchMP, cls]`.

Results. Figure 7 shows the residual programs produced by specializing the two matchers wrt `pattern=str"AAB"`.[g] The transformation time of the TSG-program `match` was 0.35 sec; the transformation time of the MP-program `matchMP` was 20 sec. To assess the efficiency of the residual programs, computation

$$\texttt{intTGS matchAAB [str"A}\ldots_{100}\ldots\texttt{AB"]} \overset{*}{\underset{\text{Gofer}}{\Rightarrow}} \texttt{'SUCCESS}$$

was performed with residual programs `matchAAB` and `matchAAB_TSG`. The run time was 0.1 sec in both cases.

Discussion. The result of supercompilation is remarkable: in both experiments the naive pattern matcher program was transformed into an efficient KMP-algorithm `matchAAB`.

Comparing the run time of the residual matchers, `matchAAB` and `matchAAB_TSG`, with the run time of the original matcher (Sec. 9), we note that on the same input the residual programs are about 25 times faster than the original matcher written in TSG. In the case of MP the overall speed-up factor of specialization (performed by `eqt-mod`) for this computation is about 2,000 times because *two optimizations* were achieved by `eqt-mod`: elimination of the interpretive level `intMP` *and* full transformation of the naive pattern matcher into an efficient KMP-algorithm.

The residual program `matchAAB_TSG` is fully optimized, although the program `matchAAB_TSG` differs[h] from program `matchAAB` both of them correspond to the KMP-matcher for pattern `"AAB"`. Supercompilation in TSG (implemented directly by `scp`) and in MP (implemented by `eqt-mod` and `intMP`) passes the KMP test.

[g] Only one syntactic change was done to the residual programs: function and variable names (automatically generated by `scp` as a letter and an unique number) were renamed to be more readable.

[h] The residual programs differ due to small differences in the source programs (see footnote *d*).

This is noteworthy because the results for MP were achieved through a standard interpreter for MP without writing an supercompiler for MP. This demonstrates that equivalence transformation, here supercompilation, can be ported from a functional language to an imperative language. Supercompilation in MP takes longer than in TSG due to the additional interpretive overhead (about 60 times).

In a related experiment [23], it was shown that a variant of Turchin's supercompiler (generated from a driving interpreter using the specializer projections [22]) is powerful enough to convert a naive pattern matcher written in a first-order functional language via a self-interpreter into a KMP-matcher. The above experiment extends this result by applying a (hand-written) supercompiler to an imperative language. In both cases, the removal of an entire level of interpretation is due to the *perfectness* of the underlying driving algorithm [26].

12. Neighborhood Analysis Modifier Applied to MP

In this section we show that neighborhood analysis can be ported from a functional language to an imperative language. After giving an algorithm for neighborhood analysis in TSG, we show how the analysis can be performed in MP. This is achieved via the standard MP/TSG-interpreter (Sec. 9).

12.1. Neighborhood Analysis: a Modifier

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 data which has the same computation process ('trace') and the same output. The neighborhood analysis for TSG is implemented in Gofer (the program-trivial neighborhood analyzer `nan`, 310 lines of pretty-printed source text).

Algorithm 26 [**NAN (outline)**] Given TSG-program `pgm`, data `dat`, let `cls` be the most general class representing all possible inputs for `pgm`. The algorithm performs two tasks: 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 `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 $(\mathtt{cls}_i\mathtt{/s, out})$.
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 $\{\mathtt{pgm\ cls}_1, \ldots, \mathtt{pgm\ cls}_n\}$ such that $\lceil pgm\ cls_1 \rceil \cup \ldots \cup \lceil pgm\ cls_n \rceil$ = $\lceil pgm\ cls_i \rceil$ and $\lceil pgm\ cls_j \rceil \cap \lceil pgm\ cls_k \rceil$ = $\emptyset$ $(n \geq j > k \geq 1)$. Configuration `pgm` $\mathtt{cls}_j$, where $\mathtt{dat}_i \in \lceil cls_j \rceil$, is selected and driven one step; all other configurations are ignored.

□

We shall not describe the technical details of the algorithm, only state the fact [3] that the algorithm computes correct results for neighborhood analysis (soundness), and that the algorithm terminates for all TSG-programs `pgm`, all data `dat` iff `pgm dat` $\overset{*}{\underset{\text{TSG}}{\Rightarrow}}$ `out` terminates (completeness).

12.2. Porting Neighborhood Analysis

The algorithm `nan` implements neighborhood analysis in TSG. According to the theoretical results of Sec. 7, neighborhood analysis can be ported to other languages using a *NAN*-modifier defined as

$$\texttt{nan-mod[intN,pgmN,dat]} \overset{\text{def}}{=} \texttt{nan [intN,[pgmN,dat]]} \;.$$

Experiment Consider the TSG- and MP-matcher and the following two experiments with neighborhood analysis:

1. $\texttt{nan [match, [pat}_1\texttt{, str}_1\texttt{]]} \overset{*}{\Rightarrow} \texttt{(cls}^{\texttt{[pat}_1\texttt{, str}_1\texttt{]}}\texttt{, y}_1\texttt{)}$
2. $\texttt{nan-mod [intMP, matchMP, [pat}_2\texttt{, str}_2\texttt{]]} \overset{*}{\Rightarrow} \texttt{(cls}^{\texttt{[matchMP, [pat}_2\texttt{, str}_2\texttt{]]}}\texttt{, y}_2\texttt{)}$

where $\texttt{pat}_1$ `= str"AB"`, $\texttt{str}_1$ `= str"XYZAB"`, $\texttt{pat}_2$ `= str"A"`, $\texttt{str}_2$ `= str"ABC"`, $\texttt{y}_1$ `=` $\texttt{y}_2$ `= 'SUCCESS`.

