On the Design of Generic Static Analyzers for Imperative Languages

ROBERTO BAGNARA
Department of Mathematics, University of Parma, Italy
and
PATRICIA M. HILL
School of Computing, University of Leeds, UK
and
ANDREA PESCETTI, and ENEA ZAFFANELLA
Department of Mathematics, University of Parma, Italy

The design and implementation of precise static analyzers for significant fragments of imperative languages like C, C++, Java and Python is a challenging problem. In this paper, we consider a core imperative language that has several features found in mainstream languages such as those including recursive functions, run-time system and user-defined exceptions, and a realistic data and memory model. For this language we provide a concrete semantics —characterizing both finite and infinite computations— and a generic abstract semantics that we prove sound with respect to the concrete one. We say the abstract semantics is generic since it is designed to be completely parametric on the analysis domains: in particular, it provides support for *relational* domains (i.e., abstract domains that can capture the relationships between different data objects). We also sketch how the proposed methodology can be extended to accommodate a larger language that includes pointers, compound data objects and non-structured control flow mechanisms. The approach, which is based on structured, big-step G^∞SOS operational semantics and on abstract interpretation, is modular in that the overall static analyzer is naturally partitioned into components with clearly identified responsibilities and interfaces, something that greatly simplifies both the proof of correctness and the implementation.

Categories and Subject Descriptors: F3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs.

General Terms: Languages, Verification.

Additional Key Words and Phrases: Abstract interpretation, structured operational semantics.

1. INTRODUCTION

The last few years have witnessed significant progress toward achieving the ideal of the program verification grand challenge [Hoa03]. Still, the distance separating us from that ideal can be measured by the substantial lack of available tools that are able to verify the absence of relevant classes of run-time errors in code written in (reasonably rich fragments of) mainstream imperative languages like C, C++, Java and Python. True: there is a handful of commercial products that target generic applications written in C, but little is known about them. In contrast, several papers explain the essence of the techniques employed by the *ASTRÉE* analyzer

This work has been partly supported by MIUR project "AIDA — Abstract Interpretation: Design and Applications" and by a Royal Society (UK) International Joint Project (ESEP) award.

to formally and automatically verify the absence of run-time errors in large safety-critical embedded control/command codes [BCC+02; BCC+03]; however, ASTRÉE is specially targeted at a particular class of programs and program properties, so that widening its scope of application is likely to require significant effort [Cou05]. It is interesting to observe that, among the dozens of software development tools that are freely available, there are hardly any that, by analyzing the program semantics, are able to certify the absence of important classes of run-time hazards such as, say, the widely known *buffer overflows* in C code.

The reason for the current, extreme scarcity of the resource "precise analyzers for mainstream programming languages" is that the design and implementation of such analyzers is a very challenging problem. The theory of abstract interpretation [CC77a; CC92a] is crucial to the management of the complexity of this problem and, in fact, both ASTRÉE and the existing commercial analyzers are (as far as we know) based on it. Static analysis via abstract interpretation is conducted by mimicking the execution of the analyzed programs on an *abstract domain*. This is a set of computable representations of program properties equipped with all the operations required to mirror, in an approximate though correct way, the real, *concrete* executions of the program. Over the last decade, research and development on the abstract domains has led to the availability of several implementations of a wide range of abstract domains: from the most efficient though imprecise, to the most precise though inefficient. Simplification and acceleration techniques have also been developed to mitigate the effects of this complexity/precision trade-off. So the lack of semantics-based static analyzers is not ascribable to a shortage of abstract domains and their implementations. The point is that there is more to a working analyzer than a collection of abstract domains:

(i) A *concrete semantics* must be selected for the analyzed language that models all the aspects of executions that are relevant to the properties of interest. This semantics must be recognizable as a sound characterization of the language at the intended level of abstraction.

(ii) An *abstract semantics* must be selected and correlated to the concrete semantics. This requires a proof of correctness that, while greatly simplified by abstract interpretation theory, can be a time-consuming task by highly qualified individuals.

(iii) An algorithm to finitely and efficiently compute (approximations of) the abstract semantics must be selected.

(iv) For good results, the abstract domain needs to be an object that is both complex and easily adaptable. So, instead of designing a new domain from scratch, it is often better if one can be obtained by combining simpler, existing, abstract domains. Even though the theory of abstract interpretation provides important conceptual instruments for the design of such a combination, a significant effort is still needed to achieve, as far as possible, the desired precision and efficiency levels. Note that this point can have an impact on points (ii) and (iii): a *generic* abstract semantics has the advantage of not requiring an entirely new proof and a new algorithm each time the abstract domain changes.

This paper, which is the first product of a long-term research plan that is meant to deal with all of the points above, specifically addresses points (i) and (ii) and refers to a slight generalization of existing techniques for point (iii).

1.1 Contribution

We build on ideas that have been around for quite some time but, as far as we know, have never been sufficiently elaborated to be applied to the description and analysis of realistic imperative languages. In extreme synthesis, the contribution consists in filling a good portion of the gaps that have impeded the application of these ideas to complex imperative programming languages such as C.[1]

More precisely, here we define the concrete and generic abstract semantics constructions for a language —called CPM— that incorporates all the features of mainstream, single-threaded imperative programming languages that can be somehow problematic from the point of view of static analysis. Most notably, the CPM language features: a non-toy memory model; exceptions; run-time errors modeled via exceptions (for instance, an exception is raised whenever a division by zero is attempted, when a stack allocation request causes a stack overflow or when other memory errors occur); array types; pointer types to both data objects and functions; short-circuit evaluation of Boolean operators; user-defined (possibly recursive) functions; and non-structured control flow mechanisms.

For the description of the concrete dynamic semantics of the language we have used a structured operational semantics (SOS) approach extended to deal with infinite computations, mainly building on the work of Kahn, Plotkin and Cousot. With respect to what can be found in the literature, we have added the treatment of all non-structured control flow mechanisms of the C language. Of course, as the ultimate goal of this research is to end up with practical analysis tools, the concrete dynamic semantics has been defined in order to facilitate as much as possible the subsequent abstraction phase. Still, our dynamic semantics retains all the traditional good features: in particular, the concrete rule schemata are plainly readable (assuming the reader becomes sufficiently familiar with the unavoidable notational conventions) and fairly concise.

For the abstract semantics, we build on the work of Schmidt by providing the concrete dynamic semantics rules with abstract counterparts. As far as we know, this is the first time that Schmidt's proposal is applied to the analysis of a realistic programming language [D. Schmidt, personal communication, 2004]. A remarkable feature of our abstract semantics is that it is truly generic in that it fully supports relational abstract domains: the key step in this direction is the identification and specification of a suitable set of operators on (concrete and abstract) memory structures, that allow for domain-independent approximations but without inherent limitations on the obtainable precision.

[1] It is worth noticing that we improperly refer to the C language to actually mean some more constrained language —like CIL, the *C Intermediate Language* described in [NMRW02]— where all ambiguities have been removed, in addition to an ABI (*Application Binary Interface*) that further defines its semantics. Similarly, by 'Python' we mean a tractable subset of the language, such as the *RPython* subset being developed by the *PyPy* project (`http://pypy.org/`).

Schmidt's proposal about the abstract interpretation of natural semantics has, in our opinion, two important advantages: concrete and abstract rules can be made executable and are easily correlated. We review these two aspects in turn.

Even though here we do not provide details in this respect, a prototype system —called ECLAIR[2]— has been developed in parallel with the writing of the present paper. The Prolog implementation exploits nice features of a semantics construction based on SOS approach: the concrete semantics rule schemata can be directly translated into Prolog clauses; and the resulting interpreter, with the help of a C++ implementation of memory structures, is efficient enough to run non-trivial programs. Similar considerations apply to the modules implementing the abstract semantics: the abstract semantics rules are almost directly translated to generic Prolog code that is interfaced with specialized libraries implementing several abstract domains, including accurate ones such as the ones provided by the Parma Polyhedra Library [BHRZ05; BHZ05; BHZ06]. So, following this approach, the distance between the expression of the concrete semantics and its executable realization is, as is well known, very little; but the same can be said about the distance between the specification of the abstract semantics and the static analyzer that results from its implementation. This prototype system therefore gives us confidence that both the concrete and abstract semantics are correctly modeled and that, in this paper, no real difficulties have been overlooked.

For space reasons, only a subset of CPM is treated in full depth in the main body of the paper (the extension of the design to the full language is only briefly described even though all the important points are covered). For this subset, we give a complete proof of correctness that relates the abstract semantics to the concrete semantics. The proofs are not complicated and suggest (also because of the way we present them) the possibility of their automatization. To summarize, at this stage of the research work it does not seem unreasonable that we may end up with: readable and executable representations of the concrete semantics of mainstream programming languages; readable and executable representations of program analyzers; correctness of the analyzers established by automatic specialized theorem provers; and, at last, availability of sophisticated program analyzers for such languages.

A final word is due to address the following concern: if the target languages are "real" imperative programming languages, why choose CPM, an unreal one? The reason is indeed quite simple: Java and Python miss some of the "hard" features of C; C misses exceptions; C++ is too hard, for the time being. So, choosing any one of these real languages would have been unlikely to provide us with the answer we were looking for, which was about the adequacy of Schmidt's approach with respect to the above goals. Moreover, in its ECLAIR realization, the CPM language is being extended so as to become a superset of C (i.e., with all the floating-point and integer types, cast and bitwise operators and so forth). Once that code has stabilized, a C and a Java subsystem will be forked.

[2]The 'Extended CLAIR' system targets the analysis of mainstream programming languages by building upon CLAIR, the 'Combined Language and Abstract Interpretation Resource', which was initially developed and used in a teaching context (see `http://www.cs.unipr.it/clair/`).

1.2 Related Work

The literature on abstract interpretation proposes several *frameworks* for static analysis, where the more general approaches put forward in foundational papers are partially specialized according to a given criterion. For a few examples of specializations based on the programming paradigm, one can mention the frameworks in [Bru91] and [GDL92] for the analysis of (constraint) logic programs; the approach in [CC94] for the analysis of functional programs; and the so called "Marktoberdorf'98 generic static analyzer" specified in [Cou99] for the analysis of imperative programs.

All of these frameworks are "generic" in that, while fixing some of the parameters of the considered problem, they are still characterized by several degrees of freedom. It is therefore natural to reason on the similarities and differences between these approaches. However, independently from the programming paradigm under analysis, direct comparisons between frameworks are extremely difficult in that each proposal typically focuses on the solution of a subset of the relevant issues, while partially disregarding other important problems. For instance, both [Bru91] and [GDL92] study the generic algebraic properties that allow for a clean and safe separation between the abstract domains and the abstract interpreter; in contrast, [Cou99] provides full details for a specific instance of the proposed framework, ranging from the parsing of literal constants to the explicit implementation of the abstract operators for the abstract domain of intervals. On the other hand, the frameworks mentioned above differ from the one presented in this paper in that they allow for significant simplifications of the language analyzed. Here we briefly discuss the main differences between the language considered in our proposal and the one in [Cou99].

At the syntactic level, as already mentioned, the language CPM is much richer than the simple imperative language adopted in [Cou99], which has no support for functions, nesting of block statements, exceptions, non-structured control flows and it allows for a single data type (in particular, no pointers and arrays). These syntactic differences are clearly mirrored at the semantics level. In particular, even though the detection of initialization and arithmetic errors is considered by the semantics in [Cou99], the actual process of error propagation is not modeled. In contrast, the semantics construction we propose can easily accommodate the sophisticated exception propagation and handling mechanisms that can be found in languages such as C++, Java and Python. Note that this choice has a non-trivial impact on the specification of the other components of the semantic construction. For example, the short-circuit evaluation of Boolean expressions cannot be normalized as proposed in [Cou99], because such a normalization process, by influencing the order of evaluation of subexpressions, is unable to preserve the concrete semantics as far as exceptional computation paths are concerned. A minor difference is in the modeling of integer variables and values: while [Cou99] considers the case of possibly uninitialized variables taking values in a finite set of machine-representable integers, for ease of presentation we have opted for definitely initialized variables storing arbitrary (i.e., unbounded) integer values. Since the CPM language supports an extensible set of RTS exceptions, the specification of a semantics modeling (the generation, propagation and handling of) uninitialization errors is rather straightforward. An extension of the semantics to the case of several sets of bounded and unbounded numerical types, with suitable type conversion functions, is under development. Another difference is in the generality of the abstract semantics construction: following the approach described here, an analyzer can take full advantage of the more accurate information provided by a relational domain such as that of polyhedra. In contrast, the work in [Cou99] only considers the simpler case of non-relational abstract domains. As mentioned above, the semantics we propose also models the case of possibly recursive functions (with a call-by-value parameter passing mechanism), which are not supported by the language syntax considered in [Cou99]. While both this paper and [Cou99] consider the specification of a *forward* static analysis framework, [Cou99] also provides a backward analysis for arithmetic expressions, to be used in reductive iterations so as to improve precision losses that are usually incurred by non-relational approximations.

1.3 Plan of the Paper

The paper is organized as follows. Section 2 introduces the notation and terminology used throughout the paper; Section 3 defines the syntax of a subset of the imperative language CPM, whereas Section 4 defines its static semantics; the concrete dynamic semantics of this fragment is presented in Section 5, whereas its abstract counterpart is defined in Section 6. The proof of correctness of the abstract semantics is the subject of Section 7, while the computation of further approximations is treated in Section 8. The integration of the full CPM language in the analysis framework presented in this paper is discussed in Section 9. Section 10 concludes.

2. PRELIMINARIES

Let S and T be sets. The notation $S \subseteq_{\mathrm{f}} T$ means that S is a *finite* subset of T. We write $S \uplus T$ to denote the union $S \cup T$, yet emphasizing the fact that $S \cap T = \varnothing$. The set of total (resp., partial) functions from S to T is denoted by $S \to T$ (resp., $S \rightarrowtail T$). We denote by $\mathrm{dom}(f)$ the *domain* of a function $f \colon S \to T$ (resp., $f \colon S \rightarrowtail T$), where $\mathrm{dom}(f) = S$ (resp., $\mathrm{dom}(f) \subseteq S$). Let $(S, \preceq)$ be a partial order and $f \colon S \to S$ be a function. An element $x \in S$ such that $x = f(x)$ (resp., $x \preceq f(x)$) is called a *fixpoint* (resp., *post-fixpoint*) of f. The notation $\mathrm{lfp}_{\preceq}(f)$ (resp., $\mathrm{gfp}_{\preceq}(f)$) stands, if it exists, for the least (resp., greatest) fixpoint of f. A complete lattice is a partial order $(S, \preceq)$ such that $\mathrm{lub}\, T$ exists for each $T \subseteq S$. If $f \colon S \to S$ is monotonic over the complete lattice S, the Knaster-Tarski theorem ensures that the set of post-fixpoints of f is itself a complete lattice. The *fixpoint coinduction* proof principle follows: if f is monotonic over the complete lattice S then, in order to prove that $x \preceq \mathrm{gfp}_{\preceq}(f)$, it is sufficient to prove that $x \preceq f(x)$.

Let $S = \{s_1, \ldots, s_n\}$ be a finite set of cardinality $n \geq 0$. Then, the notation $\{s_1 \mapsto t_1, \ldots, s_n \mapsto t_n\}$, where $\{t_1, \ldots, t_n\} \subseteq T$, stands for the function $f \colon S \to T$ such that $f(s_i) = t_i$, for each $i = 1, \ldots, n$. Note that, assuming that the codomain T is clear from context, the empty set $\varnothing$ denotes the (nowhere defined) function $f \colon \varnothing \to T$.

When denoting the application of a function $f \colon (S_1 \times \cdots \times S_n) \to T$ we omit, as customary, the outer parentheses and write $f(s_1, \ldots, s_n)$ to mean $f\bigl((s_1, \ldots, s_n)\bigr)$.

Let $f_0 \colon S_0 \rightarrowtail T_0$ and $f_1 \colon S_1 \rightarrowtail T_1$ be partial functions. Then the function $f_0[f_1] \colon (S_0 \cup S_1) \rightarrowtail (T_0 \cup T_1)$ is defined, for each $x \in \mathrm{dom}(f_0) \cup \mathrm{dom}(f_1)$, by

$$\bigl(f_0[f_1]\bigr)(x) \stackrel{\mathrm{def}}{=} \begin{cases} f_1(x), & \text{if } x \in \mathrm{dom}(f_1); \\ f_0(x), & \text{if } x \in \mathrm{dom}(f_0) \setminus \mathrm{dom}(f_1). \end{cases}$$

(Note that, if f_0 and f_1 are total functions, then $f_0[f_1]$ is total too.)

For a partial function $f \colon S \rightarrowtail T$ and a set $S' \subseteq S$, $f|_{S'}$ denotes the restriction of f to S', i.e., the function $f|_{S'} \colon S' \rightarrowtail T$ defined, for each $x \in S' \cap \mathrm{dom}(f)$, by $f|_{S'}(x) = f(x)$. (Note that, if f is a total function, then $f|_{S'}$ is total too.) With a minor abuse of notation, we will sometimes write $f \setminus S''$ to denote $f|_{S \setminus S''}$.

$S^\star$ denotes the set of all finite, possibly empty strings of symbols taken from S. The empty string is denoted by ϵ. If $w, z \in S \cup S^\star$, the concatenation of w and z is an element of $S^\star$ denoted by wz or, to avoid ambiguities, by $w \cdot z$. The length of a string z is denoted by $|z|$.

The *integer part* function $\mathrm{int} \colon \mathbb{R} \to \mathbb{Z}$ is given, for each $x \in \mathbb{R}$, by $\mathrm{int}(x) \stackrel{\mathrm{def}}{=} \lfloor x \rfloor$, if $x \geq 0$, and $\mathrm{int}(x) \stackrel{\mathrm{def}}{=} \lceil x \rceil$, if $x < 0$. The *integer division* and the *modulo* operations $\div, \mathrm{mod} \colon \bigl(\mathbb{Z} \times \mathbb{Z} \setminus \{0\}\bigr) \to \mathbb{Z}$ are defined, for each $x, y \in \mathbb{Z}$ with $y \neq 0$, respectively by $x \div y \stackrel{\mathrm{def}}{=} \mathrm{int}(x/y)$ and $x \bmod y \stackrel{\mathrm{def}}{=} x - (x \div y) \cdot y$.

We assume familiarity with the field of program analysis and verification via abstract interpretation. The reader is referred to the literature for the theory (e.g., [Cou81; CC76; CC77a; CC79; CC92a; CC92c]) and examples of applications [DRS01; Hal93; SKS00].

3. THE LANGUAGE SYNTAX

The run-time support of CPM uses exceptions to communicate run-time errors. The set of RTS exceptions is left open so that it can be extended if and when needed. That said, the basic syntactic sets of the CPM language are:

Identifiers. $\mathrm{id} \in \mathrm{Id} = \{\mathrm{main}, \mathrm{x}, \mathrm{x}_0, \mathrm{x}_1, \ldots\} \uplus \mathrm{rId}$, where $\mathrm{rId} \stackrel{\mathrm{def}}{=} \{\underline{\mathrm{x}}, \underline{\mathrm{x}}_0, \underline{\mathrm{x}}_1, \ldots\}$;

Basic types. $T \in \mathrm{Type} \stackrel{\mathrm{def}}{=} \{\mathrm{integer}, \mathrm{boolean}\}$;

Integers. $m \in \mathrm{Integer} \stackrel{\mathrm{def}}{=} \mathbb{Z}$;

Booleans. $t \in \mathrm{Bool} \stackrel{\mathrm{def}}{=} \{\mathrm{tt}, \mathrm{ff}\}$;

RTS exceptions. $\chi \in \mathrm{RTSExcept} \stackrel{\mathrm{def}}{=} \{\mathtt{divbyzero}, \mathtt{stkovflw}, \mathtt{memerror}, \ldots\}$.

The identifiers in rId are "reserved" for the specification of the concrete semantics.

From the basic sets, a number of syntactic categories are defined, along with their syntactic meta-variables, by means of the BNF rules:

Expressions.

$$\begin{aligned} \mathrm{Exp} \ni e ::= {} & m \mid -e \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_1 \mid e_0 \,/\, e_1 \mid e_0 \,\%\, e_1 \\ & \mid t \mid e_0 = e_1 \mid e_0 \neq e_1 \mid e_0 < e_1 \mid e_0 \leq e_1 \mid e_0 \geq e_1 \mid e_0 > e_1 \\ & \mid \textbf{not}\, e \mid e_0 \,\textbf{and}\, e_1 \mid e_0 \,\textbf{or}\, e_1 \mid \mathrm{id} \end{aligned}$$

Sequences of expressions.

$$\mathrm{Exps} \ni \mathrm{es} ::= \square \mid e, \mathrm{es}$$

Storable types.

$$\mathrm{sType} \ni \mathrm{sT} ::= T$$

Formal parameters.

$$\mathrm{formParams} \ni \mathrm{fps} ::= \square \mid \mathrm{id} : \mathrm{sT}, \mathrm{fps}$$

Function bodies.

$$\mathrm{Body} \ni \mathrm{body} ::= \mathbf{let}\, d \,\mathbf{in}\, s \,\mathbf{result}\, e \mid \mathbf{extern}$$

Global declarations.

$$\mathrm{Glob} \ni g ::= \mathbf{gvar}\, \mathrm{id} : \mathrm{sT} = e \mid \mathbf{function}\ \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathrm{body} \mid \mathbf{rec}\, g \mid g_0; g_1$$

Local declarations.

$$\mathrm{Decl} \ni d ::= \mathbf{nil} \mid \mathbf{lvar}\, \mathrm{id} : \mathrm{sT} = e \mid d_0; d_1$$

Catchable types.

$$\mathrm{cType} \ni \mathrm{cT} ::= \mathrm{rts\_exception} \mid \mathrm{sT}$$

Exception declarations.

$$\mathrm{exceptDecl} \ni p ::= \chi \mid \mathrm{cT} \mid \mathrm{id} : \mathrm{sT} \mid \mathbf{any}$$

Catch clauses.

$$\mathrm{Catch} \ni k ::= (p)\, s \mid k_0; k_1$$

Statements.

$$\begin{aligned}\mathrm{Stmt} \ni s ::= {} & \mathbf{nop} \mid \mathrm{id} := e \mid \mathrm{id}_0 := \mathrm{id}(\mathrm{es}) \mid s_0; s_1 \mid d; s \\ & \mid \mathbf{if}\, e \,\mathbf{then}\, s_0 \,\mathbf{else}\, s_1 \mid \mathbf{while}\, e \,\mathbf{do}\, s \\ & \mid \mathbf{throw}\, \chi \mid \mathbf{throw}\, e \mid \mathbf{try}\, s \,\mathbf{catch}\, k \mid \mathbf{try}\, s_0 \,\mathbf{finally}\, s_1\end{aligned}$$

Observe that there is no need of a separate syntactic category for programs: as we will see, a CPM program is just a global declaration defining the special function 'main', like in C and C++.

It should be noted that some apparent limitations of the abstract syntax of CPM are not real limitations. For instance: the use of function calls as expressions can be avoided by introducing temporary variables; procedures can be rendered by functions that return a dummy value; and so forth. More generally, a slight elaboration of the abstract syntax presented here and extended in Section 9 is used in the ECLAIR prototype to encode the C language almost in its entirety, plus the basic exception handling mechanisms of C++ and Java.

For notational convenience, we also define the syntactic categories of constants, storable values[3] and exceptions:

Constants.

$$\mathrm{Con} \ni \mathrm{con} ::= m \mid t$$

[3]The reason for a distinction between the roles of constants and storable values (as well as basic types and storable types) will become clear when discussing language extensions in Section 9.

Storable values.

$$\mathrm{sVal} \ni \mathrm{sval} ::= \mathrm{con}$$

Exceptions.

$$\mathrm{Except} \ni \xi ::= \chi \mid \mathrm{sval}$$

The function type: sVal $\rightarrowtail$ sType, mapping a storable value to its type name 'integer' or 'boolean', is defined by:

$$\mathrm{type}(\mathrm{sval}) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{integer}, & \text{if } \mathrm{sval} = m \in \mathrm{Integer}; \\ \mathrm{boolean}, & \text{if } \mathrm{sval} = t \in \mathrm{Bool}. \end{cases}$$

For ease of notation, we also define the overloadings type: Except $\rightarrowtail$ cType and type: exceptDecl $\rightarrowtail$ cType defined by

$$\mathrm{type}(\xi) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{rts\_exception}, & \text{if } \xi = \chi \in \mathrm{RTSExcept}; \\ \mathrm{type}(\mathrm{sval}), & \text{if } \xi = \mathrm{sval} \in \mathrm{sVal}; \end{cases}$$

$$\mathrm{type}(p) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{rts\_exception}, & \text{if } p = \chi \in \mathrm{RTSExcept}; \\ \mathrm{cT}, & \text{if } p = \mathrm{cT} \in \mathrm{cType}; \\ \mathrm{sT}, & \text{if } p = \mathrm{id} : \mathrm{sT} \text{ and } \mathrm{sT} \in \mathrm{sType}. \end{cases}$$

Note that such an overloading is consistent and the resulting function is not defined on value **any** $\in$ exceptDecl.

The helper function dom: cType $\to$ {Integer, Bool, RTSExcept}, which associates a catchable type name to the corresponding domain, is defined by

$$\mathrm{dom}(\mathrm{cT}) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{Integer}, & \text{if } \mathrm{cT} = \mathrm{integer}; \\ \mathrm{Bool}, & \text{if } \mathrm{cT} = \mathrm{boolean}; \\ \mathrm{RTSExcept}, & \text{if } \mathrm{cT} = \mathrm{rts\_exception}. \end{cases}$$

4. STATIC SEMANTICS

The static semantics of the CPM language establishes the conditions under which a program is well typed. Only well-typed programs are given a dynamic semantics.

4.1 Defined and Free Identifiers

The set of identifiers defined by sequences of formal parameters, (global or local) declarations or exception declarations is defined as follows:

$$\mathrm{DI}(\square) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathbf{nil}) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathrm{body}) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\chi) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathrm{cT}) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathbf{any}) \stackrel{\mathrm{def}}{=} \varnothing;$$

$$\mathrm{DI}(\mathrm{id} : \mathrm{sT}) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathbf{gvar}\,\mathrm{id} : \mathrm{sT} = e) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathbf{lvar}\,\mathrm{id} : \mathrm{sT} = e)$$
$$\stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathbf{function}\ \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathrm{body}) \stackrel{\mathrm{def}}{=} \{\mathrm{id}\};$$

$$\mathrm{DI}(\mathrm{id} : \mathrm{sT}, \mathrm{fps}) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathrm{id} : \mathrm{sT}) \cup \mathrm{DI}(\mathrm{fps});$$

$$\mathrm{DI}(\mathbf{rec}\, g) \stackrel{\mathrm{def}}{=} \mathrm{DI}(g);$$

$$\mathrm{DI}(g_0; g_1) \stackrel{\mathrm{def}}{=} \mathrm{DI}(g_0) \cup \mathrm{DI}(g_1);$$

$$\mathrm{DI}(d_0; d_1) \stackrel{\text{def}}{=} \mathrm{DI}(d_0) \cup \mathrm{DI}(d_1).$$

The set of identifiers that occur freely in (sequences of) expressions, (exception) declarations, statements and catch clauses is defined by:

$$\begin{aligned}
&\mathrm{FI}(m) \stackrel{\text{def}}{=} \mathrm{FI}(t) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{nop}) \stackrel{\text{def}}{=} \mathrm{FI}(\square) \stackrel{\text{def}}{=} \mathrm{FI}(\mathrm{id} : \mathrm{sT}) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{nil}) \\
&\quad \stackrel{\text{def}}{=} \mathrm{FI}(\chi) \stackrel{\text{def}}{=} \mathrm{FI}(\mathrm{cT}) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{any}) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{throw}\,\chi) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{extern}) \stackrel{\text{def}}{=} \varnothing; \\
&\mathrm{FI}(-e) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{not}\,e) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{lvar}\,\mathrm{id} : \mathrm{sT} = e) \\
&\quad \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{gvar}\,\mathrm{id} : \mathrm{sT} = e) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{throw}\,e) \stackrel{\text{def}}{=} \mathrm{FI}(e); \\
&\mathrm{FI}(e_0 \mathbin{\mathrm{op}} e_1) \stackrel{\text{def}}{=} \mathrm{FI}(e_0) \cup \mathrm{FI}(e_1), \text{ for } \mathrm{op} \in \{+, \ldots, \%, =, \ldots, >, \mathbf{and}, \mathbf{or}\}; \\
&\mathrm{FI}(\mathrm{id}) \stackrel{\text{def}}{=} \{\mathrm{id}\}; \\
&\mathrm{FI}(\mathbf{let}\,d\ \mathbf{in}\,s\,\mathbf{result}\,e) \stackrel{\text{def}}{=} \mathrm{FI}(d) \cup \bigl(\mathrm{FI}(s) \setminus \mathrm{DI}(d)\bigr) \cup \bigl(\mathrm{FI}(e) \setminus \mathrm{DI}(d)\bigr); \\
&\mathrm{FI}(\mathbf{function}\ \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathrm{body}) \stackrel{\text{def}}{=} \mathrm{FI}(\mathrm{body}) \setminus \mathrm{DI}(\mathrm{fps}); \\
&\mathrm{FI}(\mathbf{rec}\,g) \stackrel{\text{def}}{=} \mathrm{FI}(g) \setminus \mathrm{DI}(g); \\
&\mathrm{FI}(g_0; g_1) \stackrel{\text{def}}{=} \mathrm{FI}(g_0) \cup \bigl(\mathrm{FI}(g_1) \setminus \mathrm{DI}(g_0)\bigr); \\
&\mathrm{FI}(d_0; d_1) \stackrel{\text{def}}{=} \mathrm{FI}(d_0) \cup \bigl(\mathrm{FI}(d_1) \setminus \mathrm{DI}(d_0)\bigr); \\
&\mathrm{FI}(\mathrm{id} := e) \stackrel{\text{def}}{=} \{\mathrm{id}\} \cup \mathrm{FI}(e); \\
&\mathrm{FI}(e, \mathrm{es}) \stackrel{\text{def}}{=} \mathrm{FI}(e) \cup \mathrm{FI}(\mathrm{es}); \\
&\mathrm{FI}\bigl(\mathrm{id}_0 := \mathrm{id}(\mathrm{es})\bigr) \stackrel{\text{def}}{=} \{\mathrm{id}, \mathrm{id}_0\} \cup \mathrm{FI}(\mathrm{es}); \\
&\mathrm{FI}(d; s) \stackrel{\text{def}}{=} \mathrm{FI}(d) \cup \bigl(\mathrm{FI}(s) \setminus \mathrm{DI}(d)\bigr); \\
&\mathrm{FI}\bigl((p)\,s\bigr) \stackrel{\text{def}}{=} \mathrm{FI}(s) \setminus \mathrm{DI}(p); \\
&\mathrm{FI}(k_0; k_1) \stackrel{\text{def}}{=} \mathrm{FI}(k_0) \cup \mathrm{FI}(k_1); \\
&\mathrm{FI}(s_0; s_1) \stackrel{\text{def}}{=} \mathrm{FI}(\mathbf{try}\,s_0\,\mathbf{finally}\,s_1) \stackrel{\text{def}}{=} \mathrm{FI}(s_0) \cup \mathrm{FI}(s_1); \\
&\mathrm{FI}(\mathbf{if}\,e\,\mathbf{then}\,s_0\,\mathbf{else}\,s_1) \stackrel{\text{def}}{=} \mathrm{FI}(e) \cup \mathrm{FI}(s_0) \cup \mathrm{FI}(s_1); \\
&\mathrm{FI}(\mathbf{while}\,e\,\mathbf{do}\,s) \stackrel{\text{def}}{=} \mathrm{FI}(e) \cup \mathrm{FI}(s); \\
&\mathrm{FI}(\mathbf{try}\,s\,\mathbf{catch}\,k) \stackrel{\text{def}}{=} \mathrm{FI}(s) \cup \mathrm{FI}(k).
\end{aligned}$$

4.2 Type Environments

We start by defining the convenience syntactic category of

Denotable types.

$$\mathrm{dType} \ni \mathrm{dT} ::= \mathrm{sT}\ \mathrm{loc} \mid \mathrm{fps} \rightarrow \mathrm{sT}$$

A type environment associates a denotable type to each identifier of a given, finite set of identifiers.

DEFINITION 4.1. (TEnv_I, TEnv.) *For each* $I \subseteq_{\mathrm{f}} \mathrm{Id}$*, the set of* type environments over I *is* $\mathrm{TEnv}_I \stackrel{\mathrm{def}}{=} I \rightarrow \mathrm{dType}$*; the set of all type environments is given by* $\mathrm{TEnv} \stackrel{\mathrm{def}}{=} \biguplus_{I \subseteq_{\mathrm{f}} \mathrm{Id}} \mathrm{TEnv}_I$*. Type environments are denoted by* β*,* β_0*,* β_1 *and so forth. The notation* $\beta : I$ *is a shorthand for* $\beta \in \mathrm{TEnv}_I$*.*

4.3 Static Semantics Predicates

Let $I \subseteq_{\mathrm{f}} \mathrm{Id}$ and $\beta \in \mathrm{TEnv}_I$. The well-typedness of program constructs whose free identifiers are contained in I is encoded by the following predicates, here listed along with their informal meaning:

$$
\begin{array}{lr}
\beta \vdash_I e : \mathrm{sT}, & e \text{ is well-formed and has type sT in } \beta; \\
\beta \vdash_I \mathrm{body} : \mathrm{sT}, & \text{body is well-formed and has type sT in } \beta; \\
\beta, \mathrm{fps} \vdash_I \mathrm{es}, & \text{es is compatible with fps and well formed in } \beta; \\
\mathrm{fps} : \delta, & \text{fps is well formed and yields the type environment } \delta; \\
\beta \vdash_I g : \delta, & g \text{ is well formed and yields the type environment } \delta \text{ in } \beta; \\
\beta \vdash_I d : \delta, & d \text{ is well-formed and yields the type environment } \delta \text{ in } \beta; \\
p : \delta, & p \text{ is well-formed and yields the type environment } \delta; \\
\beta \vdash_I k, & k \text{ is well-formed in } \beta; \\
\beta \vdash_I s, & s \text{ is well-formed in } \beta.
\end{array}
$$

These predicates are defined inductively on the abstract syntax by means of the following rules.

