Klimov A.V.

A program specialization relation based on supercompilation and its properties

KELDYSH INSTITUTE OF APPLIED MATHEMATICS

Russian Academy of Sciences

A. V. Klimov

A program specialization relation based on supercompilation and its properties

Moscow
2008

Andrei V. Klimov. A program specialization relation based on supercompilation and its properties.

Abstract. An input-output relation for a wide class of program specializers for a simple functional language in the form of Natural Semantics inference rules is presented. It covers *polygenetic* specialization, which includes deforestation and supercompilation, and generalizes the author's previous paper on specification of *monogenetic* specialization like partial evaluation and restricted supercompilation.

The *specialization relation* expresses the idea of *what* is to be a specialized program, avoiding as much as possible the details of *how* a specializer builds it. The relation specification follows the principles of Turchin's supercompilation and captures its main notions: *configuration*, *driving*, *generalization of a configuration*, *splitting a configuration*, as well as *collapsed-jungle driving.* It is virtually a formal definition of supercompilation abstracting away the most sophisticated parts of supercompilers — strategies of *configuration analysis.*

Main properties of the program specialization relation — idempotency, transitivity, soundness, completeness, correctness — are formulated and discussed.

Андрей В. Климов. Отношение специализации программ, основанное на суперкомпиляции, и его свойства.

Аннотация. Дано формальное определение отношения между входом и выходом широкого класса специализаторов программ для простого функционального языка в виде правил вывода натуральной семантики. Отношение охватывает *полигенную* специализацию, включающую дефорестацию и суперкомпиляцию, и обобщает предыдущую работу автора по спецификации *моногенной* специализации, в которую вкладываются частичные вычисления и ограниченная суперкомпиляция.

Предложенное *отношение специализации* выражает, *что* такое правильная специализированная программа, без конкретизации того, *как* специализатор ее строит. Определение отношения формализует основные понятия суперкомпиляции по В.Ф. Турчину: *конфигурация, прогонка, обобщение конфигурации, расщепление конфигурации.* Фактически дано определение суперкомпиляции, абстрагируясь от самой сложной части суперкомпиляторов — стратегий *конфигурационного анализа.*

Обсуждаются основные свойства отношения специализации: идемпотентность, транзитивность, обоснованность (soundness), полнота, корректность.

Работа выполнена при поддержке проектов РФФИ № 06-01-00574 и № 08-07-00280 и Роснауки № 2007-4-1.4-18-02-64.

Contents

1 Introduction **4**

2 Basic notions **5**

2.1 Object Language and Semantic Domains . . . 5

2.2 Configuration . . . 8

2.3 Substitution . . . 9

2.4 Contraction . . . 11

3 Interpretation as a subset of specialization **13**

4 Interpretation and specialization of primitives **14**

4.1 Interpretation and transient driving of primitives . . . 14

4.2 Driving with branching . . . 14

5 Interpretation and specialization of control terms **17**

5.1 Correspondence between residual functions and configurations . 18

5.2 Specialization relation . . . 18

5.3 Rules for control terms . . . 19

5.3.1 Interpretation of control terms . . . 19

5.3.2 Residualization of conditional term . . . 19

5.3.3 Generalization . . . 21

5.3.4 Splitting a configuration . . . 21

5.4 Extensions . . . 22

6 Properties of specialization relation **23**

6.1 Idempotency . . . 23

6.2 Transitivity . . . 24

6.3 Soundness . . . 24

6.4 Completeness . . . 25

6.5 Preservation of semantics . . . 25

6.6 Correctness . . . 25

7 Conclusion and Related Work **26**

8 Acknowledgments **27**

1 Introduction

Program specialization is an equivalence transformation. A specializer *spec* maps a source program p to a residual program q, which is equivalent to p on a given subset D of the domain of the program p: $q = \mathit{spec}(p, D)$. The equivalence of the source and residual programs is understood *extensionally*, that is, nonconstructively: $p \approx_D q$ if for all $d \in D: \ p(d) = q(d)$ or both $p(d)$ and $q(d)$ do not terminate. The correctness of specializers is usually proven by reducing it to the extensional equivalence [4, 2, 11]:

$$q = \mathit{spec}(p, D) \ \Rightarrow \ p \approx_D q.$$

In this paper we define a constructive, *intensional* relation of specialization, that is, a relation of equivalence of source and residual programs, which many specialization methods satisfy, including partial evaluation [5], deforestation [18], supercompilation [16, 17]. Let $p \simeq_D q$ denote such intensional equivalence of a source program p and a residual one q on a set D of input data values. To be correct, it must be a subset of the extensional relation, $(\simeq) \subset (\approx)$, that is, $p \simeq_D q \ \Rightarrow \ p \approx_D q$. The intensional relation provides a shorter way for proving the correctness of specializers than the extensional equivalence does:

$$q = \mathit{spec}(p, D) \ \Rightarrow \ p \simeq_D q \ \Rightarrow \ p \approx_D q.$$

The specialization relation is defined in this paper by inference rules in the style of Natural Semantics [6]. The rules serve as formal specification of a wide class of specializers. By the nature of Natural Semantics, the specification allows for automated derivation of specializers as well as checkers of the correctness of residual programs, which can help in debugging practical specializers. Some notions (*e.g.*, driving) are defined precisely enough to unambiguously derive the corresponding algorithm by the well-known methods. Other notions (*e.g.*, generalization and splitting configurations) are defined with certain degrees of freedom to allow for various decision-taking algorithms and strategies.

The specialization relation is based on the ideas of supercompilation, but agrees with partial evaluation and deforestation as well. The inference rules model at abstract level the operational behavior of supercompilers. All essential notions of supercompilation are captured: *configuration*, *driving*, *generalization of a configuration*, *splitting a configuration*, as well as *collapsed-jungle driving* [12, 13], while abstracting from the problems of algorithmic decisions of when, what and how to generalize and when to terminate.

In paper [8] a similar specialization relation was presented for the simpler case of *monogenetic* specialization [10] where any program point in the residual program is produced from a single program point of the source program. In this paper the relation definition is developed further to the *polygenetic* specialization [10] where a residual program point is produced from one or several source program points. Monogenetic specialization includes partial evaluation but excludes deforestation and supercompilation. Polygenetic specialization covers all of them. For completeness sake, we repeat in Section 2 the basic notions from [8] as well as the definition of driving in Fig. 6 and 7.

The main contributions of the paper are as follows:

- a complete formal definition of what supercompilation is, in form of an input-output specialization relation, is given;
- several interesting properties that the presented specialization relation obeys are formulated and related to each other: idempotency, transitivity, soundness, completeness, correctness.

The paper is organized as follows. A simple object language, which is both the source and target language of specializers, is presented in Section 2.1 together with semantic domains for interpretation and supercompilation. In Section 2.2 the notion of configuration is introduced. In Section 2.3 the operation of substitution as it is used in this paper is defined. Section 2.4 contains the discussion of the peculiarities of the notion of contraction. Sections from 3 to 5 present the definition of the specialization relation: in Section 3 the specifics of our definition of the language semantics and specialization is explained; in Section 4 the semantics and driving of the language primitives and in Section 5 the semantics and specialization of control program terms are specified. In Section 6 the most interesting properties of the specialization relation are formulated and discussed, and in Section 7 we conclude.

2 Basic notions

2.1 Object Language and Semantic Domains

Figure 1 contains the definition of the abstract syntax of the object language together with semantic domains for interpretation and specialization. It is a simple first-order functional language. It has conventional control constructs **if-then-else**, **let-in**, **call**, adjusted a bit to make the specialization inference rules simpler. Figure 2 shows an example of a program and an initial configuration for specialization.

