Combining Semantics with Non-standard Interpreter Hierarchies

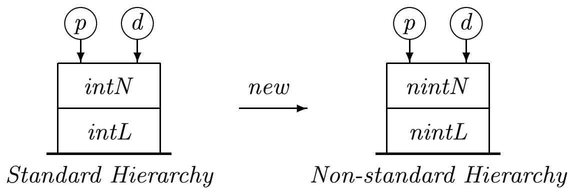
Sergei Abramov[1] and Robert Glück[2]*

[1] Program Systems Institute, Russian Academy of Sciences
RU-152140 Pereslavl-Zalessky, Russia, `abram@botik.ru`
[2] PRESTO, JST, Institute for Software Production Technology
Waseda University, Tokyo 169-8555, Japan, `glueck@acm.org`

Abstract. This paper reports on results concerning the combination of non-standard semantics via interpreters. We define what a semantics combination means and identify under which conditions a combination can be realized by computer programs (robustness, safely combinable). We develop the underlying mathematical theory and examine the meaning of several non-standard interpreter towers. Our results suggest a technique for the implementation of a certain class of programming language dialects by composing a hierarchy of non-standard interpreters.

1 Introduction

The definition of programming language semantics from simpler, more elementary parts is an intriguing question [6, 11, 17, 18]. This paper reports on new results concerning the combination of semantics via non-standard interpreters. Instead of using the familiar tower of interpreters [13] for implementing the standard semantics of a programming language, we generalize this idea to implement the non-standard semantics of a programming language by combining one or more non-standard interpreters.



The essence of the *interpreter tower* is to evaluate an N-interpreter *intN* written in L by an L-interpreter *intL* written in some ground language. This means, we give standard semantics to N-programs via L's standard semantics. But what does it mean to build a tower involving one or more non-standard interpreters? For example, what does it mean for the semantics of an N-program p if we replace interpreter *intL* by an *inverse*-interpreter *nintL*?

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

A formal answer to this question and related *non-standard towers* will be given in this paper. Using the mathematical foundations developed here some well-known results about the combination of standard interpreters are shown, as well as new results about the combination of non-standard semantics. This extends our previous work on semantics modifiers [1] and inverse computation [2] where we have observed that some non-standard semantics can be ported via standard interpreters. We can now formalize a class of non-standard semantics that can serve as semantics modifiers, and reason about new non-standard combinations, some of which look potentially useful. We focus on deterministic programming languages as an important case for practice. Since this includes universal programming languages, there is no loss of generality. Extending our results to other computation models can be considered for future work.

In practice, the implementation of a non-standard tower will be inefficient because each level of interpretation adds extra computational overhead. To improve efficiency we assume program specialization techniques. Program specialization, or *partial evaluation* [9, 5], was shown to be powerful enough to collapse towers of standard interpreters and to drastically reduce their interpretive overhead. We believe powerful program transformation tools will enable us in the future to combine non-standard interpreters with less concern about efficiency, which may make this approach more practical for the construction of software.

Finally, note that we use the term ‘programming language’ in a broad sense, that is, not only for *universal* programming languages, such as Fortran, C or ML, but also for *domain-specific* languages (*e.g.*, networks, graphics), and for languages which are *computationally incomplete* (*e.g.*, regular grammars). This means, potentially our results apply to a broad spectrum of application areas.

The main contributions of this paper are: (i) *a mathematical foundation for a theory about semantics combination*: we define what semantics combinations mean and we identify several theoretical combinations; (ii) *an approach to implementing programming language dialects* by building towers of non-standard interpreters and their correctness; (iii) *explaining the essence of known results* such as interpreter towers and the Futamura projections [7], and giving *novel insights* regarding the semantics modification of programming languages. Proofs are omitted due to space limitations.

2 Foundations for Languages and Semantics

Before introducing languages and semantics we give two preliminary definitions. We define a *projection* $(A.b)$ to form new sets given a set of tuples A and an element b, and a *preserving set definedness* relation $(A \Subset B)$ which requires $A \subseteq B$ and A to be non-empty when B is non-empty.

Definition 1 (projection). *Let A, B, C be sets, let $A \subseteq B \times C$, and let $b \in B$, then we define* projection $A.b \overset{\text{def}}{=} \{\, c' \mid (b', c') \in A, b' = b \,\}$.

Example 1. Let $A = \{(2, 3, 1), (5, 6, 7), (2, 4, 1)\}$ then $A.2 = \{(3, 1), (4, 1)\}$.

Definition 2 (preserving set definedness). *Let A, B be sets, then we define relation* preserving set definedness $(A \mathrel{\underline{⪿}} B) \overset{\text{def}}{\Longleftrightarrow} ((A \subseteq B) \wedge (B \neq \emptyset \Rightarrow A \neq \emptyset))$.