Results. Figure 8 shows the result of the first experiment. The neighborhood analysis of `match` returns a pair containing the output $\texttt{y}_1$ `= 'SUCCESS` of computation `match [str"AB", str"XYZAB"]`, and a class $\texttt{cls}^{\texttt{[pat}_1\texttt{, str}_1\texttt{]}}$ describing which part of the input `[str"AB", str"XYZAB"]` was used in the computation process. The class is represented by a list `[str"AB", str"XYZAB"]` where all components which were not relevant for the computation are marked by variables and underlined (e.g., $\underline{\mathcal{A}_6 : \texttt{'Z}}$). The variables tell us that the corresponding components can be replaced by arbitrary atoms ($\mathcal{A}_i$) or by arbitrary S-expressions ($\mathcal{E}_i$), respectively, without changing the computation process or the output of the program. The data represented by the class passes through the algorithm in the same way. The run time of the analysis was 0.1 sec.

Figure 8 shows the result of the second experiment. Applying modifier `nan-mod` to `matchMP` via `intMP` returns a pair containing the output $\texttt{y}_2$ `= 'SUCCESS` of computation `matchMP [str"A", str"ABC"]`, and a class describing which part of the input `[matchMP, [str"A", str"ABC"]]` was used in the computation process. The run time of the analysis was 5 sec. The result tells us that we can make the following modifications without changing the computation process or the output:

- *Input data:* we can use any data `(CONS pat (CONS str` $\mathcal{A}_{17}$`))` as input, where $[\texttt{pat}, \texttt{str}] \in \lceil$`[(CONS` $\mathcal{A}_{14}$ $\mathcal{A}_{15}$`), (CONS` $\mathcal{A}_{14}$ $\mathcal{E}_{16}$`)]`$\rceil$, and atom $\mathcal{A}_{17}$ represents the end of the argument list.
- *Text of MP-program*: we can (i) rename variables (by any atoms $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_4$, $\mathcal{A}_5$, $\mathcal{A}_6$ satisfying the restrictions); (ii) use any atoms $\mathcal{A}_3$ and $\mathcal{A}_7$ to represent the end of the parameter and variable list; (iii) replace (by any S-expression $\mathcal{E}_{10}$, $\mathcal{E}_{11}$, $\mathcal{E}_{12}$, $\mathcal{E}_{13}$) the corresponding fragments of the program text; (iv) replace constant `oQUOTE(CONS 'NIL 'NIL)` by any `oQUOTE(CONS` $\mathcal{E}_8$ $\mathcal{E}_9$`))`.

When comparing the results of the experiments above, the reader should note:

- In the first experiment with TSG-program `match`, the program-trivial neighborhood analyzer `nan` produces a neighborhood for the *input* of `match`: it describes which part of the input takes part in the computation process.

```
nan [match,
     [(CONS 'A (CONS 'B 'NIL)),                              -- str"AB"
      (CONS 'X (CONS 'Y (CONS 'Z (CONS 'A (CONS 'B 'NIL)))))   -- str"XYZAB"
    ]] =*=>
    (([                               -- 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] ),
     'SUCCESS                         -- Output of computation.
    )
```

```
nan-mod [intMP, matchMP, [pat, str] ] =*=> (cls^[matchMP, [pat, str]], y)
where [pat, str] =                              -- Data [str"A", str"ABC"].
          [ (CONS 'A 'NIL), (CONS 'A (CONS 'B (CONS 'C 'NIL))) ]
y = 'SUCCESS                                    -- Output of computation.
cls^[matchMP, [pat, str]] =                     -- Neighborhood of input.
  ( [oPROGRAM                                   -- Program matchMP.
    (oPARS [A1 : pattern, A2 : string A3 : ] )
    (oVARS [A4 : str, A5 : pat, A6 : goon A7 : ] )
    (oDO[   oWHILE (A2 : string) (oDO[
                A4 : str   <== A2 : string,
                A5 : pat   <== A1 : pattern,
                A6 : goon <== oQUOTE(CONS E8 : 'NIL E9 : 'NIL),
                oWHILE (A6 : goon) (
                  oIF (A5 : pat) (
                    oIF (A4 : str) (
                      oIF (oIsEqual (oCAR A5 : pat) (oCAR A4 : str)) (oDO[
                        A5 : pat <== oCDR A5 : pat,
                        A4 : str <== oCDR A4 : str
                      ]){-ELSE-}(
                        E10 : goon <== (oQUOTE 'NIL)
                    )){-ELSE-}(
                      E11 : oRETURN (oQUOTE 'FAILURE)
                  )){-ELSE-}(
                    oRETURN (oQUOTE 'SUCCESS)
                )),
                E12 : string <== oCDR string ]
            ),
            E13 : oRETURN (oQUOTE 'FAILURE) ]
    ],                                          -- Data [str"A", str"ABC"]:
  [ (CONS A14 : 'A A15 : 'NIL), (CONS A14 : 'A E16 : (CONS 'B (CONS 'C 'NIL)))A17 : ] ],
  [ A2≠A1, A4≠A1, A4≠A2, A5≠A1, A5≠A2,
    A5≠A4, A6≠A1, A6≠A2, A6≠A4, A6≠A5 ]
  )
```

Fig. 8. Nan-modifier: neighborhood analysis of TSG- and MP-matcher.