Data A data domain *Data* is a *constructor-based* domain recursively defined from a set of atoms *Atom* by applying a binary constructor Cons. The set *Atom* contains at least True, False, and Nil.

Any constructor-based domain has the nice property that it can be easily extended to meta-data without the need for encoding. In particular, a constant

$k \in \mathit{Atom}$	atomic data
$x \in \mathit{Data}$	ground data
$z \in \mathit{CData}$	configuration data
$a \in \mathit{Arg}$	source arguments
$d \in \mathit{Prim}$	source primitives
$s \in \mathit{Term}$	source program terms
$r \in \mathit{Term}$	residual program terms
$v \in \mathit{Var}$	source program variables
$u \in \mathit{Var}$	residual program variables and configuration variables
$l \in \mathit{LVar}$	liaison variables
$f \in \mathit{FName}$	function names
$p \in \mathit{Prog}$	source programs
$q \in \mathit{Prog}$	residual programs
$b \in \mathit{Args}$	argument bindings
$c \in \mathit{Contr}$	contractions
$\Delta \in \mathit{Expl}$	explications
$d, s \in \mathit{MConf}$	monogenetic configurations
$z \mid \Delta \in \mathit{PConf}$	polygenetic configurations
$m \in \mathit{CMap}$	mapping of residual function names to configurations

$$
\begin{array}{lcl}
k & ::= & \mathrm{True} \mid \mathrm{False} \mid \mathrm{Nil} \mid \dots \\
x & ::= & k \mid \mathrm{Cons}\ x\ x \\
z & ::= & k \mid \mathrm{Cons}\ z\ z \mid u \mid l \\
a & ::= & z \mid \mathrm{Cons}\ a\ a \mid v \\
d & ::= & a \mid \mathbf{fst}\ v \mid \mathbf{snd}\ v \\
  &   & \mid \mathbf{cons?}\ v \mid \mathbf{equ?}\ v\ a \\
s & ::= & d \\
  &   & \mid \mathbf{if}\ v\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \\
  &   & \mid \mathbf{let}\ v = s_1\ \mathbf{in}\ s_2 \\
  &   & \mid \mathbf{call}\ f\ b
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Prog} & = & \mathit{FName} \to \mathit{Term} \\
\mathit{Args} & = & \mathit{Var} \to \mathit{Arg} \\
\mathit{Contr} & = & \mathit{Var} \to \mathit{CData} \\
\mathit{Expl} & = & \mathit{LVar} \to \mathit{MConf} \\
\mathit{MConf} & = & \mathit{Term} \\
\mathit{PConf} & = & \mathit{CData} \times \mathit{Expl} \\
\mathit{CMap} & = & \mathit{FName} \to \mathit{PConf}
\end{array}
$$

Figure 1: Object language syntax and semantic domains

in program code coincides with the value it represents. That is, $\mathit{Data} \subset \mathit{Term}$, where Term is the domain of program terms.

Configuration data Another extension of *Data* originates from the need to constructively represent sets of data values and sets of program states. The basic method to represent sets is to embed free variables into the representation of data. In the theory of supercompilation such variables are referred to as *configuration variables.* The general principle is that a configuration variable, $u \in \mathit{Var}$, can occur in any position where a ground value is allowed.

A characteristic feature of supercompilation, which is preserved in our specialization relation definition, is that configuration variables become residual program variables.

To specify polygenetic specialization, we use a representation of configurations in form of directed acyclic graphs [12, 13], which we define in the next section and refer to as *polyconfigurations.* It requires one more extension of the

```
rev  v1              =  loop v1 []                              — a program
loop [] v2           =  v2                                        in Haskell
loop (v4 : v5) v2    =  loop v5 (v4 : v2)
```

$$
\begin{array}{rll}
p = & \{\ \mathrm{rev} \mapsto & \mathbf{call}\ \mathrm{loop}\ \{v_1 \mapsto v_1,\ v_2 \mapsto \mathrm{Nil}\}, \\
 & \ \ \mathrm{loop} \mapsto & \mathbf{let}\ v_3 = \mathbf{cons}?\ v_1\ \mathbf{in} \\
 & & \mathbf{if}\ v_3 \\
 & & \mathbf{then}\ \mathbf{let}\ v_4 = \mathbf{fst}\ v_1\ \mathbf{in} \\
 & & \qquad \mathbf{let}\ v_5 = \mathbf{snd}\ v_1\ \mathbf{in} \\
 & & \qquad \mathbf{call}\ \mathrm{loop}\ \{v_1 \mapsto v_5,\ v_2 \mapsto \mathrm{Cons}\ v_4\ v_2\} \\
 & & \mathbf{else}\ \ v_2 \\
 & \} &
\end{array}
$$

— the same program in the object language

$$s_0 = \mathbf{call}\ \mathrm{rev}\ \{v_1 \mapsto \mathrm{Cons\ A}\ (\mathrm{Cons}\ u_1\ (\mathrm{Cons\ B}\ u_2))\}$$ — an initial term

$$z \mid \Delta = l_0 \mid \{l_0 \mapsto s_0\}$$ — an initial configuration

Figure 2: An example of a program p in the object language and an initial configuration

data domain by so called *liaison*[1] *variables*, $l \in \mathit{LVar}$, bound variables that link positions in terms to subterms.

The data and configuration data domains, *Data* and *CData*, and the *Term* domain are embedded in each other: $\mathit{Data} \subset \mathit{CData} \subset \mathit{Term}$.

Primitives Data values are analyzed by primitive predicates **equ**? $v\ a$ (are two values equal?) and **cons**? v (is the value of a variable v a term of the form Cons $z_1\ z_2$?), which return atoms True or False, and selectors **fst** v and **snd** v, which require the value of v to be a Cons term and return its first and second argument respectively.

To avoid dealing with exceptions, we impose a context restriction on selectors **fst** v and **snd** v: they can occur only on the positive branch of an **if**-term with the conditional **cons**? v.

[1]The term is due to V. Turchin, who suggested the use of such a representation of configurations in supercompilers in 1970s.

Control Control terms **if-then-else** and **let-in** are the usual conditional term and **let** binding respectively. The following restriction is imposed for the simplicity of the specialization definition: the conditional must be a variable v that is bound to a conditional primitive, **equ?** or **cons?**, by an enclosing **let** term.

A function call, which usually looks like $f(a_1, \ldots, a_n)$, is written in our language as **call** $f\ \{v_1 \mapsto a_1, \ldots, v_n \mapsto a_n\}$, where $v_1, \ldots, v_n$ are the free variable names of the term that the name f is bound to in the program.

A program is a finite mapping of function names to program terms.

For simplicity, terms are in the so called *administrative normal form*, that is, the arguments of all terms except the **let-in** term and the **then** and **else** branches of the **if** term, has trivial form: $v \in \mathit{Var}$ or $a \in \mathit{Arg}$.

Notation

- FVars(t) denotes the set of free variables occurring in term t,
- LVars(t) the set of liaison variables in a term or in a configuration.
- Dom(m) and Rng(m) denote the domain and range of mapping m respectively.

2.2 Configuration

While an interpreter runs a program on a ground data, a specializer runs a source program on a set of data. A representation of a program state in interpretation and that of a set of states in specialization is referred to as a *configuration*. We follow the general rule of construction of the notion of the configuration in a supercompiler from that of the program state in an interpreter that reads as follows: add configuration variables to the data domain, and allow the variables to occur anywhere where an ordinary ground value can occur. A configuration represents the set that is obtained by replacing all configuration variables with all possible values.

There are two kinds of configurations, which we refer to as *monoconfigurations* and *polyconfigurations*. In our previous work [8] monoconfigurations were actual configurations. In this work monoconfigurations are used as configurations in the rules for primitives, and polyconfigurations are configurations in the rules for control terms. Polyconfigurations comprise monoconfigurations.