Expressions.

$$
\frac{}{\beta \vdash_I m : \mathrm{integer}} \qquad \frac{}{\beta \vdash_I t : \mathrm{boolean}}
$$

$$
\frac{\beta \vdash_I e : \mathrm{integer}}{\beta \vdash_I -e : \mathrm{integer}} \qquad \frac{\beta \vdash_I e : \mathrm{boolean}}{\beta \vdash_I \textbf{not } e : \mathrm{boolean}}
$$

$$
\frac{\beta \vdash_I e_0 : \mathrm{integer} \quad \beta \vdash_I e_1 : \mathrm{integer}}{\beta \vdash_I e_0 \boxdot e_1 : \mathrm{integer}} \quad \text{if } \boxdot \in \{+, -, *, /, \%\}
$$

$$
\frac{\beta \vdash_I e_0 : \mathrm{integer} \quad \beta \vdash_I e_1 : \mathrm{integer}}{\beta \vdash_I e_0 \circledast e_1 : \mathrm{boolean}} \quad \text{if } \circledast \in \{=, \neq, <, \leq, \geq, >\}
$$

$$
\frac{\beta \vdash_I e_0 : \mathrm{boolean} \quad \beta \vdash_I e_1 : \mathrm{boolean}}{\beta \vdash_I e_0 \diamond e_1 : \mathrm{boolean}} \quad \text{if } \diamond \in \{\textbf{and}, \textbf{or}\}
$$

$$
\frac{}{\beta \vdash_I \mathrm{id} : \mathrm{sT}} \quad \text{if } \beta(\mathrm{id}) = \mathrm{sT\ loc}
$$

Sequences of expressions.

$$
\frac{}{\beta, \square \vdash_I \square} \qquad \frac{\beta \vdash_I e : \mathrm{sT} \quad \beta, \mathrm{fps} \vdash_I \mathrm{es}}{\beta, (\mathrm{id} : \mathrm{sT}, \mathrm{fps}) \vdash_I (e, \mathrm{es})}
$$

Sequences of formal parameters.

$$\frac{}{\square : \varnothing} \qquad \frac{\mathrm{fps} : \delta}{(\mathrm{id} : \mathrm{sT}, \mathrm{fps}) : \{\mathrm{id} \mapsto \mathrm{sT\ loc}\} \cup \delta} \quad \text{if } \mathrm{id} \notin \mathrm{DI}(\mathrm{fps})$$

Function bodies.

$$\frac{\beta \vdash_I d : \beta_0 \qquad \beta[\beta_0] \vdash_{I \cup \mathrm{DI}(d)} s \qquad \beta[\beta_0] \vdash_{I \cup \mathrm{DI}(d)} e : \mathrm{sT}}{\beta \vdash_I (\mathbf{let}\, d \,\mathbf{in}\, s \,\mathbf{result}\, e) : \mathrm{sT}}$$

Declarations.

$$\frac{}{\beta \vdash_I \mathbf{nil} : \varnothing} \qquad \frac{\beta \vdash_I e : \mathrm{sT}}{\beta \vdash_I \mathbf{gvar}\, \mathrm{id} : \mathrm{sT} = e : \{\mathrm{id} \mapsto \mathrm{sT\ loc}\}}$$

$$\frac{\beta \vdash_I e : \mathrm{sT}}{\beta \vdash_I \mathbf{lvar}\, \mathrm{id} : \mathrm{sT} = e : \{\mathrm{id} \mapsto \mathrm{sT\ loc}\}}$$

$$\frac{\mathrm{fps} : \delta \qquad \beta[\delta] \vdash_{I \cup \mathrm{DI}(\mathrm{fps})} \mathrm{body} : \mathrm{sT}}{\beta \vdash_I \big(\mathbf{function}\ \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathrm{body}\big) : \big\{\mathrm{id} \mapsto (\mathrm{fps} \to \mathrm{sT})\big\}} \quad \text{if body} \neq \mathbf{extern}$$

$$\frac{\mathrm{fps} : \delta}{\beta \vdash_I \big(\mathbf{function}\ \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathbf{extern}\big) : \big\{\mathrm{id} \mapsto (\mathrm{fps} \to \mathrm{sT})\big\}}$$

$$\frac{\beta[\delta|_J] \vdash_{I \cup J} g : \delta}{\beta \vdash_I (\mathbf{rec}\, g) : \delta} \quad \text{if } J = \mathrm{FI}(g) \cap \mathrm{DI}(g) \text{ and } \forall \mathrm{id}, \mathrm{sT} : (\mathrm{id} \mapsto \mathrm{sT\ loc}) \notin \delta \tag{1}$$

$$\frac{\beta \vdash_I g_0 : \beta_0 \quad \beta[\beta_0] \vdash_{I \cup \mathrm{DI}(g_0)} g_1 : \beta_1}{\beta \vdash_I g_0; g_1 : \beta_0[\beta_1]} \qquad \frac{\beta \vdash_I d_0 : \beta_0 \quad \beta[\beta_0] \vdash_{I \cup \mathrm{DI}(d_0)} d_1 : \beta_1}{\beta \vdash_I d_0; d_1 : \beta_0[\beta_1]}$$

Note that rule (1) seems to suggest that δ must be guessed. Indeed, this is not the case, as it can be proved that the environment generated by a declaration g only depends on g and not on the environment used to establish whether g is well formed. While the right thing to do is to define two static semantics predicates for declarations —one for the generated environments and the other for well-formedness [Plo04]— we opted for a more concise presentation. Also notice that the side condition in rule (1) explicitly forbids recursive declarations of variables.[4]

Exception declarations.

$$\frac{}{\vdash_I \chi : \varnothing} \qquad \frac{}{\vdash_I \mathrm{cT} : \varnothing}$$

$$\frac{}{\vdash_I \mathrm{id} : \mathrm{sT} : \{\mathrm{id} \mapsto \mathrm{sT\ loc}\}} \qquad \frac{}{\vdash_I \mathbf{any} : \varnothing}$$

Catch clauses.

$$\frac{p : \delta \quad \beta[\delta] \vdash_{I \cup \mathrm{DI}(p)} s}{\beta \vdash_I (p)\, s} \qquad \frac{\beta \vdash_I k_0 \quad \beta \vdash_I k_1}{\beta \vdash_I k_0; k_1}$$

[4]Namely, a recursive declaration such as **rec gvar** id : sT $= e$ is not well-typed.

Statements.

$$\frac{}{\beta \vdash_I \mathbf{nop}} \qquad \frac{\beta \vdash_I e : \mathrm{sT}}{\beta \vdash_I \mathrm{id} := e} \quad \text{if } \beta(\mathrm{id}) = \mathrm{sT\ loc}$$

$$\frac{\beta, \mathrm{fps} \vdash_I \mathrm{es}}{\beta \vdash_I \mathrm{id}_0 := \mathrm{id}(\mathrm{es})} \quad \text{if } \beta(\mathrm{id}_0) = \mathrm{sT\ loc} \text{ and } \beta(\mathrm{id}) = \mathrm{fps} \to \mathrm{sT}$$

$$\frac{\beta \vdash_I s_0 \quad \beta \vdash_I s_1}{\beta \vdash_I s_0; s_1} \qquad \frac{\beta \vdash_I d : \beta_0 \quad \beta[\beta_0] \vdash_{I \cup \mathrm{DI}(d)} s}{\beta \vdash_I d; s}$$

$$\frac{\beta \vdash_I e : \mathrm{boolean} \quad \beta \vdash_I s_0 \quad \beta \vdash_I s_1}{\beta \vdash_I \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1} \qquad \frac{\beta \vdash_I e : \mathrm{boolean} \quad \beta \vdash_I s}{\beta \vdash_I \mathbf{while}\, e\, \mathbf{do}\, s}$$

$$\frac{}{\beta \vdash_I \mathbf{throw}\, \chi} \qquad \frac{\beta \vdash_I e : \mathrm{sT}}{\beta \vdash_I \mathbf{throw}\, e}$$

$$\frac{\beta \vdash_I s \quad \beta \vdash_I k}{\beta \vdash_I \mathbf{try}\, s\, \mathbf{catch}\, k} \qquad \frac{\beta \vdash_I s_0 \quad \beta \vdash_I s_1}{\beta \vdash_I \mathbf{try}\, s_0\, \mathbf{finally}\, s_1}$$

A program g is said to be *valid* if and only if it does not contain any occurrence of a reserved identifier $\mathrm{id} \in \mathrm{rId}$, $\varnothing \vdash_\varnothing g : \beta$ and $\beta(\mathrm{main}) = \square \to \mathrm{integer}$.

5. CONCRETE DYNAMIC SEMANTICS

For the specification of the concrete dynamic semantics for CPM, we adopt the G^∞SOS approach of Cousot and Cousot [CC92c]. This generalizes with infinite computations the *natural* semantics approach by Kahn [Kah87], which, in turn, is a "big-step" operational semantics defined by structural induction on program structures in the style of Plotkin [Plo04].

5.1 Absolute Locations and Indirect Locators

An *absolute location* (or, simply, *location*) is a unique identifier for a memory area of unspecified size. The (possibly infinite) set of all locations is denoted by Loc, while individual locations are denoted by l, l_0, l_1 and so forth. We also postulate the existence of a set $\mathrm{Ind} \stackrel{\mathrm{def}}{=} \mathbb{N}$ of *indirect (stack) locators* such that $\mathrm{Loc} \cap \mathrm{Ind} = \varnothing$. Indirect locators are denoted by i, i_0, i_1 and so forth. For notational convenience, we define the set of *addresses* as $\mathrm{Addr} \stackrel{\mathrm{def}}{=} \mathrm{Loc} \uplus \mathrm{Ind}$. Addresses are denoted by a, a_0, a_1 and so forth.

5.2 Concrete Execution Environments

The concrete dynamic aspect of declarations is captured by concrete execution environments. These map a finite set of identifiers to concrete denotable values. In the sequel we will simply write 'environment' to refer to execution environments.

Definition 5.1. (Abstract, dVal, Env_I.) *We define*

$$\mathrm{Abstract} \stackrel{\mathrm{def}}{=} \bigl\{ (\lambda\, \mathrm{fps} \,.\, \mathrm{body}, \mathrm{sT}) \mid \mathrm{fps} \in \mathrm{formParams}, \mathrm{body} \in \mathrm{Body}, \mathrm{sT} \in \mathrm{sType} \bigr\}.$$

The set of concrete denotable values *is*

$$\mathrm{dVal} \stackrel{\mathrm{def}}{=} (\mathrm{Addr} \times \mathrm{sType}) \uplus \mathrm{Abstract}.$$

For $I \subseteq_{\mathrm{f}} \mathrm{Id}$*,* $\mathrm{Env}_I \stackrel{\mathrm{def}}{=} I \to \mathrm{dVal}$ *is the set of* concrete environments over I*. The set of all environments is given by* $\mathrm{Env} \stackrel{\mathrm{def}}{=} \biguplus_{I \subseteq_{\mathrm{f}} \mathrm{Id}} \mathrm{Env}_I$*. Environments in* Env_I *are denoted by* ρ*,* ρ_0*,* ρ_1 *and so forth. We write* $\rho : I$ *as a shorthand for* $\rho \in \mathrm{Env}_I$*. For* $\rho : I$ *and* $\beta : I$*, we write* $\rho : \beta$ *to signify that*

$$\begin{aligned}
&\forall \mathrm{id} \in I : \bigl(\exists (a, \mathrm{sT}) \in \mathrm{Addr} \times \mathrm{sType} \mathrel{.} \beta(\mathrm{id}) = \mathrm{sT}\ \mathrm{loc} \land \rho(\mathrm{id}) = (a, \mathrm{sT})\bigr)\\
&\lor \bigl(\exists (\lambda\, \mathrm{fps} \mathrel{.} \mathrm{body}, \mathrm{sT}) \in \mathrm{Abstract} \mathrel{.} \beta(\mathrm{id}) = \mathrm{fps} \to \mathrm{sT} \land \rho(\mathrm{id}) = (\lambda\, \mathrm{fps} \mathrel{.} \mathrm{body}, \mathrm{sT})\bigr).
\end{aligned}$$

5.3 Memory Structures, Value States and Exception States

A *memory structure* uses a stack and suitable operators to allocate/deallocate, organize, read and update the locations of an absolute memory map, which is a partial function mapping a location and a storable type to a storable value. Memory structures model all the memory areas that are used in the most common implementations of imperative programming languages: the *data segment* (for global variables) and the *stack segment* (for local variables) are of interest for the language fragment we are considering; the *text segment* (where pointers to function point to) and the *heap segment* (for dynamically allocated memory) are required to deal with the extensions of Section 9. As it will be clear from the following definition, our notion of memory structure is underspecified: while we define it and its operations so that the semantics of programs is the expected one, we allow for many possible implementations by leaving out many details that are inessential to the achievement of that objective. It is for this same reason that we treat locations as unique identifiers neglecting the mathematical structure they may or may not have. More generally, what we call "concrete semantics" is indeed an abstraction of an infinite number of machines and compilation schemes that could be used to execute our programs. Furthermore, since the considered fragment of CPM does not support pointers, arrays, type casts and unions, we can here make the simplifying assumption that there is no overlap between the storage cells associated to different locations. In Section 9 we will hint at how these assumptions must be modified in order to accommodate the full language.

Memory structures will be used to describe the outcome of computations whose only observable behavior is given by their side effects. Computations yielding a proper value will be described by a *value state*, which pairs the value computed with a memory structure recording the side effects of the execution. Exceptional behavior must, of course, be taken into proper account: thus, the result of an exceptional computation path will be described by pairing the memory structure with an exception, yielding what we call an *exception state*.

Definition 5.2. **(**Map**,** Stack**,** Mem**,** ValState**,** ExceptState**.)** *The set of all absolute maps is the set of partial functions*

$$\mathrm{Map} \stackrel{\mathrm{def}}{=} (\mathrm{Loc} \times \mathrm{sType}) \rightarrowtail \mathrm{sVal}.$$

Absolute maps are denoted by μ, μ_0, μ_1 *and so forth. The absolute map update partial function*

$$\cdot[\cdot := \cdot] \colon \bigl(\mathrm{Map} \times (\mathrm{Loc} \times \mathrm{sType}) \times \mathrm{sVal}\bigr) \rightarrowtail \mathrm{Map}$$

is defined, for each $\mu \in \mathrm{Map}$, $(l, \mathrm{sT}) \in \mathrm{Loc} \times \mathrm{sType}$ *such that* $(l, \mathrm{sT}) \in \mathrm{dom}(\mu)$ *and* $\mathrm{sval} \in \mathrm{sVal}$ *such that* $\mathrm{sT} = \mathrm{type}(\mathrm{sval})$, *by*

$$\mu\bigl[(l, \mathrm{sT}) := \mathrm{sval}\bigr] \stackrel{\mathrm{def}}{=} \mu',$$

where $\mu' \in \mathrm{Map}$ *is any absolute map satisfying the following conditions:*

(i) $\mathrm{dom}(\mu') = \mathrm{dom}(\mu)$;
(ii) $\mu'(l, \mathrm{sT}) = \mathrm{sval}$;
(iii) $\mu'(l', \mathrm{sT}') = \mu(l', \mathrm{sT}')$, *for each* $(l', \mathrm{sT}') \in \mathrm{dom}(\mu)$ *such that* $l' \neq l$.

Let $W \stackrel{\mathrm{def}}{=} \bigl(\mathrm{Loc} \cup \{\dagger, \ddagger\}\bigr)^{\star}$. *An element* $w \in W$ *is a* stack *if and only if no location occurs more than once in it. The set of all stacks is denoted by* Stack. *'*$\dagger$*' is called* stack marker *and '*$\ddagger$*' is called* frame marker. *The* top-most frame *of* $w \in \mathrm{Stack}$, *denoted by* $\mathrm{tf}(w)$, *is the longest suffix of* w *containing no frame marker; formally,* $\mathrm{tf}(w) \in \bigl(\mathrm{Loc} \cup \{\dagger\}\bigr)^{\star}$ *satisfies either* $w = \mathrm{tf}(w)$ *or* $w = w' \ddagger \mathrm{tf}(w)$. *The partial infix operator* $@ \colon \mathrm{Stack} \times \mathrm{Ind} \rightarrowtail \mathrm{Loc}$ *maps, when defined, a stack* w *and an indirect locator* i *into an absolute location to be found in the top-most frame; formally, if* $\mathrm{tf}(w) = z_0 \cdots z_{n-1}$ *where, for all* $i < n$, $z_i = \dagger^{\star} \cdot l_i \cdot \dagger^{\star}$ *and* $l_i \in \mathrm{Loc}$, *then, for each* $i < n$, $w \mathbin{@} i \stackrel{\mathrm{def}}{=} l_i$.

A memory structure *is an element of* $\mathrm{Mem} \stackrel{\mathrm{def}}{=} \mathrm{Map} \times \mathrm{Stack}$. *Memory structures are denoted by* σ, σ_0, σ_1 *and so forth.*

A value state *is an element of* $\mathrm{ValState} \stackrel{\mathrm{def}}{=} \mathrm{sVal} \times \mathrm{Mem}$. *Value states are denoted by* υ, υ_0, υ_1 *and so forth.*

An exception state *is an element of* $\mathrm{ExceptState} \stackrel{\mathrm{def}}{=} \mathrm{Mem} \times \mathrm{Except}$. *Exception states are denoted by* ε, ε_0, ε_1 *and so forth.*

The overloading $@ \colon \mathrm{Mem} \times \mathrm{Addr} \rightarrowtail \mathrm{Loc}$ *of the partial infix operator* $@$ *is defined, for each* $\sigma = (\mu, w)$ *and* $a \in \mathrm{Addr}$, *as follows and under the following conditions:*

$$\sigma \mathbin{@} a \stackrel{\mathrm{def}}{=} \begin{cases} a, & \text{if } a \in \mathrm{Loc}; \\ l, & \text{if } a \in \mathrm{Ind} \text{ and } l = w \mathbin{@} a \text{ is defined.} \end{cases}$$

The memory structure read and update operators

$$\cdot[\cdot, \cdot] \colon \bigl(\mathrm{Mem} \times \mathrm{Addr} \times \mathrm{sType}\bigr) \to (\mathrm{ValState} \uplus \mathrm{ExceptState}),$$
$$\cdot[\cdot := \cdot] \colon \bigl(\mathrm{Mem} \times (\mathrm{Addr} \times \mathrm{sType}) \times \mathrm{sVal}\bigr) \to (\mathrm{Mem} \uplus \mathrm{ExceptState})$$

are respectively defined, for each $\sigma = (\mu, w) \in \mathrm{Mem}$, $a \in \mathrm{Addr}$, $\mathrm{sT} \in \mathrm{sType}$ *and* $\mathrm{sval} \in \mathrm{sVal}$, *as follows: let* $d = (\sigma \mathbin{@} a, \mathrm{sT})$; *then*

$$\sigma[a, \mathrm{sT}] \stackrel{\mathrm{def}}{=} \begin{cases} \bigl(\mu(d), \sigma\bigr), & \text{if } d \in \mathrm{dom}(\mu); \\ (\sigma, \mathtt{memerror}), & \text{otherwise;} \end{cases}$$

$$\sigma\bigl[(a, \mathrm{sT}) := \mathrm{sval}\bigr] \stackrel{\mathrm{def}}{=} \begin{cases} \bigl(\mu[d := \mathrm{sval}], w\bigr), & \text{if } d \in \mathrm{dom}(\mu) \text{ and } \mathrm{sT} = \mathrm{type}(\mathrm{sval}); \\ (\sigma, \mathtt{memerror}), & \text{otherwise.} \end{cases}$$

The data and stack memory allocation functions

$$\mathrm{new_d}\colon \mathrm{ValState} \to \bigl((\mathrm{Mem} \times \mathrm{Loc}) \uplus \mathrm{ExceptState}\bigr),$$
$$\mathrm{new_s}\colon \mathrm{ValState} \to \bigl((\mathrm{Mem} \times \mathrm{Ind}) \uplus \mathrm{ExceptState}\bigr)$$

are defined, for each $\upsilon = (\mathrm{sval}, \sigma) \in \mathrm{ValState}$*, where* $\sigma = (\mu, w)$*, by*

$$\mathrm{new_d}(\upsilon) \stackrel{\mathrm{def}}{=} \begin{cases} \bigl((\mu', w), l\bigr), & \text{if the data segment of } \sigma \text{ can be extended;} \\ (\sigma, \mathtt{datovflw}), & \text{otherwise;} \end{cases}$$

$$\mathrm{new_s}(\upsilon) \stackrel{\mathrm{def}}{=} \begin{cases} \bigl((\mu', w'), i\bigr), & \text{if the stack segment of } \sigma \text{ can be extended;} \\ (\sigma, \mathtt{stkovflw}), & \text{otherwise;} \end{cases}$$

where, in the case of $\mathrm{new_s}$*,* $w' \in \mathrm{Stack}$ *and* $i \in \mathrm{Ind}$ *are such that:*

(i) $w' = w \cdot l$*;*
(ii) $i = |\,\mathrm{tf}(w)|$*;*

and, for both $\mathrm{new_d}$ *and* $\mathrm{new_s}$*,* $\mu' \in \mathrm{Map}$ *and* $l \in \mathrm{Loc}$ *are such that:*

(iii) for each $\mathrm{sT} \in \mathrm{sType}$*,* $(l, \mathrm{sT}) \notin \mathrm{dom}(\mu)$*;*
(iv) for each $(l', \mathrm{sT}') \in \mathrm{dom}(\mu)$*,* $\mu'(l', \mathrm{sT}') = \mu(l', \mathrm{sT}')$*;*
(v) $\mu'\bigl(l, \mathrm{type}(\mathrm{sval})\bigr) = \mathrm{sval}$.

The memory structure data cleanup function $\mathrm{cleanup_d}\colon \mathrm{ExceptState} \to \mathrm{ExceptState}$ *is given, for each* $\varepsilon = (\sigma, \xi) \in \mathrm{ExceptState}$*, by*

$$\mathrm{cleanup_d}(\varepsilon) \stackrel{\mathrm{def}}{=} \bigl((\varnothing, \epsilon), \xi\bigr).$$

The stack mark function $\mathrm{mark_s}\colon \mathrm{Mem} \to \mathrm{Mem}$ *is given, for each* $\sigma \in \mathrm{Mem}$*, by*

$$\mathrm{mark_s}(\sigma) \stackrel{\mathrm{def}}{=} (\mu, w\dagger), \qquad \text{where } \sigma = (\mu, w).$$

The stack unmark partial function $\mathrm{unmark_s}\colon \mathrm{Mem} \rightarrowtail \mathrm{Mem}$ *is given, for each* $\sigma \in \mathrm{Mem}$ *such that* $\sigma = (\mu, w'\dagger w'')$ *and* $w'' \in \mathrm{Loc}^\star$*, by*

$$\mathrm{unmark_s}(\mu, w'\dagger w'') \stackrel{\mathrm{def}}{=} (\mu', w'),$$

where the absolute map $\mu' \in \mathrm{Map}$ *satisfies:*

(i) $\mathrm{dom}(\mu') = \bigl\{\, (l, \mathrm{sT}) \in \mathrm{dom}(\mu) \bigm| l \text{ does not occur in } w'' \,\bigr\}$*;*
(ii) $\mu' = \mu|_{\mathrm{dom}(\mu')}$.

The frame link partial function $\mathrm{link_s}\colon \mathrm{Mem} \rightarrowtail \mathrm{Mem}$ *is given, for each* $\sigma \in \mathrm{Mem}$ *such that* $\sigma = (\mu, w'\dagger w'')$ *and* $w'' \in \mathrm{Loc}^\star$*, by*

$$\mathrm{link_s}(\mu, w'\dagger w'') \stackrel{\mathrm{def}}{=} (\mu, w'\ddagger w'').$$

The frame unlink partial function $\mathrm{unlink_s}\colon \mathrm{Mem} \rightarrowtail \mathrm{Mem}$ *is given, for each* $\sigma \in \mathrm{Mem}$ *such that* $\sigma = (\mu, w'\ddagger w'')$ *and* $w'' \in \mathrm{Loc}^\star$*, by*

$$\mathrm{unlink_s}(\mu, w'\ddagger w'') \stackrel{\mathrm{def}}{=} (\mu, w'\dagger w'').$$

For ease of notation, the stack unmark and the frame unlink partial functions are lifted to also work on exception states. Namely, for each $\varepsilon = (\sigma, \xi) \in \mathrm{ExceptState}$,

$$\mathrm{unmark_s}(\sigma, \xi) \stackrel{\mathrm{def}}{=} \bigl(\mathrm{unmark_s}(\sigma), \xi\bigr);$$

$$\mathrm{unlink_s}(\sigma, \xi) \stackrel{\mathrm{def}}{=} \bigl(\mathrm{unlink_s}(\sigma), \xi\bigr).$$

Intuitively, global variables are allocated in the data segment using $\mathrm{new_d}$ and are accessed through absolute locations; function $\mathrm{cleanup_d}$ models their deallocation due to an RTS exception thrown during the program start-up phase. The functions $\mathrm{mark_s}$ and $\mathrm{unmark_s}$ use the stack marker '†' to implement the automatic allocation (through $\mathrm{new_s}$) and deallocation of stack slots for storing local variables, return values and actual arguments of function calls. The functions $\mathrm{link_s}$ and $\mathrm{unlink_s}$ use the frame marker '‡' to partition the stack into activation frames, each frame corresponding to a function call. All accesses to the top-most frame can be expressed in terms of indirect locators (i.e., offsets from the top-most frame marker), because at each program point the layout of the current top-most frame is statically known. As it will be clearer when considering the concrete rules for function calls, the frame marker is used to move the return value and the actual arguments, which are allocated by the caller, from the activation frame of the caller to the activation frame of the callee, and vice versa.

The memory structures and operations satisfy the following property: for each pair of memory structures σ_0 and σ_1 such that σ_1 has been obtained from σ_0 by any sequence of operations where each $\mathrm{link_s}$ is matched by a corresponding $\mathrm{unlink_s}$, for each indirect locator $i \in \mathrm{Ind}$, if $\sigma_0@i$ and $\sigma_1@i$ are both defined, then $\sigma_0@i = \sigma_1@i$.

As anticipated, we profit from the lack of aliasing in the fragment of CPM considered here, i.e., we assume there is no overlap between the storage cells associated to (l_0, sT_0) and the ones associated to (l_1, sT_1), unless $l_0 = l_1$. Moreover, we need not specify the relationship between $\mu(l, \mathrm{sT}_0)$ and $\mu(l, \mathrm{sT}_1)$ for the case where $\mathrm{sT}_0 \neq \mathrm{sT}_1$. This also implies that the absolute map update operator is underspecified, resulting in a nondeterministic operator. Of course, any real implementation will be characterized by a complete specification: for instance, a precise definition of the memory overflow conditions will take the place of the informal conditions "if the data (resp., stack) segment of σ can be extended" in the definitions of $\mathrm{new_d}$ and $\mathrm{new_s}$. As is clear from the definition above, where memory is writable if and only if it is readable, we do not attempt to model read-only memory. It is also worth observing that, in the sequel, the "meaning" of variable identifiers will depend on unrestricted elements of $\mathrm{Env} \times \mathrm{Mem}$. As a consequence we can have *dangling references*, that is, a pair $(\rho, \sigma) \in \mathrm{Env} \times \mathrm{Mem}$ with $\rho : I$ can be such that there exists an identifier $\mathrm{id} \in I$ for which $\rho(\mathrm{id}) = (a, \mathrm{sT})$ and $\sigma[a, \mathrm{sT}] = \mathtt{memerror}$.

5.4 Configurations

The dynamic semantics of CPM is expressed by means of an *evaluation (or reduction) relation*, which specifies how a *non-terminal configuration* is reduced to a *terminal configuration*. The sets of non-terminal configurations are parametric with respect to a type environment associating every identifier to its type.

DEFINITION 5.3. **(Non-terminal configurations.)** *The sets of* non-terminal configurations *for expressions, local and global declarations, statements, function bodies and catch clauses are given, respectively and for each* $\beta \in \mathrm{TEnv}_I$*, by*

$$\Gamma_{\mathrm{e}}^{\beta} \stackrel{\mathrm{def}}{=} \bigl\{ \langle e, \sigma \rangle \in \mathrm{Exp} \times \mathrm{Mem} \bigm| \exists \mathrm{sT} \in \mathrm{sType} \,.\, \beta \vdash_I e : \mathrm{sT} \bigr\},$$
$$\Gamma_{\mathrm{d}}^{\beta} \stackrel{\mathrm{def}}{=} \bigl\{ \langle d, \sigma \rangle \in \mathrm{Decl} \times \mathrm{Mem} \bigm| \exists \delta \in \mathrm{TEnv} \,.\, \beta \vdash_I d : \delta \bigr\},$$
$$\Gamma_{\mathrm{g}}^{\beta} \stackrel{\mathrm{def}}{=} \bigl\{ \langle g, \sigma \rangle \in \mathrm{Glob} \times \mathrm{Mem} \bigm| \exists \delta \in \mathrm{TEnv} \,.\, \beta \vdash_I g : \delta \bigr\},$$
$$\Gamma_{\mathrm{s}}^{\beta} \stackrel{\mathrm{def}}{=} \bigl\{ \langle s, \sigma \rangle \in \mathrm{Stmt} \times \mathrm{Mem} \bigm| \beta \vdash_I s \bigr\},$$
$$\Gamma_{\mathrm{b}}^{\beta} \stackrel{\mathrm{def}}{=} \bigl\{ \langle \mathrm{body}, \sigma \rangle \in \mathrm{Body} \times \mathrm{Mem} \bigm| \exists \mathrm{sT} \in \mathrm{sType} \,.\, \beta \vdash_I \mathrm{body} : \mathrm{sT} \bigr\},$$
$$\Gamma_{\mathrm{k}}^{\beta} \stackrel{\mathrm{def}}{=} \bigl\{ \langle k, \varepsilon \rangle \in \mathrm{Catch} \times \mathrm{ExceptState} \bigm| \beta \vdash_I k \bigr\}.$$

Each kind of terminal configuration has to allow for the possibility of both a non-exceptional and an exceptional computation path.

DEFINITION 5.4. **(Terminal configurations.)** *The sets of* terminal configurations *for expressions, local and global declarations, statements, function bodies and catch clauses are given, respectively, by*

$$T_{\mathrm{e}} \stackrel{\mathrm{def}}{=} \mathrm{ValState} \uplus \mathrm{ExceptState},$$
$$T_{\mathrm{d}} \stackrel{\mathrm{def}}{=} T_{\mathrm{g}} \stackrel{\mathrm{def}}{=} (\mathrm{Env} \times \mathrm{Mem}) \uplus \mathrm{ExceptState},$$
$$T_{\mathrm{s}} \stackrel{\mathrm{def}}{=} T_{\mathrm{b}} \stackrel{\mathrm{def}}{=} \mathrm{Mem} \uplus \mathrm{ExceptState},$$
$$T_{\mathrm{k}} \stackrel{\mathrm{def}}{=} \bigl(\{\mathtt{caught}\} \times T_{\mathrm{s}}\bigr) \uplus \bigl(\{\mathtt{uncaught}\} \times \mathrm{ExceptState}\bigr).$$

Note that T_{e} is defined as $\mathrm{ValState} \uplus \mathrm{ExceptState}$; as it will be apparent from the concrete semantics, expressions never modify the memory structure, so T_{e} could have been defined as $\mathrm{sVal} \uplus \mathrm{Except}$; but defining it as $\mathrm{ValState} \uplus \mathrm{ExceptState}$ simplifies the approximation relations in Section 6.

In the following, we write N and η to denote a non-terminal and a terminal concrete configuration, respectively. For clarity of notation, we often use angle brackets to highlight that a tuple is indeed representing a configuration. Angle brackets are not normally used for configurations made of a single element. Therefore, when $\varepsilon = (\sigma, \xi) \in \mathrm{ExceptState}$, we indifferently write $\varepsilon \in T_{\mathrm{s}}$ or $\langle \sigma, \xi \rangle \in T_{\mathrm{s}}$, as well as $\langle \mathtt{caught}, \varepsilon \rangle \in T_{\mathrm{k}}$ or $\bigl\langle \mathtt{caught}, (\sigma, \xi) \bigr\rangle \in T_{\mathrm{k}}$.

