Klimov A.V.

A program specialization relation based on supercompilation and its properties

A. V. Klimov

A program specialization relation
based on supercompilation and its properties
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.

Andrey V. Klimov. 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 \( \text{spec} \) 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 \( p \): \( q = \text{spec}(p, D) \). The equivalence of the source and residual programs is understood \textit{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 \[4, 2, 11\]:

\[ q = \text{spec}(p, D) \Rightarrow p \approx_D q. \]

In this paper we define a constructive, \textit{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:

\[ q = \text{spec}(p, D) \Rightarrow p \simeq_D q \Rightarrow p \approx_D q. \]

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: \textit{configuration}, \textit{driving}, \textit{generalization of a configuration}, \textit{splitting a configuration}, as well as \textit{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 \textit{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 \textit{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
in program code coincides with the value it represents. That is, \( \text{Data} \subset \text{Term} \), where \( \text{Term} \) is the domain of program terms.

**Configuration data** Another extension of \( \text{Data} \) 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, \( u \in \text{Var} \), 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 \ v_1 \ = \ \text{loop } v_1 \ [

\text{loop } [] v_2 \ = \ v_2

\text{loop } (v_4 : v_5) v_2 \ = \ \text{loop } v_5 (v_4 : v_2)

---

p = \{ \text{rev} \mapsto \text{call loop } \{v_1 \mapsto v_1, \ v_2 \mapsto \text{Nil}\}, \text{loop} \mapsto \text{let } v_3 = \text{cons? } v_1 \text{ in}

\text{if } v_3

\text{then let } v_4 = \text{fst } v_1 \text{ in}

\text{let } v_5 = \text{snd } v_1 \text{ in}

\text{call loop } \{v_1 \mapsto v_5, \ v_2 \mapsto \text{Cons } v_4 \ v_2\}

\text{else } v_2\}

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

z | \Delta = l_0 \ \{l_0 \mapsto s_0\} \text{ — 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*\textsuperscript{1} variables, $l \in LVar$, 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: Data $\subset$ CData $\subset$ Term.

**Primitives** Data values are analyzed by primitive predicates $\text{equ? } v \ a$ (are two values equal?) and $\text{cons? } v$ (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 $\text{fst } v$ and $\text{snd } v$, 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 $\text{fst } v$ and $\text{snd } v$: they can occur only on the positive branch of an if-term with the conditional $\text{cons? } v$.

\textsuperscript{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, \texttt{equ?} or \texttt{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 \texttt{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: \( v \in \text{Var} \) or \( a \in \text{Arg} \).

Notation
- \( \text{FVars}(t) \) denotes the set of free variables occurring in term \( t \),
- \( \text{LVars}(t) \) the set of liaison variables in a term or in a configuration.
- \( \text{Dom}(m) \) and \( \text{Rng}(m) \) 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 *polyconfiguration* 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 $\Delta$ the binding of liaison variables to terms (monoconfigurations), referred to as an *explication*.\(^3\) Topmost terms are restricted to $z \in CData$, 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:

$$z \mid \Delta = l_0 \mid \{l_0 \mapsto \text{call} \{v_1 \mapsto \text{Cons A (\ldots )}\}\}.$$  

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 $\Delta_1$ and $\Delta_2$ 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 $\theta$ and use a kind of power notation $t^\theta$.