Syntactically, a *monoconfiguration* is a source program term, in which program variables are replaced with their values.[2]

[2]An alternative is to keep program terms untouched and to represent the monoconfiguration as a pair consisting of a program term and an environment that binds program variables to their values. Although this representation is more common in implementations of interpreters and specializers, we prefer to substitute the environment into the term for conciseness of inference rules.

A *polyconfigurations* is a representation of a program state as directed acyclic graphs (as in [12, 13]). A polyconfiguration can be thought of as an entity obtained from a monoconfiguration in these steps:

1) decompose a monoconfiguration into a topmost term and some subterms,
2) bind the subterms to fresh liaison variables;
3) put the liaison variables into the topmost term instead of respective subterms;
4) analogously decompose some subterms.

A polyconfiguration is denoted by $z \mid \Delta$, where z is the topmost subterm, and Δ the binding of liaison variables to terms (monoconfigurations), referred to as an *explication.*[3] Topmost terms are restricted to $z \in \mathit{CData}$, that is, primitives and control terms must be picked out (*explicated*) and put into the binding. As an example, see the initial configuration in Fig. 2:

$$z \mid \Delta = l_0 \mid \{l_0 \mapsto \mathbf{call}\ \mathrm{rev}\, \{v_1 \mapsto \mathrm{Cons}\, \mathrm{A}\, (\dots)\}\}.$$

While initial configurations are usually trees, during specialization polyconfigurations form directed acyclic graphs in general. In the case of the applicative evaluation order, the polyconfiguration is a call stack. However, the inference rules do not fix the order of evaluation, and we consider the explication as an unordered set of bindings. Each time we write $\{l \mapsto s\}\, \Delta$, we imply $\Delta_1\{l \mapsto s\}\, \Delta_2$ for some Δ_1 and Δ_2 such that $\Delta = \Delta_1\, \Delta_2$.

When it is clear from the context what kind of configuration is meant, we say just a *configuration.* It is a monogenetic configuration when the rules of driving of primitives are considered, and a polygenetic configuration in other cases.

2.3 Substitution

To avoid the ambiguity of traditional postfix notation for substitution $t\theta$ when it is used in inference rules (either juxtaposition, or application of substitution), we lift up the substitution symbol θ and use a kind of power notation t^θ.

Thus t^θ denotes the replacement of all occurrences of variables $v \in \mathrm{Dom}(\theta)$ in t with their values from a binding θ. Notation $t^{\eta\theta}$ means sequential application of substitutions η and θ to t in that order. When the argument of a substitution is unclear, it is over-lined, *e.g.*, $a\ \overline{b\ c}^{\,\theta}\ d$.

The bindings listed in Fig. 1 are used as substitutions as follows:

- f^p gets the term bound to a function name f in a program p;

[3]The term is due to V. Turchin.

$$\overline{\_}^{\_} : \mathit{FName} \times \mathit{Prog} \to \mathit{Term}$$

$$f^{\{\ldots,\, f \mapsto s,\, \ldots\}} = s$$

$$\overline{\_}^{\_} : \mathit{Term} \times (\mathit{Contr} \cup \mathit{Args} \cup \mathit{Expl}) \to \mathit{Term}$$

$$\begin{array}{ll}
z^{\theta} & = z \ \text{ if } \ z \in \mathit{Atom} \cup \mathit{Var} \cup \mathit{LVar} \ \text{ and } \ z \notin \mathrm{Dom}(\theta) \\
v^{\{\ldots,\, v \mapsto z,\, \ldots\}} & = z \\
\overline{\mathrm{Cons}\ a_1\ a_2}^{\theta} & = \mathrm{Cons}\ a_1^{\theta}\ a_2^{\theta} \\
\overline{\mathbf{fst}\ v}^{\theta} & = \mathbf{fst}\ v^{\theta} \\
\overline{\mathbf{snd}\ v}^{\theta} & = \mathbf{snd}\ v^{\theta} \\
\overline{\mathbf{cons?}\ v}^{\theta} & = \mathbf{cons?}\ v^{\theta} \\
\overline{\mathbf{equ?}\ v\ a}^{\theta} & = \mathbf{equ?}\ v^{\theta}\ a^{\theta} \\
\overline{\mathbf{if}\ v\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2}^{\theta} & = \mathbf{if}\ v^{\theta}\ \mathbf{then}\ s_1^{\theta}\ \mathbf{else}\ s_2^{\theta} \\
\overline{\mathbf{let}\ v = s_1\ \mathbf{in}\ s_2}^{\theta} & = \mathbf{let}\ v = s_1^{\theta}\ \mathbf{in}\ s_2^{\theta} \\
\overline{\mathbf{call}\ f\ b}^{\theta} & = \mathbf{call}\ f\ b^{\theta}
\end{array}$$

$$\overline{\_}^{\_} : \mathit{Args} \times (\mathit{Contr} \cup \mathit{Args} \cup \mathit{Expl}) \to \mathit{Args}$$

$$\{v_1 \mapsto a_1,\ \ldots,\ v_n \mapsto a_n\}^{\theta} = \{v_1 \mapsto a_1^{\theta},\ \ldots,\ v_n \mapsto a_n^{\theta}\}$$

$$\overline{\_}^{\_} : \mathit{Expl} \times (\mathit{Contr} \cup \mathit{Expl}) \to \mathit{Expl}$$

$$\{l_1 \mapsto s_1,\ \ldots,\ l_n \mapsto s_n\}^{\theta} = \{l_1 \mapsto s_1^{\theta},\ \ldots,\ l_n \mapsto s_n^{\theta}\}$$

$$\overline{\_}^{\_} : \mathit{PConf} \times (\mathit{Contr} \cup \mathit{Expl}) \to \mathit{PConf}$$

$$\overline{z \mid \Delta}^{\theta} = z^{\theta} \mid \Delta^{\theta}$$

Figure 3: The definition of substitution t^{θ} for those domains which it is applied to in the specialization relation definition

- $s^b = f^{pb}$ builds a monoconfiguration from a program term $s = f^b$ and the argument binding b of a monoconfiguration $\mathbf{call}\, f\, b$;
- $\overline{z \mid \Delta}^{\{l \mapsto z'\}}$ substitutes a value z', which is the result of evaluation of some monoconfiguration d taken from a binding $\{l \mapsto z'\}$, instead of the liaison variable l in a polyconfiguration $z \mid \Delta$;
- s^c and $\overline{z \mid \Delta}^{c}$ *contracts* a monoconfiguration s and a polyconfiguration $z \mid \Delta$ respectively by replacing a configuration variable u with the configuration value z' bound to u by a *contraction* $c = \{u \mapsto z'\}$.

See Fig. 3 for the formal definition of the substitution operation.

2.4 Contraction

After evaluation of a conditional, the current configuration divides into two subconfigurations, the initial configurations of the positive and negative branches. In our definition the subconfigurations represent the subsets precisely, that is, they are disjoint.

There are two Boolean primitives, $\mathbf{equ?}\ v\ a$ and $\mathbf{cons?}\ v$, in the object language. After substitution of configuration values into the arguments of the primitives, they ultimately reduce (by rules in Fig. 7) to the following checks on configuration variables that produce branching in residual code: $\mathbf{equ?}\, u\, k$, $\mathbf{equ?}\, u\, u'$, and $\mathbf{cons?}\, u$, where k is an atom, u and u' configuration variables.

For each of the primitives, the set of the values of a configuration variable u that go to the positive branch can be represented by a substitution $\{u \mapsto k\}$, $\{u \mapsto u'\}$, or $\{u \mapsto \mathrm{Cons}\ u_1\ u_2\}$, where $k \in \mathit{Atom}$, u_1 and u_2 are new configuration variables. Such a substitution is referred to as a *contraction.* Being applied to a configuration, it produces a configuration representing a subset of the original one.