A few explanatory words are needed for T_{k}. When the evaluation of a non-terminal configuration for catch clauses $\langle k, \varepsilon \rangle \in \Gamma_{\mathrm{k}}^{\beta}$ yields the terminal configuration $\langle \mathtt{caught}, \eta \rangle \in T_{\mathrm{k}}$, then the exception ξ in $\varepsilon = (\sigma, \xi)$ was caught inside k and $\eta \in T_{\mathrm{s}}$ is the result of evaluating the corresponding exception handler statement; note that $\eta \in T_{\mathrm{s}}$ may itself be another exception state, meaning that another exception was thrown during the evaluation of the exception handler statement. In contrast, when the resulting terminal configuration is $\langle \mathtt{uncaught}, \varepsilon \rangle \in T_{\mathrm{k}}$, then the exception in ε was not caught inside k and will be propagated to the outer context.[5]

[5]Note that the names of the labels `caught` and `uncaught` have been chosen as such for clarity, but provide no special meaning: they are only needed for a correct application of the disjoint union construction, since we have $T_{\mathrm{s}} \cap \mathrm{ExceptState} \neq \varnothing$.

5.5 Concrete Evaluation Relations

For convenience, in order to represent function closures, we extend the syntactic category of local declarations with (recursive) execution environments. These syntactic constructs are meant to be only available in the dynamic semantics (in non-terminal configurations): they cannot occur in the program text. Thus we have

$$\mathrm{Decl} \ni d ::= \ldots \mid \rho \mid \mathbf{rec}\,\rho$$

Consequently, if $\rho : I$ we define $\mathrm{DI}(\rho) \stackrel{\mathrm{def}}{=} \mathrm{DI}(\mathbf{rec}\,\rho) \stackrel{\mathrm{def}}{=} I$, $\mathrm{FI}(\rho) \stackrel{\mathrm{def}}{=} \bigcup_{\mathrm{id}\in I} \mathrm{FI}\bigl(\rho(\mathrm{id})\bigr)$ and $\mathrm{FI}(\mathbf{rec}\,\rho) \stackrel{\mathrm{def}}{=} \mathrm{FI}(\rho) \setminus I$, where the function FI is defined on elements of dVal by $\mathrm{FI}(l, \mathrm{sT}) \stackrel{\mathrm{def}}{=} \mathrm{FI}(i, \mathrm{sT}) \stackrel{\mathrm{def}}{=} \varnothing$ and $\mathrm{FI}(\lambda\,\mathrm{fps}\ .\ \mathrm{body}) \stackrel{\mathrm{def}}{=} \mathrm{FI}(\mathrm{body}) \setminus \mathrm{DI}(\mathrm{fps})$. The static semantics is extended by adding the rules

$$\frac{\rho : \delta}{\beta \vdash_I \rho : \delta} \qquad \frac{\beta[\delta\,|_J] \vdash_{I\cup J} \rho : \delta}{\beta \vdash_I \mathbf{rec}\,\rho : \delta} \quad \text{if } J = \mathrm{FI}(\rho) \cap \mathrm{DI}(\rho) \text{ and } \forall \mathrm{id} : (\mathrm{id} \mapsto \mathrm{sT}\ \mathrm{loc}) \notin \delta.$$

The concrete evaluation relations that complete the definition of the concrete semantics for CPM are defined, as usual, by structural induction from a set of rule schemata. The evaluation relations are of the form $\rho \vdash_\beta N \to \eta$, where $\beta \in \mathrm{TEnv}_I$, $\rho \in \mathrm{Env}_J$, $\rho : \beta\,|_J$ and, for some $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$, $N \in \Gamma_q^\beta$ and $\eta \in T_q$.

5.5.1 *Expressions*

Constant.

$$\frac{}{\rho \vdash_\beta \langle \mathrm{con}, \sigma\rangle \to \langle \mathrm{con}, \sigma\rangle} \tag{2}$$

Identifier.

$$\frac{}{\rho \vdash_\beta \langle \mathrm{id}, \sigma\rangle \to \sigma\bigl[\rho(\mathrm{id})\bigr]} \tag{3}$$

Unary minus.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle -e, \sigma\rangle \to \varepsilon} \tag{4}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle m, \sigma_0\rangle}{\rho \vdash_\beta \langle -e, \sigma\rangle \to \langle -m, \sigma_0\rangle} \tag{5}$$

Binary arithmetic operations. Letting $\boxdot$ denote any abstract syntax operator in $\{+, -, *, /, \%\}$ and $\circ \in \{+, -, \cdot, \div, \mathrm{mod}\}$ the corresponding arithmetic operation. Then the rules for addition, subtraction, multiplication, division and remainder are given by the following schemata:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \varepsilon} \tag{6}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle m_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \varepsilon} \tag{7}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle m_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \langle m_1, \sigma_1\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \langle m_0 \circ m_1, \sigma_1\rangle} \quad \text{if } \boxdot \notin \{/, \%\} \text{ or } m_1 \neq 0 \tag{8}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle m_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \langle 0, \sigma_1\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma\rangle \to \langle \sigma_1, \mathtt{divbyzero}\rangle} \quad \text{if } \boxdot \in \{/, \%\} \tag{9}$$

Arithmetic tests. Let $\circledast \in \{=, \neq, <, \leq, \geq, >\}$ be an abstract syntax operator and denote with '$\lesseqgtr$' the corresponding test operation in $\mathbb{Z} \times \mathbb{Z} \to \mathrm{Bool}$. The rules for the arithmetic tests are then given by the following schemata:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma\rangle \to \varepsilon} \tag{10}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle m_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma\rangle \to \varepsilon} \tag{11}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma\rangle \to \langle m_0, \sigma_0\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0\rangle \to \langle m_1, \sigma_1\rangle}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma\rangle \to \langle m_0 \lesseqgtr m_1, \sigma_1\rangle} \tag{12}$$

Negation.

$$\frac{\rho \vdash_\beta \langle b, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{not}\ \ b, \sigma\rangle \to \varepsilon} \tag{13}$$

$$\frac{\rho \vdash_\beta \langle b, \sigma\rangle \to \langle t, \sigma_0\rangle}{\rho \vdash_\beta \langle \mathbf{not}\ \ b, \sigma\rangle \to \langle \neg\, t, \sigma_0\rangle} \tag{14}$$

Conjunction.

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle b_0 \mathbin{\mathbf{and}} b_1, \sigma\rangle \to \varepsilon} \tag{15}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle}{\rho \vdash_\beta \langle b_0 \mathbin{\mathbf{and}} b_1, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle} \tag{16}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle b_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle b_0 \mathbin{\mathbf{and}} b_1, \sigma\rangle \to \eta} \tag{17}$$

Disjunction.

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle b_0 \mathbin{\mathbf{or}} b_1, \sigma\rangle \to \varepsilon} \tag{18}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle}{\rho \vdash_\beta \langle b_0 \mathbin{\mathbf{or}} b_1, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle} \tag{19}$$

$$\frac{\rho \vdash_\beta \langle b_0, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle \quad \rho \vdash_\beta \langle b_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle b_0 \mathbin{\mathbf{or}} b_1, \sigma\rangle \to \eta} \tag{20}$$

5.5.2 *Declarations*

Nil.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nil}, \sigma\rangle \rightarrow \langle \varnothing, \sigma\rangle} \tag{21}$$

Environment.

$$\frac{}{\rho \vdash_\beta \langle \rho_0, \sigma\rangle \rightarrow \langle \rho_0, \sigma\rangle} \tag{22}$$

Recursive environment.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{rec}\, \rho_0, \sigma\rangle \rightarrow \langle \rho_1, \sigma\rangle} \tag{23}$$

$$\text{if } \rho_1 = \bigl\{\, \mathrm{id} \mapsto \rho_0(\mathrm{id}) \bigm| \rho_0(\mathrm{id}) = (\lambda\, \mathrm{fps}\, .\, \mathbf{extern}, \mathrm{sT}) \,\bigr\} \\ \cup \left\{ \mathrm{id} \mapsto \mathrm{abs}_1 \;\middle|\; \begin{array}{l} \forall i \in \{0,1\} : \mathrm{abs}_i = (\lambda\, \mathrm{fps}\, .\, \mathbf{let}\, d_i\ \mathbf{in}\, s\, \mathbf{result}\, e, \mathrm{sT}), \\ \rho_0(\mathrm{id}) = \mathrm{abs}_0, d_1 = \mathbf{rec}\bigl(\rho_0 \setminus \mathrm{DI}(\mathrm{fps})\bigr); d_0 \end{array} \right\}.$$

Global variable declaration.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \rightarrow \varepsilon}{\rho \vdash_\beta \langle \mathbf{gvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma\rangle \rightarrow \mathrm{cleanup}_\mathrm{d}(\varepsilon)} \tag{24}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \rightarrow \upsilon}{\rho \vdash_\beta \langle \mathbf{gvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma\rangle \rightarrow \mathrm{cleanup}_\mathrm{d}(\varepsilon)} \quad \text{if } \mathrm{new}_\mathrm{d}(\upsilon) = \varepsilon \tag{25}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \rightarrow \upsilon}{\rho \vdash_\beta \langle \mathbf{gvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma\rangle \rightarrow \langle \rho_1, \sigma_1\rangle} \tag{26}$$

if $\mathrm{new}_\mathrm{d}(\upsilon) = (\sigma_1, l)$ and $\rho_1 = \bigl\{\mathrm{id} \mapsto (l, \mathrm{sT})\bigr\}$.

Local variable declaration.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \rightarrow \varepsilon}{\rho \vdash_\beta \langle \mathbf{lvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma\rangle \rightarrow \mathrm{unmark}_\mathrm{s}(\varepsilon)} \tag{27}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \rightarrow \upsilon}{\rho \vdash_\beta \langle \mathbf{lvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma\rangle \rightarrow \mathrm{unmark}_\mathrm{s}(\varepsilon)} \quad \text{if } \mathrm{new}_\mathrm{s}(\upsilon) = \varepsilon \tag{28}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \rightarrow \upsilon}{\rho \vdash_\beta \langle \mathbf{lvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma\rangle \rightarrow \langle \rho_1, \sigma_1\rangle} \tag{29}$$

if $\mathrm{new}_\mathrm{s}(\upsilon) = (\sigma_1, i)$ and $\rho_1 = \bigl\{\mathrm{id} \mapsto (i, \mathrm{sT})\bigr\}$.

Function declaration.

$$\frac{}{\rho \vdash_\beta \bigl\langle \mathbf{function}\, \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathrm{body}_0, \sigma\bigr\rangle \rightarrow \langle \rho_0, \sigma\rangle} \tag{30}$$

if $\rho_0 = \bigl\{\mathrm{id} \mapsto (\lambda\, \mathrm{fps}\, .\, \mathrm{body}_1, \mathrm{sT})\bigr\}$ and either $\mathrm{body}_0 = \mathrm{body}_1 = \mathbf{extern}$ or, for each $i \in \{0,1\}$, $\mathrm{body}_i = \mathbf{let}\, d_i\ \mathbf{in}\, s\, \mathbf{result}\, e$, $I = \mathrm{FI}(\mathrm{body}_0) \setminus \mathrm{DI}(\mathrm{fps})$ and $d_1 = \rho\,|_I; d_0$.

Recursive declaration.

$$\frac{(\rho \setminus J) \vdash_{\beta[\beta_1]} \langle g, \sigma \rangle \to \langle \rho_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle \mathbf{rec}\, \rho_0, \sigma_0 \rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{rec}\, g, \sigma \rangle \to \eta} \tag{31}$$

if $J = \mathrm{FI}(g) \cap \mathrm{DI}(g)$, $\beta \vdash_{\mathrm{FI}(g)} g : \beta_0$ and $\beta_1 = \beta_0|_J$.

Global sequential composition.

$$\frac{\rho \vdash_\beta \langle g_0, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle g_0; g_1, \sigma \rangle \to \varepsilon} \tag{32}$$

$$\frac{\rho \vdash_\beta \langle g_0, \sigma \rangle \to \langle \rho_0, \sigma_0 \rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle g_1, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle g_0; g_1, \sigma \rangle \to \varepsilon} \quad \text{if } \beta \vdash_{\mathrm{FI}(g_0)} g_0 : \beta_0 \tag{33}$$

$$\frac{\rho \vdash_\beta \langle g_0, \sigma \rangle \to \langle \rho_0, \sigma_0 \rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle g_1, \sigma_0 \rangle \to \langle \rho_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle g_0; g_1, \sigma \rangle \to \big\langle \rho_0[\rho_1], \sigma_1 \big\rangle} \quad \text{if } \beta \vdash_{\mathrm{FI}(g_0)} g_0 : \beta_0 \tag{34}$$

Local sequential composition.

$$\frac{\rho \vdash_\beta \langle d_0, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle d_0; d_1, \sigma \rangle \to \varepsilon} \tag{35}$$

$$\frac{\rho \vdash_\beta \langle d_0, \sigma \rangle \to \langle \rho_0, \sigma_0 \rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle d_1, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle d_0; d_1, \sigma \rangle \to \varepsilon} \quad \text{if } \beta \vdash_{\mathrm{FI}(d_0)} d_0 : \beta_0 \tag{36}$$

$$\frac{\rho \vdash_\beta \langle d_0, \sigma \rangle \to \langle \rho_0, \sigma_0 \rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle d_1, \sigma_0 \rangle \to \langle \rho_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle d_0; d_1, \sigma \rangle \to \big\langle \rho_0[\rho_1], \sigma_1 \big\rangle} \quad \text{if } \beta \vdash_{\mathrm{FI}(d_0)} d_0 : \beta_0 \tag{37}$$

5.5.3 *Statements*

Nop.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nop}, \sigma \rangle \to \sigma} \tag{38}$$

Assignment.

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathrm{id} := e, \sigma \rangle \to \varepsilon} \tag{39}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \langle \mathrm{sval}, \sigma_0 \rangle}{\rho \vdash_\beta \langle \mathrm{id} := e, \sigma \rangle \to \sigma_0\big[\rho(\mathrm{id}) := \mathrm{sval}\big]} \tag{40}$$

Statement sequence.

$$\frac{\rho \vdash_\beta \langle s_0, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle s_0; s_1, \sigma \rangle \to \varepsilon} \tag{41}$$

$$\frac{\rho \vdash_\beta \langle s_0, \sigma \rangle \to \sigma_0 \quad \rho \vdash_\beta \langle s_1, \sigma_0 \rangle \to \eta}{\rho \vdash_\beta \langle s_0; s_1, \sigma \rangle \to \eta} \tag{42}$$

Block.

$$\frac{\rho \vdash_\beta \big\langle d, \mathrm{mark_s}(\sigma)\big\rangle \to \varepsilon}{\rho \vdash_\beta \langle d; s, \sigma\rangle \to \varepsilon} \tag{43}$$

$$\frac{\rho \vdash_\beta \big\langle d, \mathrm{mark_s}(\sigma)\big\rangle \to \langle \rho_0, \sigma_0\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle d; s, \sigma\rangle \to \mathrm{unmark_s}(\eta)} \quad \text{if } \beta \vdash_{\mathrm{FI}(d)} d : \beta_0 \tag{44}$$

Conditional.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \to \varepsilon} \tag{45}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s_0, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \to \eta} \tag{46}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma\rangle \to \eta} \tag{47}$$

While.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \varepsilon} \tag{48}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{ff}, \sigma_0\rangle}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \sigma_0} \tag{49}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \varepsilon} \tag{50}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_0\rangle \quad \rho \vdash_\beta \langle s, \sigma_0\rangle \to \sigma_1 \quad \rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma_1\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma\rangle \to \eta} \tag{51}$$

Throw.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{throw}\, \chi, \sigma\rangle \to \langle \sigma, \chi\rangle} \tag{52}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{throw}\, e, \sigma\rangle \to \varepsilon} \tag{53}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{sval}, \sigma_0\rangle}{\rho \vdash_\beta \langle \mathbf{throw}\, e, \sigma\rangle \to \langle \sigma_0, \mathrm{sval}\rangle} \tag{54}$$

Try blocks.

$$\frac{\rho \vdash_\beta \langle s, \sigma\rangle \to \sigma_0}{\rho \vdash_\beta \langle \mathbf{try}\, s\, \mathbf{catch}\, k, \sigma\rangle \to \sigma_0} \tag{55}$$

$$\frac{\rho \vdash_\beta \langle s, \sigma\rangle \to \varepsilon_0 \quad \rho \vdash_\beta \langle k, \varepsilon_0\rangle \to \langle u, \eta\rangle}{\rho \vdash_\beta \langle \mathbf{try}\, s\, \mathbf{catch}\, k, \sigma\rangle \to \eta} \quad \text{if } u \in \{\mathtt{caught}, \mathtt{uncaught}\} \tag{56}$$

$$\frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \to \sigma_0 \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{try}\, s_0\, \mathbf{finally}\, s_1, \sigma\rangle \to \eta} \tag{57}$$

$$\frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \to \langle \sigma_0, \xi_0\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \to \sigma_1}{\rho \vdash_\beta \langle \mathbf{try}\, s_0\, \mathbf{finally}\, s_1, \sigma\rangle \to \langle \sigma_1, \xi_0\rangle} \tag{58}$$

$$\frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \to \langle \sigma_0, \xi_0\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{try}\, s_0\, \mathbf{finally}\, s_1, \sigma\rangle \to \varepsilon} \tag{59}$$

Function call. Consider the following conditions:

$$\left.\begin{aligned} \beta(\mathrm{id}) &= (\mathrm{fps} \to \mathrm{sT}_0) \\ \rho(\mathrm{id}) &= \lambda\, \mathrm{id}_1 : \mathrm{sT}_1, \ldots, \mathrm{id}_n : \mathrm{sT}_n \,.\, \mathrm{body} \\ d &= (\mathbf{lvar}\, \underline{\mathrm{x}}_0 : \mathrm{sT}_0 = \mathrm{id}_0; \mathbf{lvar}\, \underline{\mathrm{x}}_1 : \mathrm{sT}_1 = e_1; \ldots; \mathbf{lvar}\, \underline{\mathrm{x}}_n : \mathrm{sT}_n = e_n) \end{aligned}\right\} \tag{60}$$

$$\rho_1 = \bigl\{ \underline{\mathrm{x}}_0 \mapsto (0, \mathrm{sT}_0) \bigr\} \cup \bigl\{ \mathrm{id}_j \mapsto (j, \mathrm{sT}_j) \bigm| j = 1, \ldots, n \bigr\},\ \rho_0 : \beta_0,\ \rho_1 : \beta_1. \tag{61}$$

Then the rule schemata for function calls are the following:

$$\frac{\rho \vdash_\beta \bigl\langle d, \mathrm{mark_s}(\sigma) \bigr\rangle \to \varepsilon}{\rho \vdash_\beta \bigl\langle \mathrm{id}_0 := \mathrm{id}(e_1, \ldots, e_n), \sigma \bigr\rangle \to \varepsilon} \quad \text{if (60) holds} \tag{62}$$

$$\frac{\begin{array}{c} \rho \vdash_\beta \bigl\langle d, \mathrm{mark_s}(\sigma) \bigr\rangle \to \langle \rho_0, \sigma_0 \rangle \\ \rho[\rho_1] \vdash_{\beta[\beta_1]} \bigl\langle \mathrm{body}, \mathrm{link_s}(\sigma_0) \bigr\rangle \to \varepsilon \end{array}}{\rho \vdash_\beta \bigl\langle \mathrm{id}_0 := \mathrm{id}(e_1, \ldots, e_n), \sigma \bigr\rangle \to \mathrm{unmark_s}\bigl(\mathrm{unlink_s}(\varepsilon)\bigr)} \quad \text{if (60) and (61) hold} \tag{63}$$

$$\frac{\begin{array}{l} \rho \vdash_\beta \bigl\langle d, \mathrm{mark_s}(\sigma) \bigr\rangle \to \langle \rho_0, \sigma_0 \rangle \\ \rho[\rho_1] \vdash_{\beta[\beta_1]} \bigl\langle \mathrm{body}, \mathrm{link_s}(\sigma_0) \bigr\rangle \to \sigma_1 \\ \rho[\rho_0] \vdash_{\beta[\beta_0]} \bigl\langle \mathrm{id}_0 := \underline{\mathrm{x}}_0, \mathrm{unlink_s}(\sigma_1) \bigr\rangle \to \eta_2 \end{array}}{\rho \vdash_\beta \bigl\langle \mathrm{id}_0 := \mathrm{id}(e_1, \ldots, e_n), \sigma \bigr\rangle \to \mathrm{unmark_s}(\eta_2)} \quad \text{if (60) and (61) hold} \tag{64}$$

Note that parameter passing is implemented by using reserved identifiers that reference the return value ($\underline{\mathrm{x}}_0$) and the actual arguments ($\underline{\mathrm{x}}_1$, ..., $\underline{\mathrm{x}}_n$). When evaluating the function body (i.e., after linking a new activation frame), the callee can get access to the return value and the arguments' values by using the indirect locators 0 and 1, ..., n, respectively; to this end, the callee uses the environment ρ_1, where the reserved identifier $\underline{\mathrm{x}}_0$ is still mapped to the return value, whereas the arguments are accessible using the formal parameters' names id_1, ..., id_n.

5.5.4 *Function Bodies.*

$$\frac{\rho \vdash_\beta \bigl\langle d, \mathrm{mark_s}(\sigma) \bigr\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{let}\, d\, \mathbf{in}\, s\, \mathbf{result}\, e, \sigma \rangle \to \varepsilon} \tag{65}$$

$$\frac{\rho \vdash_\beta \bigl\langle d, \mathrm{mark_s}(\sigma) \bigr\rangle \to \langle \rho_0, \sigma_0 \rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{let}\, d\, \mathbf{in}\, s\, \mathbf{result}\, e, \sigma \rangle \to \mathrm{unmark_s}(\varepsilon)} \quad \text{if } \beta \vdash_{\mathrm{FI}(d)} d : \beta_0 \tag{66}$$

$$\frac{\begin{array}{l}\rho \vdash_\beta \bigl\langle d, \mathrm{mark_s}(\sigma)\bigr\rangle \rightarrow \langle \rho_0, \sigma_0\rangle \\ \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0\rangle \rightarrow \sigma_1 \\ \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle \underline{\mathrm{x}}_0 := e, \sigma_1\rangle \rightarrow \eta_0\end{array}}{\rho \vdash_\beta \langle \mathbf{let}\, d\, \mathbf{in}\, s\, \mathbf{result}\, e, \sigma\rangle \rightarrow \mathrm{unmark_s}(\eta_0)} \quad \text{if } \beta \vdash_{\mathrm{FI}(d)} d : \beta_0 \tag{67}$$

$$\frac{}{\rho \vdash_\beta \bigl\langle \mathbf{extern}, (\mu, w)\bigr\rangle \rightarrow \eta} \tag{68}$$

if $\exists \sigma_0 = (\mu_0, w) \in \mathrm{Mem}, \xi \in \mathrm{Except} \,.\, \eta = \sigma_0 \vee \eta = \langle \sigma_0, \xi\rangle$.

5.5.5 *Catch Clauses*

Catch.

$$\frac{\rho \vdash_\beta \langle s, \sigma\rangle \rightarrow \eta_0}{\rho \vdash_\beta \bigl\langle (p)\, s, (\sigma, \xi)\bigr\rangle \rightarrow \langle \mathtt{caught}, \eta_0\rangle} \tag{69}$$

if $p = \xi \in \mathrm{RTSExcept}$, or $p = \mathrm{type}(\xi)$, or $p = \mathbf{any}$.

$$\frac{}{\rho \vdash_\beta \bigl\langle (\mathrm{id} : \mathrm{sT})\, s, (\sigma, \mathrm{sval})\bigr\rangle \rightarrow \bigl\langle \mathtt{caught}, \mathrm{unmark_s}(\varepsilon_0)\bigr\rangle} \tag{70}$$

if $\mathrm{sT} = \mathrm{type}(\mathrm{sval})$ and $\varepsilon_0 = \mathrm{new_s}\bigl(\mathrm{sval}, \mathrm{mark_s}(\sigma)\bigr)$.

$$\frac{\rho\bigl[\{\mathrm{id} \mapsto (i, \mathrm{sT})\}\bigr] \vdash_{\beta[\{\mathrm{id} \mapsto \mathrm{sT\ loc}\}]} \langle s, \sigma_0\rangle \rightarrow \eta_0}{\rho \vdash_\beta \bigl\langle (\mathrm{id} : \mathrm{sT})\, s, (\sigma, \mathrm{sval})\bigr\rangle \rightarrow \bigl\langle \mathtt{caught}, \mathrm{unmark_s}(\eta_0)\bigr\rangle} \tag{71}$$

if $\mathrm{sT} = \mathrm{type}(\mathrm{sval})$ and $(\sigma_0, i) = \mathrm{new_s}\bigl(\mathrm{sval}, \mathrm{mark_s}(\sigma)\bigr)$.

$$\frac{}{\rho \vdash_\beta \bigl\langle (p)\, s, (\sigma, \xi)\bigr\rangle \rightarrow \bigl\langle \mathtt{uncaught}, (\sigma, \xi)\bigr\rangle} \tag{72}$$

if, letting $\mathrm{cT} = \mathrm{type}(\xi)$, we have $p \notin \bigl\{\xi, \mathrm{cT}, \mathbf{any}\bigr\}$ and $\forall \mathrm{id} \in \mathrm{Id} : p \neq \mathrm{id} : \mathrm{cT}$.

Catch sequence.

$$\frac{\rho \vdash_\beta \langle k_0, \varepsilon\rangle \rightarrow \langle \mathtt{caught}, \eta_0\rangle}{\rho \vdash_\beta \langle k_0; k_1, \varepsilon\rangle \rightarrow \langle \mathtt{caught}, \eta_0\rangle} \tag{73}$$

$$\frac{\rho \vdash_\beta \langle k_0, \varepsilon\rangle \rightarrow \langle \mathtt{uncaught}, \varepsilon_0\rangle \quad \rho \vdash_\beta \langle k_1, \varepsilon_0\rangle \rightarrow \eta}{\rho \vdash_\beta \langle k_0; k_1, \varepsilon\rangle \rightarrow \eta} \tag{74}$$

5.6 Concrete Divergence Relation

In order to capture divergent computations, we follow the approach of Cousot and Cousot [CC92c], also advocated by Schmidt [Sch98] and Leroy [Ler06]. This consists in introducing a *divergence relation* by means of sequents of the form $\rho \vdash_\beta N \xrightarrow{\infty}$, where $N \in \Gamma_q$ and $q \in \{\mathrm{s}, \mathrm{b}, \mathrm{k}\}$. Intuitively, a divergence sequent of the form, say, $\rho \vdash_\beta \langle s, \sigma\rangle \xrightarrow{\infty}$ means that, in the context given by ρ and σ, the execution of statement s diverges. We now give a set of rules that (interpreted coinductively, as we will see later) allow to characterize the behavior of divergent computations. For instance, the following rule schemata characterize the divergence behavior of statement sequences:

$$\frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \xrightarrow{\infty}}{\rho \vdash_\beta \langle s_0; s_1, \sigma\rangle \xrightarrow{\infty}} \qquad \frac{\rho \vdash_\beta \langle s_0, \sigma\rangle \rightarrow \sigma_0 \quad \rho \vdash_\beta \langle s_1, \sigma_0\rangle \xrightarrow{\infty}}{\rho \vdash_\beta \langle s_0; s_1, \sigma\rangle \xrightarrow{\infty}}$$

Notice that, once the set of concrete rules characterizing finite computations is known, the concrete rules modeling divergences can be specified systematically (and thus implicitly). Namely, for each concrete rule

$$\frac{P_0 \quad \cdots \quad P_{i-1} \quad \rho_i \vdash_{\beta_i} N_i \rightarrow \eta_i \quad P_{i+1} \quad \cdots \quad P_{h-1}}{\rho \vdash_\beta N \rightarrow \eta} \quad \text{(side condition)} \tag{75}$$

such that $0 \leq i < h$ and, for $q \in \{\mathrm{s}, \mathrm{b}, \mathrm{k}\}$, $N_i \in \biguplus \Gamma_q^{\beta_i}$ and $N \in \biguplus \Gamma_q^{\beta}$, there is the corresponding divergence rule where the i-th premise is diverging, i.e.,

$$\frac{P_0 \quad \cdots \quad P_{i-1} \quad \rho_i \vdash_{\beta_i} N_i \xrightarrow{\infty}}{\rho \vdash_\beta N \xrightarrow{\infty}} \quad \text{(side condition)}$$

Therefore, there are two rules above modeling the divergence of statement sequences, which can be obtained from rule (42). It is worth noting that a single divergence rule schema can be obtained from more than one of the concrete rules in Section 5.5.

We will use the terms *negative* and *positive* to distinguish the different kinds of rules constructed in this and the previous section, respectively.

DEFINITION 5.5. **(Concrete semantics rules.)** *The set $\mathcal{R}_+$ (resp., $\mathcal{R}_-$) of* positive *(resp.,* negative*)* concrete semantics rules *is the infinite set obtained by instantiating the rule schemata of* Section 5.5 *(resp.,* Section 5.6*) in all possible ways (respecting, of course, the side conditions). Moreover, $\mathcal{R} \stackrel{\mathrm{def}}{=} \mathcal{R}_+ \uplus \mathcal{R}_-$.*

5.7 Concrete Semantics Trees

The concrete semantics of a program is a (possibly infinite) set of finite or infinite trees. Such trees are defined in terms of the (infinite) set of instances of the rules defined in the previous two sections.

Let $\mathcal{S}$ be the (infinite) set of sequents occurring in the premises and conclusions of the rules in $\mathcal{R}$. The *concrete semantics universe*, denoted by $\mathcal{U}$, is the set of finitely branching trees of at most ω-depth with labels in $\mathcal{S}$.

DEFINITION 5.6. **(Concrete semantics universe.)** *A set $P \subseteq \mathbb{N}^\star$ is* prefix-closed *if, for each $z \in \mathbb{N}^\star$ and each $n \in \mathbb{N}$, $zn \in P$ implies $z \in P$. A set $P \subseteq \mathbb{N}^\star$ is* canonical *if, for each $z \in \mathbb{N}^\star$ there exists $h \in \mathbb{N}$ such that*

$$\{\, n \in \mathbb{N} \mid zn \in P \,\} = \{0, \ldots, h-1\}.$$

*An $\mathcal{S}$-*tree *is a partial function $\theta \colon \mathbb{N}^\star \rightarrowtail \mathcal{S}$ such that* $\mathrm{dom}(\theta)$ *is prefix-closed and canonical. The* concrete semantics universe *$\mathcal{U}$ is the set of all $\mathcal{S}$-trees.*

For each $p \in \mathrm{dom}(\theta)$, the tree $\theta_{[p]}$ defined, for each $z \in \mathbb{N}^\star$, by $\theta_{[p]}(z) \stackrel{\mathrm{def}}{=} \theta(pz)$, is called a *subtree* of θ; it is called a *proper subtree* if $p \neq \epsilon$. If $\mathrm{dom}(\theta) = \varnothing$, then θ is the empty tree. If θ is not empty, then $\theta(\epsilon)$ is the *root* of θ and, if $\{0, \ldots, h-1\} \subseteq \mathrm{dom}(\theta)$ and $h \notin \mathrm{dom}(\theta)$, then $\theta_{[0]}, \ldots, \theta_{[h-1]}$ are its *immediate subtrees* (note that $h \in \mathbb{N}$ may be zero); in this case θ can be denoted by $\frac{\theta_{[0]} \; \cdots \; \theta_{[h-1]}}{\theta(\epsilon)}$.

Definition 5.7. **(Concrete semantics trees.)** *Let $\mathcal{F}_+ \colon \wp(\mathcal{U}) \to \wp(\mathcal{U})$ be the continuous function over the complete lattice $\bigl(\wp(\mathcal{U}), \subseteq\bigr)$ given, for all $U \in \wp(\mathcal{U})$, by*

$$\mathcal{F}_+(U) \stackrel{\mathrm{def}}{=} \left\{ \frac{\theta_0 \; \cdots \; \theta_{h-1}}{s} \;\middle|\; \begin{array}{l} \theta_0, \ldots, \theta_{h-1} \in U, \\ \dfrac{\theta_0(\epsilon) \; \cdots \; \theta_{h-1}(\epsilon)}{s} \in \mathcal{R}_+ \end{array} \right\}.$$

The set of positive concrete semantics trees *is $\Theta_+ \stackrel{\mathrm{def}}{=} \mathrm{lfp}_\subseteq(\mathcal{F}_+)$. Consider now the co-continuous function $\mathcal{F}_- \colon \wp(\mathcal{U}) \to \wp(\mathcal{U})$ given, for each $U \in \wp(\mathcal{U})$, by*

$$\mathcal{F}_-(U) \stackrel{\mathrm{def}}{=} \left\{ \frac{\theta_0 \; \cdots \; \theta_{h-1}}{s} \;\middle|\; \begin{array}{l} \theta_0, \ldots, \theta_{h-2} \in \Theta_+, \quad \theta_{h-1} \in U, \\ \dfrac{\theta_0(\epsilon) \; \cdots \; \theta_{h-1}(\epsilon)}{s} \in \mathcal{R}_- \end{array} \right\}.$$

The set of negative concrete semantics trees *is $\Theta_- \stackrel{\mathrm{def}}{=} \mathrm{gfp}_\subseteq(\mathcal{F}_-)$. The set of all* concrete semantics trees *is $\Theta \stackrel{\mathrm{def}}{=} \Theta_+ \uplus \Theta_-$.*

We now show that, for every concrete non-terminal configuration, there exists a concrete semantics tree with that in the root.