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

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.
\[\_^- : FName \times Prog \rightarrow Term\]
\[f^{\{\ldots, f \mapsto s, \ldots\}} = s\]
\[\_^- : Term \times (Contr \cup Args \cup Expl) \rightarrow Term\]
\[z^\theta = z \text{ if } z \in \text{Atom} \cup \text{Var} \cup \text{LVar} \text{ and } z \not\in \text{Dom}(\theta)\]
\[v^{\{\ldots, v \mapsto z, \ldots\}} = z\]
\[\text{Cons } a_1 a_2^\theta = \text{Cons } a_1^\theta a_2^\theta\]
\[\text{fst } v^\theta = \text{fst } v^\theta\]
\[\text{snd } v^\theta = \text{snd } v^\theta\]
\[\text{cons? } v^\theta = \text{cons? } v^\theta\]
\[\text{equ? } v^\theta a = \text{equ? } v^\theta a^\theta\]
\[\text{if } v^\theta \text{ then } s_1^\theta \text{ else } s_2^\theta = \text{if } v^\theta \text{ then } s_1^\theta \text{ else } s_2^\theta\]
\[\text{let } v = s_1^\theta \text{ in } s_2^\theta = \text{let } v = s_1^\theta \text{ in } s_2^\theta\]
\[\text{call } f b^\theta = \text{call } f b^\theta\]
\[\_^- : Args \times (Contr \cup Args \cup Expl) \rightarrow Args\]
\[\{v_1 \mapsto a_1, \ldots, v_n \mapsto a_n\}^\theta = \{v_1 \mapsto a_1^\theta, \ldots, v_n \mapsto a_n^\theta\}\]
\[\_^- : Expl \times (Contr \cup Expl) \rightarrow Expl\]
\[\{l_1 \mapsto s_1, \ldots, l_n \mapsto s_n\}^\theta = \{l_1 \mapsto s_1^\theta, \ldots, l_n \mapsto s_n^\theta\}\]
\[\_^- : PConf \times (Contr \cup Expl) \rightarrow PConf\]
\[z^\theta | \Delta = z^\theta | \Delta^\theta\]

Figure 3: The definition of substitution \(t^\theta\) for those domains which it is applied to in the specialization relation definition.
• \( s^b = f^b \) builds a monoconfiguration from a program term \( s = f^b \) and the argument binding \( b \) of a monoconfiguration call \( fb \);

• \( z \mid \Delta \{ l \mapsto z' \} \) 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 \( z \mid \Delta \);

• \( sc \) and \( z \mid \Delta c \) 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, \( \text{equ} \? v \ a \) and \( \text{cons} \? v \), 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: \( \text{equ} \? u \ k \), \( \text{equ} \? u \ u' \), and \( \text{cons} \? u \), 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 \( \{ u \mapsto \text{Cons} \ u_1 \ u_2 \} \), where \( k \in \text{Atom} \), \( u_1 \) 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 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 \( \{ u \mapsto u^{-\text{Cons}} \} \).
| $d \leadsto z$ | **Interpretation and transient driving** of a primitive $d$ produces a value $z$. |
| $d \prec T(c_1, c_2)$ | **Driving with branching:** Driving of a primitive $d$ produces a branching represented by a residual conditional term $T(\_, \_)$ with two free positions for positive and negative branches. In the right-hand side $T(c_1, c_2)$ 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 $l | \{ l \mapsto s \}$. |
| $p : z | \Delta \leadsto m : r$ | **Specialization to a term:** A residual term $r$ is a specialization of a source program $p$ with an initial polyconfiguration $z | \Delta$ with respect to a mapping $m$ of residual function names to polyconfigurations. |
| $p : z | \Delta \leadsto \emptyset : z$ | **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, $z \in CData$, 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 $p : l | \{ l \mapsto s \} \leadsto \emptyset : z$. |
| $p : s \overset{\circ}{\rightarrow} z$ | **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 $FVars(s) = \emptyset$ and hence $z$ is a ground value, $z \in Data$. |

Figure 4: Judgments
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 $^\circ$. 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 $p : z | \Delta \rightsquigarrow m : r$ contain a residual program $q$ and an auxiliary mapping $m$ (explained in Section 5.1 below), which has the same domain, $\text{Dom}(q) = \text{Dom}(m) \subseteq \text{FName}$, 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: $p : s \xrightarrow{\circ} z$ (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, $\text{FVars}(s) = \emptyset$. 