For uniformity's sake, the opposite case of "negative" information—the set of the values that go to the negative branch—is represented by a substitution as well. To achieve this, we assume the representation of a configuration variable contains a *negative set*: a set of "negative entities" the variable must be *unequal to.* The negative entities are atoms, configuration variables, and the word Cons, which represents inequality to all terms of the form $\mathrm{Cons}\ z_1\ z_2$.

We denote the operation to add an entity n to the negative set of a configuration variable u by u^{-n}. Thus the following substitutions are *negative contractions*: $\{u \mapsto u^{-k}\}$, $\{u \mapsto u^{-u'}\}$, and $\{u \mapsto u^{-\mathrm{Cons}}\}$.

$d \rightsquigarrow z$	**Interpretation and transient driving** of a primitive d produces a value z.
$d \prec \mathcal{T}(c_1,\, c_2)$	**Driving with branching**: Driving of a primitive d produces a branching represented by a residual conditional term $\mathcal{T}(\_,\, \_)$ with two free positions for positive and negative branches. In the right-hand side $\mathcal{T}(c_1,\, c_2)$ these positions are occupied by contractions c_1 and c_2. The contractions being substituted to the configuration before the branching produce the initial configurations for the positive and negative branches respectively.
$p : s \Rightarrow q : r$	**Specialization**: A residual program q with an initial term r is a specialization of a source program p with an initial term s. Initial polyconfiguration is $l \mid \{l \mapsto s\}$.
$p : z \mid \Delta \rightsquigarrow m : r$	**Specialization to a term**: A residual term r is a specialization of a source program p with an initial polyconfiguration $z \mid \Delta$ with respect to a mapping m of residual function names to polyconfigurations.
$p : z \mid \Delta \rightsquigarrow \emptyset : z$	**Interpretation as a subset of specialization**: In the case where the residual program is empty, and hence $m = \emptyset$, and the residual term z is a configuration value, $z \in \mathit{CData}$, the previous judgment means interpretation or transient driving.
$p : s \rightarrow z$	**Interpretation as a subset of specialization**: This is a shortcut notation for the particular case of the above judgment of the form $p : l \mid \{l \mapsto s\} \rightsquigarrow \emptyset : z$.
$p : s \overset{\circ}{\rightarrow} z$	**Interpretation (semantics)**: The judgement is equivalent to $p : s \rightarrow z$ with the requirement that it be deduced using only the interpretation and transient driving rules marked with $^\circ$. This is proper interpretation when $\mathrm{FVars}(s) = \emptyset$ and hence z is a ground value, $z \in \mathit{Data}$.

Figure 4: Judgments

TR-DRV	$\dfrac{p : l \mid \{l \mapsto s\} \ \rightsquigarrow\ \emptyset : z}{p : s\ \rightarrow\ z}$	$z \in \mathit{CData}$
INT	$\dfrac{p : s\ \rightarrow\ z}{p : s\ \overset{\circ}{\rightarrow}\ z}$	if inferred with °-rules only $z \in \mathit{CData}$

Figure 5: Transient driving and semantics as a subset of specialization

3 Interpretation as a subset of specialization

A specialization relation is an extension of the semantics of a language, that is, its interpretation relation, which is usually a function (for deterministic languages). We could give the language semantics, develop separately the specialization relation, and then prove the statement that the specialization relation includes the semantics.

However, to save space and mental effort we follow another line. We define the specialization relation by inference rules in such a way that a subset of the rules defines the semantics (interpretation). The interpretation rules have labels marked with °. The rules with unmarked labels extend interpretation to specialization.

The interpretation and specialization judgments are listed and commented in Fig. 4.

The specialization judgments $p : s \Rightarrow q : r$ and $p : z \mid \Delta \rightsquigarrow m : r$ contain a residual program q and an auxiliary mapping m (explained in Section 5.1 below), which has the same domain, $\mathrm{Dom}(q) = \mathrm{Dom}(m) \subset \mathit{FName}$, the set of residual function names. When there are no residual functions, that is, $p = q = \emptyset$, the residual code is merely a tree represented by the term r that cannot contain **call** terms. The particular case where the residual term r is a value virtually defines the semantics of the language. It can be proven that judgments of this form are deducible by the interpretation rules.

We define a shortcut notation $p : s \rightarrow z$ for this case by rule TR-DRV, and denote by a circle over the arrow the fact that it is inferred by the interpretation rules only: $p : s \overset{\circ}{\rightarrow} z$ (see rule INT in Fig. 5).

The interpretation rules also define *transient driving*, which is the basic case of driving where the configuration and liaison variables do not prevent a specializer from unambiguously performing a step. The case of proper interpretation can be distinguished from the case of transient driving by the restriction that the initial configuration does not contain configuration variables, $\mathrm{FVars}(s) = \emptyset$.

Definition 1 (Interpretation, semantics) *A source program p with an initial term s (in which arguments have been substituted) without configuration variables,* $\mathrm{FVars}(s) = \emptyset$*, evaluates to a term $x \in \mathit{Data}$ if the following judgment is deducible:*

$$p : s \overset{\circ}{\rightarrow} x$$

4 Interpretation and specialization of primitives

4.1 Interpretation and transient driving of primitives

A judgment of the form $d \rightsquigarrow z$ means *interpretation or transient driving* of a primitive monoconfiguration d (*i.e.*, a source program primitive term, in which program variables have been replaced with their values) produces a value z.

In the rules we distinguish between the case where configuration values can occur and the case where only ground values occur by the names of free variables: $k \in \mathit{Atom} \subset \mathit{Data}$, $x \in \mathit{Data}$, $z \in \mathit{CData}$, that is, k and x range over ground values and z ranges over configuration values that may contain configuration and liaison variables. Note that only rules I-CONS-F$^\circ$ and I-EQ-F$^\circ$, which defines inequality, require values to be ground.

The last two rules in Fig. 6, D-EQ-CK and D-EQ-CC, and the rules in Fig. 7 that infer judgments of the form $d \rightsquigarrow z$ define the cases of transient driving that are not covered by the interpretation rules. The latter rules mean:

- D-CONS-F and D-EQ-UCF — returning False in case where a configuration variable u contains in its negative set the symbol Cons and hence is unequal to any Cons term;
- D-EQ-COM — commutativity of **equ?**;
- D-EQ-UKF — returning False in case where a configuration variable u contains in its negative set the atom k it is compared to;
- D-EQ-UUF — returning False in case where a configuration variable u_1 contains in its negative set the configuration variable u_2 it is compared to.

4.2 Driving with branching

A judgment of the form $d \prec \mathcal{T}(c_1, c_2)$ means *driving*[4] of a primitive monoconfiguration d produces a branching in residual code represented by a conditional

[4]What is usually called driving is the process of unfolding an *infinite process tree* [17, 3, 1]. This sense could be captured by the $^\circ$-rules together with rule PS-BRANCH if we consider *infinite* residual terms r. However, it would be another theory.