We define languages, semantics and functional equivalence using a relational approach. When we speak of languages we mean formal languages. As is customary, we use the same universal data domain (D) for all languages and for representing all programs. Mappings between different data domains and different program representations are straightforward to define and not essential for our discussion.

The reader should be aware of the difference between the abstract language definitions given in this section, which may be non-constructive, and the definitions for programming languages in Sect. 3 which are constructive. The formalization is geared towards the definition of deterministic programming languages.

Definition 3 (language). *A* language *L is a triple $L = (P_L, D_L, \llbracket\ \rrbracket_L)$, where $P_L \subseteq D$ is the set of L-programs, $D_L = D$ is the data domain for L, and $\llbracket\ \rrbracket_L$ is the semantics of L: $\llbracket\ \rrbracket_L \subseteq P_L \times D \times D$. We denote by $\mathcal{L}$ the* set of all languages.

Definition 4 (program semantics, application). *Let $L = (P_L, D, \llbracket\ \rrbracket_L)$ be a language, let $p \in P_L$ be an L-program, let $d \in D$ be data, then the* semantics of *p is defined by $\llbracket\ \rrbracket_L.p \subseteq D \times D$, and the* application of *p* to *d by $\llbracket\ \rrbracket_L.p.d \subseteq D$.*

Definition 5 (functional equivalence). *Let $L_1 = (P_{L_1}, D, \llbracket\ \rrbracket_{L_1})$ and $L_2 = (P_{L_2}, D, \llbracket\ \rrbracket_{L_2})$ be languages, let $p_1 \in P_{L_1}$ and $p_2 \in P_{L_2}$ be programs, then p_1* and *p_2* are functionally equivalent *iff $\llbracket\ \rrbracket_{L_1}.p_1 = \llbracket\ \rrbracket_{L_2}.p_2$.*

Note that we defined the semantics $\llbracket\ \rrbracket_L$ of a language as a relation $(P_L \times D \times D)$. For convenience, we will sometimes use notation $\llbracket p \rrbracket_L\, d$ for *application* $\llbracket\ \rrbracket_L.p.d$, and notation $\llbracket p \rrbracket_L$ for *program semantics* $\llbracket\ \rrbracket_L.p$. As Def. 4 shows, the result of an application is always a set of data, and we can distinguish three cases:

$\llbracket p \rrbracket_L\, d = \emptyset$	— application *undefined*,
$\llbracket p \rrbracket_L\, d = \{a\}$	— application *defined* (deterministic case),
$\llbracket p \rrbracket_L\, d = \{a_1, a_2, \ldots\}$	— application *defined* (non-deterministic case).

Definition 6 (deterministic language). *A language $L = (P_L, D, \llbracket\ \rrbracket_L)$ is* deterministic *iff* $\forall (p_1, d_1, a_1), (p_2, d_2, a_2) \in \llbracket\ \rrbracket_L :\ (p_1 = p_2 \wedge d_1 = d_2) \Rightarrow (a_1 = a_2)$. *We denote by $\mathcal{D}$ the* set of all deterministic languages $(\mathcal{D} \subseteq \mathcal{L})$.

Relations $\subseteq$ and $\mathrel{\underline{⪿}}$ have a clear meaning for application: $\llbracket p \rrbracket_{L_1} d \subseteq \llbracket p \rrbracket_{L_2} d$ tells us that the left application may be undefined even when the right application is defined (*definedness is not preserved*); $\llbracket p \rrbracket_{L_1} d \mathrel{\underline{⪿}} \llbracket p \rrbracket_{L_2} d$ tells us that both applications are either defined or undefined (*definedness is preserved*). In Def. 7 we use $\mathrel{\underline{⪿}}$ to define a definedness preserving relation between semantics ($\mathrel{\underline{\subset\!\!\!\bullet}}$).

Definition 7 (preserving semantics definedness). *Let $L_1 = (P_{L_1}, D, \llbracket\,\rrbracket_{L_1})$ and $L_2 = (P_{L_2}, D, \llbracket\,\rrbracket_{L_2})$ be languages such that $P_{L_1} = P_{L_2}$, then we define relation* preserving semantics definedness ($\mathrel{\underline{\subset\!\!\!\bullet}}$) *as follows:*

$$(\llbracket\ \rrbracket_{L_1} \mathrel{\underline{\subset\!\!\!\bullet}} \llbracket\ \rrbracket_{L_2}) \overset{\text{def}}{\Longleftrightarrow} (\forall p \in P_{L_1}\ \forall d \in D\colon\ \llbracket\ \rrbracket_{L_1}.p.d \mathrel{\underline{⪿}} \llbracket\ \rrbracket_{L_2}.p.d)\ .$$

2.1 Semantics Properties and Language Dialects

A *property* S is a central concept for the foundations of non-standard semantics. It specifies a modification of the standard semantics of a language. When we speak of an S-*dialect* L' of a language L, then the relation of input/output of all L-programs applied under L' must satisfy property S. For example, we require that the output of applying an L-program under an *inverse*-dialect L' of L [2] is a possible input of that program applied under L's standard semantics. Given a *request* r for S-computation, there may be infinitely many *answers* a that satisfy property S.[1] We consider each of them as a correct *wrt* S.