Proposition 5.8. *For each $\beta \in \mathrm{TEnv}$, $\rho \in \mathrm{Env}$ such that $\rho : \beta$ and $N \in \Gamma_q^\beta$, where $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$, there exists $\theta \in \Theta$ such that*

$$\theta(\epsilon) \in \bigl\{\, (\rho \vdash_\beta N \to \eta) \bigm| \eta \in T_q \,\bigr\} \uplus \bigl\{\, (\rho \vdash_\beta N \xrightarrow{\infty}) \,\bigr\}.$$

Proof. If $q = \mathrm{e}$ and $\eta \in T_\mathrm{e}$, we say that the sequent $(\rho \vdash_\beta N \to \eta)$ is well-typed if $N = \langle e, \sigma_0 \rangle$ and $\eta = \langle \mathrm{sval}, \sigma_1 \rangle$ imply $\beta \vdash e : \mathrm{type}(\mathrm{sval})$. For the proof, let

$$S_+(\rho, \beta, N) \stackrel{\mathrm{def}}{=} \bigl\{\, s \bigm| s = (\rho \vdash_\beta N \to \eta), \eta \in T_q, (q = \mathrm{e} \implies s \text{ is well-typed}) \,\bigr\}.$$

We now assume that $N \in \Gamma_q^\beta$ is a fixed but arbitrary non-terminal configuration. It suffices to show there exists $\theta \in \Theta$ such that $\theta(\epsilon) \in S_+(\rho, \beta, N) \uplus \bigl\{\, (\rho \vdash_\beta N \xrightarrow{\infty}) \,\bigr\}$. Let R_0 be the set of all rules in $\mathcal{R}_+$ whose conclusions are in $S_+(\rho, \beta, N)$. By inspecting the concrete evaluation rule schemata in Section 5.5, $R_0 \neq \varnothing$. Let $j \geq 0$ be the maximal value for which there exist finite trees $\theta_0, \ldots, \theta_{j-1} \in \Theta_+$ where $P_0 = \theta_0(\epsilon), \ldots, P_{j-1} = \theta_{j-1}(\epsilon)$ are the first j premises of a rule in R_0. Let $R_j \subseteq R_0$ be the set of all rules in R_0 with $P_0, \ldots, P_{j-1}$ as their first j premises; then $R_j \neq \varnothing$. By inspecting the rule schemata in Section 5.5, it can be seen that, if there exists

$\frac{P_0 \cdots P_{j-1}\, P'_j \cdots}{s'} \in R_j$ for some $P'_j \in S_+(\rho_j, \beta_j, N_j)$ and $s' \in S_+(\rho, \beta, N)$, then[6]

$$\forall P_j \in S_+(\rho_j, \beta_j, N_j) : \exists s \in S_+(\rho, \beta, N) \;.\; \frac{P_0 \;\cdots\; P_{j-1}\; P_j \;\cdots}{s} \in R_j. \tag{76}$$

Suppose that $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}\}$ so that we can also assume $N = \langle u, \sigma\rangle$. We show by structural induction on u that there exists $\theta \in \Theta_+$ such that $\theta(\epsilon) \in S_+(\rho, \beta, N)$. By inspecting the rule schemata in Section 5.5, it can be seen that, if u is atomic, the rules in R_0 have no premises (so that $j = 0$) and hence, letting $\theta \in \Theta_+$ be the singleton tree consisting of the conclusion of a rule in R_0, we obtain that $\theta(\epsilon) \in S_+(\rho, \beta, N)$. Otherwise, u is not atomic, we show that each of the rules in R_j has exactly j premises; to do this, we assume there exists a rule in R_j with a $(j+1)$-th premise P_j and derive a contradiction. Let $N_j \in \Gamma_{q_j}^{\beta_j}$ be the non-terminal configuration in P_j. By inspecting the rule schemata in Section 5.5 in the case that $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}\}$, it can be seen that:

(i) $q_j \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}\}$ so that N_j has the form $\langle u_j, \sigma_j\rangle$;

(ii) u_j is a substructure of u unless R_j consists of instances of the schematic rule (31) and $j = 1$.

If u_j is a substructure of u, by property (i), we can apply structural induction to obtain that there exists a finite tree $\theta_j \in \Theta_+$ such that $P_j = \theta_j(\epsilon) \in S_+(\rho_j, \beta_j, N_j)$; hence, by property (76), there exists a rule in R_j having P_j as its $(j+1)$-th premise; contradicting the assumption that j was maximal. Otherwise, by property (ii), if u_j is not a substructure of u, the rules in R_0 must be instances of rule schema (31) and $j = 1$; in this case, rule schema (23), which has no premises, can be instantiated with the second premise of a rule in R_j as its conclusion; and again we have a contradiction. Thus, for any u_j, all rules in R_j have exactly j premises. By Definition 5.7, $\theta = \frac{\theta_0 \cdots \theta_{j-1}}{s} \in \Theta_+$ for some $s \in S_+(\rho, \beta, N)$. Therefore, since $\Theta_+ \subseteq \Theta$, the thesis holds when $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}\}$.

Suppose now that $q \in \{\mathrm{s}, \mathrm{b}, \mathrm{k}\}$. We prove that, if there does not exist a tree $\theta \in \Theta_+$ such that $\theta(\epsilon) \in S_+(\rho, \beta, N)$, then, for all $n \geq 0$, there exists a tree θ such that $\theta(\epsilon) = s_\infty \stackrel{\mathrm{def}}{=} (\rho \vdash_\beta N \xrightarrow{\infty})$ and $\theta \in \mathcal{F}^n_-(\mathcal{U})$. To this end, we reason by induction on $n \geq 0$. By our assumption that there is no tree $\theta \in \Theta_+$ such that $\theta(\epsilon) \in S_+(\rho, \beta, N)$, there must exist a rule

$$\frac{P_0 \;\cdots\; P_{j-1}\; P_j \;\cdots}{s} \in R_j$$

for some $P_j \in S_+(\rho_j, \beta_j, N_j)$; let q_j be such that $N_j \in \Gamma_{q_j}^{\beta_j}$. By the maximality of j, there is no tree in Θ_+ whose root is P_j. We have already shown that, if

[6]To help understand this property, we illustrate it in the case that $q = \mathrm{e}$ and the non-terminal configuration is $N = \langle b_0\ \mathbf{and}\ b_1, \sigma\rangle$; hence the concrete rule schemata (15)–(17) will apply. In all the rule instances, the first premise is of the form $P_0 = (\rho \vdash_\beta N_0 \to \eta_0)$, where $N_0 = \langle b_0, \sigma\rangle$; as a consequence, we have $S_+(\rho, \beta, N_0) = \{\, (\rho \vdash_\beta N_0 \to \eta_0) \mid \eta_0 \in B \,\}$, where $B \stackrel{\mathrm{def}}{=} \mathrm{ExceptState} \uplus \{\, \langle t, \sigma_0\rangle \in T_\mathrm{e} \mid t \in \mathrm{Bool}, \sigma_0 \in \mathrm{Mem} \,\}$. Thus, for each terminal configuration $\eta_0 \in B$, there is a rule instance having η_0 in its first premise — that is we instantiate rule (15) when $\eta_0 = \varepsilon$, rule (16) when $\eta_0 = \langle \mathrm{ff}, \sigma_0\rangle$ and rule (17) when $\eta_0 = \langle \mathrm{tt}, \sigma_0\rangle$. Thus property (76) holds for $j = 0$. Moreover, although only rule (17) applies when $j = 1$, the terminal configuration for the second premise (P_1) is just any terminal configuration in T_e. Thus property (76) also holds for $j = 1$.

$q_j \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}\}$, then there exists a tree $\theta_j \in \Theta_+$ such that $\theta_j(\epsilon) \in S_+(\rho_j, \beta_j, N_j)$; thus, by property (76), there must be a rule in R_j whose $(j+1)$-th premise is $\theta_j(\epsilon)$; contradicting the assumption that $j \geq 0$ is maximal. Hence $q_j \in \{\mathrm{s}, \mathrm{b}, \mathrm{k}\}$. By the definition of the negative concrete semantics rules in Section 5.6, there exists a corresponding negative rule

$$\frac{P_0 \ \cdots \ P_{j-1} \ P_\infty}{s_\infty} \in \mathcal{R}_-$$

such that $P_\infty = (\rho_j \vdash_{\beta_j} N_j \xrightarrow{\infty})$. Hence, by Definition 5.6, there exists a tree in $\mathcal{U} = \mathcal{F}_-^0(\mathcal{U})$ with root s_∞, so that the inductive hypothesis holds for $n = 0$. Suppose now that $n > 0$. By the inductive hypothesis, there exists a tree $\theta_\infty \in \mathcal{F}_-^{n-1}(\mathcal{U})$ such that $\theta_\infty(\epsilon) = P_\infty$. Hence, by Definition 5.7, $\frac{\theta_0 \ \cdots \ \theta_{j-1} \ \theta_\infty}{s_\infty} \in \mathcal{F}_-^n(\mathcal{U})$. Thus, for all $n \geq 0$, there exists a tree in $\mathcal{F}_-^n(\mathcal{U})$ with root s_∞ and hence, by Definition 5.7, there exists a tree in Θ_- with root s_∞. Since $\Theta = \Theta_+ \uplus \Theta_-$, the thesis holds when $q \in \{\mathrm{s}, \mathrm{b}, \mathrm{k}\}$. $\square$

The concrete semantics of a valid program g with respect to the initial memory structure $\sigma_\mathrm{i} \stackrel{\mathrm{def}}{=} (\varnothing, \epsilon) \in \mathrm{Mem}$ is a set of concrete semantics trees. This set will always include a tree $\theta_0 \in \Theta$ (which, by Proposition 5.8, must exist) such that

$$\theta_0(\epsilon) = \Big(\varnothing \vdash_\varnothing \big\langle (g; \mathbf{gvar}\, \underline{\mathrm{x}} : \mathrm{integer} = 0), \sigma_\mathrm{i} \big\rangle \rightarrow \eta_0\Big).$$

If $\eta_0 = \varepsilon_0$, i.e., an RTS exception is thrown during the evaluation of g, then the concrete semantics is $\{\theta_0\}$. If, instead, $\eta_0 = \langle \rho_0, \sigma_0 \rangle$, then the concrete semantics is

$$\{\theta_0\} \cup \big\{\, \theta \in \Theta \bigm| \theta(\epsilon) = (\rho_0 \vdash_\beta N \rightarrow \eta) \ \text{ or } \ \theta(\epsilon) = (\rho_0 \vdash_\beta N \xrightarrow{\infty}) \,\big\},$$

where $N = \big\langle \big(\underline{\mathrm{x}} := \mathrm{main}(\square)\big), \sigma_0 \big\rangle \in \Gamma_\mathrm{s}^\beta$ and $\varnothing \vdash_\varnothing (g; \mathbf{gvar}\, \underline{\mathrm{x}} : \mathrm{integer} = 0) : \beta$.

The concrete semantics for CPM we have just presented, extended as indicated in Section 9, allows us to reason on a number of interesting program safety properties (such as the absence of division-by-zero and other run-time errors) as well as termination and computational complexity. In the next section, we will see how the usually non-computable concrete semantics can be given an abstract counterpart that is amenable to effective computation.

6. ABSTRACT DYNAMIC SEMANTICS

For the specification of the abstract semantics, we mainly follow the approach outlined in the works by Schmidt [Sch95; Sch97; Sch98]. The specification of the abstract semantics requires that appropriate abstract domains are chosen to provide correct approximations for the values that are involved in the concrete computation [CC77a; CC79; CC92a; CC92c]. For the sake of generality and extensibility, we will not target any specific abstraction, but rather consider arbitrary abstract domains that satisfy a limited set of properties that are sufficient to provide the correctness of the overall analysis without compromising its potential precision.

6.1 Abstract Semantic Domains

We adopt the framework proposed in [CC92a, Section 7], where the correspondence between the concrete and the abstract domains is induced from a concrete approximation relation and a concretization function. For the sole purpose of simplifying the presentation, we will consider a particular instance of the framework by assuming a few additional but non-essential domain properties. The resulting construction is adequate for our purposes and still allows for algebraically weak abstract domains, such as the domain of convex polyhedra [CH78].

A concrete domain is modeled as a complete lattice $(C, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$ of semantic properties; as usual, the concrete approximation relation $c_1 \sqsubseteq c_2$ holds if c_1 is a stronger property than c_2 (i.e., c_2 approximates c_1). An abstract domain is modeled as a bounded join-semilattice $(D^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$, so that it has a bottom element $\bot^\sharp$ and the least upper bound $d_1^\sharp \sqcup^\sharp d_2^\sharp$ exists for all $d_1^\sharp, d_2^\sharp \in D^\sharp$. When the abstract domain is also provided with a top element $\top^\sharp \in D^\sharp$, we will write $(D^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp)$. The abstract domain $D^\sharp$ is related to C by a monotonic concretization function $\gamma \colon D^\sharp \to C$: in words, C is approximated by $D^\sharp$ through γ; this approximation is said to be *strict* if γ is a strict function.[7]

In order to compute approximations for specific concrete objects, we assume the existence of a partial abstraction function $\alpha \colon C \rightarrowtail D^\sharp$ such that, for each $c \in C$, if $\alpha(c)$ is defined then $c \sqsubseteq \gamma\bigl(\alpha(c)\bigr)$. In particular, we assume that $\alpha(\bot) = \bot^\sharp$ is always defined; if an abstract top element exists, then $\alpha(\top) = \top^\sharp$ is also defined. When needed or useful, we will require a few additional properties.

Most of the concrete domains used in the concrete semantics construction are obtained as the powerset lattice $\bigl(\wp(D), \subseteq, \varnothing, D, \cap, \cup\bigr)$ of some set of concrete objects D. In such a situation, for each concrete object $d \in D$ and abstract element $d^\sharp \in D^\sharp$ such that the corresponding domains are related by the concretization function $\gamma \colon D^\sharp \to \wp(D)$, we write $d \propto d^\sharp$ and $d \not\propto d^\sharp$ to denote the assertions $d \in \gamma(d^\sharp)$ and $d \notin \gamma(d^\sharp)$, respectively. For a lighter notation, we denote $\sqsubseteq^\sharp$, $\bot^\sharp$, $\top^\sharp$ and $\sqcup^\sharp$ by $\sqsubseteq$, $\bot$, $\top$ and $\sqcup$, respectively. We also overload the symbols $\sqsubseteq$, $\bot$, $\top$, $\sqcup$, γ and α: the context will always make clear which incarnation has to be considered.

The approximations of composite concrete domains are typically obtained by suitably combining the approximations already available for their basic components. For $i = 1$, 2, let D_i be a set of concrete objects and consider the corresponding powerset lattice $\bigl(\wp(D_i), \subseteq, \varnothing, D_i, \cap, \cup\bigr)$; let also $D_i^\sharp$ be an abstract domain related to $\wp(D_i)$ by the concretization function $\gamma_i \colon D_i^\sharp \to \wp(D_i)$.

6.1.1 *Approximation of Cartesian Products.* Values of the Cartesian product $D_1 \times D_2$ can be approximated by elements of the Cartesian product $D_1^\sharp \times D_2^\sharp$. Namely, the component-wise ordered abstract domain $\bigl(D_1^\sharp \times D_2^\sharp, \sqsubseteq, \bot, \sqcup\bigr)$ is related to the concrete powerset lattice $\bigl(\wp(D_1 \times D_2), \subseteq, \varnothing, D_1 \times D_2, \cap, \cup\bigr)$ by the concretization function $\gamma \colon (D_1^\sharp \times D_2^\sharp) \to \wp(D_1 \times D_2)$ defined, for each $(d_1^\sharp, d_2^\sharp) \in D_1^\sharp \times D_2^\sharp$, by

$$\gamma(d_1^\sharp, d_2^\sharp) \stackrel{\mathrm{def}}{=} \bigl\{\, (d_1, d_2) \in D_1 \times D_2 \bigm| d_1 \in \gamma_1(d_1^\sharp), d_2 \in \gamma_2(d_2^\sharp) \,\bigr\}. \tag{77}$$

Hence, $(d_1, d_2) \propto (d_1^\sharp, d_2^\sharp)$ holds if and only if $d_1 \propto d_1^\sharp$ and $d_2 \propto d_2^\sharp$.

If the underlying approximations $D_1^\sharp$ and $D_2^\sharp$ are both strict, then a better approximation scheme can be obtained by adopting the *strict product* (also called

---

[7]Let $f \colon D_1 \times \cdots \times D_n \to D_0$, where $(D_i, \sqsubseteq_i, \bot_i, \sqcup_i)$ is a bounded join-semilattice, for each $i = 0$, $\dots$, n. Then, function f is *strict on the i-th argument* if $d_i = \bot_i$ implies $f(d_1, \dots, d_n) = \bot_0$.

smash product) construction, which performs a simple form of reduction by collapsing $(d_1^\sharp, d_2^\sharp)$ to the bottom element whenever $d_1^\sharp = \bot$ or $d_2^\sharp = \bot$. Namely,

$$D_1^\sharp \otimes D_2^\sharp \stackrel{\mathrm{def}}{=} \bigl\{\, (d_1^\sharp, d_2^\sharp) \in D_1^\sharp \times D_2^\sharp \bigm| d_1^\sharp = \bot \text{ if and only if } d_2^\sharp = \bot \,\bigr\}.$$

The concretization function is defined exactly as in (77). The constructor function $\cdot \otimes \cdot \colon (D_1^\sharp \times D_2^\sharp) \to (D_1^\sharp \otimes D_2^\sharp)$ is defined by

$$d_1^\sharp \otimes d_2^\sharp \stackrel{\mathrm{def}}{=} \begin{cases} (d_1^\sharp, d_2^\sharp), & \text{if } d_1^\sharp \neq \bot \text{ and } d_2^\sharp \neq \bot; \\ \bot, & \text{otherwise.} \end{cases}$$

6.1.2 *Approximation of Disjoint Unions.* In order to provide an abstract domain approximating sets of concrete objects drawn from a disjoint union, we use the following well-known construction several times.

Suppose that $D_1 \cap D_2 = \varnothing$. Then, values of the disjoint union $D = D_1 \uplus D_2$ can be approximated by elements of the Cartesian product $D^\sharp = D_1^\sharp \times D_2^\sharp$. In this case, the abstract domain $D^\sharp$ is related to the concrete powerset lattice $\bigl(\wp(D), \subseteq, \varnothing, D, \cap, \cup\bigr)$ by means of the concretization function $\gamma \colon (D_1^\sharp \times D_2^\sharp) \to \wp(D_1 \uplus D_2)$ defined, for each $(d_1^\sharp, d_2^\sharp) \in D_1^\sharp \times D_2^\sharp$, by

$$\gamma(d_1^\sharp, d_2^\sharp) \stackrel{\mathrm{def}}{=} \gamma_1(d_1^\sharp) \uplus \gamma_2(d_2^\sharp).$$

Therefore, the approximation provided by $D^\sharp$ is strict if both $D_1^\sharp$ and $D_2^\sharp$ are so. In order to simplify notation, if $d_1^\sharp \in D_1^\sharp$ then we will sometimes write $d_1^\sharp$ to also denote the abstract element $(d_1^\sharp, \bot) \in D^\sharp$; similarly, $d_2^\sharp \in D_2^\sharp$ also denotes the abstract element $(\bot, d_2^\sharp) \in D^\sharp$. As usual, for each $i = 1$, 2 and $d_i \in D_i$, the notation $d_i \propto (d_1^\sharp, d_2^\sharp)$ stands for the assertion $d_i \in \gamma(d_1^\sharp, d_2^\sharp)$, which is equivalent to $d_i \in \gamma_i(d_i^\sharp)$. For the sake of clarity, the abstract domain $D^\sharp$ as specified above will be denoted by $D_1^\sharp \uplus^\sharp D_2^\sharp$. It is worth stressing that $D_1^\sharp \uplus^\sharp D_2^\sharp \neq D_1^\sharp \uplus D_2^\sharp$.

6.2 Approximation of Integers

The concrete domain of integers $\bigl(\wp(\mathrm{Integer}), \subseteq, \varnothing, \mathrm{Integer}, \cap, \cup\bigr)$ is correctly approximated by an abstract domain $\bigl(\mathrm{Integer}^\sharp, \sqsubseteq, \bot, \top, \sqcup\bigr)$, where we assume that γ is strict. Elements of $\mathrm{Integer}^\sharp$ are denoted by $m^\sharp$, $m_0^\sharp$, $m_1^\sharp$ and so forth. We assume that the partial abstraction function $\alpha \colon \wp(\mathrm{Integer}) \rightarrowtail \mathrm{Integer}^\sharp$ is defined on all singletons $\{m\} \in \wp(\mathrm{Integer})$. We also assume that there are abstract binary operations '$\oplus$', '$\ominus$', '$\odot$', '$\oslash$' and '$\varobar$' on $\mathrm{Integer}^\sharp$ that are strict on each argument and sound with respect to the corresponding operations on $\wp(\mathrm{Integer})$ which, in turn, are the obvious pointwise extensions of addition, subtraction, multiplication, division and remainder over the integers. More formally, we require $\gamma(m_0^\sharp \oplus m_1^\sharp) \supseteq \bigl\{\, m_0 + m_1 \bigm| m_0 \in \gamma(m_0^\sharp), m_1 \in \gamma(m_1^\sharp) \,\bigr\}$ for each $m_0^\sharp, m_1^\sharp \in \mathrm{Integer}^\sharp$, to ensure that '$\oplus$' is sound with respect to addition. Likewise for '$\ominus$' and '$\odot$' with respect to subtraction and multiplication, respectively. For the '$\oslash$' operation we require soundness with respect to integer division i.e., that, for each $m_0^\sharp, m_1^\sharp \in \mathrm{Integer}^\sharp$, $\gamma(m_0^\sharp \oslash m_1^\sharp) \supseteq \bigl\{\, m_0 \div m_1 \bigm| m_0 \in \gamma(m_0^\sharp), m_1 \in \gamma(m_1^\sharp), m_1 \neq 0 \,\bigr\}$. Likewise for '$\varobar$' with respect to the 'mod' operation. We also assume there is a unary abstract operation, denoted by '$\ominus$', which is strict and sound with respect to the unary minus concrete operation, that is, $\gamma(\ominus m^\sharp) \supseteq \bigl\{\, -m \bigm| m \in \gamma(m^\sharp) \,\bigr\}$.

6.3 Approximation of Booleans

We assume a complete lattice $\bigl(\mathrm{Bool}^\sharp, \sqsubseteq, \bot, \top, \sqcap, \sqcup\bigr)$ is given that is related to the concrete domain of Booleans $\bigl(\wp(\mathrm{Bool}), \subseteq, \varnothing, \mathrm{Bool}, \cap, \cup\bigr)$ by means of a Galois connection where γ is strict. Elements of $\mathrm{Bool}^\sharp$ are denoted by $t^\sharp$, $t_0^\sharp$, $t_1^\sharp$ and so forth. We assume that there are abstract operations '$\ominus$', '$\oslash$' and '$\oslash$' on $\mathrm{Bool}^\sharp$ that are strict on each argument and sound with respect to the pointwise extensions of Boolean negation, disjunction and conjunction over $\wp(\mathrm{Bool})$. For instance, for the operation '$\oslash$' to be sound with respect to disjunction on $\wp(\mathrm{Bool})$, it is required that, $\gamma(t_0^\sharp \oslash t_1^\sharp) \supseteq \bigl\{\, t_0 \vee t_1 \bigm| t_0 \in \gamma(t_0^\sharp), t_1 \in \gamma(t_1^\sharp) \,\bigr\}$ for each $t_0^\sharp$ and $t_1^\sharp$ in $\mathrm{Bool}^\sharp$. Likewise for '$\oslash$'. For operation '$\ominus$' to be sound with respect to negation on $\wp(\mathrm{Bool})$, we require that, for each $t^\sharp$ in $\mathrm{Bool}^\sharp$, $\gamma(\ominus\, t^\sharp) \supseteq \bigl\{\, \neg\, t \bigm| t \in \gamma(t^\sharp) \,\bigr\}$.

Furthermore, we assume that there are abstract operations '$\triangleq$', '$\not\triangleq$', '$\lhd$', '$\unlhd$', '$\unrhd$' and '$\rhd$' on $\mathrm{Integer}^\sharp$ that are strict on each argument and sound with respect to the pointwise extensions over $\wp(\mathrm{Integer})$ of the corresponding relational operators '$=$', '$\neq$', '$<$', '$\leq$', '$\geq$' and '$>$' over the integers, considered as functions taking values in Bool. For instance, for the operation '$\triangleq$' to be sound with respect to equality on $\wp(\mathrm{Integer})$, we require that $\gamma(m_0^\sharp \triangleq m_1^\sharp) \supseteq \bigl\{\, m_0 = m_1 \bigm| m_0 \in \gamma(m_0^\sharp), m_1 \in \gamma(m_1^\sharp) \,\bigr\}$ for each $m_0^\sharp, m_1^\sharp \in \mathrm{Integer}^\sharp$. Likewise for '$\not\triangleq$', '$\lhd$', '$\unlhd$', '$\unrhd$' and '$\rhd$'.

6.4 Approximation of Storable Values

The concrete domain of storable values $\bigl(\wp(\mathrm{sVal}), \subseteq, \varnothing, \mathrm{sVal}, \cap, \cup\bigr)$, including both integers and Booleans, is abstracted by the domain $\mathrm{sVal}^\sharp \stackrel{\mathrm{def}}{=} \mathrm{Integer}^\sharp \uplus^\sharp \mathrm{Bool}^\sharp$. The hypotheses on $\mathrm{Integer}^\sharp$ and $\mathrm{Bool}^\sharp$ imply that the approximation is strict.

6.5 Approximation of Exceptions

For the approximation of RTS exceptions, we assume that there is an abstract domain $\bigl(\mathrm{RTSExcept}^\sharp, \sqsubseteq, \bot, \top, \sqcup\bigr)$, which is related to the concrete powerset domain $\bigl(\wp(\mathrm{RTSExcept}), \subseteq, \varnothing, \mathrm{RTSExcept}, \cap, \cup\bigr)$ by a strict concretization function. The partial abstraction function $\alpha \colon \wp(\mathrm{RTSExcept}) \rightarrowtail \mathrm{RTSExcept}^\sharp$ is assumed to be defined on all singletons. Elements of $\mathrm{RTSExcept}^\sharp$ are denoted by $\chi^\sharp$, $\chi_0^\sharp$, $\chi_1^\sharp$ and so forth.

Generic exceptions, including both RTS exceptions and user-defined exceptions, are approximated by elements of the domain $\mathrm{Except}^\sharp \stackrel{\mathrm{def}}{=} \mathrm{RTSExcept}^\sharp \uplus^\sharp \mathrm{sVal}^\sharp$. The hypotheses on its components imply that the approximation is strict. Elements of $\mathrm{Except}^\sharp$ are denoted by $\xi^\sharp$, $\xi_0^\sharp$, $\xi_1^\sharp$ and so forth.

6.6 Approximation of Memory Structures, Value States and Exception States

Here we differ from other published abstract semantics in that we explicitly cater for *relational* abstract domains as well as for *attribute-independent* ones [CC79]. While this complicates the presentation, it results in a truly generic abstract semantics. Moreover, the approach presented here is —all things considered— quite simple and reflects into a modular, clean design of the analyzer.

DEFINITION 6.1. **($\mathrm{Mem}^\sharp$, $\mathrm{ValState}^\sharp$, $\mathrm{ExceptState}^\sharp$.)** *We assume there exists an abstract domain* $\bigl(\mathrm{Mem}^\sharp, \sqsubseteq, \bot, \sqcup\bigr)$ *that is related, by means of a strict concretization function, to the concrete powerset domain* $\bigl(\wp(\mathrm{Mem}), \subseteq, \varnothing, \mathrm{Mem}, \cap, \cup\bigr)$. *Elements of*

$\mathrm{Mem}^\sharp$ *are denoted by* $\sigma^\sharp$, $\sigma^\sharp_0$, $\sigma^\sharp_1$ *and so forth. We assume that, for each* $\sigma \in \mathrm{Mem}$, *there exists* $\sigma^\sharp \in \mathrm{Mem}^\sharp$ *such that* $\sigma \propto \sigma^\sharp$.

The abstract domain of value states is $\mathrm{ValState}^\sharp \stackrel{\mathrm{def}}{=} \mathrm{sVal}^\sharp \otimes \mathrm{Mem}^\sharp$. *Elements of* $\mathrm{ValState}^\sharp$ *will be denoted by* $\upsilon^\sharp$, $\upsilon^\sharp_0$, $\upsilon^\sharp_1$ *and so forth.*

The abstract domain of exception states is $\mathrm{ExceptState}^\sharp \stackrel{\mathrm{def}}{=} \mathrm{Mem}^\sharp \otimes \mathrm{Except}^\sharp$. *Elements of* $\mathrm{ExceptState}^\sharp$ *will be denoted by* $\varepsilon^\sharp$, $\varepsilon^\sharp_0$, $\varepsilon^\sharp_1$ *and so forth. To improve readability,* $\mathtt{none}^\sharp$ *will denote the bottom element* $\bot \in \mathrm{ExceptState}^\sharp$, *indicating that no exception is possible.*

The abstract memory structure read and update operators

$$\begin{aligned}
&\cdot[\cdot,\cdot] \colon (\mathrm{Mem}^\sharp \times \mathrm{Addr} \times \mathrm{sType}) \to (\mathrm{ValState}^\sharp \uplus^\sharp \mathrm{ExceptState}^\sharp),\\
&\cdot[\cdot :=^\sharp \cdot] \colon \bigl(\mathrm{Mem}^\sharp \times (\mathrm{Addr} \times \mathrm{sType}) \times \mathrm{sVal}^\sharp\bigr) \to (\mathrm{Mem}^\sharp \uplus^\sharp \mathrm{ExceptState}^\sharp)
\end{aligned}$$

are assumed to be such that, for each $\sigma^\sharp \in \mathrm{Mem}^\sharp$, $a \in \mathrm{Addr}$, $\mathrm{sT} \in \mathrm{sType}$ *and* $\mathrm{sval}^\sharp \in \mathrm{sVal}^\sharp$*:*

$$\begin{aligned}
\gamma\bigl(\sigma^\sharp[a,\mathrm{sT}]\bigr) &\supseteq \bigl\{\, \sigma[a,\mathrm{sT}] \bigm| \sigma \in \gamma(\sigma^\sharp) \,\bigr\},\\
\gamma\bigl(\sigma^\sharp\bigl[(a,\mathrm{sT}) :=^\sharp \mathrm{sval}^\sharp\bigr]\bigr) &\supseteq \bigl\{\, \sigma\bigl[(a,\mathrm{sT}) := \mathrm{sval}\bigr] \bigm| \sigma \in \gamma(\sigma^\sharp), \mathrm{sval} \in \gamma(\mathrm{sval}^\sharp) \,\bigr\}.
\end{aligned}$$

The abstract data and stack memory allocation functions

$$\begin{aligned}
&\mathrm{new_d}^\sharp \colon \mathrm{ValState}^\sharp \to \bigl((\mathrm{Mem}^\sharp \times \mathrm{Loc}) \uplus^\sharp \mathrm{ExceptState}^\sharp\bigr),\\
&\mathrm{new_s}^\sharp \colon \mathrm{ValState}^\sharp \to \bigl((\mathrm{Mem}^\sharp \times \mathrm{Ind}) \uplus^\sharp \mathrm{ExceptState}^\sharp\bigr)
\end{aligned}$$

are assumed to be such that, for each $\upsilon \in \mathrm{ValState}$ *and* $\upsilon^\sharp \in \mathrm{ValState}^\sharp$ *such that* $\upsilon \in \gamma(\upsilon^\sharp)$, *and each* $h \in \{\mathrm{d},\mathrm{s}\}$*: if* $\mathrm{new}_h(\upsilon) = (\sigma, a)$ *(resp.,* $\mathrm{new}_h(\upsilon) = \varepsilon$*) and* $\mathrm{new}_h{}^\sharp(\upsilon^\sharp) = \bigl((\sigma^\sharp, a'), \varepsilon^\sharp\bigr)$, *then* $\sigma \in \gamma(\sigma^\sharp)$ *and* $a = a'$ *(resp.,* $\varepsilon \in \gamma(\varepsilon^\sharp)$*).*

The abstract memory structure data cleanup function

$$\mathrm{cleanup_d}^\sharp \colon \mathrm{ExceptState}^\sharp \to \mathrm{ExceptState}^\sharp$$

is such that, for each $\varepsilon^\sharp \in \mathrm{ExceptState}^\sharp$, *we have*

$$\gamma\bigl(\mathrm{cleanup_d}^\sharp(\varepsilon^\sharp)\bigr) \supseteq \bigl\{\, \mathrm{cleanup_d}(\varepsilon) \bigm| \varepsilon \in \gamma(\varepsilon^\sharp) \,\bigr\}.$$

The abstract functions

$$\{\mathrm{mark}^\sharp_\mathrm{s}, \mathrm{unmark}^\sharp_\mathrm{s}, \mathrm{link}^\sharp_\mathrm{s}, \mathrm{unlink}^\sharp_\mathrm{s}\} \subseteq \mathrm{Mem}^\sharp \to \mathrm{Mem}^\sharp$$

are defined to be such that, for each $\sigma^\sharp \in \mathrm{Mem}^\sharp$*:*

$$\begin{aligned}
\gamma\bigl(\mathrm{mark}^\sharp_\mathrm{s}(\sigma^\sharp)\bigr) &\supseteq \bigl\{\, \mathrm{mark_s}(\sigma) \bigm| \sigma \in \gamma(\sigma^\sharp) \,\bigr\},\\
\gamma\bigl(\mathrm{unmark}^\sharp_\mathrm{s}(\sigma^\sharp)\bigr) &\supseteq \bigl\{\, \mathrm{unmark_s}(\sigma) \bigm| \sigma \in \gamma(\sigma^\sharp) \text{ and } \mathrm{unmark_s}(\sigma) \text{ is defined} \bigr\},\\
\gamma\bigl(\mathrm{link}^\sharp_\mathrm{s}(\sigma^\sharp)\bigr) &\supseteq \bigl\{\, \mathrm{link_s}(\sigma) \bigm| \sigma \in \gamma(\sigma^\sharp) \text{ and } \mathrm{link_s}(\sigma) \text{ is defined} \bigr\},\\
\gamma\bigl(\mathrm{unlink}^\sharp_\mathrm{s}(\sigma^\sharp)\bigr) &\supseteq \bigl\{\, \mathrm{unlink_s}(\sigma) \bigm| \sigma \in \gamma(\sigma^\sharp) \text{ and } \mathrm{unlink_s}(\sigma) \text{ is defined} \bigr\}.
\end{aligned}$$

It is assumed that all the abstract operators mentioned above are strict on each of their arguments taken from an abstract domain.