\begin{center}
\begin{tabular}{ll}
TR-DRV & $p : l | \{l \mapsto s\} \rightsquigarrow \emptyset : z$ \\
& \hspace{1cm} $p : s \rightarrow z$ $\quad z \in CData$
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ll}
INT & $p : s \rightarrow z$ \\
& $p : s \xrightarrow{\circ} z$ if inferred with $^\circ$-rules only \\
& $\quad z \in CData$
\end{tabular}
\end{center}

Figure 5: Transient driving and semantics as a subset of specialization
Definition 1 (Interpretation, semantics) A source program $p$ with an initial term $s$ (in which arguments have been substituted) without configuration variables, $\text{FVars}(s) = \emptyset$, evaluates to a term $x \in \text{Data}$ if the following judgment is deducible:

$$p : s \rightarrow x$$

4 Interpretation and specialization of primitives

4.1 Interpretation and transient driving of primitives

A judgment of the form $d \rightarrow z$ 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: $k \in \text{Atom} \subset \text{Data}$, $x \in \text{Data}$, $z \in \text{CData}$, 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 $\text{i-CONS-F}^\circ$ and $\text{i-EQ-F}^\circ$, which defines inequality, require values to be ground.

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

- $\text{D-CONS-F}$ and $\text{D-EQ-UCF}$ — 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;
- $\text{D-EQ-COM}$ — commutativity of $\text{equ}$?
- $\text{D-EQ-UKF}$ — returning False in case where a configuration variable $u$ contains in its negative set the atom $k$ it is compared to;
- $\text{D-EQ-UUF}$ — 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 $\text{PS-BRANCH}$ if we consider infinite residual terms $r$. However, it would be another theory.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Expression</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-VALUE&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( z \leadsto z )</td>
<td>( z \in \text{CData} )</td>
</tr>
<tr>
<td>I-FST&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( \text{fst} \ (\text{Cons} \ z_1 \ z_2) \leadsto z_1 )</td>
<td></td>
</tr>
<tr>
<td>I-SND&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( \text{snd} \ (\text{Cons} \ z_1 \ z_2) \leadsto z_2 )</td>
<td></td>
</tr>
<tr>
<td>I-CONS-T&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( \text{cons}? \ (\text{Cons} \ z_1 \ z_2) \leadsto \text{True} )</td>
<td></td>
</tr>
<tr>
<td>I-CONS-F&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( \text{cons}? \ k \leadsto \text{False} )</td>
<td>( k \in \text{Atom} )</td>
</tr>
<tr>
<td>I-EQ-T&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( \text{equ}? \ z \ z \leadsto \text{True} )</td>
<td></td>
</tr>
<tr>
<td>I-EQ-F&lt;sup&gt;°&lt;/sup&gt;</td>
<td>( \text{equ}? \ x_1 \ x_2 \leadsto \text{False} )</td>
<td>( x_1 \neq x_2, \ x_1, x_2 \in \text{Data} )</td>
</tr>
<tr>
<td>D-EQ-CK</td>
<td>( \text{equ}? \ (\text{Cons} \ z_1 \ z_2) \ k \leadsto \text{False} )</td>
<td>( k \in \text{Atom} )</td>
</tr>
<tr>
<td>D-EQ-CC</td>
<td>( \text{equ}? \ z_{1i} \ z_{2i} \leadsto \text{False} )</td>
<td>( \text{equ}? \ (\text{Cons} \ z_{1i} \ z_{12}) \ (\text{Cons} \ z_{2i} \ z_{22}) \leadsto \text{False} )</td>
</tr>
</tbody>
</table>