- In the second experiment with MP-program `matchMP`, modifier `nan-mod` produces a neighborhood for list `[matchMP, [pat`$_2$`, str`$_2$`]]`—not only for the *input* $[\mathrm{pat}_2, \mathrm{str}_2]$ of `matchMP`, but also for the *program* `matchMP`. This is a consequence of inserting an additional interpretive level: the text of *program* `matchMP` becomes a part of the *input* of `intMP`.

The experimental results correspond to the results of Sec. 7.1 (Def. 20, 22, 23 and Theorem 4).

Discussion. Both experiments, neighborhood analysis in TSG (implemented directly by `nan`), as well as neighborhood analysis in MP (implemented by `eqt-mod` and `intMP`), return non-trivial results. In particular, the second experiment is remarkable because the analysis was done through a standard interpreter for MP (not by a neighborhood analyzer for MP). This shows that neighborhood analysis can be ported from a functional language to an imperative language. Neighborhood analysis in MP takes longer than in TSG due to the additional interpretive overhead (about 50 times).

In a related experiment [4], neighborhood analysis was successfully ported from TSG to a small assembler-like programming language (called Norma [8]). The experiments above give further practical evidence for the viability of the approach by porting the analysis to yet another programming language (MP). The result of the analysis can be used for various applications, for example, for program testing [2, 3].

13. Eliminating Interpretive Overhead: the Modifier Projections

The scheme for non-standard computation used in the previous sections gives remarkable results, but is inefficient because it involves two levels of interpretation. In this section we study different approaches for improving the efficiency of non-standard computation by program specialization. We show that *non-standard interpreters* and *non-standard compilers* can be generated by specializing semantics modifiers with respect to standard interpreters.

13.1. Ways to Efficiency

Non-standard computation by a semantics modifier *sem-modL* involves interpretation at two levels: (i) an L-program *intN* is interpreted by *sem-modL*, and (ii) an N-program *pgm* is interpreted by *intN*:

$$\textit{sem-modL}\ [\textit{intN}, \textit{pgm}, \textit{req}] \underset{M}{\overset{*}{\Rightarrow}} \textit{ans}\ .$$

Programs written in three languages are involved: N-program *pgm*, L-program *intN*, and M-program *sem-modL*. This hierarchy of programs and languages is illustrated in Figure 9. The overall performance of this scheme can be improved by removing some or all of the interpretive overhead, either by removing language N, or language L, or both from the program hierarchy. The reduced hierarchies (a, b, c) are shown in Fig. 9. But how can we remove the intermediate languages?

Program specialization, or partial evaluation, is an optimization technique which is known for drastically improving the performance of programs by reducing interpretive overhead and collapsing towers of interpreters [9, 31, 15]. Impressive results have been achieved for interpreter specialization by offline partial evaluation (e.g., [10, 34]). Such an optimization is very important as a practical basis for building real non-standard tools. In the remainder of this section we examine how program specialization can make non-standard computation more efficient.

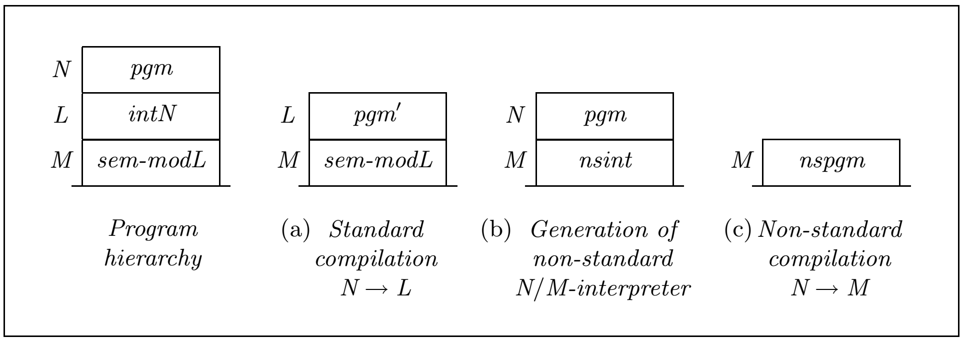
N
pgm
L
intN
M
sem-modL
Program hierarchy
L
pgm′
M
sem-modL
(a) Standard compilation N → L
N
pgm
M
nsint
(b) Generation of non-standard N/M-interpreter
M
nspgm
(c) Non-standard compilation N → M

Fig. 9. Removing interpretive layers from the program hierarchy.

Let *sem-modL* be an L/M-semantics modifier for a non-standard semantics S, *intN* be an N/L-interpreter, *pgm* be an N-program, and *req* be a request for S-non-standard computation. Answer *ans* is the result for request *req*.

In the following we assume that *pe* is an $L{\rightarrow}L/L$-specializer in case (a), and an $M{\rightarrow}M/M$-specializer in cases (b) and (c); for multi-language specialization with different source, target and implementation languages see [22]. For describing program specialization, we use the SD-notation from Sec. 6.2. We assume that the reader is familiar with the basic notions of self-applicable partial evaluation, for example as presented in [31, Part II].