A property describes a semantics modification for a set of languages. The specification can be non-constructive and non-deterministic. We specify a property S for a set of languages as a set of tuples (L, p, r, a). We say a language L' is an S-dialect of L if both languages have the same syntax, and the semantics of L' is a subset of $S.L$. We define three types of dialects that can be derived from S. Later in Sect. 3 we consider only those dialects that are constructive.

Definition 8 (property). *Let $\mathcal{N} \subseteq \mathcal{L}$, then set S* is a property for $\mathcal{N}$ *iff*

$$S \subseteq \bigcup_{L \in \mathcal{N}} \{L\} \times P_L \times D \times D \ .$$

Example 2 (properties). Let $\mathcal{N} = \mathcal{L}$ and $R \in \mathcal{L}$, then *Id*, *Inv*, *Trans*$_R$ and *Copy* are properties for $\mathcal{L}$, namely identity, inversion, translation, and copying of programs. Other, more sophisticated properties may be defined that way.

$$\begin{aligned}
\mathit{Id} &\stackrel{\text{def}}{=} \{\, (L,p,r,a) \ \mid \ L \in \mathcal{L}, p \in P_L, r \in D, a \in [\![p]\!]_L\, r \,\} \\
\mathit{Inv} &\stackrel{\text{def}}{=} \{\, (L,p,r,a) \ \mid \ L \in \mathcal{L}, p \in P_L, a \in D, r \in [\![p]\!]_L\, a \,\} \\
\mathit{Trans}_R &\stackrel{\text{def}}{=} \{\, (L,p,r,p') \mid L \in \mathcal{L}, p \in P_L, r \in D, p' \in P_R : [\![p]\!]_L = [\![p']\!]_R \,\} \\
\mathit{Copy} &\stackrel{\text{def}}{=} \{\, (L,p,r,p) \ \mid \ L \in \mathcal{L}, p \in P_L, r \in D \,\}
\end{aligned}$$

Definition 9 (dialects). *Let S be a property for $\mathcal{N}$, let $L \in \mathcal{N}$, then $S.L \subseteq P_L \times D \times D$ is the* most general S-semantics for L. *Let $L = (P_L, D, [\![\ ]\!]_L)$, then a language $L' = (P_L, D, [\![\ ]\!]_{L'}) \in \mathcal{L}$ is (i) the* most general S-dialect of L *iff* $[\![\ ]\!]_{L'} = S.L$, *(ii) an* S-dialect of L *iff* $[\![\ ]\!]_{L'} \overset{\bullet}{\subseteq} S.L$, *and (iii) an* S-semi-dialect of L *iff* $[\![\ ]\!]_{L'} \subseteq S.L$. *We denote by $S|L$ the* most general S-dialect of L *and by $\mathcal{D}_{S|L}$ the* set of all deterministic S-dialects of L.

The most general S-semantics $S.L$ specifies all correct answers for an application $S.L.p.r$ given S, L, p, r. In general, the most general dialect $S|L$ of a language L will be non-deterministic. This allows the definition of different S-dialects for L.

[1] When we talk about non-standard semantics, we use the terms *request* and *answer* to distinguish them from input and output of standard computation.

Example 3 (dialects). There are usually infinitely many deterministic and non-deterministic *Inv*-dialects of L (they differ in which and how many inverse answers they return). For property *Copy*, the most general dialect $\mathit{Copy}|L$ is always deterministic and there exists only one *Copy*-dialect for each L. Another example is property *Id*. If L is non-deterministic, then there are usually infinitely many deterministic and non-deterministic *Id*-dialects. But if L is deterministic, then there exists only *one* deterministic *Id*-dialect L' and $L' = \mathit{Id}|L = L$.

Definition 10 (robust property). *Let $\mathcal{N}$ be a set of languages, let S be a property for $\mathcal{N}$, then S* is robust *iff all functionally equivalent programs are also functionally equivalent under the most general S-dialect:*

$$\forall L_1, L_2 \in \mathcal{N}\ \forall p_1 \in P_{L_1}\ \forall p_2 \in P_{L_2} : (\llbracket p_1 \rrbracket_{L_1} = \llbracket p_2 \rrbracket_{L_2}) \Rightarrow (\llbracket p_1 \rrbracket_{S|L_1} = \llbracket p_2 \rrbracket_{S|L_2}) \ .$$

Example 4 (robustness). All properties in Ex. 2 are robust ($\mathit{Id}, \mathit{Inv}, \mathit{Trans}_R$), except *Copy*, which returns different results for fct. equivalent programs $p \neq p'$.