As done in the concrete, the abstract stack unmark and the abstract frame unlink functions are lifted to also work on abstract exception states. Namely, for each

$\varepsilon^\sharp = (\sigma^\sharp, \xi^\sharp) \in \mathrm{ExceptState}^\sharp$,

$$\mathrm{unmark}^\sharp_{\mathrm{s}}(\sigma^\sharp, \xi^\sharp) \stackrel{\mathrm{def}}{=} \bigl(\mathrm{unmark}^\sharp_{\mathrm{s}}(\sigma^\sharp), \xi^\sharp\bigr),$$
$$\mathrm{unlink}^\sharp_{\mathrm{s}}(\sigma^\sharp, \xi^\sharp) \stackrel{\mathrm{def}}{=} \bigl(\mathrm{unlink}^\sharp_{\mathrm{s}}(\sigma^\sharp), \xi^\sharp\bigr).$$

Besides the abstract operators specified above, which closely mimic the concrete operators related to concrete memory structures and exception states, other abstract operators will be used in the abstract semantics construction so as to enhance its precision.

When dealing with Boolean guards during the abstract evaluation of conditional and iteration statements, it might be the case that no definite information is available. In such a situation, the abstract execution can be made more precise if the abstract memory structure is *filtered* according to the condition holding in the considered computation branch.

Definition 6.2. **(Memory structure filter.)** *An* abstract memory structure filter *is any computable function* $\phi\colon (\mathrm{Env} \times \mathrm{Mem}^\sharp \times \mathrm{Exp}) \to \mathrm{Mem}^\sharp$ *such that, for each* $e \in \mathrm{Exp}$*, each* $\beta : I$ *with* $\mathrm{FI}(e) \subseteq I$ *and* $\beta \vdash_I e : \mathrm{boolean}$*, for each* $\rho \in \mathrm{Env}$ *with* $\rho : \beta$ *and each* $\sigma^\sharp \in \mathrm{Mem}^\sharp$*, if* $\phi(\rho, \sigma^\sharp, e) = \sigma^\sharp_{\mathrm{tt}}$*, then*

$$\gamma(\sigma^\sharp_{\mathrm{tt}}) \supseteq \bigl\{\, \sigma_{\mathrm{tt}} \in \mathrm{Mem} \bigm| \sigma \in \gamma(\sigma^\sharp), \rho \vdash_\beta \langle e, \sigma\rangle \to \langle \mathrm{tt}, \sigma_{\mathrm{tt}}\rangle \,\bigr\}.$$

Similarly, abstract exception states can be filtered according to whether or not they can be caught by the guard of a catch clause.

Definition 6.3. **(Exception state filters and selectors.)** *The* abstract exception state filters *are computable functions*

$$\phi^+, \phi^- : (\mathrm{exceptDecl} \times \mathrm{ExceptState}^\sharp) \to \mathrm{ExceptState}^\sharp$$

such that, for each $p \in \mathrm{exceptDecl}$ *and each* $\varepsilon^\sharp \in \mathrm{ExceptState}^\sharp$*,*

$$\gamma\bigl(\phi^+(p, \varepsilon^\sharp)\bigr) \supseteq \begin{cases} \gamma(\varepsilon^\sharp), & \text{if } p = \mathbf{any}; \\ \bigl\{\, (\sigma, \xi) \in \gamma(\varepsilon^\sharp) \bigm| \xi = p \,\bigr\}, & \text{if } p \in \mathrm{RTSExcept}; \\ \bigl\{\, (\sigma, \xi) \in \gamma(\varepsilon^\sharp) \bigm| \xi \in \mathrm{dom}\bigl(\mathrm{type}(p)\bigr) \,\bigr\}, & \text{otherwise}; \end{cases}$$

$$\gamma\bigl(\phi^-(p, \varepsilon^\sharp)\bigr) \supseteq \begin{cases} \varnothing, & \text{if } p = \mathbf{any}; \\ \bigl\{\, (\sigma, \xi) \in \gamma(\varepsilon^\sharp) \bigm| \xi \neq p \,\bigr\}, & \text{if } p \in \mathrm{RTSExcept}; \\ \bigl\{\, (\sigma, \xi) \in \gamma(\varepsilon^\sharp) \bigm| \xi \notin \mathrm{dom}\bigl(\mathrm{type}(p)\bigr) \,\bigr\}, & \text{otherwise}. \end{cases}$$

The abstract memory structure and abstract exception selectors

$$\mathrm{mem}\colon \mathrm{ExceptState}^\sharp \to \mathrm{Mem}^\sharp,$$
$$\mathrm{sel}\colon (\mathrm{cType} \times \mathrm{ExceptState}^\sharp) \to (\mathrm{RTSExcept}^\sharp \uplus \mathrm{Integer}^\sharp \uplus \mathrm{Bool}^\sharp)$$

are defined, for each $\varepsilon^\sharp = \bigl(\sigma^\sharp, \bigl(\chi^\sharp, (m^\sharp, t^\sharp)\bigr)\bigr) \in \mathrm{ExceptState}^\sharp$ *and* $\mathrm{cT} \in \mathrm{cType}$*, by*

$$\mathrm{mem}(\varepsilon^\sharp) \stackrel{\mathrm{def}}{=} \sigma^\sharp;$$

$$\mathrm{sel}(\mathrm{cT}, \varepsilon^\sharp) \stackrel{\mathrm{def}}{=} \begin{cases} \chi^\sharp, & \text{if } \mathrm{cT} = \mathrm{rts\_exception}; \\ m^\sharp, & \text{if } \mathrm{cT} = \mathrm{integer}; \\ t^\sharp, & \text{if } \mathrm{cT} = \mathrm{boolean}. \end{cases}$$

To simplify notation, we will write $\mathrm{cT}(\varepsilon^\sharp)$ *to denote* $\mathrm{sel}(\mathrm{cT}, \varepsilon^\sharp)$.

The generic specification provided above for abstract memory structures and the corresponding abstract operators plays a central role for the modularity of the overall construction. By exploiting this "black box" approach, we achieve orthogonality not only from the specific abstract domains used to approximate (sets of tuples of) storable values, but also from the critical design decisions that have to be taken when approximating the concrete stack, which may be unbounded in size due to recursive functions. Hence, while still staying in the boundaries of the current framework, we can flexibly explore, combine, and finely tune the sophisticated proposals that have been put forward in the literature, such as the work in [JS03; JS04], which encompasses both the functional and the call string approaches to interprocedural analysis [CC77b; SP81].

6.7 Abstract Configurations

Terminal and non-terminal configurations of the abstract transition system are now defined.

Definition 6.4. **(Non-terminal abstract configurations.)** *The sets of* non-terminal abstract configurations *for expressions, local and global declarations, statements, function bodies and catch clauses are given, for each* $\beta \in \mathrm{TEnv}_I$ *and respectively, by*

$$\Gamma_{\mathrm{e}}^{\beta\sharp} \stackrel{\mathrm{def}}{=} \bigl\{ \langle e, \sigma^\sharp \rangle \in \mathrm{Exp} \times \mathrm{Mem}^\sharp \bigm| \exists \mathrm{sT} \in \mathrm{sType} \,.\, \beta \vdash_I e : \mathrm{sT} \bigr\},$$
$$\Gamma_{\mathrm{d}}^{\beta\sharp} \stackrel{\mathrm{def}}{=} \bigl\{ \langle d, \sigma^\sharp \rangle \in \mathrm{Decl} \times \mathrm{Mem}^\sharp \bigm| \exists \delta \in \mathrm{TEnv} \,.\, \beta \vdash_I d : \delta \bigr\},$$
$$\Gamma_{\mathrm{g}}^{\beta\sharp} \stackrel{\mathrm{def}}{=} \bigl\{ \langle g, \sigma^\sharp \rangle \in \mathrm{Glob} \times \mathrm{Mem}^\sharp \bigm| \exists \delta \in \mathrm{TEnv} \,.\, \beta \vdash_I g : \delta \bigr\},$$
$$\Gamma_{\mathrm{s}}^{\beta\sharp} \stackrel{\mathrm{def}}{=} \bigl\{ \langle s, \sigma^\sharp \rangle \in \mathrm{Stmt} \times \mathrm{Mem}^\sharp \bigm| \beta \vdash_I s \bigr\},$$
$$\Gamma_{\mathrm{b}}^{\beta\sharp} \stackrel{\mathrm{def}}{=} \bigl\{ \langle \mathrm{body}, \sigma^\sharp \rangle \in \mathrm{Body} \times \mathrm{Mem}^\sharp \bigm| \exists \mathrm{sT} \in \mathrm{sType} \,.\, \beta \vdash_I \mathrm{body} : \mathrm{sT} \bigr\},$$
$$\Gamma_{\mathrm{k}}^{\beta\sharp} \stackrel{\mathrm{def}}{=} \bigl\{ \langle k, \varepsilon^\sharp \rangle \in \mathrm{Catch} \times \mathrm{ExceptState}^\sharp \bigm| \beta \vdash_I k \bigr\}.$$

We write $N^\sharp$ *to denote a non-terminal abstract configuration.*

The approximation relation between concrete and abstract non-terminal configurations is defined as follows. For each $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}\}$, $N = \langle q_1, \sigma \rangle \in \Gamma_q^\beta$ *and* $N^\sharp = \langle q_2, \sigma^\sharp \rangle \in \Gamma_q^{\beta\sharp}$,

$$N \propto N^\sharp \iff (q_1 = q_2 \land \sigma \propto \sigma^\sharp). \tag{78}$$

For each $N = \langle k_1, \varepsilon \rangle \in \Gamma_{\mathrm{k}}^\beta$ *and* $N^\sharp = \langle k_2, \varepsilon^\sharp \rangle \in \Gamma_{\mathrm{k}}^{\beta\sharp}$,

$$N \propto N^\sharp \iff (k_1 = k_2 \land \varepsilon \propto \varepsilon^\sharp). \tag{79}$$

Definition 6.5. **(Terminal abstract configurations.)** *The sets of* terminal abstract configurations *for expressions, local and global declarations, statements, function bodies and catch clauses are given, respectively, by*

$$T_{\mathrm{e}}^\sharp \stackrel{\mathrm{def}}{=} \mathrm{ValState}^\sharp \uplus^\sharp \mathrm{ExceptState}^\sharp,$$
$$T_{\mathrm{d}}^\sharp \stackrel{\mathrm{def}}{=} T_{\mathrm{g}}^\sharp \stackrel{\mathrm{def}}{=} (\mathrm{Env} \times \mathrm{Mem}^\sharp) \uplus^\sharp \mathrm{ExceptState}^\sharp,$$

$$T^\sharp_{\mathrm{s}} \stackrel{\mathrm{def}}{=} T^\sharp_{\mathrm{b}} \stackrel{\mathrm{def}}{=} \mathrm{Mem}^\sharp \uplus^\sharp \mathrm{ExceptState}^\sharp,$$
$$T^\sharp_{\mathrm{k}} \stackrel{\mathrm{def}}{=} T^\sharp_{\mathrm{s}} \uplus^\sharp \mathrm{ExceptState}^\sharp.$$

We write $\eta^\sharp$ to denote a terminal abstract configuration.

The approximation relation $\eta \propto \eta^\sharp$ between concrete and abstract terminal configurations is defined as follows. For expressions,

$$\eta \propto \langle \upsilon^\sharp, \varepsilon^\sharp \rangle \iff \begin{cases} \upsilon \propto \upsilon^\sharp, & \text{if } \eta = \upsilon; \\ \varepsilon \propto \varepsilon^\sharp, & \text{if } \eta = \varepsilon. \end{cases} \tag{80}$$

For local and global declarations,

$$\eta \propto \big\langle (\rho_2, \sigma^\sharp), \varepsilon^\sharp \big\rangle \iff \begin{cases} (\rho_1 = \rho_2 \wedge \sigma \propto \sigma^\sharp), & \text{if } \eta = \langle \rho_1, \sigma \rangle; \\ \varepsilon \propto \varepsilon^\sharp, & \text{if } \eta = \varepsilon. \end{cases} \tag{81}$$

For statements and function bodies,

$$\eta \propto \langle \sigma^\sharp, \varepsilon^\sharp \rangle \iff \begin{cases} \sigma \propto \sigma^\sharp, & \text{if } \eta = \sigma; \\ \varepsilon \propto \varepsilon^\sharp, & \text{if } \eta = \varepsilon. \end{cases} \tag{82}$$

For catch sequences,

$$\eta \propto \langle \eta^\sharp_{\mathrm{s}}, \varepsilon^\sharp \rangle \iff \begin{cases} \eta_{\mathrm{s}} \propto \eta^\sharp_{\mathrm{s}}, & \text{if } \eta = \langle \mathtt{caught}, \eta_{\mathrm{s}} \rangle; \\ \varepsilon \propto \varepsilon^\sharp, & \text{if } \eta = \langle \mathtt{uncaught}, \varepsilon \rangle. \end{cases} \tag{83}$$

The approximation relation for sequents is trivially obtained from the approximation relations defined above for configurations.

DEFINITION 6.6. **('$\propto$' on sequents.)** *The approximation relation between concrete (positive and negative) sequents and abstract sequents is defined, for each $\beta \in \mathrm{TEnv}_I$, for each $\rho_0, \rho_1 \in \mathrm{Env}_J$ such that $\rho_0 : \beta|_J$ and $\rho_1 : \beta|_J$, for each $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$, $N \in \Gamma^\beta_q$, $\eta \in T_q$, $N^\sharp \in \Gamma^{\beta\sharp}_q$ and $\eta^\sharp \in T^\sharp_q$, by*

$$(\rho_0 \vdash_\beta N \to \eta) \propto (\rho_1 \vdash_\beta N^\sharp \to \eta^\sharp) \iff (\rho_0 = \rho_1 \wedge N \propto N^\sharp \wedge \eta \propto \eta^\sharp); \tag{84}$$
$$(\rho_0 \vdash_\beta N \xrightarrow{\infty}) \propto (\rho_1 \vdash_\beta N^\sharp \to \eta^\sharp) \iff (\rho_0 = \rho_1 \wedge N \propto N^\sharp). \tag{85}$$

6.8 Supported Expressions, Declarations and Statements

Each abstract domain has to provide a relation saying which (abstract configuration for) expressions, declarations and statements it directly supports, as well as an abstract evaluation function providing safe approximations of any supported expressions, declarations and statements.

DEFINITION 6.7. **($\mathrm{supported}^\sharp$, $\mathrm{eval}^\sharp$.)** *For each $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}\}$, we assume there exists a computable relation and a partial and computable operation,*

$$\mathrm{supported}^\sharp \subseteq \mathrm{Env} \times \Gamma^{\beta\sharp}_q \qquad \text{and} \qquad \mathrm{eval}^\sharp \colon (\mathrm{Env} \times \Gamma^{\beta\sharp}_q) \rightarrowtail T^\sharp_q,$$

such that whenever $\rho : \beta$ and $\mathrm{supported}^\sharp(\rho, N^\sharp)$ holds, $\mathrm{eval}^\sharp(\rho, N^\sharp)$ is defined and has value $\eta^\sharp \in T^\sharp_q$ and, for each $N \in \Gamma^\beta_q$ and each $\eta \in T_q$ such that $N \propto N^\sharp$ and $\rho \vdash_\beta N \to \eta$, we have $\eta \propto \eta^\sharp$.

An appropriate use of 'supported$^\sharp$' and 'eval$^\sharp$' allows the design of the domain of abstract memory structures to be decoupled from the design of the analyzer. In particular, it enables the use of relational as well as non-relational domains. For example, using the domain of convex polyhedra the proper way, one can easily implement a safe evaluation function for (the non-terminal abstract configuration of) any affine expression e. As a consequence, one can specify the support relation so that $\mathrm{supported}^\sharp\bigl(\rho, \langle e, \sigma^\sharp\rangle\bigr)$ holds. Similarly, one can specify $\mathrm{supported}^\sharp\bigl(\rho, \langle \mathrm{id} := e, \sigma^\sharp\rangle\bigr)$ holds for any affine assignment, i.e., an assignment where e is an affine expression. Other implementation choices are possible. For instance, besides supporting affine expressions, the implementer could specify that $\mathrm{supported}^\sharp\bigl(\rho, \langle \mathrm{id}_1 * \mathrm{id}_2, \sigma^\sharp\rangle\bigr)$ holds provided $\rho : I$, $\mathrm{id}_1, \mathrm{id}_2 \in I$ and, for at least one $i \in \{1, 2\}$, $\gamma\bigl(\sigma^\sharp\bigl[\rho(\mathrm{id}_i)\bigr]\bigr) = \{m\}$, for some integer value m. Similarly, the design can impose that $\mathrm{supported}^\sharp\bigl(\rho, \langle \mathrm{id} * \mathrm{id}, \sigma^\sharp\rangle\bigr)$ always holds.

6.9 Abstract Evaluation Relations

The abstract evaluation relations that provide the first part of the specification of the abstract interpreter for CPM are now defined. These relations are of the form

$$\rho \vdash_\beta N^\sharp \to \eta^\sharp,$$

where $\beta \in \mathrm{TEnv}$, $\rho : \beta$ and, for some $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$, $N^\sharp \in \Gamma_q^{\beta\sharp}$ and $\eta^\sharp \in T_q^\sharp$. The definition is again by structural induction from a set of rule schemata. In order to allow for the arbitrary weakening of the abstract descriptions in the conclusion, without having to introduce precondition strengthening and postcondition weakening rules, and to save typing at the same time, we will use the notation

$$\frac{P_0 \cdots P_{\ell-1}}{\rho \vdash_\beta N^\sharp \rightsquigarrow \eta_0^\sharp} \quad \text{(side condition)}$$

to denote

$$\frac{P_0 \cdots P_{\ell-1}}{\rho \vdash_\beta N^\sharp \to \eta^\sharp} \quad \text{(side condition) and } \eta_0^\sharp \sqsubseteq \eta^\sharp$$

where '$\sqsubseteq$' is the natural ordering relation on the appropriate abstract lattice (i.e., one of the $T_q^\sharp$, for $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$.

Recalling the shorthand notation introduced in Section 6.1.2, when an abstract storable value $\mathrm{sval}^\sharp$ is expected and we write an abstract integer $m^\sharp$ or an abstract Boolean $t^\sharp$, then we are actually meaning the abstract storable value $(m^\sharp, \bot)$ or $(\bot, t^\sharp)$, respectively; similarly, when an abstract exception $\xi^\sharp$ is expected and we write an abstract RTS exception $\chi^\sharp$ or an abstract storable value $\mathrm{sval}^\sharp$, then we are actually meaning the abstract exceptions $(\chi^\sharp, \bot)$ or $(\bot, \mathrm{sval}^\sharp)$, respectively.

6.9.1 *Unsupported Expressions.* The following rules for the abstract evaluation of expressions apply only if $\mathrm{supported}^\sharp\bigl(\rho, \langle e, \sigma^\sharp\rangle\bigr)$ does not hold, where e is the expression being evaluated. This side condition will be left implicit in order not to clutter the presentation.

Constant.

$$\frac{}{\rho \vdash_\beta \langle \mathrm{con}, \sigma^\sharp \rangle \rightsquigarrow \big\langle \alpha(\{\mathrm{con}\}) \otimes \sigma^\sharp, \mathtt{none}^\sharp \big\rangle} \tag{86}$$

Identifier.

$$\frac{}{\rho \vdash_\beta \langle \mathrm{id}, \sigma^\sharp \rangle \rightsquigarrow \sigma^\sharp\big[\rho(\mathrm{id})\big]} \tag{87}$$

Unary minus.

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp \rangle \rightarrow \big\langle (m^\sharp, \sigma_0^\sharp), \varepsilon^\sharp \big\rangle}{\rho \vdash_\beta \langle -e, \sigma^\sharp \rangle \rightsquigarrow \big\langle (\ominus\, m^\sharp, \sigma_0^\sharp), \varepsilon^\sharp \big\rangle} \tag{88}$$

Binary arithmetic operations. Let $\boxdot \in \{+, -, *, /, \%\}$ be a syntactic operator and $\circledcirc \in \{\oplus, \ominus, \odot, \oslash, \varobar\}$ denote the corresponding abstract operation. Then the abstract rules for addition, subtraction, multiplication, division and remainder are given by the following schemata:

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp \rangle \rightarrow \big\langle (m_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp \rangle \rightarrow \big\langle (m_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp \big\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma^\sharp \rangle \rightsquigarrow \big\langle (m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \big\rangle} \tag{89}$$

if $\boxdot \notin \{/, \%\}$ or $0 \not\propto m_1^\sharp$.

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp \rangle \rightarrow \big\langle (m_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp \rangle \rightarrow \big\langle (m_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp \big\rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma^\sharp \rangle \rightsquigarrow \big\langle (m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp \big\rangle} \tag{90}$$

if $\boxdot \in \{/, \%\}$, $0 \propto m_1^\sharp$ and $\varepsilon_2^\sharp = \sigma_1^\sharp \otimes \alpha(\{\mathtt{divbyzero}\})$.

Arithmetic tests. Let $\circledast \in \{=, \neq, <, \leq, \geq, >\}$ be an abstract syntax operator and let $\bowtie\colon (\mathrm{Integer}^\sharp \times \mathrm{Integer}^\sharp) \rightarrow \mathrm{Bool}^\sharp$ denote the corresponding abstract test operation in $\{\triangleq, \not\triangleq, \lhd, \unlhd, \unrhd, \rhd\}$. Then the rules for the abstract arithmetic tests are given by

$$\frac{\rho \vdash_\beta \langle e_0, \sigma^\sharp \rangle \rightarrow \big\langle (m_0^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0^\sharp \rangle \rightarrow \big\langle (m_1^\sharp, \sigma_1^\sharp), \varepsilon_1^\sharp \big\rangle}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma^\sharp \rangle \rightsquigarrow \big\langle (m_0^\sharp \bowtie m_1^\sharp, \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \big\rangle} \tag{91}$$

Negation.

$$\frac{\rho \vdash_\beta \langle b, \sigma^\sharp \rangle \rightarrow \big\langle (t^\sharp, \sigma_0^\sharp), \varepsilon^\sharp \big\rangle}{\rho \vdash_\beta \langle \mathbf{not}\ \ b, \sigma^\sharp \rangle \rightsquigarrow \big\langle (\circleddash\, t^\sharp, \sigma_0^\sharp), \varepsilon^\sharp \big\rangle} \tag{92}$$

Conjunction.

$$\frac{\rho \vdash_\beta \langle b_0, \sigma^\sharp \rangle \rightarrow \langle \upsilon_0^\sharp, \varepsilon_0^\sharp \rangle \quad \rho \vdash_\beta \langle b_1, \sigma_{\mathrm{tt}}^\sharp \rangle \rightarrow \langle \upsilon_1^\sharp, \varepsilon_1^\sharp \rangle}{\rho \vdash_\beta \langle b_0\, \mathbf{and}\, b_1, \sigma^\sharp \rangle \rightsquigarrow \big\langle \upsilon_{\mathrm{ff}}^\sharp \sqcup \upsilon_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \big\rangle}, \tag{93}$$

if $\sigma_{\mathrm{tt}}^\sharp = \phi(\rho, \sigma^\sharp, b_0)$, $\sigma_{\mathrm{ff}}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, b_0)$ and $\upsilon_{\mathrm{ff}}^\sharp = \alpha(\{\mathrm{ff}\}) \otimes \sigma_{\mathrm{ff}}^\sharp$.

Disjunction.

$$\frac{\rho \vdash_\beta \langle b_0, \sigma^\sharp \rangle \rightarrow \langle \upsilon_0^\sharp, \varepsilon_0^\sharp \rangle \quad \rho \vdash_\beta \langle b_1, \sigma_{\mathrm{ff}}^\sharp \rangle \rightarrow \langle \upsilon_1^\sharp, \varepsilon_1^\sharp \rangle}{\rho \vdash_\beta \langle b_0\, \mathbf{or}\, b_1, \sigma^\sharp \rangle \rightsquigarrow \langle \upsilon_{\mathrm{tt}}^\sharp \sqcup \upsilon_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \rangle} \tag{94}$$

if $\sigma^\sharp_{\mathrm{tt}} = \phi(\rho, \sigma^\sharp, b_0)$, $\sigma^\sharp_{\mathrm{ff}} = \phi(\rho, \sigma^\sharp, \mathbf{not}\, b_0)$ and $\upsilon^\sharp_{\mathrm{tt}} = \alpha(\{\mathrm{tt}\}) \otimes \sigma^\sharp_{\mathrm{tt}}$.

6.9.2 *Unsupported Declarations.* The following rules only apply if the condition $\mathrm{supported}^\sharp\big(\rho, \langle q, \sigma^\sharp\rangle\big)$ does not hold, where $q \in \mathrm{Decl} \uplus \mathrm{Glob}$ is the declaration being evaluated. Again, this side condition is left implicit.

Nil.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nil}, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\varnothing, \sigma^\sharp), \mathtt{none}^\sharp \big\rangle} \tag{95}$$

Environment.

$$\frac{}{\rho \vdash_\beta \langle \rho_0, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_0, \sigma^\sharp), \mathtt{none}^\sharp \big\rangle} \tag{96}$$

Recursive environment.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{rec}\, \rho_0, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_1, \sigma^\sharp), \mathtt{none}^\sharp \big\rangle} \tag{97}$$

$$\text{if } \rho_1 = \big\{\, \mathrm{id} \mapsto \rho_0(\mathrm{id}) \bigm| \rho_0(\mathrm{id}) = (\lambda\, \mathrm{fps}\, .\, \mathbf{extern}, \mathrm{sT}) \,\big\} \\ \cup \left\{ \mathrm{id} \mapsto \mathrm{abs}_1 \;\middle|\; \begin{array}{l} \forall i \in \{0,1\} : \mathrm{abs}_i = (\lambda\, \mathrm{fps}\, .\, \mathbf{let}\, d_i \ \mathbf{in}\, s\, \mathbf{result}\, e, \mathrm{sT}), \\ \rho_0(\mathrm{id}) = \mathrm{abs}_0, d_1 = \mathbf{rec}\big(\rho_0 \setminus \mathrm{DI}(\mathrm{fps})\big); d_0 \end{array} \right\}.$$

Global variable declaration.

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \rightarrow \langle \upsilon^\sharp, \varepsilon^\sharp_0\rangle}{\rho \vdash_\beta \langle \mathbf{gvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_1, \sigma^\sharp_1), \mathrm{cleanup_d}^\sharp(\varepsilon^\sharp_0 \sqcup \varepsilon^\sharp_1)\big\rangle} \tag{98}$$

if $\mathrm{new_d}^\sharp(\upsilon^\sharp) = \big((\sigma^\sharp_1, l), \varepsilon^\sharp_1\big)$ and $\rho_1 = \big\{\mathrm{id} \mapsto (l, \mathrm{sT})\big\}$.

Local variable declaration.

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \rightarrow \langle \upsilon^\sharp, \varepsilon^\sharp_0\rangle}{\rho \vdash_\beta \langle \mathbf{lvar}\, \mathrm{id} : \mathrm{sT} = e, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_1, \sigma^\sharp_1), \mathrm{unmark}^\sharp_{\mathrm{s}}(\varepsilon^\sharp_0 \sqcup \varepsilon^\sharp_1)\big\rangle} \tag{99}$$

if $\mathrm{new_s}^\sharp(\upsilon^\sharp) = \big((\sigma^\sharp_1, i), \varepsilon^\sharp_1\big)$ and $\rho_1 = \big\{\mathrm{id} \mapsto (i, \mathrm{sT})\big\}$.

Function declaration.

$$\frac{}{\rho \vdash_\beta \big\langle \mathbf{function}\, \mathrm{id}(\mathrm{fps}) : \mathrm{sT} = \mathrm{body}_0, \sigma^\sharp\big\rangle \rightsquigarrow \big\langle (\rho_0, \sigma^\sharp), \mathtt{none}^\sharp \big\rangle} \tag{100}$$

if $\rho_0 = \big\{\mathrm{id} \mapsto (\lambda\, \mathrm{fps}\, .\, \mathrm{body}_1, \mathrm{sT})\big\}$ and either $\mathrm{body}_0 = \mathrm{body}_1 = \mathbf{extern}$ or, for each $i \in \{0,1\}$, $\mathrm{body}_i = \mathbf{let}\, d_i \ \mathbf{in}\, s\, \mathbf{result}\, e$, $I = \mathrm{FI}(\mathrm{body}_0) \setminus \mathrm{DI}(\mathrm{fps})$ and $d_1 = \rho\,|_I; d_0$.

Recursive declaration.

$$\frac{(\rho \setminus J) \vdash_{\beta[\beta_1]} \langle g, \sigma^\sharp\rangle \rightarrow \big\langle (\rho_0, \sigma^\sharp_0), \mathtt{none}^\sharp \big\rangle \quad \rho \vdash_\beta \langle \mathbf{rec}\, \rho_0, \sigma^\sharp_0\rangle \rightarrow \eta^\sharp}{\rho \vdash_\beta \langle \mathbf{rec}\, g, \sigma^\sharp\rangle \rightsquigarrow \eta^\sharp} \tag{101}$$

if $J = \mathrm{FI}(g) \cap \mathrm{DI}(g)$, $\beta \vdash_{\mathrm{FI}(g)} g : \beta_0$ and $\beta_1 = \beta_0\,|_J$.

Global sequential composition.

$$\frac{\rho \vdash_\beta \langle g_0, \sigma^\sharp\rangle \rightarrow \big\langle (\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle g_1, \sigma_0^\sharp\rangle \rightarrow \big\langle (\rho_1, \sigma_1^\sharp), \varepsilon_1^\sharp \big\rangle}{\rho \vdash_\beta \langle g_0; g_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_0[\rho_1], \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \big\rangle} \tag{102}$$

if $\beta \vdash_I g_0 : \beta_0$ and $\mathrm{FI}(g_0) \subseteq I$.

Local sequential composition.

$$\frac{\rho \vdash_\beta \langle d_0, \sigma^\sharp\rangle \rightarrow \big\langle (\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle d_1, \sigma_0^\sharp\rangle \rightarrow \big\langle (\rho_1, \sigma_1^\sharp), \varepsilon_1^\sharp \big\rangle}{\rho \vdash_\beta \langle d_0; d_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle (\rho_0[\rho_1], \sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \big\rangle} \tag{103}$$

if $\beta \vdash_I d_0 : \beta_0$ and $\mathrm{FI}(d_0) \subseteq I$.

6.9.3 *Unsupported Statements.* The following rules only apply if the implicit side condition $\mathrm{supported}^\sharp\big(\rho, \langle s, \sigma^\sharp\rangle\big)$ does not hold, where s is the statement being evaluated.

Nop.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{nop}, \sigma^\sharp\rangle \rightsquigarrow \sigma^\sharp} \tag{104}$$

Assignment.

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \rightarrow \big\langle (\mathrm{sval}^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle}{\rho \vdash_\beta \langle \mathrm{id} := e, \sigma^\sharp\rangle \rightsquigarrow \langle \sigma_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\rangle} \quad \text{if } \sigma_0^\sharp\big[\rho(\mathrm{id}) :=^\sharp \mathrm{sval}^\sharp\big] = (\sigma_1^\sharp, \varepsilon_1^\sharp) \tag{105}$$

Statement sequence.

$$\frac{\rho \vdash_\beta \langle s_0, \sigma^\sharp\rangle \rightarrow \langle \sigma_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0^\sharp\rangle \rightarrow \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle}{\rho \vdash_\beta \langle s_0; s_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle \sigma_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \big\rangle} \tag{106}$$

Block.

$$\frac{\rho \vdash_\beta \big\langle d, \mathrm{mark}_\mathrm{s}^\sharp(\sigma^\sharp)\big\rangle \rightarrow \big\langle (\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle \quad \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0^\sharp\rangle \rightarrow \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle}{\rho \vdash_\beta \langle d; s, \sigma^\sharp\rangle \rightsquigarrow \big\langle \mathrm{unmark}_\mathrm{s}^\sharp(\sigma_1^\sharp), \varepsilon_0^\sharp \sqcup \mathrm{unmark}_\mathrm{s}^\sharp(\varepsilon_1^\sharp)\big\rangle} \tag{107}$$

if $\beta \vdash_{\mathrm{FI}(d)} d : \beta_0$.

Conditional.

$$\frac{\begin{array}{r}\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \rightarrow \langle \upsilon_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle s_0, \sigma_\mathrm{tt}^\sharp\rangle \rightarrow \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle \\ \rho \vdash_\beta \langle s_1, \sigma_\mathrm{ff}^\sharp\rangle \rightarrow \langle \sigma_2^\sharp, \varepsilon_2^\sharp\rangle\end{array}}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \sigma^\sharp\rangle \rightsquigarrow \langle \sigma_1^\sharp \sqcup \sigma_2^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp\rangle} \tag{108}$$

if $\sigma_\mathrm{tt}^\sharp = \phi(\rho, \sigma^\sharp, e)$ and $\sigma_\mathrm{ff}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, e)$.