(a) *Standard compilation*:

$$pe\ [intN^{SD}, pgm] \overset{*}{\underset{L}{\Rightarrow}} pgm' \ .$$

Specializing interpreter *intN* wrt N-program *pgm* yields an L-program *pgm′* which is functionally equivalent to *pgm*. To perform non-standard computation, modifier *sem-modL* has to be applied to *pgm′* (in general, this requires a self-interpreter for L, cf. Sec. 3.5). Interpretation of *pgm* by *intN* is avoided; but non-standard interpretation of *pgm′* remains.

(b) *Generation of a non-standard interpreter*:

$$pe\ [sem\text{-}modL^{SDD}, intN] \overset{*}{\underset{M}{\Rightarrow}} nsint \quad \text{where} \quad nsint\ [pgm, req] \overset{*}{\underset{M}{\Rightarrow}} ans \ .$$

Specializing *sem-modL* wrt N-interpreter *intN* yields a non-standard interpreter *nsint* for N. Non-standard interpretation of *intN* by *sem-modL* is avoided; but non-standard interpretation of *pgm* by *nsint* remains. One level of non-standard interpretation is removed.

(c) *Non-standard compilation*:

$$pe\ [sem\text{-}modL^{SSD}, intN, pgm] \overset{*}{\underset{M}{\Rightarrow}} nspgm \quad \text{where} \quad nspgm\ [req] \overset{*}{\underset{M}{\Rightarrow}} ans \ .$$

Specializing *sem-modL* wrt L-program *intN* and N-program *pgm* yields a non-standard M-program *nspgm*. All standard/non-standard interpretation levels are removed; the non-standard semantics is *internalized* in M-program *nspgm*.

13.2. Modifier Projections for Non-Standard Tools

Each of the three cases (a, b, c) can be carried further by applying a program specializer to the initial projection. The idea of *self-applying* a program specializer is known from the *Futamura projections* [20]. Self-application can be used to convert interpreters into compilers and, more generally, programs into two-level generating extensions (known as Ershov's *generating extension* [19]). Program specializers capable of this transformation have been developed for a variety of programming languages (see [9, 31, 15]). We are interested in this transformation because we want to produce efficient tools for non-standard interpretation and non-standard compilation.

Repeatedly applying a specializer to the initial projection in case (a) leads to the 2nd and 3rd Futamura projection, namely the conversion of an interpreter into a compiler, and the generation of a compiler generator. This application is well-known; we shall not pursue it further.

Cases (b) and (c) are more interesting because a semantics modifier is specialized, instead of a standard interpreter. Repeatedly applying specializer *pe* to the initial projection in case (b) leads to two new projections for non-standard interpretation (i2, i3). All three projections for non-standard interpretation are shown in Figure 10. Projection (i1) produces a non-standard interpreter *nsint* from a standard interpreter; (i2) produces a generator of non-standard interpreters *g-nsint* from a semantics modifier; and (i3) produces a generator of generators of non-standard interpreters *gg-nsint* by double self-application of *pe* (note that program *gg-nsint* is precisely the compiler generator produced by the 3rd Futamura projection).

Repeatedly applying specializer *pe* to the initial projection in case (c) leads to three new projections for non-standard compilation (c2–c4). All four projections for non-standard compilation are shown in Fig 10. Projection (c1) produces a non-standard program *nspgm* from a standard program; (c2) produces a non-standard compiler *nscomp* from a standard interpreter; (c3) produces a generator of non-standard compilers *g-nscomp* from a semantics modifier; and (c4) produces a generator of generators of non-standard compilers *gg-nscomp* by triple self-application of specializer *pe* (note that *gg-nscomp* is a three-level compiler generator, a special case of a *multi-level compiler generator* [25]).

Figure 11 shows the application of the programs given by projections (i1–i3) and (c1–c4). It is easy to verify the correctness of these equations using Def. 19.

Example 3 Let us illustrate projection (c1) which transforms a standard program *pgm* into a non-standard program *nspgm*. Consider the semantics modifier *inv-mod* for inverse computation (Def. 15). According to (c1), *nspgm* is the result of non-standard compilation of *pgm* from *N* to *M*,

$$pe\ [\mathit{inv\text{-}mod}^{SSD}, \mathit{intN}, \mathit{pgm}] \overset{*}{\Rightarrow} \mathit{nspgm} \qquad \text{(inverse compilation)}$$

such that $\forall \mathit{req} \in \mathit{Dat}$:

$$(\mathit{nspgm}\ [\mathit{req}] \underset{M}{\overset{*}{\Rightarrow}} \mathit{ans}) \Longleftrightarrow (\mathit{inv\text{-}mod}\ [\mathit{intN}, \mathit{pgm}, \mathit{req}] \underset{M}{\overset{*}{\Rightarrow}} \mathit{ans})\ .$$