The motivation for defining *robustness* is that we are mainly interested in a class of properties that can be combined by interpreters. When we use a robust property S we cannot distinguish by the semantics of the most general dialect $S|L$ two programs which are functionally equivalent under L's standard semantics. A robust property specifies an extensional modification of a language semantics which is independent of the particular operational features of a program.

2.2 Combining Properties

Two properties S' and S'' can be combined into a new property $S' \circ S''$. Intuitively speaking, one gets an ($S' \circ S''$)-dialect of a language L by taking an S'-dialect of an S''-dialect of L. This combination is captured by projection $S'.L''.p.r$ in the following definition. The reason for choosing language L'' from the set of deterministic S''-dialects $\mathcal{D}_{S''|L}$ of L is that later we will use deterministic programming languages for implementing property combinations.

Definition 11 (combination). *Let S', S'' be properties for $\mathcal{D}$, then we define*

$$S' \circ S'' \overset{\text{def}}{=} \{(L, p, r, a) \mid L \in \mathcal{D}, p \in P_L, r \in D, a \in D, L'' \in \mathcal{D}_{S''|L}, a \in S'.L''.p.r\} \, .$$

Example 5 (combination). Let S be a property for $\mathcal{D}$, then some of the combinations of the properties in Example 2 are as follows:

$\boxed{S \circ \mathit{Id} = S}$: Right combination with identity does not change property S.

$\boxed{\mathit{Id} \circ S = S}$: Left combination with identity does not change property S.

$\boxed{\mathit{Trans}_R \circ S = S\_\mathit{Trans}_R}$: S-translation to R (will be explained in Sec. 4.3).

$\boxed{\mathit{Inv} \circ S = S^{-1}}$: Inversion of property S (will be explained in Sec. 4.4).

$\boxed{\mathit{Copy} \circ S = \mathit{Copy}}$: "Left zero" for property S.

In addition, we are interested in combinations $(S'{\circ}S'')$ that guarantee that all applications $S'.L''.p.r$ are defined for the same set of program-request pairs (p, r) regardless which deterministic S''-dialect L'' we select for L. This requires that S' and S'' satisfy the condition given in the following definition. In this case we say, S' and S'' are *safely combinable* $(S' \bowtie S'')$.

Definition 12 (safely combinable). *Let S', S'' be properties for $\mathcal{D}$, then S'* is safely combinable with *S'' $(S' \bowtie S'')$ iff*

$$\forall L \in \mathcal{D},\ \forall L''_1, L''_2 \in \mathcal{D}_{S''|L},\ \forall p \in P_L,\ \forall d \in D : \\ (S'.L''_1.p.d \neq \emptyset) \Leftrightarrow (S'.L''_2.p.d \neq \emptyset)\ .$$

Example 6 (safely combinable). Let S', S'' be properties, and let S' be robust, then the following combinations are always safely combinable:
$\boxed{Id \bowtie S''}$, $\boxed{S' \bowtie Id}$, $\boxed{S' \bowtie Copy}$.

3 Programming Languages

We now turn to *programming languages*, and focus on *deterministic* programming languages as an important case for practice. Since this includes universal programming languages, there is no loss of generality. All computable functions can be expressed. First, we give definitions for programming languages and interpreters, then we introduce *non-standard interpreters* which we define as programs that implement non-standard dialects.

As before we assume a universal data domain D for programming languages, but require D to be constructive (recursively enumerable) and to be closed under tupling: $d_1, \ldots, d_k \in D \Rightarrow [\ d_1, \ldots,\ d_k\ ] \in D$. For instance, a suitable choice for D is the set of S-expressions familiar from Lisp [13]. Since we consider only deterministic programming languages, the result of an application is either a singleton set or the empty set.

Definition 13 (programming language). *A* programming language *L is a deterministic language $L = (P_L, D_L, [\![\ ]\!]_L)$ where $P_L \subseteq D$ is the recursively enumerable set of L-programs, $D_L = D$ is the recursively enumerable data domain for L, and $[\![\ ]\!]_L$ is the recursively enumerable semantics of L: $[\![\ ]\!]_L \subseteq P_L \times D \times D$. We denote by $\mathcal{P}$ the* set of all programming languages*.*

Definition 14 (interpreter). *Let $L = (P_L, D, [\![\ ]\!]_L)$, $M = (P_M, D, [\![\ ]\!]_M)$ be programming languages, then an M-program intL is an* interpreter for L in M *iff*

$$\forall p \in P_L,\ \forall d \in D :\ [\![intL]\!]_M\ [\ p, d\ ] = [\![p]\!]_L\ d\ .$$