I-VALUE$^\circ$	$z \rightsquigarrow z$	$z \in \mathit{CData}$
I-FST$^\circ$	$\textbf{fst}\ (\text{Cons}\ z_1\ z_2) \rightsquigarrow z_1$	
I-SND$^\circ$	$\textbf{snd}\ (\text{Cons}\ z_1\ z_2) \rightsquigarrow z_2$	
I-CONS-T$^\circ$	$\textbf{cons}?\ (\text{Cons}\ z_1\ z_2) \rightsquigarrow \text{True}$	
I-CONS-F$^\circ$	$\textbf{cons}?\ k \rightsquigarrow \text{False}$	$k \in \mathit{Atom}$
I-EQ-T$^\circ$	$\textbf{equ}?\ z\ z \rightsquigarrow \text{True}$	
I-EQ-F$^\circ$	$\textbf{equ}?\ x_1\ x_2 \rightsquigarrow \text{False}$	$x_1 \neq x_2$ $x_1, x_2 \in \mathit{Data}$
D-EQ-CK	$\textbf{equ}?\ (\text{Cons}\ z_1\ z_2)\ k \rightsquigarrow \text{False}$	$k \in \mathit{Atom}$
D-EQ-CC	$\dfrac{\textbf{equ}?\ z_{1i}\ z_{2i} \rightsquigarrow \text{False}}{\textbf{equ}?\ (\text{Cons}\ z_{11}\ z_{12})\ (\text{Cons}\ z_{21}\ z_{22}) \rightsquigarrow \text{False}}$	$i \in \{1, 2\}$

Figure 6: Interpretation and transient driving of primitives

term $\mathcal{T}(\_, \_)$ with two free positions for positive and negative branches and two contractions c_1 and c_2. The contractions c_1 and c_2, being applied as substitutions to the configuration d, divide it into two subconfigurations, which are initial configurations for positive and negative branches respectively.

For the sake of notation brevity, the contractions c_1 and c_2 occupy in $\mathcal{T}(\_, \_)$ the positions where the residual terms for the positive and negative branches will occur in the final residual code.

Figure 7 contains the rules that infer branching in residual code for the source primitives **cons?** and **equ?**. The branching happens when a configuration variable u (or two variables u_1 and u_2 in the case of the **equ?** term) prohibits from performing an evaluation step. Notice the branching rules perform no evaluation step of the source program, but just produce a residual **if** term and contractions. The evaluation step is performed by transient driving rules for the same primitive after contractions c_1 and c_2 has been substituted into the current configuration by rule PS-BRANCH, which produces initial configurations for branches, and transient driving of the primitive has been "invoked" by rule PS-PRIM$^\circ$ in Fig. 9.

D-CONS-F
$$\textbf{cons?}\ u \leadsto \text{False} \qquad \begin{array}{l} u = u^{-\text{Cons}} \\ u \in \mathit{Var} \end{array}$$

D-CONS
$$\textbf{cons?}\ u \prec \begin{array}{l} \textbf{let}\ u_0 = \textbf{cons?}\ u\ \textbf{in} \\ \textbf{if}\ u_0 \\ \textbf{then}\ \textbf{let}\ u_1 = \textbf{fst}\ u\ \textbf{in} \\ \qquad \textbf{let}\ u_2 = \textbf{snd}\ u\ \textbf{in} \\ \qquad \{u \mapsto \text{Cons}\ u_1\ u_2\} \\ \textbf{else}\ \ \{u \mapsto u^{-\text{Cons}}\} \end{array} \qquad \begin{array}{l} u \neq u^{-\text{Cons}} \\ u_0,\ u_1,\ u_2\ \text{new} \\ u,\ u_i \in \mathit{Var} \end{array}$$

D-EQ-COM
$$\frac{\textbf{equ?}\ u\ z \leadsto \mathcal{T}}{\textbf{equ?}\ z\ u \leadsto \mathcal{T}} \qquad u \in \mathit{Var}$$

D-EQ-UKF
$$\textbf{equ?}\ u\ k \leadsto \text{False} \qquad \begin{array}{l} u = u^{-k} \\ u \in \mathit{Var} \\ k \in \mathit{Atom} \end{array}$$

D-EQ-UK
$$\textbf{equ?}\ u\ k \prec \begin{array}{l} \textbf{let}\ u_0 = \textbf{equ?}\ u\ k\ \textbf{in} \\ \textbf{if}\ u_0 \\ \textbf{then}\ \{u \mapsto k\} \\ \textbf{else}\ \ \{u \mapsto u^{-k}\} \end{array} \qquad \begin{array}{l} u \neq u^{-k} \\ u_0\ \text{new} \\ u,\ u_0 \in \mathit{Var} \\ k \in \mathit{Atom} \end{array}$$

D-EQ-UUF
$$\textbf{equ?}\ u_1\ u_2 \leadsto \text{False} \qquad \begin{array}{l} u_1 = u_1^{-u_2} \\ u \in \mathit{Var} \end{array}$$

D-EQ-UU
$$\textbf{equ?}\ u_1\ u_2 \prec \begin{array}{l} \textbf{let}\ u_0 = \textbf{equ?}\ u_1\ u_2\ \textbf{in} \\ \textbf{if}\ u_0 \\ \textbf{then}\ \{u_1 \mapsto u_2\} \\ \textbf{else}\ \ \{u_1 \mapsto u_1^{-u_2},\ u_2 \mapsto u_2^{-u_1}\} \end{array} \qquad \begin{array}{l} u_1 \neq u_1^{-u_2} \\ u_0\ \text{new} \\ u_i \in \mathit{Var} \end{array}$$

D-EQ-UCF
$$\textbf{equ?}\ u\ (\text{Cons}\ z_1\ z_2) \leadsto \text{False} \qquad \begin{array}{rl} & u = u^{-\text{Cons}} \\ & u \in \mathit{Var} \\ \text{or} & u \in \text{FVars}(z_1) \\ \text{or} & u \in \text{FVars}(z_2) \end{array}$$

D-EQ-UC
$$\frac{\textbf{cons?}\ u \prec \mathcal{T}}{\textbf{equ?}\ u\ (\text{Cons}\ z_1\ z_2) \prec \mathcal{T}} \qquad \begin{array}{l} u \neq u^{-\text{Cons}} \\ u \in \mathit{Var} \\ u \notin \text{FVars}(z_1) \\ u \notin \text{FVars}(z_2) \end{array}$$

Figure 7: Driving of primitives

The correctness of this derivation is based on the perfectness [3] of contractions c_1 and c_2 and on the fact that after substitution of c_i into d some transient driving rule for the judgement $d^{c_i} \rightsquigarrow z$ is applicable.

In each of the three use cases of the term $\mathcal{T}(\_,\,\_)$ in Fig. 7 it meets the following property: the value of $\mathcal{T}(x_1,\,x_2)$ is either x_1 for the configuration obtained by contraction c_1, or x_2 for the configuration obtained by contraction c_2. The correctness of rule PS-BRANCH relies on this property.

In Fig. 7, the rules of driving of **equ?** and **cons?** terms that infer judgments of the form $d \prec \mathcal{T}(c_1,\,c_2)$ mean:

- Rule D-CONS defines the branching in a residual program that corresponds to a monoconfiguration of the form **cons**? u. The right-hand side consists of the residual **if** term that tests the value of the variable u, assignments of the parts of the u value to fresh variables u_1 and u_2 on the positive branch, and two complementary contractions $\{u \mapsto \text{Cons}\ u_1\ u_2\}$ and $\{u \mapsto u^{-\text{Cons}}\}$ occupying in the term $\mathcal{T}(\_,\,\_)$ the positions of the positive and negative branches respectively.

- Rules D-EQ-UK and D-EQ-UU analogously define the branchings corresponding to monoconfigurations **equ**? $u\ k$ and **equ**? $u_1\ u_2$ in case where there is no information in negative sets of configuration variables about the equalities under the respective tests.

- Rule D-EQ-UC reduces the test of equality of a variable u and a Cons term, **equ**? u (Cons $z_1\ z_2$), to the test of whether u is a Cons term or not, **cons**? u. Recall the judgment $d \prec \mathcal{T}(c_1,\,c_2)$ defines no evaluation step, only a branching. So, the rule reads as follows: to advance driving of the configuration **equ**? u (Cons $z_1\ z_2$) residualize the same branching as for **cons**? u. After that, the evaluation step will be performed by the driving rules for the **equ?** term.