Non-standard interpretation tools:		
$\textit{nsint} = \textit{pe}[\textit{sem-modL}^{SDD}, \textit{intN}]$	*S-non-standard N-interpreter*	(i1)
$\textit{g-nsint} = \textit{pe}[\textit{pe}^{SD}, \textit{sem-modL}^{SDD}]$	*Generator of S-non-standard interpreters*	(i2)
$\textit{gg-nsint} = \textit{pe}[\textit{pe}^{SD}, \textit{pe}^{SD}]$	*Generator of generators of ns-interpreters*	(i3)
Non-standard compilation tools:		
$\textit{nspgm} = \textit{pe}[\textit{sem-modL}^{SSD}, \textit{intN}, \textit{pgm}]$	*S-non-standard N→M-compilation*	(c1)
$\textit{nscomp} = \textit{pe}[\textit{pe}^{SSD}, \textit{sem-modL}^{SSD}, \textit{intN}]$	*S-non-standard N→M-compiler*	(c2)
$\textit{g-nscomp} = \textit{pe}[\textit{pe}^{SSD}, \textit{pe}^{SSD}, \textit{sem-modL}^{SSD}]$	*Generator of S-non-standard compilers*	(c3)
$\textit{gg-nscomp} = \textit{pe}[\textit{pe}^{SSD}, \textit{pe}^{SSD}, \textit{pe}^{SSD}]$	*Generator of generators of ns-compilers*	(c4)

Fig. 10. Seven modifier projections for non-standard tools.

Non-standard interpretation:

$$\begin{aligned} \textit{sem-modL}\ [\textit{intN}, \textit{pgm}, \textit{req}] &= \textit{nsint}\ [\textit{pgm}, \textit{req}] \\ &= \textit{g-nsint}\ [\textit{intN}]\ [\textit{pgm}, \textit{req}] \\ &= \textit{gg-nsint}\ [\textit{sem-modL}^{SDD}]\ [\textit{intN}]\ [\textit{pgm}, \textit{req}] \end{aligned}$$

Non-standard compilation:

$$\begin{aligned} \textit{sem-modL}\ [\textit{intN}, \textit{pgm}, \textit{req}] &= \textit{nspgm}\ [\textit{req}] \\ &= \textit{nscomp}\ [\textit{pgm}]\ [\textit{req}] \\ &= \textit{g-nscomp}\ [\textit{intN}]\ [\textit{pgm}]\ [\textit{req}] \\ &= \textit{gg-nscomp}\ [\textit{sem-modL}^{SSD}]\ [\textit{intN}]\ [\textit{pgm}]\ [\textit{req}] \end{aligned}$$

Fig. 11. Equations for applying the non-standard tools.

Program *nspgm* performs inverse computation given a request *req*. It runs *pgm*'s computation backwards. Therefore we have good reasons to call *nspgm* the *inverse program* of *pgm*, and to refer to the non-standard compilation of *pgm* as *inverse compilation* [1]. We expect that program *nspgm* is an efficient implementation of the inverse program. This provides a novel connection between inverse computation and program inversion. To summarize, inverse compilation implies two operations:

1. inversion of the function of program *pgm*;
2. representation of the inverted function by an *M*-program *nspgm*.

Non-standard compilation for an arbitrary semantics modifier *sem-modL*:

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

□

The modifier projections in Figure 10 show what can be achieved by program specialization in one transformation step. In some cases there exist different ways to obtain the same (functionally equivalent) programs.

For example, non-standard compilation (c1) can also be achieved in two steps. First by converting a standard interpreter *intN* into a non-standard interpreter *nsint* using (i1), and then specializing the non-standard interpreter *nsint* wrt program *pgm*. The result is a non-standard program *nspgm*′.

$$\left.\begin{array}{l} 1.\ pe\ [\textit{sem-modL}^{SDD}, \textit{intN}] \overset{*}{\Rightarrow} \textit{nsint} \\ 2.\ pe\ [\textit{nsint}^{SD}, \textit{pgm}] \overset{*}{\Rightarrow} \textit{nspgm}' \end{array}\right\} \textit{non-standard compilation}\ (\mathrm{c1}')$$

Another example is the generation of a non-standard compiler (c2) in two steps. First by converting a standard interpreter *intN* into a non-standard interpreter *nsint* using (i1), and then generating a non-standard compiler *nscomp*′ from *nsint* by self-application of specializer *pe*.

$$\left.\begin{array}{l} 1.\ pe\ [\textit{sem-modL}^{SDD}, \textit{intN}] \overset{*}{\Rightarrow} \textit{nsint} \\ 2.\ pe\ [pe^{SD}, \textit{nsint}^{SD}] \overset{*}{\Rightarrow} \textit{nscomp}' \end{array}\right\} \textit{non-standard compiler}\ (\mathrm{c2}')$$

13.3. Comparison of Modifier Projections and Futamura Projections

The Futamura projections [20] tell us how to convert an interpreter into a compiler by self-application of a program specializer. Formally speaking, the modifier projections (i1–i3) use the same specialization pattern as the Futamura projections, except that modifier *sem-modL* is used instead of an interpreter, and interpreter *intN* instead of a source program. From this perspective, the modifier projections (i1–i3) can be viewed as extending the Futamura projections by adding a new application, namely the conversion of a standard interpreter into a non-standard interpreter by specialization of a semantics modifier (e.g., transforming a standard interpreter into an inverse interpreter!). This application may not be surprising since the Futamura projections, reduced to their essence, define the conversion of arbitrary programs (interpreters, semantics modifiers, etc.) into *two-level generating extensions* (compilers, non-standard interpreters, etc.).

In our framework, we can recover the classical Futamura projections [20] from projections (i1–i3) by choosing a translation modifier (Sec. 6.2), and the specializer projections [22] by choosing a specialization modifier (Sec. 6.2). In the first case only *pgm* is present and request *req* is empty; in the second case *pgm* is present and request *req* contains *x*, a part of *pgm*'s input. Consequently, in our terminology, a compiler is a non-standard interpreter for translation semantics, and a specializer is a non-standard interpreter for specialization semantics.