Definition 15 (partially fixed argument). *Let $L = (P_L, D, [\![\ ]\!]_L)$ be a programming language, let $p, p' \in P_L$, and let $d_1 \in D$ such that*

$$\forall d_2 \in D :\ [\![p']\!]_L\ d_2 = [\![p]\!]_L\ [\ d_1, d_2\ ]\ .$$

If program p' exists we denote *it by "$[\, p, [\, d_1, \bullet\,]\,]$", and we have*

$$\forall d_2 \in D : [\![\, [\, p, [\, d_1, \bullet\,]\,]\,]\!]_L\, d_2 = [\![p]\!]_L\, [\, d_1, d_2\,]\ .$$

In a universal programming language we can always write program $[\, p, [\, d_1, \bullet\,]\,]$ given $p \in P_L$ and $d_1 \in D$ (this is similar to Kleene's S-m-n theorem). In a programming language that supports abstraction and application as in the lambda-calculus we can define: $[\, p, [\, d_1, \bullet\,]\,] \overset{\text{def}}{=} \lambda d_2.p\, [\, d_1, d_2\,]$.

Definition 16 (prog. lang. dialects). *Let $\mathcal{P}' \subseteq \mathcal{P}$, let S be a property for $\mathcal{P}'$, and let $L = (P_L, D, [\![\ ]\!]_L) \in \mathcal{P}'$, then a prog. language $L' = (P_L, D, [\![\ ]\!]_{L'}) \in \mathcal{P}$ is an S-*dialect *of L iff $[\![\ ]\!]_{L'} \overset{\bullet}{\subseteq} [\![\ ]\!]_{S|L}$, and an S-*semi-dialect *of L iff $[\![\ ]\!]_{L'} \subseteq [\![\ ]\!]_{S|L}$.*

Definition 17 ($S|L/M$-interpreter). *Let L, M be programming languages, let $\mathcal{P}' \subseteq \mathcal{P}$, let S be a property for $\mathcal{P}'$, and let $L \in \mathcal{P}'$, then an M-program nintL is an S-*interpreter for L in M ($S|L/M$-interpreter) *if there exists an S-dialect L' of L such that nintL is an interpreter for L' in M.*

An interpreter for a language L is an implementation of the standard semantics of L, while an S-interpreter is an implementation, if it exists, of an S-dialect L' of L. Since a property S may specify infinitely many S-dialects for L (see Sect. 2.1), we say that *any* program that implements one of these dialects is an S-interpreter.[2]

In general, not every non-standard S-dialect is computable. Some dialects may be undecidable, others (semi-)decidable. A non-standard interpreter *nintL* realizes an S-dialect for a given language L, and having *nintL* we can say that S can be realized constructively for L. If this is the case for two properties S' and S'', then $(S' \circ S'')$ can be implemented by a tower of non-standard interpreters.

4 Towers of Non-standard Interpreters

Definition 18 (non-standard tower). *Let $\mathcal{P}' \subseteq \mathcal{P}$, let $M \in \mathcal{P}$, let $N, L \in \mathcal{P}'$, let S', S'' be properties for $\mathcal{P}'$, let S' be robust, let M-program nintL$'$ be an S'-interpreter for L in M, let L-program nintN$''$ be an S''-interpreter for N in L, and let $p \in P_N$, $d \in D$, then a* non-standard tower *is defined by application*

$$[\![nintL']\!]_M\, [\, [\, nintN'', [\, p, \bullet\,]\,], d\,]\ .$$

Theorem 1 (correctness of non-standard tower). *Let $\mathcal{P}' \subseteq \mathcal{P}$, let $M \in \mathcal{P}$, let $N, L \in \mathcal{P}'$, let S', S'' be properties for $\mathcal{P}'$, let M-program nintL$'$ be an S'-interpreter for L in M, let L-program nintN$''$ be an S''-interpreter for N in L:*

– *If S' is robust then the following non-standard tower implements an $(S' \circ S'')$-*semi-dialect of N in M *(cf. Fig. 1):*

$$\forall p \in P_N,\ \forall d \in D : [\![nintL']\!]_M\, [\, [\, nintN'', [\, p, \bullet\,]\,], d\,] \subseteq (S' \circ S'').N.p.d\ .$$

[2] In general, deterministic programs cannot implement all S-dialects since some dialects may be non-deterministic (*e.g.*, *Inv*-dialects).

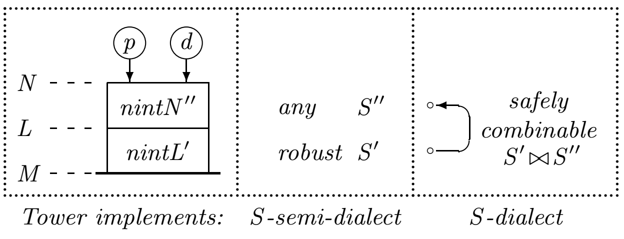


Fig. 1. Two-level non-standard tower

– *If S' is robust and safely combinable with S'' ($S' \bowtie S''$) then the following non-standard tower implements an* $(S' \circ S'')$-dialect of N in M *(cf. Fig. 1):*
$\forall p \in P_N,\ \forall d \in D : [\![nintL']\!]_M\,[\,[\,nintN'',\,[\,p,\,\bullet\,]\,],\,d\,] \Subset (S' \circ S'').N.p.d$.