While.

$$\frac{\begin{array}{r}\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \rightarrow \langle \upsilon_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle s, \sigma_\mathrm{tt}^\sharp\rangle \rightarrow \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle \\ \rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma_1^\sharp\rangle \rightarrow \langle \sigma_2^\sharp, \varepsilon_2^\sharp\rangle\end{array}}{\rho \vdash_\beta \langle \mathbf{while}\, e\, \mathbf{do}\, s, \sigma^\sharp\rangle \rightsquigarrow \langle \sigma_\mathrm{ff}^\sharp \sqcup \sigma_2^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp\rangle} \tag{109}$$

if $\sigma_\mathrm{tt}^\sharp = \phi(\rho, \sigma^\sharp, e)$ and $\sigma_\mathrm{ff}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, e)$.

Throw.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{throw}\,\chi, \sigma^\sharp\rangle \rightsquigarrow \langle \bot, \varepsilon^\sharp\rangle} \quad \text{if } \varepsilon^\sharp = \sigma^\sharp \otimes \alpha(\{\chi\}) \tag{110}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma^\sharp\rangle \to \big\langle (\mathrm{sval}^\sharp, \sigma_0^\sharp), \varepsilon_0^\sharp \big\rangle}{\rho \vdash_\beta \langle \mathbf{throw}\,e, \sigma^\sharp\rangle \rightsquigarrow \langle \bot, \varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp\rangle} \quad \text{if } \varepsilon_1^\sharp = \sigma_0^\sharp \otimes \mathrm{sval}^\sharp \tag{111}$$

Try blocks.

$$\frac{\rho \vdash_\beta \langle s, \sigma^\sharp\rangle \to \langle \sigma_0^\sharp, \varepsilon_0^\sharp\rangle \quad \rho \vdash_\beta \langle k, \varepsilon_0^\sharp\rangle \to \big\langle (\sigma_1^\sharp, \varepsilon_1^\sharp), \varepsilon_2^\sharp \big\rangle}{\rho \vdash_\beta \langle \mathbf{try}\,s\,\mathbf{catch}\,k, \sigma^\sharp\rangle \rightsquigarrow \langle \sigma_0^\sharp \sqcup \sigma_1^\sharp, \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp\rangle} \tag{112}$$

$$\frac{\begin{array}{r}\rho \vdash_\beta \langle s_0, \sigma^\sharp\rangle \to \big\langle \sigma_0^\sharp, (\sigma_1^\sharp, \xi_1^\sharp)\big\rangle \quad \rho \vdash_\beta \langle s_1, \sigma_0^\sharp\rangle \to \langle \sigma_2^\sharp, \varepsilon_2^\sharp\rangle \\ \rho \vdash_\beta \langle s_1, \sigma_1^\sharp\rangle \to \langle \sigma_3^\sharp, \varepsilon_3^\sharp\rangle\end{array}}{\rho \vdash_\beta \langle \mathbf{try}\,s_0\,\mathbf{finally}\,s_1, \sigma^\sharp\rangle \rightsquigarrow \big\langle \sigma_2^\sharp, \varepsilon_2^\sharp \sqcup \varepsilon_3^\sharp \sqcup (\sigma_3^\sharp \otimes \xi_1^\sharp)\big\rangle} \tag{113}$$

Function call. With reference to conditions (60) and (61) of the concrete rules for function calls, the corresponding abstract rule schema is

$$\frac{\begin{array}{l}\rho \vdash_\beta \big\langle d, \mathrm{mark}_\mathrm{s}^\sharp(\sigma^\sharp)\big\rangle \to \big\langle (\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle \\ \rho[\rho_1] \vdash_{\beta[\beta_1]} \big\langle \mathrm{body}, \mathrm{link}_\mathrm{s}^\sharp(\sigma_0^\sharp)\big\rangle \to \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle \\ \rho[\rho_0] \vdash_{\beta[\beta_0]} \big\langle \mathrm{id}_0 := \underline{\mathrm{x}}_0, \mathrm{unlink}_\mathrm{s}^\sharp(\sigma_1^\sharp)\big\rangle \to \langle \sigma_2^\sharp, \varepsilon_2^\sharp\rangle\end{array}}{\rho \vdash_\beta \big\langle \mathrm{id}_0 := \mathrm{id}(e_1, \ldots, e_n), \sigma^\sharp\big\rangle \rightsquigarrow \big\langle \mathrm{unmark}_\mathrm{s}^\sharp(\sigma_2^\sharp), \varepsilon^\sharp\big\rangle} \tag{114}$$

if (60) and (61) hold and $\varepsilon^\sharp = \varepsilon_0^\sharp \sqcup \mathrm{unmark}_\mathrm{s}^\sharp\big(\mathrm{unlink}_\mathrm{s}^\sharp(\varepsilon_1^\sharp)\big) \sqcup \mathrm{unmark}_\mathrm{s}^\sharp(\varepsilon_2^\sharp)$.

6.9.4 *Function Bodies.*

$$\frac{\begin{array}{l}\rho \vdash_\beta \big\langle d, \mathrm{mark}_\mathrm{s}^\sharp(\sigma^\sharp)\big\rangle \to \big\langle (\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp\big\rangle \\ \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle s, \sigma_0^\sharp\rangle \to \langle \sigma_1^\sharp, \varepsilon_1^\sharp\rangle \\ \rho[\rho_0] \vdash_{\beta[\beta_0]} \langle \underline{\mathrm{x}}_0 := e, \sigma_1^\sharp\rangle \to \langle \sigma_2^\sharp, \varepsilon_2^\sharp\rangle\end{array}}{\rho \vdash_\beta \langle \mathbf{let}\,d\,\mathbf{in}\,s\,\mathbf{result}\,e, \sigma^\sharp\rangle \rightsquigarrow \big\langle \mathrm{unmark}_\mathrm{s}^\sharp(\sigma_2^\sharp), \varepsilon_3^\sharp\big\rangle} \tag{115}$$

if $\beta \vdash_{\mathrm{FI}(d)} d : \beta_0$, $\varepsilon_3^\sharp = \varepsilon_0^\sharp \sqcup \mathrm{unmark}_\mathrm{s}^\sharp(\varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp)$.

$$\frac{}{\rho \vdash_\beta \langle \mathbf{extern}, \sigma^\sharp\rangle \rightsquigarrow \big\langle \sigma_0^\sharp, (\sigma_0^\sharp, \top)\big\rangle} \tag{116}$$

if $\forall \sigma, \sigma_0 \in \mathrm{Mem} : \big(\sigma = (\mu, w) \wedge \sigma \propto \sigma^\sharp \wedge \sigma_0 = (\mu_0, w)\big) \implies \sigma_0 \propto \sigma_0^\sharp$.

6.9.5 *Catch Clauses*

Catch.

$$\frac{\rho \vdash_\beta \big\langle s, \mathrm{mem}(\varepsilon_0^\sharp)\big\rangle \to \eta_1^\sharp}{\rho \vdash_\beta \big\langle (p)\,s, \varepsilon^\sharp\big\rangle \rightsquigarrow \langle \eta_1^\sharp, \varepsilon_1^\sharp\rangle} \tag{117}$$

if $p = \mathbf{any}$ or $p = \chi$ or $p = \mathrm{cT}$, $\varepsilon_0^\sharp = \phi^+(p, \varepsilon^\sharp)$ and $\varepsilon_1^\sharp = \phi^-(p, \varepsilon^\sharp)$.

$$\frac{\rho\big[\{\mathrm{id} \mapsto (i, \mathrm{sT})\}\big] \vdash_{\beta[\{\mathrm{id} \mapsto \mathrm{sT\ loc}\}]} \langle s, \sigma_2^\sharp\rangle \rightarrow \langle \sigma_3^\sharp, \varepsilon_3^\sharp\rangle}{\rho \vdash_\beta \big\langle (\mathrm{id} : \mathrm{sT})\, s, \varepsilon^\sharp\big\rangle \rightsquigarrow \big\langle (\sigma_4^\sharp, \varepsilon_4^\sharp), \varepsilon_1^\sharp\big\rangle} \tag{118}$$

if $\varepsilon_0^\sharp = \phi^+(\mathrm{sT}, \varepsilon^\sharp)$, $\varepsilon_1^\sharp = \phi^-(\mathrm{sT}, \varepsilon^\sharp)$, $\mathrm{new_s}^\sharp\Big(\mathrm{sT}(\varepsilon_0^\sharp), \mathrm{mark}_\mathrm{s}^\sharp\big(\mathrm{mem}(\varepsilon_0^\sharp)\big)\Big) = \big((\sigma_2^\sharp, i), \varepsilon_2^\sharp\big)$, $\sigma_4^\sharp = \mathrm{unmark}_\mathrm{s}^\sharp(\sigma_3^\sharp)$ and $\varepsilon_4^\sharp = \mathrm{unmark}_\mathrm{s}^\sharp(\varepsilon_2^\sharp) \sqcup \mathrm{unmark}_\mathrm{s}^\sharp(\varepsilon_3^\sharp)$.

Catch sequence.

$$\frac{\rho \vdash_\beta \langle k_0, \varepsilon^\sharp\rangle \rightarrow \big\langle (\sigma_0^\sharp, \varepsilon_0^\sharp), \varepsilon_1^\sharp\big\rangle \quad \rho \vdash_\beta \langle k_1, \varepsilon_1^\sharp\rangle \rightarrow \big\langle (\sigma_1^\sharp, \varepsilon_2^\sharp), \varepsilon_3^\sharp\big\rangle}{\rho \vdash_\beta \langle k_0; k_1, \varepsilon^\sharp\rangle \rightsquigarrow \big\langle (\sigma_0^\sharp \sqcup \sigma_1^\sharp, \varepsilon_0^\sharp \sqcup \varepsilon_2^\sharp), \varepsilon_3^\sharp\big\rangle} \tag{119}$$

6.9.6 *Supported Expressions, Declarations and Statements.* Let $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}\}$ and $N^\sharp \in \Gamma_q^{\beta\sharp}$. Then, whenever $\mathrm{supported}^\sharp(\rho, N^\sharp)$ holds, alternate versions of the rules above apply. For each of the rules above,

$$\frac{P_0 \quad \cdots \quad P_{\ell-1}}{\rho \vdash_\beta N^\sharp \rightsquigarrow \eta^\sharp} \quad \text{if (side condition) and not } \mathrm{supported}^\sharp(\rho, N^\sharp)$$

we also have the rule

$$\frac{P_0 \quad \cdots \quad P_{\ell-1}}{\rho \vdash_\beta N^\sharp \rightsquigarrow \mathrm{eval}^\sharp(\rho, N^\sharp)} \quad \text{if (side condition) and } \mathrm{supported}^\sharp(\rho, N^\sharp)$$

Notice that even if $\mathrm{eval}^\sharp(\rho, N^\sharp)$ does not depend on the rule antecedents $P_0, \ldots, P_{\ell-1}$, these cannot be omitted, as this would neglect the sub-computations spawned by the unsupported evaluation of $N^\sharp$.

6.10 Abstract Semantics Trees

We now define possibly infinite abstract semantics trees along the lines of what we did in Section 5.7. Notice that the need to consider infinite abstract trees goes beyond the need to observe infinite concrete computations. For instance, there is no finite abstract tree corresponding to a program containing a **while** command, because (109) is the only abstract rule for **while** and it recursively introduces a new **while** node into the tree.

DEFINITION 6.8. **(Abstract semantics rules.)** *The set* $\mathcal{R}^\sharp$ *of* abstract semantics rules *is the infinite set obtained by instantiating the rule schemata of* Section 6.9 *in all possible ways (respecting the side conditions).*

Let $\mathcal{S}^\sharp$ be the (infinite) set of sequents occurring in the premises and conclusions of the rules in $\mathcal{R}^\sharp$. Matching Definition 5.6, the *abstract semantics universe*, denoted by $\mathcal{U}^\sharp$, is the set of finitely branching trees of at most ω-depth with labels in $\mathcal{S}^\sharp$.

DEFINITION 6.9. **(Abstract semantics trees.)** *Let* $\mathcal{F}^\sharp \colon \wp(\mathcal{U}^\sharp) \rightarrow \wp(\mathcal{U}^\sharp)$ *be given, for each* $U^\sharp \in \wp(\mathcal{U}^\sharp)$*, by*

$$\mathcal{F}^\sharp(U^\sharp) \stackrel{\mathrm{def}}{=} \left\{ \frac{\theta_0^\sharp \ \cdots \ \theta_{\ell-1}^\sharp}{s} \;\middle|\; \{\theta_0^\sharp, \ldots, \theta_{\ell-1}^\sharp\} \subseteq U^\sharp,\ \frac{\theta_0^\sharp(\epsilon) \ \cdots \ \theta_{\ell-1}^\sharp(\epsilon)}{s} \in \mathcal{R}^\sharp \right\}.$$

The set of abstract semantics trees *is* $\Theta^\sharp \stackrel{\mathrm{def}}{=} \mathrm{gfp}_\subseteq(\mathcal{F}^\sharp)$.

We now show that, for every non-terminal abstract configuration, there exists an abstract tree with that in the root.

PROPOSITION 6.10. *For each* $\beta \in \mathrm{TEnv}$, $\rho \in \mathrm{Env}$ *such that* $\rho : \beta$ *and* $N^\sharp \in \Gamma_q^{\beta\sharp}$, *where* $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$, *there exists* $\theta^\sharp \in \Theta^\sharp$ *such that,*

$$\theta^\sharp(\epsilon) \in \left\{\, (\rho \vdash_\beta N^\sharp \to \eta^\sharp) \;\middle|\; \eta^\sharp \in T_q^\sharp \,\right\}.$$

PROOF. For the proof, let[8]

$$S_+^\sharp(\rho, \beta, N^\sharp) \stackrel{\text{def}}{=} \left\{\, s^\sharp \;\middle|\; s^\sharp = (\rho \vdash_\beta N^\sharp \to \eta^\sharp), (s \propto s^\sharp \implies s \text{ is well-typed}) \,\right\}.$$

We now assume that $N^\sharp \in \Gamma_q^{\beta\sharp}$ is a fixed but arbitrary non-terminal abstract configuration. Suppose that $\mathrm{supported}^\sharp(\rho, N^\sharp)$ does not hold. By inspecting the abstract evaluation rules given in Section 6.9, it can be seen that there exists $\ell \geq 0$ and a nonempty set of rules $R_0 \in \mathcal{R}^\sharp$ with ℓ premises and a conclusion in $S_+^\sharp(\rho, \beta, N^\sharp)$. If, on the other hand, $\mathrm{supported}^\sharp(\rho, N^\sharp)$ does hold, then it follows from Section 6.9.6 that, by Definition 6.7, $\mathrm{eval}^\sharp(\rho, N^\sharp)$ is defined and, for each rule in R_0, there is a rule with the same set of premises but where the conclusion $\bigl(\rho \vdash_\beta N^\sharp \to \mathrm{eval}^\sharp(\rho, N^\sharp)\bigr)$ is also in $S_+^\sharp(\rho, \beta, N^\sharp)$. Thus, in both cases, by definition of $\mathcal{U}^\sharp$, there exists a tree in $\mathcal{U}^\sharp$ with root in $S_+^\sharp(\rho, \beta, N^\sharp)$.

We prove that, for any $n \in \mathbb{N}$, there exists a tree $\theta^\sharp \in \mathcal{F}^{\sharp n}(\mathcal{U}^\sharp)$ such that $\theta^\sharp(\epsilon) \in S_+^\sharp(\rho, \beta, N^\sharp)$. To this end, we reason by induction on $n \geq 0$. In the case $n = 0$, $\mathcal{U} = \mathcal{F}^{\sharp n}(\mathcal{U}^\sharp)$ so that the hypothesis holds.

We now suppose that $n > 0$. Let $j \in \{0, \dots, \ell\}$ be the maximal value for which there exist trees $\theta_0^\sharp, \dots, \theta_{j-1}^\sharp \in \mathcal{F}^{\sharp(n-1)}(\mathcal{U}^\sharp)$ where $P_0 = \theta_0^\sharp(\epsilon), \dots, P_{j-1} = \theta_{j-1}^\sharp(\epsilon)$ are the first j premises of a rule in R_0; let $R_j \subseteq R_0$ be the set of all rules in R_0 with $P_0, \dots, P_{j-1}$ as their first j premises; then $R_j \neq \varnothing$. We assume that $j < \ell$ and derive a contradiction. By inspecting the rule schemata in Section 6.9, it can be seen that, if there exists $\frac{P_0 \,\cdots\, P_{j-1} \; P'_j \,\cdots}{\acute{s}^\sharp} \in R_j$ for some $P'_j \in S_+^\sharp(\rho_j, \beta_j, N_j^\sharp)$ and $\acute{s}^\sharp \in S_+^\sharp(\rho, \beta, N^\sharp)$, then

$$\forall P_j \in S_+^\sharp(\rho_j, \beta_j, N_j^\sharp) : \exists s^\sharp \in S_+^\sharp(\rho, \beta, N^\sharp) \,.\, \frac{P_0 \;\cdots\; P_{j-1} \; P_j \;\cdots}{s^\sharp} \in R_j. \tag{120}$$

By the inductive hypothesis, there exists $\theta_j^\sharp \in \mathcal{F}^{\sharp(n-1)}(\mathcal{U}^\sharp)$ such that $P_j = \theta_j^\sharp(\epsilon) \in S_+^\sharp(\rho_j, \beta_j, N_j^\sharp)$; hence, by (120), there must be a rule in R_j whose $(j+1)$-th premise is P_j; contradicting the assumption that $j < \ell$ is maximal. Hence $j = \ell$. Thus there exists a rule $\frac{P_0 \,\cdots\, P_{\ell-1}}{s^\sharp} \in R_0$ for some $s^\sharp \in S_+^\sharp(\rho, \beta, N^\sharp)$; hence, by Definition 6.9, the tree $\frac{\theta_0^\sharp \,\cdots\, \theta_{\ell-1}^\sharp}{s^\sharp} \in \mathcal{F}^{\sharp n}(\mathcal{U}^\sharp)$. Therefore since, by Definition 6.9, $\Theta^\sharp = \mathrm{gfp}_\subseteq(\mathcal{F}^\sharp)$, there exists a tree $\theta^\sharp$ in $\Theta^\sharp$ such that $\theta^\sharp(\epsilon) \in S_+^\sharp(\rho, \beta, N^\sharp)$. $\square$

7. CORRECTNESS OF THE ABSTRACT SEMANTICS

In Section 6, we introduced the notion of *sound approximation* for configurations and sequents in terms of the concretization function γ defined for each abstract domain. We now proceed to define the notion of sound approximation for trees.

[8] For the definition of a well-typed sequent, see the proof of Proposition 5.8.

Definition 7.1. (**'$\propto$' for trees.**) *Let $\overline{\propto}\colon \wp(\Theta \times \Theta^\sharp) \to \wp(\Theta \times \Theta^\sharp)$ be given, for each $U \in \wp(\Theta \times \Theta^\sharp)$, by*

$$\overline{\propto}(U) \stackrel{\mathrm{def}}{=} \left\{ (\theta, \theta^\sharp) \in \Theta \times \Theta^\sharp \;\middle|\; \begin{array}{l} \theta(\epsilon) \propto \theta^\sharp(\epsilon), \\ \forall i \in \mathrm{dom}(\theta) \cap \mathbb{N} : \\ \qquad \exists j \in \mathrm{dom}(\theta^\sharp) \cap \mathbb{N} \,.\, \bigl(\theta_{[i]}, \theta^\sharp_{[j]}\bigr) \in U \end{array} \right\}.$$

Then $\theta \propto \theta^\sharp$ if and only if $(\theta, \theta^\sharp) \in \mathrm{gfp}_{\subseteq}(\overline{\propto})$.

In words, $\theta \propto \theta^\sharp$ means that the root of θ is approximated by the root of $\theta^\sharp$ and every immediate subtree of θ is approximated by some immediate subtrees of $\theta^\sharp$. Notice that one immediate subtree in $\theta^\sharp$ may be related by '$\propto$' to none, one or more than one immediate subtree of θ.

The following result states that, for each concrete tree, there is always an abstract tree that is generated from a corresponding non-terminal abstract configuration.

Theorem 7.2. *Let $\theta \in \Theta$ be a concrete tree such that $\theta(\epsilon) = (\rho \vdash_\beta N \to \eta)$ or $\theta(\epsilon) = (\rho \vdash_\beta N \xrightarrow{\infty})$. Then there exists $\theta^\sharp \in \Theta^\sharp$ such that, $\theta^\sharp(\epsilon) = (\rho \vdash_\beta N^\sharp \to \eta^\sharp)$ and $N \propto N^\sharp$.*

Proof. Suppose first that $N = \langle q, \sigma \rangle$ where $q \in \{e, d, g, s, b\}$. By Definition 6.1, we can always find $\sigma^\sharp \in \mathrm{Mem}^\sharp$ such that $\sigma \propto \sigma^\sharp$. Hence, letting $N^\sharp = \langle q, \sigma^\sharp \rangle$, by (78) in Definition 6.4, we obtain $N \propto N^\sharp$. Next suppose $N = \langle k, \varepsilon \rangle$, where $\varepsilon = (\sigma, \xi)$. As before, by Definition 6.1, we can always find $\sigma^\sharp \in \mathrm{Mem}^\sharp$ such that $\sigma \propto \sigma^\sharp$. Moreover, by the definition of the approximation for exceptions, we can always find $\xi^\sharp \in \mathrm{Except}^\sharp$ such that $\xi \propto \xi^\sharp$. Hence, letting $N^\sharp = \langle k, \sigma^\sharp \otimes \xi^\sharp \rangle$, by (79) in Definition 6.4, we again obtain $N \propto N^\sharp$. In both cases, by Proposition 6.10, there exists an abstract tree $\theta^\sharp$ such that $\theta^\sharp(\epsilon) = (\rho \vdash_\beta N^\sharp \to \eta^\sharp)$ and $N \propto N^\sharp$. $\square$

The next result states that our abstract rules only generate abstract trees that are correct approximations of their concrete counterparts (i.e., concrete trees rooted with the same statement, the same environment and initial memory structure).

Theorem 7.3. *Let $\theta \in \Theta$ and $\theta^\sharp \in \Theta^\sharp$ be such that $\theta(\epsilon) = \bigl(\rho \vdash_\beta N \to \eta\bigr)$ or $\theta(\epsilon) = \bigl(\rho \vdash_\beta N \xrightarrow{\infty}\bigr)$ and $\theta^\sharp(\epsilon) = \bigl(\rho \vdash_\beta N^\sharp \to \eta^\sharp\bigr)$, where $N \propto N^\sharp$. Then $\theta \propto \theta^\sharp$.*

Theorem 7.3 is a trivial corollary of the following

Proposition 7.4. *Let*

$$S \stackrel{\mathrm{def}}{=} \left\{ (\theta, \theta^\sharp) \in \Theta \times \Theta^\sharp \;\middle|\; \begin{array}{l} \theta(\epsilon) \in \bigl\{\rho \vdash_\beta N \to \eta,\ \rho \vdash_\beta N \xrightarrow{\infty}\bigr\}, \\ \theta^\sharp(\epsilon) = \rho \vdash_\beta N^\sharp \to \eta^\sharp, \\ N \propto N^\sharp \end{array} \right\}. \tag{121}$$

Then, for all $(\theta, \theta^\sharp) \in S$, $\theta \propto \theta^\sharp$.

Proof. Let $\theta \in \Theta$ and $\theta^\sharp \in \Theta^\sharp$. We define:

$$r \stackrel{\mathrm{def}}{=} \frac{\theta_{[0]}(\epsilon)\ \cdots\ \theta_{[h-1]}(\epsilon)}{\theta(\epsilon)} \qquad\qquad r^\sharp \stackrel{\mathrm{def}}{=} \frac{\theta^\sharp_{[0]}(\epsilon)\ \cdots\ \theta^\sharp_{[\ell-1]}(\epsilon)}{\theta^\sharp(\epsilon)}$$

where, for some $h, \ell \geq 0$, $\{0, \ldots, h-1\} \subseteq \mathrm{dom}(\theta)$, $\{0, \ldots, \ell-1\} \subseteq \mathrm{dom}(\theta^\sharp)$, $h \notin \mathrm{dom}(\theta)$ and $\ell \notin \mathrm{dom}(\theta^\sharp)$. By Definitions 5.7 and 6.9, $r \in \mathcal{R}$ and $r^\sharp \in \mathcal{R}^\sharp$. Note that, to simplify the proof, we will use the schematic concrete and abstract rules given in Sections 5.5 and 6.9 to denote the actual rule instances r and $r^\sharp$.

Letting $(\theta, \theta^\sharp) \in S$, we need to show that $\theta \propto \theta^\sharp$; by Definition 7.1, this is equivalent to showing that $(\theta, \theta^\sharp) \in \mathrm{gfp}_{\subseteq}(\overline{\propto})$. To this end, by the principle of fixpoint coinduction, we will show that $(\theta, \theta^\sharp) \in \overline{\propto}(S)$.

By Definition 7.1, we need to show that the following properties hold:

(i) $\theta(\epsilon) \propto \theta^\sharp(\epsilon)$;

(ii) for each $i = 0, \ldots, h-1$ there exists $j \in \{0, \ldots, \ell-1\}$ such that $(\theta_{[i]}, \theta^\sharp_{[j]}) \in S$.

The proof that properties (i) and (ii) hold is by (well-founded) induction on the structure of the concrete tree θ. Observe that the "immediate subtree" relation between trees in Θ_+ is a well-founded partial ordering because, if $\theta \in \Theta_+$ then, by Definition 5.7, there are no infinite descending chains. We extend this ordering relation to the *immediate positive subtree* relation between trees in Θ: θ' is said to be an *immediate positive subtree* of θ if and only if $\theta' \in \Theta_+$ and is an immediate subtree of θ. Clearly, by Definition 5.7, the immediate positive subtree ordering on trees in Θ is also well-founded.

We first note that it is not restrictive to only consider unsupported expressions, declarations or statements: as noted in Section 6.9, the tree for any supported expression (resp., declaration or statement) has the same structure as the tree for the same expression (resp., declaration or statement) as if it were unsupported. Hence, once correctness of the approximation for unsupported expressions, declarations or statements is proved, the correctness for their supported counterparts will immediately follow from Definition 6.7.

Let

$$\theta(\epsilon) = \bigl(\rho \vdash_\beta N \rightarrow \eta\bigr) \text{ or } \theta(\epsilon) = \bigl(\rho \vdash_\beta N \xrightarrow{\infty}\bigr),$$
$$\theta^\sharp(\epsilon) = \bigl(\rho \vdash_\beta N^\sharp \rightsquigarrow \eta^\sharp\bigr).$$

By (121), $N \propto N^\sharp$. Therefore, by condition (85) of Definition 6.6, property (i) holds trivially whenever $\theta \in \Theta_-$ (i.e., when r is a negative concrete rule). In addition, to prove that property (i) holds for each $\theta \in \Theta_+$ (i.e., when r is a positive concrete rule), by condition (84) of Definition 6.6, we just need to show $\eta \propto \eta^\sharp$.

Consider next property (ii). The base cases are when the concrete rule r has no premises (i.e., $h = 0$); and this property holds trivially in these cases. For the inductive steps (i.e., $h > 0$) suppose $i \in \{0, \ldots, h-1\}$ and $j \in \{0, \ldots, \ell-1\}$ are such that $(\theta_{[i]}, \theta^\sharp_{[j]}) \in S$. If $\theta \in \Theta_+$ then, by the inductive hypothesis, we can assume that $(\theta_{[i]}, \theta^\sharp_{[j]}) \in \overline{\propto}(S)$; similarly, if $\theta \in \Theta_-$ and $i \neq h-1$, by the inductive hypothesis, we can assume that, $(\theta_{[i]}, \theta^\sharp_{[j]}) \in \overline{\propto}(S)$. Hence, in both cases, by Definition 7.1, $\theta_{[i]}(\epsilon) \propto \theta^\sharp_{[j]}(\epsilon)$. Also, if $\theta \in \Theta_-$, by Definition 5.7, $\theta_{[h-1]}(\epsilon)$ is a divergent sequent so that, by Definitions 6.6 and 7.1, $\theta_{[h-1]}(\epsilon) \propto \theta^\sharp_{[j]}(\epsilon)$. Thus, for all concrete trees $\theta \in \Theta$, we can safely assume the following:

$$\forall i \in \{0, \ldots, h-1\}, j \in \{0, \ldots, \ell-1\} : (\theta_{[i]}, \theta^\sharp_{[j]}) \in S \implies \theta_{[i]}(\epsilon) \propto \theta^\sharp_{[j]}(\epsilon). \quad (122)$$

Moreover, we need only explicitly prove property (ii) for each of the positive rules since, by the definition of the concrete divergence (negative) rules, (122) and Defini-

Table I. Corresponding concrete and abstract rules and terminals for expressions

E	r	$r^\sharp$	η_e	$\eta_\mathrm{e}^\sharp = \langle(\mathrm{sval}_a^\sharp, \sigma_a^\sharp), \varepsilon_a^\sharp\rangle$	
				$(\mathrm{sval}_a^\sharp, \sigma_a^\sharp)$	$\varepsilon_a^\sharp$
con	2	86	$\langle \mathrm{con}, \sigma\rangle$	$\alpha(\{\mathrm{con}\}) \otimes \sigma^\sharp$	$\texttt{none}^\sharp$
id	3	87	$\sigma[\rho(\mathrm{id})]$	$\sigma^\sharp[\rho(\mathrm{id})]$	
$-e$	4	88	ε	$(\ominus m^\sharp, \sigma_0^\sharp)$	$\varepsilon^\sharp$
	5		$\langle -m, \sigma_0\rangle$		
$e_0 \boxdot e_1$	6/7	89	ε	$(m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
		90		$(m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp$
	8	89	$\langle m_0 \circ m_1, \sigma_1\rangle$	$(m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
		90		$(m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp$
	9	90	$\langle \sigma_1, \mathtt{divbyzero}\rangle$	$(m_0^\sharp \circledcirc m_1^\sharp, \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp$
$m_0 \boxast m_1$	10/11	91	ε	$(m_0^\sharp \bowtie m_1^\sharp, \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
	12		$\langle m_0 \lesseqgtr m_1, \sigma_1\rangle$		
not b	13	92	ε	$(\circleddash\, t^\sharp, \sigma_0^\sharp)$	$\varepsilon^\sharp$
	14		$\langle \neg\, t, \sigma_0\rangle$		
b_0 **and** b_1	15	93	ε	$\upsilon_\mathrm{ff}^\sharp \sqcup \upsilon_1^\sharp$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
	16		$\langle \mathrm{ff}, \sigma_0\rangle$		
	17		η		
b_0 **or** b_1	18–20	94	Similar to the rows for 'b_0 **and** b_1'		

tion 6.4, if property (ii) holds for any positive rule it also holds for the corresponding negative rules. Thus in the detailed proofs of properties (i) and (ii) for the inductive steps, we only consider the positive rules.

To help the reader, Tables I, II, III, IV and V, contain a summary of the conclusions of rules r and $r^\sharp$. The first column $Q \in \{E, D, G, S, B, K\}$, gives the syntactic forms in the first component of the non-terminal configurations N and $N^\sharp$ (which, by Definition 6.4, must be the same); the second and third columns give a concrete rule r and abstract rule $r^\sharp$, respectively, that apply to Q. Note that we do not pair concrete rules with abstract rules that have mutually inconsistent side conditions. Justification for the omission of any abstract rules for a particular concrete rule r is given in the detailed proof for that case. The column headed η_q, where $q \in \{\mathrm{e}, \mathrm{d}, \mathrm{g}, \mathrm{s}, \mathrm{b}, \mathrm{k}\}$ gives the concrete terminal configuration for r, while the columns headed by $\eta_q^\sharp$ give the components of the abstract terminal configuration for $r^\sharp$. A blank entry in any table cell means that the value is exactly the same as the value found in the same column of the previous row. To save space in Tables II, III, IV and V, we have denoted the operations '$\mathrm{cleanup}_\mathrm{d}$', '$\mathrm{unmark}_\mathrm{s}$', '$\mathrm{unlink}_\mathrm{s}$', '$\mathrm{unmark}_\mathrm{s}^\sharp$' and '$\mathrm{unlink}_\mathrm{s}^\sharp$' by '$\mathrm{cu}_\mathrm{d}$', '$\mathrm{um}_\mathrm{s}$', '$\mathrm{ul}_\mathrm{s}$', '$\mathrm{um_s}^\sharp$' and '$\mathrm{ul_s}^\sharp$', respectively. Note that the premises and the side conditions for the rules are not provided in any of the tables; reference must be made to the actual rules for this information.