Modifier projections (c1–c4) do *not* fit the specialization pattern of the Futamura projections. They involve the generation of a *three-level generating extension* which requires double self-application (c3), or a three-level compiler generator such as *gg-nscomp* produced by triple self-application (c4). It is not possible to produce a three-level generating extension directly by a classical (two-level) compiler generator, such as *gg-nsint* (i3) (this would require *incremental generation* in two or more steps as explained in [25]). Clearly, *g-nsint* (i2) is Ershov's classical (two-level) generating extension [19], while *g-nscomp* (c3) is a three-level generating extension. Both programs are examples of *multi-level generating extensions* [25].

Projections (i1–i3) and (c1–c4) tell us how non-standard interpreters and non-standard compilers can be generated from semantics modifiers. To stress the importance of these transformations, and because we believe that these projections pose new challenges for program specialization, for example the generation of inverse compilers, and because they suggest new ways of using program specialization by abstracting from the semantics under consideration, we call these seven projections collectively *modifier projections*.

14. Related Work

This section relates some of the existing work to semantics modifiers, including translation, specialization and inversion, and reviews application of program specialization. Because this work relates to theoretical and practical results in different research areas, the exposition here will likely be incomplete. Also, some related work has been discussed in the respective sections above.

Interpreters are popular in many areas of meta-programming, and have been used for the development of programming languages since the sixties [18, 38]. For example, meta-interpreters are used extensively in logic programming for instrumenting programs, and for providing powerful ways for reasoning [6, 47]. However, these applications are usually restricted to changing inference rules of the underlying logic system and, in general, do not attempt the radical semantics changes possible with semantics modifiers.

Instead of just providing the means for implementing standard semantics of programming languages, such as interpreter hierarchies [18], semantics modifiers provide a new formalism for implementing a large family of non-standard semantics in a very systematic manner. Some of these ideas have been used earlier, but have not been identified as instances of a more general scheme. For example, the well-known interpreter hierarchies, which are often used for porting programming languages, are nothing but a special case of semantics modification (identity modifier).

The Futamura projections [20] can be seen an early example of semantics modification, namely the assertion that a translation semantics can be given to any language via an interpreter. A similar assertion is expressed by the specializer projections [22] which state that a specialization semantics can be given to any language via an interpreter. Both projections have been put to work using partial evaluation (first in [32] and [23]). The results in this paper show that these ideas can be extended to all equivalence transformers.

The possibility of inverse computation by inverting a standard interpreter of a language was found in [1]. A logic programming system, say Prolog, can be considered [4] as an implementation of inverse computation (it allows, depending on the query, to run programs forwards and backwards [36]). The only other work in this direction we are aware of, inverses imperative programs by treating their relational semantics as logic program [43]. We have seen that this application is a particular instance of a more general scheme which is independent of a particular language paradigm. In this paper (and in earlier work [4]), inverse computation was successfully ported to imperative languages using URA [50, 3, 5, 42], an algorithm for inverse computation in a functional language.

The major problem of running programs interpretively is that each new level of interpretation multiplies the run time of the interpreted program by a significant factor. Much work has been done to reduce the costs of interpretation using various methods of program specialization (see [9, 31, 15]). Partial deduction, a form of program specialization, has been used to reduce the interpretive overhead of meta-interpreters, e.g. [48]. Considerable success has been achieved and the refinement of these methods is an ongoing effort [11, 37, 55]. Offline partial evaluation is well-known for its power to substantially reduce interpretive overhead, e.g. [10, 12, 33, 34], and has been used successfully to optimize two levels of interpretation, e.g. [35]. Recent work [13] applies offline partial evaluation to interpreters of domain-specific languages written in C. These applications may also be viewed as working examples of translation modifiers.

All three specializer projections have been supported by computer experiments, e.g. [23, 24, 46]. Among others, they have been used to bootstrap program transformers for non-trivial language extensions (e.g., transforming programs written in a higher-order language by a transformer for a first-order language [46]).

While substantial practical evidence exists for equivalence-transformation modifiers, not much work has been done regarding inversion modifiers. Experiments [40, 53] show that specializing an interpreter for inverse computation can invert programs. Only a very small number of publications studies the problem of program inversion [27, 28, 41, 42, 54]. The modifier projections suggest a new way of constructing program inverters, namely by transforming an inverse interpreter.

Inserting an interpreter between a source program and a program transformer can achieve certain powerful transformations [34, 49, 53]. Our use of the interpretive approach is different from this work: our primary goal is not to expose more information to a program transformer by instrumenting a standard interpreter, but rather to combine a standard interpreter with different semantics modifiers (however, both uses of interpreters may be combined).

Reflective languages and continuation semantics have been suggested for modifying and controlling computations at different meta-levels, (e.g. [7, 14, 35, 56]); more should be known about their relation to semantics modifiers.

15. Conclusion and Future Work

In this paper we investigated semantics modifiers starting from theoretical considerations to an experimental assessment. We specified the class of semantics modifiers, proved the modifier property of several non-standard semantics, showed that constructive definitions exist, performed experiments porting semantics from a functional to an imperative language, and showed how, in principle, efficient implementations can be obtained by program specialization or other advanced methods for automatic program transformation.