The theorem guarantees that a non-standard tower consisting of an S'-interpreter *nintL'* and an S''-interpreter *nintN''* returns a result (if defined) that is correct *wrt* $S' \circ S''$, provided property S' is robust. Regardless of how the two interpreters are implemented, we obtain an implementation of (at least) an $(S' \circ S'')$*-semi-dialect*. If in addition S' and S'' are safely combinable ($S' \bowtie S''$), we obtain an implementation of an $(S' \circ S'')$*-dialect*. In contrast to the mathematical combination of two properties (Sect. 2.2), a combination of two non-standard interpreters requires that the source language of *nintL'* and the implementation language of *nintN''* match (*i.e.* language L). This is illustrated in Fig. 1. We showed which properties are robust (Sect. 2.1) and which are safely combinable (Sect. 2.2).

Figure 2 summarizes relation safely combinable for combinations of properties defined in Ex. 2. For $Trans_R$ we assume R is a universal language. Property *Inv* is not always safely combinable. While some properties S' and S'' are not safely combinable for all languages, they may be safely combinable for some languages. Two cases when properties are safely combinable for a subset of $\mathcal{D}$:

1. *Only one S''-dialect exists for N*. For instance, for property *Inv* this condition is satisfied for programming languages in which all programs are injective (this is not true for all programming languages).
2. *Property S' is total for N''*. For example, if R is a universal programming language in property $Trans_R$, then every source program can be translated to R. Thus, $Trans_R$ is totally defined. More formally, S' is a total property for N'' if we have: $\forall p \in P_N,\ \forall d \in D : S'.N''.p.d \neq \emptyset$.

We now examine several semantics combinations and their non-standard towers. The results are summarized in Fig. 3. (Multi-level towers can be constructed by repeating the construction of a two-level tower.)

4.1 Classical Interpreter Tower

Two classical results about standard interpreters can be obtained in our framework using two facts: (i) property *Id* is robust (Sect. 2.1), and (ii) *Id* is safely

$S' \bowtie S''$	*Id*	*Inv*	$Trans_Q$	*Copy*
Id	Yes	Yes	Yes	Yes
Inv	Yes	No	No	Yes
$Trans_R{}^*$	Yes	Yes	Yes	Yes

$^{(*)}R$ is a universal programming language

Fig. 2. S' and S'' are safely combinable

combinable with any property S for $\mathcal{D}$: $Id \bowtie S$ (Sect. 2.2). We also observe that an interpreter *intL* is an *Id*-interpreter because $Id|L = L$ is an *Id*-dialect of L, and accord. to Def. 17 *intL* is an interpreter for this *Id*-dialect. Thus we have:

Corollary 1 (*Id*-interpreter). *Let $L, M \in \mathcal{P}$, and let M-program intL be an interpreter for L in M, then intL is an Id-interpreter for L in M.*

$\boxed{Id \circ Id = Id}$ Since we consider only deterministic programming languages, there exists only one deterministic *Id*-dialect, and since $Id \bowtie Id$ is safely combinable, we can build the following non-standard tower consisting of an L/M-interpreter *intL* and an N/L-interpreters *intN*:

$$\forall p \in P_N,\ \forall d \in D :\ [\![intL]\!]_M\, [\,[\, intN, [\, p, \bullet\,]\,], d\,] = [\![p]\!]_N\ d\ .$$

It is easy to see that this combination is the classical *interpreter tower*. The key point is that the semantics of N is preserved by combination $Id \circ Id$. Property *Id* can be regarded as *identity operation* in the algebra of semantics combination.
$\boxed{Id \circ S = S}$ More generally, any S-interpreter *nintN* for N in L can be evaluated in M given an *Id*-interpreter *intL* for L in M. The non-standard tower is a faithful implementation of an S-dialect in M. Not surprisingly, an S-interpreter can be ported from L to M using an *Id*-interpreter *intL*.

$$\forall p \in P_N,\ \forall d \in D :\ [\![intL]\!]_M\, [\,[\, nintN, [\, p, \bullet\,]\,], d\,] \Subset S.N.p.d = [\![p]\!]_{S|N}\ d\ .$$