5 Interpretation and specialization of control terms

A judgment of the form $p : z \mid \Delta \rightsquigarrow m : r$ inferred by the rules in Fig. 9 asserts that a residual term r is a specialization of a source program p with an initial polyconfiguration $z \mid \Delta$ with respect to a mapping m of residual function names to polyconfigurations.

The mapping m assigns meaning to the residual **call** terms occurring in r, which can be explained in terms of the language semantics as follows. For all values of the configuration variables of the configuration $z \mid \Delta$ and that of the residual term r, evaluation of the configuration $z \mid \Delta$ with the source program p gives the same result as evaluation of the term r in two steps:

1) for each subterm of the form **call** $f_i\ b_i$ occurring in r, evaluate the configuration $f_i^{mb_i} = \overline{z_i \mid \Delta_i}^{b_i}$ with the source program p. Here the arguments b_i are substituted into the configuration $z_i \mid \Delta_i$ bound to the function name f_i by the mapping m;

2) evaluate the term r using the thus obtained values of the **call** terms.

5.1 Correspondence between residual functions and configurations

Each residual function f is equivalent to some configuration expressed in terms of the source program. The mapping $m : \mathrm{Dom}(q) \to \mathit{PConf}$ keeps this correspondence. To define correct specialization, the mapping m must be *consistent* with the source and residual programs p and q. This means each residual function body in q is equivalent to the corresponding configuration in m, provided the mapping m is used to assign values to **call** terms as explained above.

Definition 2 (Consistency) *A mapping* $m : \mathrm{Dom}(q) \to \mathit{PConf}$ *of residual function names to configurations is* consistent with source and residual programs p and q *if for every residual function name* $f \in \mathrm{Dom}(q)$ *the following judgment is deducible:*

$$p : f^m \rightsquigarrow m : f^q$$

provided the judgment is not inferred immediately from axiom PS-GEN.

The equivalence of configurations and residual terms represented by the mapping m is virtually an inductive hypotheses, the derivation of the judgements in the definition being an induction step. When proving by induction, care must be taken to avoid the vicious circle of premature use of the inductive hypothesis. This is the role of the provision in the definition. Otherwise, the residual program may contain a loop that is absent in the source program, and the residual program may not terminate when the source program terminates.

5.2 Specialization relation

Now we are ready to define the specialization relation, a relation between pairs consisting of a program and an initial term. We denote the pairs by $p : s$ and $q : r$ for a source program pair and a residual program pair respectively, $p, q \in \mathit{Prog}$, $s, r \in \mathit{CData}$. Note only the terms s and r that have the same configuration variables can relate by the specialization relation, $\mathrm{FVars}(s) = \mathrm{FVars}(r)$.

$$\textsc{spec}\quad \frac{p : l \mid \{l \mapsto s\} \;\rightsquigarrow\; m : r}{p : s \;\Rightarrow\; q : r} \quad \text{if } m \text{ is consistent with } p \text{ and } q$$

Figure 8: Specialization relation

For the sake of uniformity of input and output, we consider the specialization relation over plain terms rather polygenetic configurations. An initial term s maps to the initial polyconfiguration $l \mid \{l \mapsto s\}$.

Definition 3 (Specialization) *A pair $q : r$ of a residual program q and an initial term r is a* specialization *of a pair $p : s$ of a source program p and an initial term s if there exist a mapping m of residual function names to configurations, consistent with p and q, such that the following judgment is deducible:*

$$p : l \mid \{l \mapsto s\} \rightsquigarrow m : r.$$

We denote the fact that pairs $p : s$ and $q : r$ satisfy the specialization relation by judgment $p : s \Rightarrow q : r$. Formally it is defined by rule SPEC in Fig. 8.

5.3 Rules for control terms

Figure 9 contains the main part of the specialization relation definition.

5.3.1 Interpretation of control terms

Rule PS-BASE° asserts the evident fact that a constructor term $z \in \mathit{CData}$ is equivalent to itself considered as either a configuration, or a residual term.

Rule PS-PRIM° describes the case where a monoconfiguration d taken from a polyconfiguration $z \mid \{l \mapsto d\}\, \Delta$ can be evaluated to a value z', that is, $d \rightsquigarrow z'$.

Rules PS-IF-T° and PS-IF-F° define the semantics of the **if** term.

Rule PS-LET° defines the **let** term by decomposing it into two parts and reducing to a configuration where the parts s_1 and s_2 are bound to separate liaison variables l' and l respectively.

Rule PS-CALL° defines the term **call** $f\, b$ by picking up the body f^p of a function f from a program p and applying the argument substitution to it.

Other rules specify specialization proper.

5.3.2 Residualization of conditional term

Rule PS-BRANCH uses the result of driving of a Boolean primitive to build a branching in residual code. It was commented above in Section 4.2.

PS-BASE$^\circ$
$$p : z \mid \{\} \rightsquigarrow m : z \qquad z \in \mathit{CData}$$

PS-PRIM$^\circ$
$$\frac{\begin{array}{c} d \rightsquigarrow z' \\ p : \overline{z \mid \Delta}^{\{l \mapsto z'\}} \rightsquigarrow m : r \end{array}}{p : z \mid \{l \mapsto d\}\, \Delta \rightsquigarrow m : r} \qquad \begin{array}{r} d \in \mathit{Prim} \\ z' \in \mathit{CData} \end{array}$$

PS-BRANCH
$$\frac{\begin{array}{c} d \prec \mathcal{T}(c_1,\, c_2) \\ p : \overline{z \mid \{l \mapsto d\}\, \Delta}^{c_1} \rightsquigarrow m : r_1 \\ p : \overline{z \mid \{l \mapsto d\}\, \Delta}^{c_2} \rightsquigarrow m : r_2 \end{array}}{p : z \mid \{l \mapsto d\}\, \Delta \rightsquigarrow m : \mathcal{T}(r_1,\, r_2)} \qquad \begin{array}{r} \mathrm{FVars}(d) \subseteq \mathrm{Dom}(\rho) \\ d \in \mathit{Prim} \end{array}$$

PS-IF-T$^\circ$
$$\frac{p : z \mid \{l \mapsto s_1\}\, \Delta \rightsquigarrow m : r}{p : z \mid \{l \mapsto \textbf{if}\ \mathrm{True}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\}\, \Delta \rightsquigarrow m : r}$$

PS-IF-F$^\circ$
$$\frac{p : z \mid \{l \mapsto s_2\}\, \Delta \rightsquigarrow m : r}{p : z \mid \{l \mapsto \textbf{if}\ \mathrm{False}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\}\, \Delta \rightsquigarrow m : r}$$

PS-LET$^\circ$
$$\frac{p : z \mid \{l' \mapsto s_1,\ l \mapsto s_2^{\{v \mapsto l'\}}\}\, \Delta \rightsquigarrow m : r}{p : z \mid \{l \mapsto \textbf{let}\ v = s_1\ \textbf{in}\ s_2\}\, \Delta \rightsquigarrow m : r} \qquad \begin{array}{r} l'\ \text{new} \\ l' \in \mathit{LVar} \end{array}$$

PS-CALL$^\circ$
$$\frac{p : z \mid \{l \mapsto f^{pb}\}\, \Delta \rightsquigarrow m : r}{p : z \mid \{l \mapsto \textbf{call}\ f\ b\}\, \Delta \rightsquigarrow m : r} \qquad \begin{array}{r} \mathrm{FVars}(f^p) \subseteq \mathrm{Dom}(b) \\ f \in \mathrm{Dom}(p) \\ f \in \mathit{FName} \\ b \in \mathit{Var} \to \mathit{Arg} \end{array}$$