We explored semantics modifiers ranging from inverse computation to equivalence transformation and demonstrated that non-trivial algorithms exist in each case. This suggests that a large class of computational problems, which superficially seem very different, can be studied in a language-independent and uniform manner. This is noteworthy because designing and implementing non-standard semantics, such as equivalence transformers and inverse interpreters, is far from trivial.

To summarize, the approach presented above allows to

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

It is clear that several challenging problems lie ahead regarding further theoretical and practical ramifications of this approach. The modifier projections are promising, but have not been fully implemented, although partial solutions exist. Their implementation may require further improvements of program specializers and other program transformers. The theoretical constructions exhibit the underlying metasystem structure of the problems, but assert nothing about the quality of the obtained results. Another challenging task is the development of structured methods for constructing semantics modifiers from a combination of existing 'atomic' modifiers. The proofs of the modifier theorems seem to follow a certain pattern. This suggests there might be a meta-theorem. These are some of the topics for future work on semantics modifiers.

Acknowledgments

The authors are very grateful to Valentin Turchin for introducing them to supercompilation and the principles of metacomputation. Special thanks are due to Neil D. Jones and the anonymous referees for providing many valuable comments on an earlier version of this paper. Thanks to Kazuhiko Kakehi for careful proofreading. The authors would like to thank their colleagues from the Refal group and the Topps group for stimulating discussions on various topics related to this work.

References

1. S. M. Abramov, "Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming)," *Programmirovanie* **3** (1991) 31–44 in Russian.
2. S. M. Abramov, "Metacomputation and program testing," in *1st International Workshop on Automated and Algorithmic Debugging*, Linköping University, Sweden, 1993, pp. 121–135.
3. S. M. Abramov, *Metavychislenija i ikh prilozhenija (Metacomputation and its applications)* (Nauka-Fizmatlit, Moscow, 1995) in Russian.
4. S. M. Abramov and R. Glück, "Semantics modifiers: an approach to non-standard semantics of programming languages," in *Third Fuji International Symposium on Functional and Logic Programming*, eds. M. Sato and Y. Toyama (World Scientific, 1998) pp. 247–270.
5. S. M. Abramov and R. Glück, "The universal resolving algorithm: inverse computation in a functional language," in *Mathematics of Program Construction. Proceedings*, eds. R. Backhouse and J. Oliveira, *LNCS* (Springer-Verlag, 2000) to appear.
6. K. Apt and F. Turini, *Meta-Logics and Logic Programming* (MIT Press, Cambridge, Massachusetts, 1995).

7. K. Asai, S. Matsuoka and A. Yonezawa, "Duplication and partial evaluation—for a better understanding of reflective languages," *Lisp and Symbolic Computation* **9(2&3)** (1996) 203–241.
8. R. Bird, *Programs and Machines* (John Wiley & Sons, London, 1976).
9. D. Bjørner, A. P. Ershov and N. D. Jones, eds., *Partial Evaluation and Mixed Computation,* (North-Holland, Amsterdam, 1988).
10. A. Bondorf and J. Palsberg, "Generating action compilers by partial evaluation," *Journal of Functional Programming* **6(2)** (1996) 269–298.
11. A. F. Bowers and C. A. Gurr, "Towards fast and declarative meta-programming," in [6], pp. 137–166.
12. C. Consel, "New insights into partial evaluation: the Schism experiment," in *ESOP'88,* ed. H. Ganzinger, *LNCS* **300** (Springer-Verlag, 1988) 236–247.
13. C. Consel and R. Marlet, "Architecture software using: a methodology for language development," in *Principles of Declarative Programming. Proceedings*, eds. C. Palamidessi, H. Glaser and K. Meinke, *LNCS* **1490** (Springer-Verlag, 1998) 170–194.
14. O. Danvy, "Across the bridge between reflection and partial evaluation," in [9], pp. 83–116.
15. O. Danvy, R. Glück and P. Thiemann, eds., *Partial Evaluation. Proceedings*, *LNCS* **724** (Springer-Verlag, 1996).
16. J. Darlington, "An experimental program transformation and synthesis system," *Artificial Intelligence* **16(1)** (1981) 1–46.
17. N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in *Handbook of Theoretical Computer Science*, ed. J. v. Leeuwen (Elsevier, 1990) pp. 244–320.
18. J. Earley and H. Sturgis, "A formalism for translator interactions," *Communications of the ACM* **13(10)** (1970) 607–617.
19. A. P. Ershov, "On the essence of compilation," in *Formal Description of Programming Concepts*, ed. E. Neuhold (North-Holland, Amsterdam, 1978), pp. 391–420.
20. Y. Futamura, "Partial evaluation of computing process – an approach to a compiler-compiler," *Systems, Computers, Controls* **2(5)** (1971) 45–50.
21. Y. Futamura, K. Nogi and A. Takano, "Essence of generalized partial computation," *Theoretical Computer Science* **90(1)** (1991) 61–79.
22. R. Glück, "On the generation of specializers," *Journal of Functional Programming* **4(4)** (1994) 499–514.
23. R. Glück and J. Jørgensen, "Generating optimizing specializers," in *IEEE International Conference on Computer Languages* (IEEE Computer Society Press, 1994) pp. 183–194.
24. R. Glück and J. Jørgensen, "Generating transformers for deforestation and supercompilation," in *Static Analysis. Proceedings*, ed. B. Le Charlier, *LNCS* **864** (Springer-Verlag, 1994) 432–448.
25. R. Glück and J. Jørgensen, "An automatic program generator for multi-level specialization," *Lisp and Symbolic Computation* **10(2)** (1997) 113–158.
26. R. Glück and A. V. Klimov, "Occam's razor in metacomputation: the notion of a perfect process tree," in *Static Analysis. Proceedings*, eds. P. Cousot, M. Falaschi, G. Filé and A. Rauzy, *LNCS* **724** (Springer-Verlag, 1993) 112–123.
27. D. Gries, *The Science of Programming*, Texts and Monographs in Computer Science (Springer-Verlag, New York, 1981).