4.2 Semantics Modifiers

A novel application of *Id*-interpreters can be obtained from combination $S \circ Id$.
$\boxed{S \circ Id = S}$ If property S for $\mathcal{D}$ is robust then $S \bowtie Id$ is safely combinable, and we can write the following non-standard tower consisting of an *Id*-interpreter *intN* for N in L and an S-interpreter *nintL* for L in M:

$$\forall p \in P_N,\ \forall d \in D :\ [\![nintL]\!]_M\, [\,[\, intN, [\, p, \bullet\,]\,], d\,] \Subset S.N.p.d = [\![p]\!]_{S|N}\ d\ .$$

The equation asserts that an S-interpretation of N-programs can be performed by combining an *Id*-interpreter for N in L and an S-interpreter for L. The non-standard tower implements an S-interpreter for N. Every S-interpreter captures

$S' \circ S''$	*Id*	*Inv*	Trans_Q	*Copy*
Id	*Id* int-tower	*Inv* porting	Trans_Q porting	*Copy* porting
Inv	*Inv* semmod	*Id* identity	Cert_Q certifier	*Recog* recognizer
Trans_R	Trans_R semmod	$\mathit{InvTrans}_R$ inverter	$G_{Q/R}$	Arch_R archiver

Fig. 3. Examples of property combinations

the essence of S-computation regardless of its source language. This is radically different from other forms of program reuse because all interpreters implementing robust properties can be ported to new programming languages by means of *Id*-interpreters. In other words, the entire class of robust properties is suited as *semantics modifiers* [1]. This idea was demonstrated for the following examples.

$\boxed{\mathit{Inv} \circ \mathit{Id} = \mathit{Inv}}$ Since *Inv* is a robust property (Sect. 2.1), we can reduce the problem of writing an *Inv*-interpreter for N to the simpler problem of writing an *Id*-interpreter for N in L, provided an inverse interpreter for L exists. For experimental results see [16, 1, 2].

$\boxed{\mathit{Trans}_R \circ \mathit{Id} = \mathit{Trans}_R}$ A translator is a classical example of an equivalence transformer. Since property Trans_R is robust for all universal programming languages R, this equations asserts that translation from N to R can be performed by combining a standard interpreter for N in L and a translator from L to R. A realization of this idea are the *Futamura projections* [7]: it was shown [9] that partial evaluation can implement this equation efficiently (for details see also [1]).

4.3 Non-standard Translation

$\boxed{\mathit{Trans}_R \circ S = S\_\mathit{Trans}_R}$ where S is a property for $\mathcal{D}$ and

$$S\_\mathit{Trans}_R \overset{\text{def}}{=} \\ \{\ (L, p, r, p') \mid L \in \mathcal{D}, p \in P_L, r \in D, L' \in \mathcal{D}_{S|L}, p' \in P_R, [\![p']\!]_R = [\![p]\!]_{L'}\ \}$$

This combination describes the semantics of translating an L-program p into a standard R-program p' which is functionally equivalent to p evaluated under a deterministic S-dialect of N. In other words, non-standard computation of p is performed by standard computation of p' in R. We say S_Trans_R is the semantics of *S-compilation into R*. This holds regardless of S. We have already met the case of *Id*-translation (Sect. 4.2). Let us examine two examples:

$\boxed{\mathit{Trans}_R \circ \mathit{Inv} = \mathit{InvTrans}_R}$: semantics of an *program inverter* which produces an inverse R-program p^{-1} given an L-program p.

$\boxed{\mathit{Trans}_R \circ \mathit{Copy} = \mathit{Arch}_R}$: semantics of an *archival program* which converts an L-program p into a "self-extracting archive" written in R.

4.4 Semantics Inversion

$\boxed{Inv \circ S = S^{-1}}$ where S is a property for $\mathcal{D}$ and

$$S = \{ (L, p, a, r) \mid L \in \mathcal{D}, p \in P_L, a \in D, r \in S.L.p.a \}$$
$$S^{-1} = \{ (L, p, r, a) \mid L \in \mathcal{D}, p \in P_L, a \in D, r \in S.L.p.a \}$$

The combination describes the inversion of a property S. Three examples:
$\boxed{Inv \circ Trans_Q = Cert_Q}$: semantics of a *program certifier* which, given Q-program p' and L-programs p, verifies whether p' is a translated version of p.
$\boxed{Inv \circ Copy = Recog}$: semantics of a *recognizer*, a program checking whether two L-programs are textually identical – a rather simple-minded semantics.
$\boxed{Inv \circ Inv = Id}$: the inverse semantics of an inverse semantics is the *Id*-semantics (in general they are not safely combinable and a tower of two *Inv*-interpreters ensures only a semi-dialect).

5 Related Work

Interpreters are a convenient way for designing and implementing programming languages [13, 6, 14, 19, 10]. Early operational semantics [12] and definitional interpreters [15] concerned the definition of one programming language using another which, in our terms, relies on the robustness of *Id*-semantics.