Figure 6: Interpretation and transient driving of primitives

Term \( T(\_, \_) \) 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 \( T(\_, \_) \) 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 \( \text{cons}? \) and \( \text{equ}? \). The branching happens when a configuration variable \( u \) (or two variables \( u_1 \) and \( u_2 \) in the case of the \( \text{equ}? \) term) prohibits from performing an evaluation step. Notice the branching rules perform no evaluation step of the source program, but just produce a residual \textbf{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 \textit{PS-BRANCH}, which produces initial configurations for branches, and transient driving of the primitive has been “invoked” by rule \textit{PS-PRIM<sup>°</sup>} in Fig. 9.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D-CONS-F</strong></td>
<td>( \text{cons? } u \leadsto \text{False} )</td>
</tr>
<tr>
<td><strong>D-CONS</strong></td>
<td>( \text{cons? } u \prec ) let ( u_0 = \text{cons? } u ) in ( \text{if } u_0 ) then let ( u_1 = \text{fst } u ) in let ( u_2 = \text{snd } u ) in ( { u \mapsto \text{Cons } u_1 u_2 } ) else ( { u \mapsto u^{-\text{Cons}} } )</td>
</tr>
<tr>
<td><strong>D-EQ-COM</strong></td>
<td>( \text{equ? } u \leadsto T ) ( \text{equ? } z \leadsto T )</td>
</tr>
<tr>
<td><strong>D-EQ-UUF</strong></td>
<td>( \text{equ? } u_1 u_2 \leadsto \text{False} )</td>
</tr>
<tr>
<td><strong>D-EQ-UU</strong></td>
<td>( \text{equ? } u_1 u_2 \prec ) let ( u_0 = \text{equ? } u_1 u_2 ) in ( \text{if } u_0 ) then ( { u_1 \mapsto u_2 } ) else ( { u_1 \mapsto u_1^{-u_2}, u_2 \mapsto u_2^{-u_1} } )</td>
</tr>
<tr>
<td><strong>D-EQ-UCF</strong></td>
<td>( \text{equ? } u (\text{Cons } z_1 z_2) \leadsto \text{False} )</td>
</tr>
<tr>
<td><strong>D-EQ-UC</strong></td>
<td>( \text{cons? } u \prec T ) ( \text{equ? } u (\text{Cons } z_1 z_2) \prec T )</td>
</tr>
</tbody>
</table>

Figure 7: Driving of primitives
The correctness of this derivation is based on the perfectness [3] of contrac-
tions $c_1$ 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 $T(\_, \_)$ in Fig. 7 it meets the
following property: the value of $T(x_1, x_2)$ 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 $\text{equ?}$ and $\text{cons?}$ terms that infer judgments
of the form $d \ll T(c_1, c_2)$ mean:

- Rule D-CONS defines the branching in a residual program that corresponds
to a monoconfiguration of the form $\text{cons?} u$. 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 posi-
tive branch, and two complementary contractions $\{u \mapsto \text{Cons } u_1 u_2\}$ and
$\{u \mapsto u^{\text{Cons}}\}$ occupying in the term $T(\_, \_)$ the positions of the positive
and negative branches respectively.

- Rules D-EQ-UK and D-EQ-UU analogously define the branchings corre-
sponding to monoconfigurations $\text{equ?} u$ and $\text{equ?} u_1 u_2$ 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, $\text{equ?} u (\text{Cons } z_1 z_2)$, to the test of whether $u$ is a Cons term or not,
$\text{cons?} u$. Recall the judgment $d \ll T(c_1, c_2)$ defines no evaluation step,
only a branching. So, the rule reads as follows: to advance driving of the
configuration $\text{equ?} u (\text{Cons } z_1 z_2)$ residualize the same branching as for
$\text{cons?} u$. After that, the evaluation step will be performed by the driving
rules for the $\text{equ?}$ term.

5 Interpretation and specialization of control
terms

A judgment of the form $p : z | \Delta \rightsquigarrow m : r$ 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 $z | \Delta$ 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 $z | \Delta$ and that of the
residual term $r$, evaluation of the configuration $z | \Delta$ 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 $\text{call} f_i b_i$ occurring in $r$, evaluate the configuration $f_i^{mb_i} = z_i | \Delta_i^{b_i}$ with the source program $p$. Here the arguments $b_i$ are substituted into the configuration $z_i | \Delta_i$ bound to the function name $f_i$ by the mapping $m$;

2) evaluate the term $r$ using the thus obtained values of the $\text{call}$ 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 $m : \text{Dom}(q) \rightarrow \text{PConf}$ 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 $\text{call}$ terms as explained above.