PS-GEN
$$p : f^{mb} \rightsquigarrow m : \textbf{call}\ f\ b \qquad \begin{array}{r} \mathrm{FVars}(f^m) \subseteq \mathrm{Dom}(b) \\ f \in \mathrm{Dom}(m) \\ f \in \mathit{FName} \\ b \in \mathit{Var} \to \mathit{Arg} \end{array}$$

Figure 9: Polygenetic specialization

PS-SPLIT

$$\frac{\begin{array}{r} p : \overline{z \mid \Delta_1}^{\{l \mapsto u\}} \rightsquigarrow m : r_1 \\ p : l \mid \{l \mapsto s\}\, \Delta_2 \rightsquigarrow m : r_2 \end{array}}{p : z \mid \Delta_1\, \{l \mapsto s\}\, \Delta_2 \rightsquigarrow m : \textbf{let } u = r_2 \textbf{ in } r_1} \quad \begin{array}{l} V_1 \cap V_2 = \{l\} \\ u \text{ new} \\ u \in \mathit{Var} \end{array}$$

$$\text{where} \quad \begin{array}{l} V_1 = \mathrm{LVars}(z \mid \Delta_1) \\ V_2 = \mathrm{LVars}(\{l \mapsto s\}\, \Delta_2) \end{array}$$

Figure 10: Splitting a Configuration

Notice the case where a value of a conditional a is a configuration variable is absent. It is useless due to the syntactic restriction on the term a (see Section 2.1).

5.3.3 Generalization

The most interesting rule is axiom PS-GEN that defines the notion of generalization of a configuration together with folding into a residual **call** term. Reading the judgment

$$p : f^{mb} \rightsquigarrow m : \textbf{call } f\ b$$

from left to right in terms of production of residual code rather than its specification, we say that some configuration $f^{mb} = z \mid \Delta$ is generalized to configuration $f^m = z' \mid \Delta'$ with the substitution b such that $z \mid \Delta = \overline{z' \mid \Delta'}^{b}$. The term **call** $f\ b$ is residualized, where the function f has such a body $r = f^q$ that satisfies the specialization relation

$$p : z' \mid \Delta' \rightsquigarrow m : r$$

which is implied by the consistency requirement to the mapping m of residual function names to configurations.

5.3.4 Splitting a configuration

The definition of polygenetic specialization is incomplete without a rule that allows for *composition* of configurations if inference rules are read forwards, or *splitting* a configuration into two ones if inference rules are read backwards. Such rule PS-SPLIT is presented in Fig. 10.

Rules PS-GEN and PS-SPLIT are the only rules that do not allow for unambiguously constructing the specialization algorithm from the inference rules. They reveal the place in construction of supercompilers where decision-taking

$$\begin{array}{lll}
\textsc{ps-collapse} & \dfrac{p : \overline{z \mid \{l \mapsto s\}\,\Delta}^{\{l'\mapsto l\}} \rightsquigarrow m : r}{p : z \mid \{l \mapsto s,\ l' \mapsto s\}\,\Delta \rightsquigarrow m : r} & \\[2ex]
\textsc{ps-nonstrict}^{\circ} & \dfrac{p : z \mid \Delta \rightsquigarrow m : r}{p : z \mid \{l \mapsto s\}\,\Delta \rightsquigarrow m : r} & \begin{array}{l} l \notin \mathrm{LVars}(z \mid \Delta) \\ l \in \mathit{LVar} \end{array}
\end{array}$$

Figure 11: Extensions of interpretation and specialization

strategies when and how to generalize and when and how to split configurations are to be developed. This is the most sophisticated part of supercompilers. The *quality* of residual programs depends mainly on them, while the *correctness* is guaranteed by the mere fact the result matches these simple inference rules.

5.4 Extensions

The specialization relation can be infinitely extended by adding more and more rules that describe additional equivalences between source and residual programs. Figure 11 demonstrates two of them.

The first extension is *collapsed-jungle driving* [12, 13] defined by rule PS-COLLAPSE. It avoids multiple evaluation of equal terms by deleting one of two liaison variable bindings of the form $\{l \mapsto s,\ l' \mapsto s\}$ and replacing all occurrences of the deleted variable l' by the second variable l. The classic example of application of this rule is the transformation of the naive recursive definition of the Fibonacci function with exponential complexity to the definition with linear complexity.

The second rule PS-NONSTRICT$^\circ$ extends the relation to *non-strict semantics*, which means the possibility of evaluation of a function call without evaluation of its arguments. The specialization rules allow for arbitrary order of evaluation. All of the other rules preserve unevaluated terms even if their results are unneeded. With rule PS-NONSTRICT$^\circ$, which removes a binding of an unused liaison variable, the relation allows for both *non-strict semantics* and *lazy evaluation* in an interpreter as well as in a specializer.

Notice rule PS-COLLAPSE is considered a specialization relation rule (having no $^\circ$ mark), while PS-NONSTRICT$^\circ$ is marked with $^\circ$ as an interpretation rule. The reason for the difference is that the former rule does not change the language denotational semantics, while the latter does. The collapsed-jungle driving allows for merely achieving additional speed-up by specialization, while the non-strict extension changes the termination behavior of a program.

If rule PS-NONSTRICT$^\circ$ were applied only to specialization, then a residual program might have a larger domain than the source program. This is often the

Idempotency $\dfrac{p : s \;\Rightarrow\; q : r}{q : r \;\Rightarrow\; q : r}$	Transitivity $\dfrac{\begin{array}{l} p : s \;\Rightarrow\; q : r \\ q : r^c \;\Rightarrow\; q' : r' \end{array}}{p : s^c \;\Rightarrow\; q' : r'}$	Preservation of semantics by specialization $(p : s \overset{\circ}{\to} x) \Leftrightarrow (p : s \to x)$
Completeness $\dfrac{\begin{array}{l} p : s \;\Rightarrow\; q : r \\ p : s^c \;\to\; x \end{array}}{q : r^c \;\to\; x}$	Soundness $\dfrac{\begin{array}{l} p : s \;\Rightarrow\; q : r \\ q : r^c \;\to\; x \end{array}}{p : s^c \;\to\; x}$	Correctness $\dfrac{p : s \Rightarrow q : r}{(p : s^c \overset{\circ}{\to} x) \Leftrightarrow (q : r^c \overset{\circ}{\to} x)}$

Figure 12: Properties of the specialization relation

case of practical supercompilers for strict languages (*e.g.*, for Refal [9]), since supercompilers use lazy evaluation for achieving a better result. This is the reason for the widespread myth that supercompilation in essence violates the termination behavior. However, if both source and residual programs as well as a specializer use the same semantics—either strict, or non-strict—the semantics is preserved (provided some other rules do not violate it).

6 Properties of specialization relation

As it is common in mathematics, relations obey more interesting properties than functions. In Fig. 12 the most important properties of the specialization relation are summed up. For readability, the statements are written out in form of inference rules. Their validity can be proven from the specialization rules.

The properties are rather natural for such a relation. Some of the properties are mandatory: preservation of semantics, soundness, completeness, and their corollary—correctness. Others—idempotency and transitivity—are additional nice properties that allow for simpler and more natural reasoning about specialization in form of relation rather than function.

6.1 Idempotency

Intuitively, we consider returning an unchanged program to be a trivial case of specialization. One may expect that $p : s \Rightarrow p : s$ is true, that is, the specialization relation is *reflexive*. However, our rules require some specialization of function bodies always be performed, and hence many programs cannot occur in the right-hand sides of deducible judgments. In principle, it is easy to add several rules that would allow for "doing nothing", but we prefer the present version, which guide us in construction of non-trivial specializers. Nevertheless, any residual program is allowed to be its own specialization by our relation. Such property is referred to as *idempotency*.