28. P. G. Harrison, "Function inversion," in [9], pp. 153–166.
29. J. Hatcliff and R. Glück, "Reasoning about hierarchies of online specialization systems," in [15], pp. 161–182.
30. P. Hudak, et al., "Report on the programming language Haskell, a non-strict purely functional language," *SIGPLAN Notices* **27(5)** (1992) R1–R164.
31. N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation* (Prentice-Hall, 1993).
32. N. D. Jones, P. Sestoft and H. Søndergaard, "An experiment in partial evaluation: the generation of a compiler generator," in *Rewriting Techniques and Applications*, ed. J.-P. Jouannaud, *LNCS* **202** (Springer-Verlag, 1985) 124–140.
33. N. D. Jones, P. Sestoft and H. Søndergaard, "Mix: a self-applicable partial evaluator for experiments in compiler generation," *LISP and Symbolic Computation* **2(1)** (1989) 9–50.
34. J. Jørgensen, "Generating a compiler for a lazy language by partial evaluation," in *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1992) pp. 258–268.
35. A. Kishon and P. Hudak, "Semantics directed program execution monitoring," *Journal of Functional Programming* **5(4)** (1995) 501–547.
36. R. Kowalski, "Algorithm = logic + control," *Communications of the ACM* **22(7)** (1979) 424–436.
37. M. Leuschel, "On the power of homeomorphic embedding for online termination," in *Static Analysis. Proceedings*, ed. G. Levi, *LNCS* **1503** (Springer-Verlag, 1998) 230–245.
38. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin, *Lisp 1.5 Programmer's Manual* (MIT Press, Cambridge, Massachusetts, 1962) 2nd edition (1965).
39. T. Æ. Mogensen, "Partially static structures in a self-applicable partial evaluator," in [9], pp. 325–347.
40. A. P. Nemytykh and V. A. Pinchuk, "Program transformation with metasystem transitions: experiments with a supercompiler," in *Perspectives of System Informatics. Proceedings*, eds. D. Bjørner, M. Broy and I. V. Pottosin, *LNCS* **1181** (Springer-Verlag, 1996) 249–260.
41. A. Y. Romanenko, "The generation of inverse functions in Refal," In [9], pp. 427–444.
42. A. Y. Romanenko, "Inversion and metacomputation," in *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (ACM Press, 1991) pp. 12–22.
43. B. J. Ross, "Running programs backwards: the logical inversion of imperative computation," *Formal Aspects of Computing* **9** (1997) 331–348.
44. P. Sestoft, "The structure of a self-applicable partial evaluator," Technical Report 85/11, DIKU, University of Copenhagen, Denmark, Nov. 1985.
45. M. H. Sørensen, R. Glück and N. D. Jones, "A positive supercompiler," *Journal of Functional Programming* **6(6)** (1996) 811–838.
46. M. Sperber, R. Glück and P. Thiemann, "Bootstrapping higher-order program transformers from interpreters," in *Proceedings of the 1996 ACM Symposium on Applied Computing*, eds. K. M. George, J. H. Carroll, D. Oppenheim and J. Hightower (ACM Press, 1996) pp. 408–413.

47. L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, Cambridge, Massachusetts, 1986).
48. A. Takeuchi, "Affinity between meta interpreters and partial evaluation," in *Information Processing 86*, ed. H.-J. Kugler (Elsevier Science Publishers, 1986) pp. 279–282.
49. P. Thiemann and M. Sperber, "Polyvariant expansion and compiler generators," in *Perspectives of System Informatics. Proceedings*, eds. D. Bjørner, M. Broy and I. V. Pottosin, *LNCS* **1181** (Springer-Verlag, 1996) 285–296.
50. V. F. Turchin, "Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal)," in *Teorija Jazykov i Metody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods)*, Kiev-Alushta, USSR, 1972, pp. 31–42, in Russian.
51. V. F. Turchin, "The concept of a supercompiler," *Transactions on Programming Languages and Systems* **8(3)** (1986) 292–325.
52. V. F. Turchin, "The algorithm of generalization in the supercompiler," in [9], pp. 531–549.
53. V. F. Turchin, "Program transformation with metasystem transitions," *Journal of Functional Programming* **3(3)** (1993) 283–313.
54. J. L. A. van de Snepscheut, *What computing is all about* (Springer-Verlag, 1993).
55. W. Vanhoof and B. Martens, "To parse or not to parse," in *Logic Program Synthesis and Transformation.*, ed. N. Fuchs, *LNCS* **1463** (Springer-Verlag, 1998) 314–333.
56. M. Wand and D. P. Friedman, "The mystery of the tower revealed: a nonreflective description of the reflective tower," *Lisp and Symbolic Computation* **1(1)** (1988) 11–37.