**Definition 2 (Consistency)** A mapping $m : \text{Dom}(q) \rightarrow \text{PConf}$ of residual function names to configurations is consistent with source and residual programs $p$ and $q$ if for every residual function name $f \in \text{Dom}(q)$ 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, $p,q \in \text{Prog}$, $s,r \in \text{CData}$. Note only the terms $s$ and $r$ that have the same configuration variables can relate by the specialization relation, $\text{FVars}(s) = \text{FVars}(r)$. 
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 \( l \{l \mapsto s\} \).

**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:

\[
p : l \{l \mapsto s\} \rightsquigarrow m : r.
\]

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 \( \text{spec} \) 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 \( \text{ps-base} \) asserts the evident fact that a constructor term \( z \in \text{CData} \) is equivalent to itself considered as either a configuration, or a residual term.

Rule \( \text{ps-prim} \) describes the case where a monoconfiguration \( d \) taken from a polyconfiguration \( z \{l \mapsto d\} \Delta \) can be evaluated to a value \( z' \), that is, \( d \rightsquigarrow z' \).

Rules \( \text{ps-if-t} \) and \( \text{ps-if-f} \) define the semantics of the \( \text{if} \) term.

Rule \( \text{ps-let} \) defines the \( \text{let} \) 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 \( \text{ps-call} \) defines the term \( \text{call} fb \) 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 \( \text{ps-branch} \) uses the result of driving of a Boolean primitive to build a branching in residual code. It was commented above in Section 4.2.
<table>
<thead>
<tr>
<th>PS-BASE°</th>
<th>$p : z \mid {} \rightsquigarrow m : z$</th>
<th>$z \in \text{CData}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS-PRIM°</td>
<td>$\frac{d \rightsquigarrow z'}{p : z \mid {l \mapsto d} \Delta \rightsquigarrow m : r}$</td>
<td>$d \in \text{Prim}$</td>
</tr>
<tr>
<td></td>
<td>$\frac{p : z \mid {l \mapsto d} \Delta \rightsquigarrow m : r}{z' \in \text{CData}}$</td>
<td></td>
</tr>
<tr>
<td>PS-BRANCH</td>
<td>$\frac{d &lt; T(c_1, c_2)}{p : z \mid {l \mapsto d} \Delta \rightsquigarrow m : r_1}$</td>
<td>FVars($d$) $\subseteq \text{Dom}(\rho)$</td>
</tr>
<tr>
<td></td>
<td>$\frac{p : z \mid {l \mapsto d} \Delta \rightsquigarrow m : r_2}{p : z \mid {l \mapsto d} \Delta \rightsquigarrow m : T(r_1, r_2)}$</td>
<td>$d \in \text{Prim}$</td>
</tr>
<tr>
<td>PS-IF-T°</td>
<td>$p : z \mid {l \mapsto s_1} \Delta \rightsquigarrow m : r$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$p : z \mid {l \mapsto \text{if True then } s_1 \text{ else } s_2} \Delta \rightsquigarrow m : r$</td>
<td></td>
</tr>
<tr>
<td>PS-IF-F°</td>
<td>$p : z \mid {l \mapsto s_2} \Delta \rightsquigarrow m : r$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$p : z \mid {l \mapsto \text{if False then } s_1 \text{ else } s_2} \Delta \rightsquigarrow m : r$</td>
<td></td>
</tr>
<tr>
<td>PS-LET°</td>
<td>$p : z \mid {l \mapsto s_1, l \mapsto s_2} \Delta \rightsquigarrow m : r$</td>
<td>$l' \text{ new}$</td>
</tr>
<tr>
<td></td>
<td>$p : z \mid {l \mapsto \text{let } v = s_1 \text{ in } s_2} \Delta \rightsquigarrow m : r$</td>
<td>$l' \in \text{LVar}$</td>
</tr>
<tr>
<td>PS-CALL°</td>
<td>$p : z \mid {l \mapsto f^{p}} \Delta \rightsquigarrow m : r$</td>
<td>FVars($f^p$) $\subseteq \text{Dom}(b)$</td>
</tr>
<tr>
<td></td>
<td>$p : z \mid {l \mapsto \text{call } f \ b} \Delta \rightsquigarrow m : r$</td>
<td>$f \in \text{Dom}(p)$</td>
</tr>
<tr>
<td></td>
<td>$FVars(f^{m}) \subseteq \text{Dom}(b)$</td>
<td>$f \in \text{FName}$</td>
</tr>
<tr>
<td></td>
<td>$f \in \text{Dom}(m)$</td>
<td>$b \in \text{Var} \rightarrow \text{Arg}$</td>
</tr>
<tr>
<td>PS-GEN</td>
<td>$p : f^{mb} \rightsquigarrow m : \text{call } f \ b$</td>
<td>FVars($f^{m}$) $\subseteq \text{Dom}(b)$</td>
</tr>
<tr>
<td></td>
<td>$f \in \text{Dom}(m)$</td>
<td>$b \in \text{FName}$</td>
</tr>
<tr>
<td></td>
<td>$b \in \text{Var} \rightarrow \text{Arg}$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Polygenetic specialization
PS-SPLIT