Proposition 1 (Idempotency) *For all p, s, q, r such that*

$$p : s \Rightarrow q : r$$

the following judgment is deducible:

$$q : r \Rightarrow q : r$$

6.2 Transitivity

Specialization can be performed stepwise: specialization of a source program $p : s$ with respect to a part of arguments (let them be already substituted into s) followed by specialization of the residual program $q : r$ with respect to a part of the rest arguments (let them be represented by a substitution c) produces the second residual program $q' : r'$, which may be expected to be a specialization of the source program with respect to all information known so far. This property (referred to as *transitivity*) is not generally true when specializer *functions* are concerned, but it may hold for a specialization *relation*. This is indeed our case.

Proposition 2 (Transitivity) *For all p, s, c, q, r, q', r' such that*

$$\begin{array}{l} p : s \;\Rightarrow q : r \\ q : r^c \Rightarrow q' : r' \end{array}$$

the following judgment is deducible:

$$p : s^c \Rightarrow q' : r'$$

6.3 Soundness

Consider the special case of transitivity where the second specialization is interpretation, that is, the residual program is empty, $q' = \emptyset$. In this case transitivity means: if a residual program $q : r$ when run with arguments given by a contraction c produces some result x, the source program $p : s$ run with the same arguments also terminates and gives the same result. This property is *soundness* of specialization.

Proposition 3 (Soundness) *For all p, s, c, q, r, x such that*

$$\begin{array}{l} p : s \;\Rightarrow q : r \\ q : r^c \rightarrow x \end{array}$$

the following judgment is deducible:

$$p : s^c \rightarrow x$$

Soundness is an immediate corollary of transitivity and a necessary condition for correctness.

6.4 Completeness

The converse property to soundness is completeness: if a source program $p : s$ when run with arguments c produces some result x, a residual program $q : r$ run with the same arguments also terminates and gives the same result.

Proposition 4 (Completeness) *For all p, s, q, r, c, x such that*

$$\begin{array}{l} p : s \Rightarrow q : r \\ p : s^c \rightarrow x \end{array}$$

the following judgment is deducible:

$$q : r^c \rightarrow x$$

Completeness is one more necessary condition for correctness.

6.5 Preservation of semantics

Recall we use a subset of the specialization rules as the definition of the language semantics. Hence, the fact that specialization includes interpretation is trivial. However, we must ensure that the specialization rules do not occasionally extend the semantics. Formally speaking, the following proposition must hold and it holds for our specialization relation indeed.

Proposition 5 (Preservation of semantics) *For all p, s, x such that*

$$p : s \rightarrow x$$

the following judgment is deducible:

$$p : s \overset{\circ}{\rightarrow} x$$

6.6 Correctness

Since the semantics of the object language is represented by a part of the inference rules, the correctness of the specialization relation is its internal property that can be expressed as follows.

The last two judgments mean interpretation of source and residual programs $p : s$ and $q : r$ with values supplied by a contraction c produces equal results x.

Proposition 6 (Correctness) *For all p, s, q, r such that*

$$p : s \Rightarrow q : r$$

it holds that for all c and x the following judgments are deducible or not deducible simultaneously:

$$\begin{array}{l} p : s^c \overset{\circ}{\rightarrow} x \\ q : r^c \overset{\circ}{\rightarrow} x \end{array}$$

The correctness is an immediate corollary of soundness, completeness and preservation of semantics.

7 Conclusion and Related Work

This paper presents a formal specification of a class of specializers by inference rules in the style of Natural Semantics [6]. The rules define a relation between source and residual programs, which partial evaluation and supercompilation obey. The proposed intensional relation lies between algorithmic definitions of specializers and the extensional equivalence of programs.

The specialization relation definition declaratively captures the essential notions of supercompilation: *configuration*, *driving*, *generalization of a configuration*, *splitting a configuration*, as well as advanced notions like *collapsed-jungle driving* and the variations of the strictness of semantics, while abstracting from algorithmic problems of when, what and how to generalize and split, and when to terminate. It provides the basis for the correctness proofs of supercompilers and an alternative to that of partial evaluators [4, 2]. To prove the correctness of a particular specializer we just need to prove that its inputs and outputs satisfy the relation. By nature of Natural Semantics, the definition in form of inference rules allows for automated derivation of specializers that satisfy it as well as checkers of the correctness of residual programs.

An earlier version of specialization relation definition was presented at the Dagstuhl Seminar on Partial Evaluation, where only abstract [7] was published. It continues the work started in [3] and aimed at clarifying and formalizing the ideas of supercompilation. This paper gives a generalization of the definition presented in [8] from monogenetic to polygenetic case.

In Turchin's original papers [16, 17] and others, the essential ideas of supercompilation and technical details of algorithms were not separated enough to give their short formal definition. Later on, several works have been done to fill this gap, *e.g.*, [3, 13, 14, 15]. All of them formalize the function of the supercompiler, while our work is, to our knowledge, the first attempt to define an input-output relation, which specializers based on both supercompilation and partial evaluation satisfy. The closest related work is [13, 14] where the notion of the graph of configurations is formalized by inference rules that deduce the arcs of the graph.

The specialization relation obeys a number of nice properties: idempotency, transitivity and its corollary soundness, completeness, correctness, and others.

Future work will include development of specialization relation definitions for more sophisticated languages, including object-oriented ones, further investigation into their properties, and construction of supercompilers that satisfy the specialization relation and hence are provably correct.

8 Acknowledgments

The material of this paper is a refinement of Valentin Turchin's ideas of supercompilation. I would like to express my sincere gratitude to him for introducing me into this exciting field more than three decades ago and deeply influencing my life and work during all years.

I am very grateful to Sergei Romanenko and Yuri Klimov for their valuable comments and suggestions to make the formalism simpler and more readable.

References

[1] S. M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications).* Nauka, Moscow, 1995. (In Russian).

[2] C. Consel and S. C. Khoo. On-line and off-line partial evaluation: semantic specifications and correctness proofs. *Journal of Functional Programming*, 5(4):461–500, Oct. 1995.

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

[4] C. K. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM TOPLAS*, 14(2):147–172, 1992.

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

[6] G. Kahn. Natural Semantics. In F. G. Brandenburg, G. Vidal-Naquet, and W. Wirsing, editors, *STACS87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[7] A. V. Klimov. A specification of a class of supercompilers. In O. Danvy, R. Glück, and P. Thiemann, editors, *Draft Proceedings of the Dagstuhl Seminar on Partial Evaluation*, page 232. Technical Report WSI-96-6, Universität Tübingen, Germany, 1996.

[8] A. V. Klimov. Specifying monogenetic specializers by means of a relation between source and residual programs. In *Perspectives of Systems Informatics (Proc. 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006)*, volume 4378 of *Lecture Notes in Computer Science*, pages 248–259. Springer-Verlag, 2007.

[9] A. P. Nemytykh. *Superkompilyator SCP4: Obschaya struktura (Supercompiler SCP4: general structure).* Editorial URSS, Moscow, 2007. (In Russian).

[10] S. A. Romanenko. Arity raiser and its use in program specialization. In N. D. Jones, editor, *ESOP'90, Copenhagen, Denmark*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360. Springer-Verlag, May 1990.

[11] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(10), October 1996.

[12] J. P. Secher. Driving in the jungle. In O. Danvy and A. Filinski, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 198–217, Aarhus, Denmark, May 2001. Springer-Verlag.

[13] J. P. Secher and M. H. B. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2000.

[14] M. H. Sørensen and R. Glück. Introduction to supercompilation. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.

[15] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[16] V. F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.

[17] V. F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[18] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.