7.1 Expressions

For this part of the proof, we use Table I. By (121), $N \propto N^\sharp$. Thus letting $N = \langle E, \sigma\rangle$ and $N^\sharp = \langle E, \sigma^\sharp\rangle$, by Definition 6.4, we have the implicit hypothesis $\sigma \propto \sigma^\sharp$. We show using (80) in Definition 6.5, that $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Constant. Suppose r is an instance of (2). By definition of $\alpha\colon \wp(\mathrm{Integer}) \rightarrowtail \mathrm{Integer}^\sharp$ and $\alpha\colon \wp(\mathrm{Bool}) \rightarrowtail \mathrm{Bool}^\sharp$, we have $\mathrm{con} \propto \alpha(\{\mathrm{con}\})$; by hypothesis, $\sigma \propto \sigma^\sharp$

so that $\langle \text{con}, \sigma \rangle \propto \alpha(\{\text{con}\}) \otimes \sigma^\sharp$. Hence $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Identifier. Suppose r is an instance of (3). Since, by hypothesis, $\sigma \propto \sigma^\sharp$, by Definition 6.1 we obtain $\sigma\bigl[\rho(\text{id})\bigr] \propto \sigma^\sharp\bigl[\rho(\text{id})\bigr]$. Hence, $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Unary Minus. Suppose r is an instance of (4) or (5). Then, by hypothesis, $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$ and, hence, as $h = 1$, property (ii) holds. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$. Thus, if r is an instance of (4), then $\varepsilon \propto \varepsilon^\sharp$; if r is an instance of (5), then $m \propto m^\sharp$ and $\sigma_0 \propto \sigma_0^\sharp$. In the latter case, by the soundness of '$\ominus$', $-m \propto \ominus\, m^\sharp$. Hence, in both cases, $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Binary Arithmetic Operations. Suppose that r is an instance of one of the rules (6)–(9). Then, by hypothesis, $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$. Note that, in the condition for abstract rule (90), $\varepsilon_2^\sharp = \sigma_1^\sharp \otimes \alpha(\{\mathtt{divbyzero}\})$.

If r is an instance of (6), then $h = 1$ so that property (ii) holds. The property $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\varepsilon \propto \varepsilon_0^\sharp$. Therefore,[9] $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

If r is an instance of (7), (8) or (9), then $h = 2$. Property $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\sigma_0 \propto \sigma_0^\sharp$ and $m_0 \propto m_0^\sharp$; hence $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$.

If r is an instance of (7), then property $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ implies $\varepsilon \propto \varepsilon_1^\sharp$; thus $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$. If r is an instance of (8), then property $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ implies $\sigma_1 \propto \sigma_1^\sharp$ and $m_1 \propto m_1^\sharp$ so that, by the soundness of '$\circledcirc$', $(m_0 \boxdot m_1) \propto (m_0^\sharp \circledcirc m_1^\sharp)$; and hence $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$. If r is an instance of (9), then the condition $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ implies $\sigma_1 \propto \sigma_1^\sharp$ and $0 \propto m_1^\sharp$. Hence, by the side conditions, $r^\sharp$ must be an instance of (90); so that, as $\langle \sigma_1, \mathtt{divbyzero} \rangle \propto \sigma_1^\sharp \otimes \alpha(\{\mathtt{divbyzero}\})$, we have $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Test Operators. Suppose r is an instance of one of rules (10)–(12). Then, by hypothesis, $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$.

If r is an instance of (10), then $h = 1$ and property (ii) holds. $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\varepsilon \propto \varepsilon_0^\sharp$. Hence $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

If r is an instance of (11) or (12), then $h = 2$. $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\sigma_0 \propto \sigma_0^\sharp$ and $m_0 \propto m_0^\sharp$. Thus $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$. If r is an instance of (11), then $\varepsilon \propto \varepsilon_1^\sharp$; and if r is an instance of (12), $\sigma_1 \propto \sigma_1^\sharp$ and $m_1 \propto m_1^\sharp$ so that, by soundness of '$\bowtie$', $(m_0 \lesseqgtr m_1) \propto (m_0^\sharp \bowtie m_1^\sharp)$. Hence, for both concrete rules, $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Negation. The proof when r is an instance of (13) or (14) has the same structure of the proof for the unary minus case shown before.

Conjunction. Suppose r is an instance of one of rules (15)–(17). By hypothesis, $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$.

[9]Here and in the following, whenever we need to prove $\iota \propto \iota_0^\sharp \sqcup \iota_1^\sharp$, we just prove either $\iota \propto \iota_0^\sharp$ or $\iota \propto \iota_1^\sharp$ and implicitly use the monotonicity of γ.

Table II. Corresponding concrete and abstract rules and terminals for declarations

Q	r	$r^\sharp$	η_q	$\eta_\mathrm{q}^\sharp = \langle(\rho_a^\sharp, \sigma_a^\sharp), \varepsilon_a^\sharp\rangle$	
				$(\rho_a^\sharp, \sigma_a^\sharp)$	$\varepsilon_a^\sharp$
nil	21	95	$\langle\varnothing, \sigma\rangle$	$(\varnothing, \sigma^\sharp)$	$\mathtt{none}^\sharp$
ρ_0	22	96	$\langle\rho_0, \sigma\rangle$	$(\rho_0, \sigma^\sharp)$	$\mathtt{none}^\sharp$
rec ρ_0	23	97	$\langle\rho_1, \sigma\rangle$	$(\rho_1, \sigma^\sharp)$	$\mathtt{none}^\sharp$
gvar $\mathrm{id} : \mathrm{sT} = e$	24/25 26	98	$\mathrm{cu}_\mathrm{d}(\varepsilon)$ $\langle\rho_1, \sigma_1\rangle$	$(\rho_1, \sigma_1^\sharp)$	$\mathrm{cu}_\mathrm{d}{}^\sharp(\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp)$
lvar $\mathrm{id} : \mathrm{sT} = e$	27/28 29	99	$\mathrm{um}_\mathrm{s}(\varepsilon)$ $\langle\rho_1, \sigma_1\rangle$	$(\rho_1, \sigma_1^\sharp)$	$\mathrm{um}_\mathrm{s}{}^\sharp(\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp)$
function $\mathrm{id}(\mathrm{fps}) : \mathrm{sT} = e$	30	100	$\langle\rho_0, \sigma\rangle$	$(\rho_0, \sigma^\sharp)$	$\mathtt{none}^\sharp$
rec g	31	101	η	$\eta^\sharp$	
$g_0; g_1$	32/33 34	102	ε $\langle\rho_0[\rho_1], \sigma_1\rangle$	$(\rho_0[\rho_1], \sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
$d_0; d_1$	35–37	103	Similar to the rows for '$g_0; g_1$'		

If r is an instance of (15) or (16), then $h = 1$ and property (ii) holds. If r is an instance of (15), by (122), we have $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$, which implies $\varepsilon \propto \varepsilon_0^\sharp$. If r is an instance of (16), by Definition 6.2, $\sigma_0 \propto \sigma_\mathrm{ff}^\sharp = \phi(\rho, \sigma^\sharp, \mathbf{not}\, b_0)$. Thus, since $\mathrm{ff} \propto \alpha(\{\mathrm{ff}\})$ holds by definition, we have $\langle\mathrm{ff}, \sigma_0\rangle \propto \upsilon_\mathrm{ff}^\sharp$. Hence, for both concrete rules, $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

If r is an instance of (17), then $h = 2$. By Definition 6.2, $\sigma_0 \propto \sigma_\mathrm{tt}^\sharp$, so that $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ so that $\eta \propto \langle\upsilon_1^\sharp, \varepsilon_1^\sharp\rangle$. Hence, $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Disjunction. The proof when r is an instance of one of rules (18)–(20) is similar to that for conjunction.

7.2 Declarations

In Table II, Q denotes a local declaration D or a global declaration G. Moreover, $\eta_\mathrm{q} \in \{T_\mathrm{d}, T_\mathrm{g}\}$ and $\eta_\mathrm{q}^\sharp \in \{T_\mathrm{d}^\sharp, T_\mathrm{g}^\sharp\}$, the actual domains for η_q and $\eta_\mathrm{q}^\sharp$ will depend on context.

By (121) we have $N \propto N^\sharp$. Thus letting $N = \langle Q, \sigma\rangle$ and $N^\sharp = \langle Q, \sigma^\sharp\rangle$ for any $Q \in \{D, G\}$, by Definition 6.4, we have the implicit hypothesis $\sigma \propto \sigma^\sharp$. We show using (81) in Definition 6.5, that $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

Nil. If r is an instance of (21) then, by the hypothesis, $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

(Recursive) Environment. If r is an instance of (22) or (23) then, by the hypothesis, $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

Global Variable Declaration. If r is an instance of one of rules (24)–(26) then, by the hypothesis $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$ so that, as $h = 1$, property (ii) holds. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$. If r is an instance of (24), then $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\varepsilon \propto \varepsilon_0^\sharp$; by Definition 6.1 and monotonicity of γ, we have $\mathrm{cleanup}_\mathrm{d}(\varepsilon) \propto \mathrm{cleanup}_\mathrm{d}{}^\sharp(\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp)$, i.e., $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$. If r is an instance of (25) or (26), then $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\upsilon \propto \upsilon^\sharp$. By Definition 6.1, $\mathrm{new}_\mathrm{d}(\upsilon) \propto \mathrm{new}_\mathrm{d}{}^\sharp(\upsilon^\sharp)$. By the side condition for abstract rule (98), $\mathrm{new_d}^\sharp(\upsilon) = \bigl((\sigma_1^\sharp, l), \varepsilon_1^\sharp\bigr)$. By the side conditions for (25) and (26), either $\mathrm{new_d}(\upsilon) = \varepsilon \propto \varepsilon_1^\sharp$ —and hence $\mathrm{cleanup_d}(\varepsilon) \propto \mathrm{cleanup_d}^\sharp(\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp)$ by Definition 6.1— or $\mathrm{new_d}(\upsilon) = (\sigma_1, l) \propto (\sigma_1^\sharp, l)$. Thus, in both cases, $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

Local Variable Declaration. The proof for local variable declaration, when r is an instance of one of rules (27)–(29), is the same as that for global variable declaration, with the few necessary adjustments (i.e., using $\mathrm{unmark_s}$, $\mathrm{unmark_s^\sharp}$, $\mathrm{new_s}$, $\mathrm{new_s}^\sharp$ and i in place of $\mathrm{cleanup_d}$, $\mathrm{cleanup_d}^\sharp$, $\mathrm{new_d}$, $\mathrm{new_d}^\sharp$ and l).

Function Declaration. If r is an instance of (30) then, by the hypothesis, $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

Recursive Declaration. If r is an instance of (31), then $h = 2$ and, by the hypothesis, $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$, which implies that ρ_0 denotes the same environment in both r and $r^\sharp$ and $\sigma_0 \propto \sigma_0^\sharp$. Hence, $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ which implies $\eta \propto \eta^\sharp$. Hence, $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

Global Sequential Composition. If r is an instance of one of rules (32)–(34), then $1 \le h \le 2$ and $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$.

If r is an instance of (32), then $h = 1$ and property (ii) holds. Also, $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ implies $\varepsilon \propto \varepsilon_0^\sharp$ and hence $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

If r is an instance of (33) or (34), then $h = 2$ and, since $\sigma_0 \propto \sigma_0^\sharp$, $(\theta_{[1]} \propto \theta_{[1]}^\sharp) \in S$, so that property (ii) holds. By (122), we have $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$. If r is an instance of (33), then $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ implies $\varepsilon \propto \varepsilon_1^\sharp$, so that $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$. If r is an instance of (34), then $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$ and $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ imply that $\sigma_1 \propto \sigma_1^\sharp$ and that the two environments ρ_0 and ρ_1 are the same in both r and $r^\sharp$. Hence, their composition $\rho_0[\rho_1]$ is the same in both rules r and $r^\sharp$, so that $\eta_\mathrm{q} \propto \eta_\mathrm{q}^\sharp$.

Local Sequential Composition. The proof when r is an instance of one of rules (35)–(37) is similar to that for global sequential composition.

7.3 Statements

For this part of the proof, we use Table III. By (121), $N \propto N^\sharp$. Thus letting $N = \langle s, \sigma \rangle$ and $N^\sharp = \langle s, \sigma^\sharp \rangle$, by Definition 6.4, we have the implicit hypothesis $\sigma \propto \sigma^\sharp$. We show using (82) in Definition 6.5, that $\eta_\mathrm{s} \propto \eta_\mathrm{s}^\sharp$.

Nop. If r is an instance of (38) then, by the hypothesis, $\eta_\mathrm{e} \propto \eta_\mathrm{e}^\sharp$.

Assignment. Suppose r is an instance of (39) or (40). Then $h = 1$ and, by the hypothesis, $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$ and hence property (ii) holds. By (122) we have $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$. If r is an instance of (39), $\varepsilon \propto \varepsilon_0^\sharp$. Moreover, if r is an instance of (40), $\langle \mathrm{sval}, \sigma_0 \rangle \propto \langle \mathrm{sval}_0^\sharp, \sigma_0^\sharp \rangle$ so that, by Definition 6.1, $\sigma_0\bigl[\rho(\mathrm{id}) := \mathrm{sval}\bigr] \propto \sigma_0^\sharp\bigl[\rho(\mathrm{id}) := \mathrm{sval}^\sharp\bigr]$; letting $\sigma_0^\sharp\bigl[\rho(\mathrm{id}) := \mathrm{sval}^\sharp\bigr] = (\sigma_1^\sharp, \varepsilon_1^\sharp)$, this means that either we have $\sigma_0\bigl[\rho(\mathrm{id}) := \mathrm{sval}\bigr] \in \mathrm{ExceptState}$, so that $\sigma_0\bigl[\rho(\mathrm{id}) := \mathrm{sval}\bigr] \propto \varepsilon_1^\sharp$, or we have $\sigma_0\bigl[\rho(\mathrm{id}) := \mathrm{sval}\bigr] \in \mathrm{Mem}$, so that $\sigma_0\bigl[\rho(\mathrm{id}) := \mathrm{sval}\bigr] \propto \sigma_1^\sharp$. In all cases, $\eta_\mathrm{s} \propto \eta_\mathrm{s}^\sharp$.

Table III. Corresponding concrete and abstract rules and terminals for statements

S	r	$r^\sharp$	η_s	$\eta_s^\sharp = \langle \sigma_a^\sharp, \varepsilon_a^\sharp \rangle$	
				$\sigma_a^\sharp$	$\varepsilon_a^\sharp$
nop	38	104	σ	$\sigma^\sharp$	$\mathtt{none}^\sharp$
$\mathrm{id} := e$	39 40	105	ε $\sigma_0\bigl[\rho(\mathrm{id}) := \mathrm{sval}\bigr]$	$\sigma_1^\sharp$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
$s_0; s_1$	41 42	106	ε η	$\sigma_1^\sharp$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
$d; s$	43 44	107	ε $\mathrm{um}_\mathrm{s}(\eta)$	$\mathrm{um_s}^\sharp(\sigma_1^\sharp)$	$\varepsilon_0^\sharp \sqcup \mathrm{um_s}^\sharp(\varepsilon_1^\sharp)$
if e **then** s_0 **else** s_1	45 46/47	108	ε η	$\sigma_1^\sharp \sqcup \sigma_2^\sharp$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp$
while e **do** s_0	48/50 49 51	109	ε σ_0 η	$\sigma_{\mathrm{ff}}^\sharp \sqcup \sigma_2^\sharp$	$\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp$
throw s	52 53 54	110 111	$\langle \sigma, \chi \rangle$ ε $\langle \sigma_0, \mathrm{sval} \rangle$	$\bot$	$\varepsilon^\sharp$ $\varepsilon_0^\sharp \sqcup \varepsilon_1^\sharp$
try s **catch** k	55 56	112	σ_0 η	$\sigma_0^\sharp \sqcup \sigma_1^\sharp$	$\varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp$
try s_0 **finally** s_1	57 58 59	113	η $\langle \sigma_1, \xi_0 \rangle$ ε	$\sigma_2^\sharp$	$\varepsilon_2^\sharp \sqcup \varepsilon_3^\sharp \sqcup (\sigma_3^\sharp \otimes \xi_1^\sharp)$
$\mathrm{id} := \mathrm{id}_0(e_1, \ldots, e_n)$	62 63 64	114	ε $\mathrm{um_s}\bigl(\mathrm{ul_s}(\varepsilon)\bigr)$ $\mathrm{um_s}(\eta_2)$	$\mathrm{um_s}^\sharp(\sigma_2^\sharp)$	$\varepsilon^\sharp = \varepsilon_0^\sharp$ $\sqcup\, \mathrm{um_s}^\sharp\bigl(\mathrm{ul_s}^\sharp(\varepsilon_1^\sharp)\bigr)$ $\sqcup\, \mathrm{um_s}(\varepsilon_2^\sharp)$

Statement Sequence. Suppose r is an instance of (41) or (42). Then $1 \le h \le 2$ and, by the hypothesis, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$. If r is an instance of rule (41), as $h = 1$, property (ii) holds and also $\varepsilon \propto \varepsilon_0^\sharp$. If r is an instance of (42), then $\sigma_0 \propto \sigma_0^\sharp$ so that $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$; also, as $h = 2$, property (ii) holds; by (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$ so that $\eta \propto \langle \sigma_1^\sharp, \varepsilon_1^\sharp \rangle$. Hence, in both cases, $\eta_\mathrm{s} \propto \eta_\mathrm{s}^\sharp$.

Block. Suppose r is an instance of (43) or (44). Then $1 \le h \le 2$ and, by the hypothesis and Definition 6.1, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$. If r is an instance of (43), as $h = 1$, property (ii) holds and also $\varepsilon \propto \varepsilon_0^\sharp$. If r is an instance of (44), then $\sigma_0 \propto \sigma_0^\sharp$ so that $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$; also, as $h = 2$, property (ii) holds; by (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$; so that $\eta \propto \langle \sigma_1^\sharp, \varepsilon_1^\sharp \rangle$ and therefore, by Definition 6.1, $\mathrm{unmark_s}(\eta) \propto \bigl\langle \mathrm{unmark}_\mathrm{s}^\sharp(\sigma_1^\sharp), \mathrm{unmark}_\mathrm{s}^\sharp(\varepsilon_1^\sharp) \bigr\rangle$. Hence, in both cases $\eta_\mathrm{s} \propto \eta_\mathrm{s}^\sharp$.

Conditional. Suppose r is an instance of one of rules (45)–(47). Then $1 \le h \le 2$ and, by the hypothesis, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$.

If r is an instance of (45), $h = 1$, property (ii) holds and, as $\varepsilon \propto \varepsilon_0^\sharp$, $\eta_\mathrm{s} \propto \eta_\mathrm{s}^\sharp$.

If r is an instance of (46) or (47), then $h = 2$ and $\sigma_0 \propto \sigma_0^\sharp$. By the side conditions and Definition 6.2, if $\mathrm{tt} \propto t^\sharp$, then $\langle \mathrm{tt}, \sigma_0 \rangle \propto \langle t^\sharp, \sigma_\mathrm{tt} \rangle$ and, if $\mathrm{ff} \propto t^\sharp$, then $\langle \mathrm{ff}, \sigma_0 \rangle \propto \langle t^\sharp, \sigma_\mathrm{ff} \rangle$. Hence, if (46) applies, $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$ so that $\eta \propto \langle \sigma_1^\sharp, \varepsilon_1^\sharp \rangle$; and, if (47) applies, $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[2]}(\epsilon)$ so that $\eta \propto \langle \sigma^\sharp_2, \varepsilon^\sharp_2 \rangle$. Hence, in both cases, $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

While. Suppose r is an instance of one of rules (48)–(51). Then $1 \leq h \leq 3$ and, by hypothesis, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$.

If r is an instance of (48), $h = 1$, property (ii) holds and, as $\varepsilon \propto \varepsilon^\sharp_0$, $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

Suppose r is an instance of (49), (50) or (51). By the side conditions and Definition 6.2, if $\mathrm{tt} \propto t^\sharp$, then $\langle \mathrm{tt}, \sigma_0 \rangle \propto \langle t^\sharp, \sigma_\mathrm{tt} \rangle$ and, if $\mathrm{ff} \propto t^\sharp$, then $\langle \mathrm{ff}, \sigma_0 \rangle \propto \langle t^\sharp, \sigma_\mathrm{ff} \rangle$.

If r is an instance of (49), then, as $h = 1$, property (ii) holds and hence $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

If r is an instance of (50), then $h = 2$. Thus $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$ so that $\varepsilon \propto \varepsilon^\sharp_1$. Hence $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

If r is an instance of (51), then $h = 3$. Thus $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$. By (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$ so that $\sigma_1 \propto \sigma^\sharp_1$. Thus $(\theta_{[2]}, \theta^\sharp_{[2]}) \in S$ and property (ii) holds. By (122), $\theta_{[2]}(\epsilon) \propto \theta^\sharp_{[2]}(\epsilon)$ so that $\eta \propto \langle \sigma^\sharp_2, \varepsilon^\sharp_2 \rangle$. Hence $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

Throw. Suppose r is an instance of (52). Then $s = \chi \in \mathrm{RTSExcept}$ (so that rule (111) is not applicable). By definition of $\alpha \colon \wp(\mathrm{RTSExcept}) \rightarrowtail \mathrm{RTSExcept}^\sharp$, $\chi \propto \alpha(\{\chi\})$. Since, by hypothesis, $\sigma \propto \sigma^\sharp$, $\sigma^\sharp \otimes \alpha(\{\chi\}) = \bigl\langle \sigma^\sharp, \alpha(\{\chi\}) \bigr\rangle$ so that, by the side condition for (110), $(\sigma, \chi) \propto \varepsilon^\sharp$. Hence $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

Suppose r is an instance of (53) or (54). Then $s = e \in \mathrm{Exp}$ (so that rule (110) is not applicable). By hypothesis, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$ and, as $h = 1$, property (ii) holds. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$. If r is an instance of (53), then $\varepsilon \propto \varepsilon^\sharp_0$, while, if r is an instance of (54), $\mathrm{sval} \propto \mathrm{sval}^\sharp$ and $\sigma_0 \propto \sigma^\sharp_0$. Hence, in both cases, $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

Try Blocks. Suppose r is an instance of (55)–(59). By hypothesis, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$. Note that if r is an instance of (55) or (56), only abstract rule (112) will be applicable while if r is an instance of (57)–(59), only abstract rule (113) will be applicable.

If r is an instance of (55), $h = 1$, property (ii) holds and, as $\sigma_0 \propto \sigma^\sharp_0$, $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

If r is an instance of (56), then $\varepsilon_0 \propto \varepsilon^\sharp_0$ so that $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$. Thus, as $h = 2$, property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$ so that $\langle u, \eta \rangle \propto \bigl\langle (\sigma^\sharp_1, \varepsilon^\sharp_1), \varepsilon^\sharp_2 \bigr\rangle$ where $u \in \{\mathtt{caught}, \mathtt{uncaught}\}$. By Definition 6.5, if $u = \mathtt{caught}$, then $\eta \propto \langle \sigma^\sharp_1, \varepsilon^\sharp_1 \rangle$ and, if $u = \mathtt{uncaught}$, then $\eta \propto \varepsilon^\sharp_2$. Hence, in both cases, $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

If r is an instance of rule (57), $\sigma_0 \propto \sigma^\sharp_0$; hence $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$ so that $\eta \propto \langle \sigma^\sharp_2, \varepsilon^\sharp_2 \rangle$. Hence $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

If r is an instance of (58) or (59), $\langle \sigma_0, \xi_0 \rangle \propto \langle \sigma^\sharp_1, \xi^\sharp_1 \rangle$; hence $\sigma_0 \propto \sigma^\sharp_1$ and $\xi_0 \propto \xi^\sharp_1$ so that $(\theta_{[1]}, \theta^\sharp_{[2]}) \in S$ and property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[2]}(\epsilon)$. Thus, if (58) applies, $\sigma_1 \propto \langle \sigma^\sharp_3, \varepsilon^\sharp_3 \rangle$ so that $\langle \sigma_1, \xi_0 \rangle \propto (\sigma^\sharp_3 \otimes \xi^\sharp_1)$; and, if (59) applies, $\varepsilon \propto \langle \sigma^\sharp_3, \varepsilon^\sharp_3 \rangle$ so that $\varepsilon \propto \varepsilon^\sharp_3$. Hence, in both cases, $\eta_\mathrm{s} \propto \eta^\sharp_\mathrm{s}$.

Function call. If r is an instance of one of rules (62)–(64), then $1 \leq h \leq 3$ and $\ell = 3$. Then the conditions (60) and (61) are also conditions for abstract rule (114). By hypothesis and Definition 6.1, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$; by (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$.

Table IV. Corresponding concrete and abstract rules and terminals for function bodies

B	r	$r^\sharp$	η_{b}	$\eta_{\mathrm{b}}^\sharp = \langle (\mathrm{sval}_a^\sharp, \sigma_a^\sharp), \varepsilon_a^\sharp \rangle$	
				$(\mathrm{sval}_a^\sharp, \sigma_a^\sharp)$	$\varepsilon_a^\sharp$
let d **in** s **result** e	65	115	ε	$\mathrm{ums_s}^\sharp(\sigma_2^\sharp)$	$\varepsilon_3^\sharp = \varepsilon_0^\sharp$
	66		$\mathrm{um_s}(\varepsilon)$		$\sqcup\, \mathrm{ums_s}^\sharp(\varepsilon_1^\sharp \sqcup \varepsilon_2^\sharp)$
	67		$\mathrm{um_s}(\eta_0)$		
extern	68	116	$\sigma_0 \mid \langle \sigma_0, \xi \rangle$	$\sigma_0^\sharp$	$(\sigma_0^\sharp, \top)$

If r is an instance of (62), then $\varepsilon \propto \varepsilon_0^\sharp$, $h = 1$ and property (ii) holds. Hence $\eta_{\mathrm{s}} \propto \eta_{\mathrm{s}}^\sharp$.

If r is an instance of (63), then $\sigma_0 \propto \sigma_0^\sharp$ so that, by Definition 6.1, $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$; also, as $h = 2$, property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ and $\varepsilon \propto \varepsilon_1^\sharp$; by Definition 6.1, $\mathrm{unmark_s}\bigl(\mathrm{unlink_s}(\varepsilon)\bigr) \propto \mathrm{unmark}_{\mathrm{s}}^\sharp\bigl(\mathrm{unlink}_{\mathrm{s}}^\sharp(\varepsilon_1^\sharp)\bigr)$. Hence $\eta_{\mathrm{s}} \propto \eta_{\mathrm{s}}^\sharp$.

If r is an instance of (64), then $\sigma_1 \propto \sigma_1^\sharp$ so that, by Definition 6.1, $(\theta_{[2]}, \theta_{[2]}^\sharp) \in S$; also, as $h = 3$, property (ii) holds. By (122), $\theta_{[2]}(\epsilon) \propto \theta_{[2]}^\sharp(\epsilon)$ and $\eta_2 \propto \langle \sigma_2^\sharp, \varepsilon_2^\sharp \rangle$; by Definition 6.1, $\mathrm{unmark_s}(\eta_2) \propto \bigl\langle \mathrm{unmark}_{\mathrm{s}}^\sharp(\sigma_2^\sharp), \mathrm{unmark}_{\mathrm{s}}^\sharp(\varepsilon_2^\sharp) \bigr\rangle$. Hence $\eta_{\mathrm{s}} \propto \eta_{\mathrm{s}}^\sharp$.

7.4 Function Bodies

For this part of the proof, we use Table IV. By (121), $N \propto N^\sharp$. Thus letting $N = \langle B, \sigma \rangle$ and $N^\sharp = \langle B, \sigma^\sharp \rangle$, by Definition 6.4, we have the implicit hypothesis $\sigma \propto \sigma^\sharp$. We show using (82) in Definition 6.5, that $\eta_{\mathrm{b}} \propto \eta_{\mathrm{b}}^\sharp$.

Suppose r is an instance of one of rules (65)–(67). By hypothesis and Definition 6.1, $\mathrm{marks_s}(\sigma) \propto \mathrm{marks}_{\mathrm{s}}^\sharp(\sigma^\sharp)$, so that $(\theta_{[0]}, \theta_{[0]}^\sharp) \in S$. By (122), $\theta_{[0]}(\epsilon) \propto \theta_{[0]}^\sharp(\epsilon)$.

If r is an instance of (65), $\varepsilon \propto \varepsilon_0^\sharp$, $h = 1$ and property (ii) holds. Hence $\eta_{\mathrm{b}} \propto \eta_{\mathrm{b}}^\sharp$.

If r is an instance of (66), $\sigma_0 \propto \sigma_0^\sharp$; hence $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$ and, as $h = 2$, property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$ so that $\varepsilon \propto \varepsilon_1^\sharp$. By Definition 6.1, $\mathrm{unmark_s}(\varepsilon) \propto \mathrm{unmark}_{\mathrm{s}}^\sharp(\varepsilon_1^\sharp)$; hence $\eta_{\mathrm{b}} \propto \eta_{\mathrm{b}}^\sharp$.

If r is an instance of (67), $\sigma_0 \propto \sigma_0^\sharp$; hence $(\theta_{[1]}, \theta_{[1]}^\sharp) \in S$. By (122) we have $\theta_{[1]}(\epsilon) \propto \theta_{[1]}^\sharp(\epsilon)$, so that $\sigma_1 \propto \sigma_1^\sharp$; hence $(\theta_{[2]}, \theta_{[2]}^\sharp) \in S$; as $h = 3$, property (ii) holds. Again, by (122), $\theta_{[2]}(\epsilon) \propto \theta_{[2]}^\sharp(\epsilon)$; hence, $\eta_0 \propto \langle \sigma_2^\sharp, \varepsilon_2^\sharp \rangle$. By Definition 6.1, $\mathrm{unmark_s}(\eta_0) \propto \bigl\langle \mathrm{unmark}_{\mathrm{s}}^\sharp(\sigma_2^\sharp), \mathrm{unmark}_{\mathrm{s}}^\sharp(\varepsilon_2^\sharp) \bigr\rangle$; hence $\eta_{\mathrm{b}} \propto \eta_{\mathrm{b}}^\sharp$.

Suppose r is an instance of (68). Then $\sigma = (\mu, w)$ and $\sigma_0 = (\mu_0, w)$. By the hypothesis, $\sigma \propto \sigma^\sharp$; hence, by the side conditions, $\sigma_0 \propto \sigma_0^\sharp$; also, $\xi \propto \top$, so that $\eta_{\mathrm{b}} \propto \eta_{\mathrm{b}}^\sharp$.

7.5 Catch Clauses

For this part of the proof, we use Table V. By (121), $N \propto N^\sharp$. Thus, letting $N = \langle K, \varepsilon \rangle$ and $N^\sharp = \langle K, \varepsilon^\sharp \rangle$, by Definition 6.4, we have the implicit hypothesis $\varepsilon \propto \varepsilon^\sharp$. We show using (83) in Definition 6.5, that $\eta_{\mathrm{k}} \propto \eta_{\mathrm{k}}^\sharp$.

Catch. Let K have the form $(p)\, s$ for some exception declaration p.

Suppose r is an instance of one of rules (69)–(71). Then, by the hypothesis and Definition 6.3, $\varepsilon \propto \phi^+(p, \varepsilon^\sharp)$; by the side conditions for the abstract rules, $\varepsilon \propto \varepsilon_0^\sharp$.

Table V. Corresponding concrete and abstract rules and terminals for catch clauses

K	r	$r^\sharp$	η_k	$\eta^\sharp_\mathrm{k} = \langle \eta^\sharp_a, \varepsilon^\sharp_a \rangle$	
				$\eta^\sharp_a$	$\varepsilon^\sharp_a$
$(\mathbf{any})\, s \mid (\chi)\, s \mid (\mathrm{sT})\, s$	69	117	$\langle \mathtt{caught}, \eta_0 \rangle$	$\eta^\sharp_1$	$\varepsilon^\sharp_1$
$(\mathrm{id} : \mathrm{sT})\, s$	70	118	$\big\langle \mathtt{caught}, \mathrm{um_s}(\varepsilon_0) \big\rangle$	$(\sigma_4, \varepsilon_4) = \big(\mathrm{um_s}^\sharp(\sigma^\sharp_3),$	$\varepsilon^\sharp_1$
	71		$\langle \mathtt{caught}, \eta_0 \rangle$	$\mathrm{um_s}^\sharp(\varepsilon^\sharp_2) \sqcup \mathrm{um_s}^\sharp(\varepsilon^\sharp_3)\big)$	
$(\chi)\, s \mid (\mathrm{sT})\, s$	72	117	$\big\langle \mathtt{uncaught}, (\sigma, \xi) \big\rangle$	$\eta^\sharp_1$	$\varepsilon^\sharp_1$
$(\mathrm{id} : \mathrm{sT})\, s$	72	118	$\big\langle \mathtt{uncaught}, (\sigma, \xi) \big\rangle$	$(\sigma^\sharp_3, \varepsilon^\sharp_2 \sqcup \varepsilon^\sharp_3)$	$\varepsilon^\sharp_1$
$k_0; k_1$	73	119	$\langle \mathtt{caught}, \eta_0 \rangle$	$(\sigma^\sharp_0 \sqcup \sigma^\sharp_1, \varepsilon^\sharp_0 \sqcup \varepsilon^\sharp_2)$	$\varepsilon^\sharp_3$
	74		η		

If r is an instance of (69) then $\varepsilon = (\sigma, \xi)$; by Definition 6.1, $\sigma \propto \mathrm{mem}(\varepsilon^\sharp_0)$; Hence $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$ and, as $h = 1$, property (ii) holds. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$, which implies $\eta_0 \propto \eta^\sharp_1$ so that $\eta_\mathrm{k} \propto \eta^\sharp_\mathrm{k}$.