Monadic interpreters have been studied recently to support features of a programming language, such as profiling, tracing, and error messages (*e.g.*, [11, 18]). These works are mostly concerned with modifying operational aspects of a particular language, rather than modifying extensional semantics properties of a class of languages. We studied language-independent conditions for analyzing semantics changes and provided a solid mathematical basis for their correctness.

Meta-interpreters have been used in logic programming for instrumenting programs and for changing ways of formal reasoning [20, 3]. These modifications usually change the inference rules of the underlying logic system, and in general do not attempt the deep semantics changes covered by our framework.

Reflective languages have been advocated to enable programs to semantically extend the source language itself, by permitting them to run at the level of the language implementation with access to their own context [4, 8]. The reflective tower [17] is the principle architecture of such languages. More should be known to what extent reflective changes can be captured by robust semantics properties.

Experimental evidence for porting S-semantics via *Id*-interpreters ($S \circ Id$) has been given for inverse semantics (*Inv*) [16, 1, 2], and for translation semantics ($Trans_R$) in the area of partial evaluation [9, 5]. We are not aware of other work developing mathematical foundations for a theory about semantics combinations, but should mention related work [1] studying the class of semantics modifiers.

6 Conclusion and Future Work

The semantics conditions we identified, allow us to reason about the combination of semantics on an abstract level without referring to a particular implementation, and to examine a large class of non-standard semantics instead of particular instances (*e.g.*, a specializer and a translator both implement a translation semantics Trans_R). Our results suggest a technique for the implementation of a certain class of programming language dialects by composing a hierarchy of non-standard interpreters (*e.g.*, inverse compilation by $\mathit{Trans}_R \circ \mathit{Inv}$).

Among others, we can now answer the question raised in the introduction, namely what it means for the semantics of a language N if the implementation language L of its standard interpreter *intN* is interpreted in a non-standard way ($S \circ \mathit{Id} = ?$). As an example we showed that an inverse interpretation of L implements an inverse interpreter for N (even though we have *never* written an inverse interpreter for N, only a standard interpreter *intN*). This is possible because *Inv* is a *robust property* that can be *safely combined* with *Id*.

For some of the properties presented in this paper, practical demonstrations of their combination exist (*e.g.*, $\mathit{Id} \circ \mathit{Id}$, $\mathit{Trans}_R \circ \mathit{Id}$, $\mathit{Inv} \circ \mathit{Id}$). In fact, for the first two combinations it was shown that partial evaluation is strong enough to achieve efficient implementations. It is clear that more experimental work will be needed to examine to what extent these and other transformation techniques can optimize non-standard towers, and to what extent stronger techniques are required. We presented a dozen property combinations. Which of these combinations will be useful for which application is another practical question for future work.

Acknowledgements Special thanks to Neil D. Jones for suggesting a relational semantics in an earlier version of this paper. The second author was partly supported by an Invitation Fellowship of the Japan Society for the Promotion of Science (JSPS). Thanks to David Sands and the four anonymous reviewers for constructive feedback.

References

1. S. M. Abramov, R. Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science.* to appear.
2. S. M. Abramov, R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse, J. N. Oliveira (eds.), *Mathematics of Program Construction. Proceedings*, *LNCS* 1837, 187–212. Springer-Verlag, 2000.
3. K. Apt, F. Turini. *Meta-Logics and Logic Programming.* MIT Press, 1995.
4. O. Danvy. Across the bridge between reflection and partial evaluation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *PEMC*, 83–116. North-Holland, 1988.
5. O. Danvy, R. Glück, P. Thiemann (eds.). *Partial Evaluation. Proceedings*, *LNCS* 1110. Springer-Verlag, 1996.
6. J. Earley, H. Sturgis. A formalism for translator interactions. *CACM*, 13(10):607–617, 1970.
7. Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

8. S. Jefferson, D. P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2/3):181–202, 1996.
9. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
10. S. N. Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, 1990.
11. A. Kishon, P. Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–547, 1995.
12. P. Lucas, P. Lauer, H. Stigleitner. Method and notation for the formal definition of programming languages. Technical report, IBM Lab Vienna, 1968.
13. J. McCarthy. Recursive functions of symb. expressions. *CACM*, 3(4):184–195, 1960.
14. F. G. Pagan. On interpreter-oriented definitions of programming languages. *Computer Journal*, 19(2):151–155, 1976.
15. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, 717–740. ACM, 1972.
16. B. J. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348, 1997.
17. B. C. Smith. Reflection and semantics in Lisp. In *POPL*, 23–35. ACM Press, 1984.
18. G. L. Steele. Building interpreters by composing monads. In *POPL*, 472–492. ACM Press, 1994.
19. G. L. Steele, G. J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, two). MIT AI Memo 453, MIT AI Laboratory, 1978.
20. L. Sterling, E. Shapiro. *The Art of Prolog*. MIT Press, 1986.