\[
\begin{array}{c}
p : z \mid \Delta_1 \leadsto m : r_1 \\
p : l \mid \{l \mapsto s\} \Delta_2 \leadsto m : r_2 \\
p : z \mid \Delta_1 \{l \mapsto s\} \Delta_2 \leadsto m : \text{let } u = r_2 \text{ in } r_1
\end{array}
\]

\[V_1 \cap V_2 = \{l\}\]

\[u \text{ new}\]

\[u \in \text{Var}\]

where \(V_1 = \text{LVars}(z \mid \Delta_1)\)

\[V_2 = \text{LVars}(\{l \mapsto s\} \Delta_2)\]

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 \textsc{ps-gen} that defines the notion of generalization of a configuration together with folding into a residual \textbf{call} term. Reading the judgment

\[p : f^{mb} \leadsto m : \textbf{call } f \ b\]

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 \(z \mid \Delta = z' \mid \Delta'^b\). The term \textbf{call } f \ b\) is residualized, where the function \(f\) has such a body \(r = f^g\) that satisfies the specialization relation

\[p : z' \mid \Delta' \leadsto m : r\]

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 \textsc{ps-split} is presented in Fig. 10.

Rules \textsc{ps-gen} and \textsc{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
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 \cite{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

<table>
<thead>
<tr>
<th>PS-Collapse</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p : z \mid {l \mapsto s} \Delta \leadsto m : r )</td>
</tr>
<tr>
<td>( p : z \mid {l \mapsto s, l' \mapsto s} \Delta \leadsto m : r )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PS-Nonstrict(^{\circ})</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p : z \mid \Delta \leadsto m : r ) ( l \notin \text{LVars}(z \mid \Delta) )</td>
</tr>
<tr>
<td>( p : z \mid {l \mapsto s} \Delta \leadsto m : r ) ( l \in \text{LVar} )</td>
</tr>
</tbody>
</table>

Figure 11: Extensions of interpretation and specialization
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  
\[
p : s \Rightarrow q : r
\quad q : r^c \Rightarrow q' : r'
\]
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  
\[
p : s \Rightarrow q : r
\quad q : r^c \rightarrow x
\]
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
\[
\begin{align*}
 p : s & \Rightarrow q : r \\
 p : s^c & \rightarrow x
\end{align*}
\]
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:
\[
p : s^\circ \rightarrow x
\]

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:
\[
\begin{align*}
p : s^c & \rightarrow x \\
q : r^c & \rightarrow x
\end{align*}
\]
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 super-compilation. 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