If r is an instance of (70) or (71), then $\varepsilon = (\sigma, \mathrm{sval})$ and $\mathrm{type}(\mathrm{sval}) = \mathrm{sT}$; by Definition 6.1, $\sigma \propto \mathrm{mem}(\varepsilon^\sharp_0)$ and $\mathrm{sval} \propto \mathrm{sT}(\varepsilon^\sharp_0)$. Hence, by Definition 6.1,

$$\mathrm{new_s}\big(\mathrm{sval}, \mathrm{mark_s}(\sigma)\big) \propto \mathrm{new_s}^\sharp\Big(\mathrm{sT}(\varepsilon^\sharp_0), \mathrm{mark}^\sharp_\mathrm{s}\big(\mathrm{mem}(\varepsilon^\sharp_0)\big)\Big) = \big((\sigma^\sharp_2, i), \varepsilon^\sharp_2\big). \quad (123)$$

If (70) applies, then $h = 0$, so that property (ii) holds trivially, and, by the side condition, $\varepsilon_0 = \mathrm{new_s}\big(\mathrm{sval}, \mathrm{mark_s}(\sigma)\big)$ so that by (123), $\varepsilon_0 \propto \varepsilon^\sharp_2$; by Definition 6.1, $\mathrm{unmark_s}(\varepsilon_0) \propto \mathrm{unmark}^\sharp_\mathrm{s}(\varepsilon^\sharp_2)$. If (71) applies, then, by the side condition, $(\sigma_0, i) = \mathrm{new_s}\big(\mathrm{sval}, \mathrm{mark_s}(\sigma)\big)$ so that by (123), $\sigma_0 \propto \sigma^\sharp_2$. Hence, $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$ and, as $h = 1$, property (ii) holds. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$, which implies $\eta_0 \propto \langle \sigma^\sharp_3, \varepsilon^\sharp_3 \rangle$. Thus, by Definition 6.1, $\mathrm{unmark_s}(\eta_0) \propto \big(\mathrm{unmark}^\sharp_\mathrm{s}(\sigma^\sharp_3), \mathrm{unmark}^\sharp_\mathrm{s}(\varepsilon^\sharp_2)\big)$. Hence, in both cases, $\eta_\mathrm{k} \propto \eta^\sharp_\mathrm{k}$.

If r is an instance of (72), then $h = 0$, so that property (ii) holds trivially. We have $\varepsilon = (\sigma, \xi)$ and, by the side condition, $p \notin \{\xi, \mathrm{cT}, \mathbf{any}\}$, where $\mathrm{cT} = \mathrm{type}(\xi)$. If $p \in \{\chi, \mathrm{sT}\}$ then abstract rule (117) applies so that, by the hypothesis, the side conditions and Definition 6.3, $(\sigma, \xi) \propto \phi^-(p, \varepsilon^\sharp) = \varepsilon^\sharp_1$. Similarly, if $p = \mathrm{id} : \mathrm{sT}$ and abstract rule (118) applies, $(\sigma, \xi) \propto \phi^-(\mathrm{sT}, \varepsilon^\sharp) = \varepsilon^\sharp_1$. Hence, in both cases, $\eta_\mathrm{k} \propto \eta^\sharp_\mathrm{k}$.

Catch Sequence. If r is an instance of (73), then as $h = 1$ and $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$, property (ii) holds. By (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$, so that $\langle \mathtt{caught}, \eta_0 \rangle \propto \big\langle (\sigma^\sharp_0, \varepsilon^\sharp_0), \varepsilon^\sharp_1 \big\rangle$. By (83) in Definition 6.5, $\eta_0 \propto (\sigma^\sharp_0, \varepsilon^\sharp_0)$, which implies $\eta_\mathrm{k} \propto \eta^\sharp_\mathrm{k}$.

If r is an instance of (74), then $(\theta_{[0]}, \theta^\sharp_{[0]}) \in S$ and, by (122), $\theta_{[0]}(\epsilon) \propto \theta^\sharp_{[0]}(\epsilon)$. Thus, $\langle \mathtt{uncaught}, \varepsilon_0 \rangle \propto \big\langle (\sigma^\sharp_0, \varepsilon^\sharp_0), \varepsilon^\sharp_1 \big\rangle$, so that, by (83) in Definition 6.5, $\varepsilon_0 \propto \varepsilon^\sharp_1$. Hence $(\theta_{[1]}, \theta^\sharp_{[1]}) \in S$ and, as $h = 2$, property (ii) holds. By (122), $\theta_{[1]}(\epsilon) \propto \theta^\sharp_{[1]}(\epsilon)$, so that $\eta \propto \big\langle (\sigma^\sharp_1, \varepsilon^\sharp_2), \varepsilon^\sharp_3 \big\rangle$, which implies $\eta_\mathrm{k} \propto \eta^\sharp_\mathrm{k}$. $\square$

A few observations regarding the precision of the proposed approximations are in order. Consider an abstract tree $\theta^\sharp \in \Theta^\sharp$ such that $\theta^\sharp(\epsilon) = (\rho \vdash_\beta N^\sharp \to \eta^\sharp)$, where $N^\sharp \in \Gamma^{\beta\sharp}_\mathrm{s}$ and $\eta^\sharp \in T^\sharp_\mathrm{s}$. If the concretization functions relating the concrete and abstract domains are strict, then the abstract tree above will encode the following *definite information*:

—non-terminating computations (i.e., unreachable code), if $\eta^\sharp = \bot$;

—non-exceptional computations, if $\eta^\sharp = \langle \sigma^\sharp, \mathtt{none}^\sharp \rangle$ and $\sigma^\sharp \neq \bot$;

—exceptional computations, if $\eta^\sharp = \langle \bot, \varepsilon^\sharp \rangle$ and $\varepsilon^\sharp \neq \mathtt{none}^\sharp$.

Obviously, a precise propagation of this definite information requires that all of the abstract domain operators are strict too. Hence, if $\theta^\sharp(\epsilon) = \bigl(\rho \vdash_\beta \langle s, \bot \rangle \rightarrow \eta^\sharp\bigr)$, we will also have $\eta^\sharp = \bot$. Similar properties hold when considering expressions, declarations and catch clauses.

8. COMPUTING ABSTRACT TREES

The results of the previous section (Theorems 7.2 and 7.3) guarantee that each concrete tree can be safely approximated by an abstract tree, provided the non-terminal configurations in the roots satisfy the approximation relation.

For expository purposes, suppose we are interested in a whole-program analysis. For each (concrete and abstract) pair of initial memories satisfying $\sigma_\mathrm{i} \propto \sigma_\mathrm{i}^\sharp$ and each $g_0 = (g; \mathbf{gvar}\, \underline{\mathrm{x}} : \mathrm{integer} = 0)$, where g is a valid program, we obtain that any abstract tree $\theta_0^\sharp \in \Theta^\sharp$ such that $\theta_0^\sharp(\epsilon) = \bigl(\varnothing \vdash_\varnothing \langle g_0, \sigma_\mathrm{i}^\sharp \rangle \rightarrow \eta_0^\sharp\bigr)$ correctly approximates each concrete tree $\theta_0 \in \Theta$ such that $\theta_0(\epsilon) = \bigl(\varnothing \vdash_\varnothing \langle g_0, \sigma_\mathrm{i} \rangle \rightarrow \eta_0\bigr)$. Notice that $\theta_0^\sharp$ is a finite tree. Letting $\eta_0^\sharp = \bigl\langle (\rho_0, \sigma_0^\sharp), \varepsilon_0^\sharp \bigr\rangle$ and assuming $\eta_0 \notin \mathrm{ExceptState}$, we obtain $\eta_0 = \langle \rho_0, \sigma_0 \rangle$, where $\sigma_0 \propto \sigma_0^\sharp$. Hence, letting $s_0 = \bigl(\underline{\mathrm{x}} := \mathrm{main}(\square)\bigr)$ and $\rho_0 : \beta$, any abstract tree $\theta_1^\sharp \in \Theta^\sharp$ such that $\theta_1^\sharp(\epsilon) = \bigl(\rho_0 \vdash_\beta \langle s_0, \sigma_0^\sharp \rangle \rightarrow \eta_1^\sharp\bigr)$ correctly approximates each concrete tree $\theta_1 \in \Theta$ such that either $\theta_1(\epsilon) = \bigl(\rho_0 \vdash_\beta \langle s_0, \sigma_0 \rangle \rightarrow \eta_1\bigr)$ or $\theta_1(\epsilon) = \bigl(\rho_0 \vdash_\beta \langle s_0, \sigma_0 \rangle \xrightarrow{\infty}\bigr)$. We are thus left with the problem of computing (any) one of these abstract trees, which are usually infinite. In particular, we are interested in choosing $\theta_1^\sharp$ in a subclass of trees admitting finite representations and, within this class, in maintaining a level of accuracy that is compatible with the complexity/precision trade-off dictated by the application.

A classical choice is to restrict attention to rational trees, that is, trees with only finitely many subtrees: the algorithm sketched in [Sch95; Sch97; Sch98], which assumes that the abstract domain is *Noetherian* (i.e., all of its ascending chains are finite), guides the analysis toward the computation of a rational tree by forcing each infinite path to contain a repetition node. Here below we describe a variation, also working for abstract domains that admit infinite ascending chains, that exploits *widening operators* [CC76; CC77a; CC92b].

Definition 8.1. **(Widening operators.)** *Let* $(D^\sharp, \sqsubseteq, \bot, \sqcup)$ *be an abstract domain. The partial operator* $\nabla \colon D^\sharp \times D^\sharp \rightarrowtail D^\sharp$ *is a* widening *if:*

—*for all* $x^\sharp, y^\sharp \in D^\sharp$, $y^\sharp \sqsubseteq x^\sharp$ *implies that* $y^\sharp \mathbin{\nabla} x^\sharp$ *is defined and* $x^\sharp \sqsubseteq y^\sharp \mathbin{\nabla} x^\sharp$;

—*for all increasing chains* $x_0^\sharp \sqsubseteq x_1^\sharp \sqsubseteq \cdots$, *the increasing chain defined by* $y_0^\sharp \stackrel{\mathrm{def}}{=} x_0^\sharp$ *and* $y_{i+1}^\sharp \stackrel{\mathrm{def}}{=} y_i^\sharp \mathbin{\nabla} (y_i^\sharp \sqcup x_{i+1}^\sharp)$, *for* $i \in \mathbb{N}$, *is not strictly increasing.*

The algorithm works by recursively constructing a finite approximation for the abstract subtree rooted in the current node (initially, the root of the whole tree). Let $n = \bigl(\rho \vdash_\beta \langle q, y_n^\sharp \rangle \rightarrow r_n\bigr)$ be the current node, where q is a uniquely labeled program phrase,[10] $y^\sharp \in D^\sharp$ is either an abstract memory $\sigma^\sharp \in \mathrm{Mem}^\sharp$ or an abstract exception state $\varepsilon^\sharp \in \mathrm{ExceptState}^\sharp$, and r_n is a placeholder for the "yet to be computed" conclusion. The node n is processed according to the following alternatives.

(i) If no ancestor of n is labeled by the program phrase q, the node has to be expanded using an applicable abstract rule instance. Namely, descendants of the premises of the rule are (recursively) processed, one at a time and from left to right. When the expansion of all the premises has been completed, including the case when the rule has no premise at all, the marker r_n is replaced by an abstract value computed according to the conclusion of the rule.

(ii) If there exists an ancestor node $m = \bigl(\rho \vdash_\beta \langle q, y_m^\sharp \rangle \to r_m\bigr)$ of n labeled by the same program phrase q and such that $y_n^\sharp \sqsubseteq y_m^\sharp$, i.e., if node n is subsumed by node m, then the node is not expanded further and the placeholder r_n is replaced by the least fixpoint of the equation $r_n = f_m(r_n)$, where f_m is the expression corresponding to the conclusion of the abstract rule that was used for the expansion of node m.[11] Intuitively, an infinite subtree rooted in node m has been identified and the "repetition node" n is transformed to a back edge to the root m of this subtree.

(iii) Otherwise, there must be an ancestor node $m = \bigl(\rho \vdash_\beta \langle q, y_m^\sharp \rangle \to r_m\bigr)$ of n labeled by the same program phrase q, but the subsumption condition $y_n^\sharp \sqsubseteq y_m^\sharp$ does not hold. Then, to ensure convergence, the abstract element $y_n^\sharp$ in node n is further approximated by $y_m^\sharp \mathbin{\nabla} (y_m^\sharp \sqcup y_n^\sharp)$ and we proceed as in case (i).

Termination of the algorithm can be proved thanks to the following observations: an infinite abstract tree necessarily has infinite paths (since the tree is finitely branching); each infinite path necessarily has an infinite number of nodes labeled by the same program phrase (since the set of program phrases is finite); the application of case (iii) leads to the computation, along each infinite path, of increasing chains of abstract elements and, by Definition 8.1, these chains are necessarily finite; hence, case (ii) is eventually applied to all infinite paths, leading to a finite representation of the rational tree where all the infinite paths are expressed by using back edges.

It should be stressed that, as far as efficiency is concerned, the algorithm outlined above can be improved by the adoption of well studied memoization techniques; as noted in [Sch97], by clearly separating design concerns from implementation concerns, the adopted methodology produces simpler proofs of correctness. Also note that the choice of the widening operator has a deep impact on the precision of the results obtained and, moreover, even a precise widening can lead to inaccurate results if applied too eagerly. However, precision problems can be mitigated by the application of suitable "widening delay" techniques [CC92b; HPR97; BHRZ05].

[10] Unique labels (e.g., given by the address of the root node for q in the program parse tree) ensure that different occurrences of the same syntax are not confused [Sch95]; this also means that, in each node n, the type and execution environments ρ and β are uniquely determined by q.

[11] As explained in [Sch95; Sch97; Sch98], the computation of such a *least* fixpoint (in the context of a coinductive interpretation of the abstract rules) is justified by the fact that here we only need to approximate the conclusions produced by the terminating concrete computations, i.e., by the concrete rules that are interpreted inductively. Also note that the divergence rules have no conclusion at all.

9. EXTENSIONS

In this section we outline how the techniques presented in the first part of the paper can be extended so as to encompass the C language and all the imperative aspects of C++ (including, of course, exceptions): Section 9.1 shows how the set of primitive types can be extended by discussing the introduction of bounded integer and floating-point types; Section 9.2 provides a sketch of how C-like pointers, arrays and records can be dealt with; dynamic memory allocation and deallocation is treated in Section 9.3; and Section 9.4 illustrates how all the non-structured control flow mechanisms of C and C++ can be accounted for.

Once an ABI (*Application Binary Interface*) has been fixed and its characteristics have been reflected into concrete and abstract memory structures, C struct and union compound types can be accommodated, even in presence of pointer casts and unrestricted pointer arithmetics, by compiling down all their uses to memory reads and writes performed through pointer dereferencing [Min06].

While we have not yet tried to incorporate object-oriented features (like classes, inheritance, method calls with dynamic binding and so forth) we do not see what, in the current design, would prevent such an extension.

9.1 Additional Arithmetic Types

The addition of more arithmetic types such as (signed and unsigned) finite integer and floating-point types is fairly straightforward. It is assumed that a preprocessor will add, as needed, a value cast operator that, for a given numeric type and constant expression, ensures that either the returned value is in the domain of that type or an appropriate exception is thrown. With this assumption, all the operations need only to be specified for operands of the very same type.

9.1.1 *Syntax.* For floating-point numbers, we add a new basic type float that represents a fixed and finite subset of the reals together with a set of special values denoting infinities, NaN (*Not a Number*) value and so forth. The exact format and range of a floating-point literal is unspecified. The addition of other floating-point types to represent double and extended precision numbers can be done the same way. To exemplify the inclusion of signed and unsigned bounded integer types, we also add the signed_char and unsigned_char basic types.

Integer types. $\mathrm{iT} \in \mathrm{iType} \stackrel{\mathrm{def}}{=} \{\mathrm{integer}, \mathrm{signed\_char}, \mathrm{unsigned\_char}, \ldots\}$;

Numeric types. $\mathrm{nT} \in \mathrm{nType} \stackrel{\mathrm{def}}{=} \mathrm{iType} \cup \{\mathrm{float}, \ldots\}$;

Basic types. $T \in \mathrm{Type} \stackrel{\mathrm{def}}{=} \mathrm{nType} \cup \{\mathrm{boolean}\}$;

Floating-point literals. $\mathrm{fl} \in \mathrm{Float}$;

Signed char literals. $\mathrm{sc} \in \mathrm{sChar}$;

Unsigned char literals. $\mathrm{uc} \in \mathrm{uChar}$.

Expressions and constants. Expressions are extended with floating-point constants, bounded integer constants, and **vcast**, a *value cast* operator for converting values from one basic type to another, when possible, or yielding an appropriate exception:

$$\begin{aligned} \mathrm{Exp} \ni e &::= \ldots \mid \mathrm{fl} \mid \mathrm{sc} \mid \mathrm{uc} \mid \mathbf{vcast}(\mathrm{nT}, e) \\ \mathrm{Con} \ni \mathrm{con} &::= \ldots \mid \mathrm{fl} \mid \mathrm{sc} \mid \mathrm{uc}. \end{aligned}$$

The functions dom: cType $\to$ {Integer, Bool, RTSExcept, Float, sChar, uChar} and type: sVal $\rightarrowtail$ sType are easily extended:

$$\mathrm{dom}(\mathrm{float}) \stackrel{\mathrm{def}}{=} \mathrm{Float}, \qquad \mathrm{type}(\mathrm{fl}) \stackrel{\mathrm{def}}{=} \mathrm{float},$$
$$\mathrm{dom}(\mathrm{signed\_char}) \stackrel{\mathrm{def}}{=} \mathrm{sChar}, \qquad \mathrm{type}(\mathrm{sc}) \stackrel{\mathrm{def}}{=} \mathrm{signed\_char},$$
$$\mathrm{dom}(\mathrm{unsigned\_char}) \stackrel{\mathrm{def}}{=} \mathrm{uChar}, \qquad \mathrm{type}(\mathrm{uc}) \stackrel{\mathrm{def}}{=} \mathrm{unsigned\_char}.$$

9.1.2 *Static Semantics.* The required adjustments to functions FI and DI are straightforward and thus omitted. Then, we add the following static semantic rules, where $\boxdot \in \{+, -, *, /, \%\}$ and $\circledast \in \{=, \neq, <, \leq, \geq, >\}$:

Expressions.

$$\frac{}{\beta \vdash_I \mathrm{fl} : \mathrm{float}} \qquad \frac{}{\beta \vdash_I \mathrm{sc} : \mathrm{signed\_char}}$$

$$\frac{\beta \vdash_I e : \mathrm{nT}}{\beta \vdash_I -e : \mathrm{nT}} \qquad \frac{}{\beta \vdash_I \mathrm{uc} : \mathrm{unsigned\_char}}$$

$$\frac{\beta \vdash_I e_0 : \mathrm{nT} \quad \beta \vdash_I e_1 : \mathrm{nT}}{\beta \vdash_I e_0 \boxdot e_1 : \mathrm{nT}} \qquad \frac{\beta \vdash_I e_0 : \mathrm{nT} \quad \beta \vdash_I e_1 : \mathrm{nT}}{\beta \vdash_I e_0 \circledast e_1 : \mathrm{boolean}}$$

$$\frac{\beta \vdash_I e : T_0}{\beta \vdash_I \mathbf{vcast}(T_1, e) : T_1} \quad \text{if casting } T_0 \text{ to } T_1 \text{ is legal.}$$

9.1.3 *Concrete Dynamic Semantics.* The added numeric types and the operations upon them bring in a considerable degree of complexity. Consider the C language, for example: unsigned bounded integers employ modular arithmetic; for signed bounded integers, overflow yields undefined behavior; the results of floating-point operations depend on the rounding mode in effect and on the settings that cause floating-point exceptions to be trapped or ignored; relational operators may or may not raise a floating-point exception when one or both arguments are NaN. In order to factor out these details and delegate them to the memory structure, we resort to a device like the one used to model supported and unsupported language elements in the abstract semantics. We thus postulate the existence of the partial functions

$$\begin{aligned}
\mathrm{eval}_{\mathrm{vc}} &: (\mathrm{nType} \times \mathrm{Con} \times \mathrm{Mem}) \rightarrowtail \mathrm{ValState} \uplus \mathrm{ExceptState},\\
\mathrm{eval}_{-_1} &: (\mathrm{Con} \times \mathrm{Mem}) \rightarrowtail \mathrm{ValState} \uplus \mathrm{ExceptState},\\
\mathrm{eval}_{\boxdot} &: (\mathrm{Con} \times \mathrm{Con} \times \mathrm{Mem}) \rightarrowtail \mathrm{ValState} \uplus \mathrm{ExceptState},\\
\mathrm{eval}_{\circledast} &: (\mathrm{Con} \times \mathrm{Con} \times \mathrm{Mem}) \rightarrowtail \mathrm{ValState} \uplus \mathrm{ExceptState},
\end{aligned}$$

that model the cast operator, unary minus, binary operators $\boxdot \in \{+, -, *, /, \%\}$ and relational operators $\circledast \in \{=, \neq, <, \leq, \geq, >\}$, respectively. Such functions need not be always defined: for example, there is no need to define $\mathrm{eval}_+(\mathrm{con}_0, \mathrm{con}_1, \sigma)$ for the case $\mathrm{type}(\mathrm{con}_0) \neq \mathrm{type}(\mathrm{con}_1)$.

Value casts. The following concrete rule schemata use the corresponding evaluation function to specify the execution of the **vcast** operator.

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{vcast}(\mathrm{nT}, e), \sigma \rangle \to \varepsilon} \qquad \frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \langle \mathrm{con}, \sigma_0 \rangle}{\rho \vdash_\beta \langle \mathbf{vcast}(\mathrm{nT}, e), \sigma \rangle \to \mathrm{eval}_{\mathrm{vc}}(\mathrm{nT}, \mathrm{con}, \sigma_0)}$$

Arithmetic evaluation. By using the evaluation functions, we can substitute rules (5), (8) and (9) with the following (note that they also capture the case when a divide-by-zero exception is thrown):

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \langle \mathrm{con}, \sigma_0 \rangle}{\rho \vdash_\beta \langle -e, \sigma \rangle \to \mathrm{eval}_{-_1}(\mathrm{nT}, \mathrm{con}, \sigma_0)}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle \mathrm{con}_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle \mathrm{con}_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \mathrm{eval}_{\boxdot}(\mathrm{con}_0, \mathrm{con}_1, \sigma_1)}$$

Arithmetic tests. Similarly, rule (12) is replaced by the more general rule

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle \mathrm{con}_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle \mathrm{con}_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 \boxast e_1, \sigma \rangle \to \mathrm{eval}_{\boxast}(\mathrm{con}_0, \mathrm{con}_1, \sigma_1)}$$

9.2 C-like Pointers, Arrays and Records

9.2.1 *Syntax.* Recall that in Sections 3 and 4 we defined the set of storable types, whose values can be read from and written to memory, and the set of denotable types, that can occur in declarations. The introduction of pointer, array and record types requires the adoption of a finer classification. The set of all *memory types* is partitioned into *object types* and *function types*: the latter differ in that we cannot read or update the "value" of a function; rather, we execute it. Object types are further partitioned into *elementary types* (also called scalar types, including basic types and pointer types) and *aggregate types* (arrays and records). All the elementary types are storable, meaning that their values can be read directly from or written directly to memory, as well as passed to and returned from functions. Regarding aggregate types, the C language prescribes that record types are storable, whereas array types are not. Pointer, array and record type derivations can be applied repeatedly to obtain, e.g., multi-dimensional arrays.

Types.

$$\begin{aligned}
\mathrm{eType} \ni \mathrm{eT} &::= T \mid \mathrm{pT} & \mathrm{oType} \ni \mathrm{oT} &::= \mathrm{sT} \mid \mathrm{aT} \\
\mathrm{pType} \ni \mathrm{pT} &::= \mathrm{mT}{*} & \mathrm{fType} \ni \mathrm{fT} &::= \mathrm{fps} \to \mathrm{sT} \\
\mathrm{sType} \ni \mathrm{sT} &::= \mathrm{eT} \mid \mathrm{rT} & \mathrm{mType} \ni \mathrm{mT} &::= \mathrm{oT} \mid \mathrm{fT} \\
\mathrm{aType} \ni \mathrm{aT} &::= \mathbf{array}\, m\, \mathbf{of}\ \mathrm{oT} & \mathrm{dType} \ni \mathrm{dT} &::= \mathrm{mT}\ \mathrm{loc} \\
\mathrm{rType} \ni \mathrm{rT} &::= \mathbf{record}\, \mathrm{id}\, \mathbf{of}\ \mathrm{id}_1 : \mathrm{oT}_1, \ldots, \mathrm{id}_j : \mathrm{oT}_j
\end{aligned}$$

We assume, without loss of generality, that the field names of record types are unique across the entire program (for example, id_1, ..., id_j could contain id as some kind of special prefix).

Identifiers are no longer the only way to denote a memory structure location. This can also be referred to by combining a pointer with the indirection operator

'$*$', an array with the indexing operator, or a record with the field selection operator. Hence, we introduce the concept of *lvalue*, which can be read as "location-valued expression."

Offsets and lvalues.

$$\mathrm{Offset} \ni o ::= \square \mid [e] \cdot o \mid \mathbf{.}\,\mathrm{id} \cdot o$$
$$\mathrm{LValue} \ni \mathrm{lval} ::= \mathrm{id} \cdot o \mid (*\,e) \cdot o$$

Consequently, the syntactic production for expressions generating identifiers, as well as the productions for statements generating assignments and function calls, are replaced by more general versions using lvalues; expressions and declarations are also extended with the address-of operator, null pointers and array variables.

Expressions, declarations and statements.

$$\mathrm{Exp} \ni e ::= \ldots \mid \mathbf{val}\,\mathrm{lval} \mid \&\,\mathrm{lval} \mid (\mathrm{pT})\,0 \qquad \mathrm{Glob} \ni g ::= \ldots \mid \mathbf{gvar}\,\mathrm{id} : \mathrm{aT} = e$$
$$\mathrm{Stmt} \ni s ::= \ldots \mid \mathrm{lval} := e \mid \mathrm{lval} := e(\mathrm{es}) \qquad \mathrm{Decl} \ni d ::= \ldots \mid \mathbf{lvar}\,\mathrm{id} : \mathrm{aT} = e$$

9.2.2 *Static Semantics.* The required adjustments to functions FI and DI are straightforward and thus omitted. The well-typedness of offsets and lvalues is encoded by the following predicates:

$$\beta, \mathrm{dT}_0 \vdash_I o : \mathrm{dT}_1, \qquad o \text{ is compatible with } \mathrm{dT}_0 \text{ and has type } \mathrm{dT}_1 \text{ in } \beta;$$
$$\beta \vdash_I \mathrm{lval} : \mathrm{dT}, \qquad \mathrm{lval} \text{ is well-formed and has type dT in } \beta.$$

The static semantics is thus extended by the following rules.[12] Note that the evaluation of an lvalue as an expression —**val** lval— causes a suitable type conversion, sometimes referred to as "type decay." Pointer arithmetics can only be applied to object types. In function calls, the callee is specified via an expression having function pointer type (typically resulting from a type decay).

Offset.

$$\frac{}{\beta, \mathrm{dT} \vdash_I \square : \mathrm{dT}} \qquad \frac{\beta \vdash_I e : \mathrm{integer} \quad \beta, \mathrm{oT}\ \mathrm{loc} \vdash_I o : \mathrm{dT}}{\beta, (\mathbf{array}\, m\, \mathbf{of}\ \mathrm{oT})\ \mathrm{loc} \vdash_I [e] \cdot o : \mathrm{dT}}$$

$$\frac{\beta, \mathrm{oT}_i\ \mathrm{loc} \vdash_I o : \mathrm{dT}}{\beta, (\mathbf{record}\,\mathrm{id}\,\mathbf{of}\ \mathrm{id}_1 : \mathrm{oT}_1; \ldots; \mathrm{id}_j : \mathrm{oT}_j)\ \mathrm{loc} \vdash_I \mathbf{.}\,\mathrm{id}_i \cdot o : \mathrm{dT}} \quad \text{if } i \in \{1, \ldots, j\}$$

Lvalue.

$$\frac{\beta, \mathrm{dT}_0 \vdash_I o : \mathrm{dT}_1}{\beta \vdash_I \mathrm{id} \cdot o : \mathrm{dT}_1} \quad \text{if } \beta(\mathrm{id}) = \mathrm{dT}_0 \qquad \frac{\beta \vdash_I e : \mathrm{mT}\,* \quad \beta, \mathrm{mT}\ \mathrm{loc} \vdash_I o : \mathrm{dT}}{\beta \vdash_I (*\,e) \cdot o : \mathrm{dT}}$$

Null pointer and address-of operator.

$$\frac{}{\beta \vdash_I (\mathrm{pT})\,0 : \mathrm{pT}} \qquad \frac{\beta \vdash_I \mathrm{lval} : \mathrm{mT}\ \mathrm{loc}}{\beta \vdash_I \&\,\mathrm{lval} : \mathrm{mT}*}$$

[12]The previous rules for identifier, assignment and function call are no longer used.

Type decay.

$$\frac{\beta \vdash_I \mathrm{lval} : \mathrm{sT\ loc}}{\beta \vdash_I \mathbf{val}\,\mathrm{lval} : \mathrm{sT}} \qquad \frac{\beta \vdash_I \mathrm{lval} : (\mathbf{array}\, m\, \mathbf{of}\ \mathrm{oT})\ \mathrm{loc}}{\beta \vdash_I \mathbf{val}\,\mathrm{lval} : \mathrm{oT}*} \qquad \frac{\beta \vdash_I \mathrm{lval} : \mathrm{fT\ loc}}{\beta \vdash_I \mathbf{val}\,\mathrm{lval} : \mathrm{fT}*}$$

Pointer arithmetics.

$$\frac{\beta \vdash_I e_0 : \mathrm{oT}* \quad \beta \vdash_I e_1 : \mathrm{integer}}{\beta \vdash_I e_0 + e_1 : \mathrm{oT}*} \qquad \frac{\beta \vdash_I e_0 : \mathrm{integer} \quad \beta \vdash_I e_1 : \mathrm{oT}*}{\beta \vdash_I e_0 + e_1 : \mathrm{oT}*}$$

$$\frac{\beta \vdash_I e_0 : \mathrm{oT}* \quad \beta \vdash_I e_1 : \mathrm{integer}}{\beta \vdash_I e_0 - e_1 : \mathrm{oT}*} \qquad \frac{\beta \vdash_I e_0 : \mathrm{oT}* \quad \beta \vdash_I e_1 : \mathrm{oT}*}{\beta \vdash_I e_0 - e_1 : \mathrm{integer}}$$

Pointer comparison.

$$\frac{\beta \vdash_I e_0 : \mathrm{pT} \quad \beta \vdash_I e_1 : \mathrm{pT}}{\beta \vdash_I e_0 \circledast e_1 : \mathrm{boolean}} \quad \text{where } \circledast \in \{=, \neq, <, \leq, \geq, >\}.$$

Assignment and function call.

$$\frac{\beta \vdash_I \mathrm{lval} : \mathrm{sT\ loc} \quad \beta \vdash_I e : \mathrm{sT}}{\beta \vdash_I \mathrm{lval} := e}$$

$$\frac{\beta \vdash_I \mathrm{lval} : \mathrm{sT\ loc} \quad \beta \vdash_I e : (\mathrm{fps} \to \mathrm{sT})* \quad \beta, \mathrm{fps} \vdash_I \mathrm{es}}{\beta \vdash_I \mathrm{lval} := e(\mathrm{es})}$$

(Multi-dimensional) Global array declaration.

$$\frac{\beta \vdash_I \mathbf{gvar}\,\mathrm{id} : \mathrm{oT} = e : \{\mathrm{id} \mapsto \mathrm{oT\ loc}\}}{\beta \vdash_I \mathbf{gvar}\,\mathrm{id} : \mathbf{array}\, m\, \mathbf{of}\ \mathrm{oT} = e : \bigl\{\mathrm{id} \mapsto (\mathbf{array}\, m\, \mathbf{of}\ \mathrm{oT})\ \mathrm{loc}\bigr\}} \quad \text{if } m > 0$$

The static semantics rule for a local array declaration is similar.

9.2.3 *Concrete Dynamic Semantics.* Concrete execution environments now map function identifiers to (properly typed) locations, rather than function abstracts: hence, we redefine $\mathrm{dVal} \stackrel{\mathrm{def}}{=} \mathrm{Addr} \times \mathrm{mType}$.

A proper handling of aggregate and function types in memory structures requires a few semantic adjustments and extensions. New memory functions allow the allocation of function abstracts in the text segment, as well as the contiguous allocation of a number of memory cells, so as to model (multi-dimensional) arrays:

$$\begin{aligned} \mathrm{new_t} &: (\mathrm{Abstract} \times \mathrm{Mem}) \to \bigl((\mathrm{Mem} \times \mathrm{Loc}) \uplus \mathrm{ExceptState}\bigr), \\ \mathrm{newarray_d} &: (\mathrm{Integer} \times \mathrm{ValState}) \to \bigl((\mathrm{Mem} \times \mathrm{Loc}) \uplus \mathrm{ExceptState}\bigr), \\ \mathrm{newarray_s} &: (\mathrm{Integer} \times \mathrm{ValState}) \to \bigl((\mathrm{Mem} \times \mathrm{Ind}) \uplus \mathrm{ExceptState}\bigr). \end{aligned}$$

It can be observed that the properties stated in Definition 5.2 still hold as long as we consider locations having non-aggregate type and properly extend the domain and codomain of the absolute memory map:

$$\mathrm{Map} \stackrel{\mathrm{def}}{=} (\mathrm{Loc} \times (\mathrm{eType} \uplus \mathrm{fType})) \rightarrowtail (\mathrm{Con} \uplus \mathrm{Loc} \uplus \mathrm{Abstract}).$$

These "elementary" memory maps need to be extended to read or update record values. To this end, we assume the existence of a couple of helper functions working on locations having aggregate type:

$$\begin{aligned}\text{locfield}\colon (\text{Id} \times \text{Loc} \times \text{rType}) &\rightarrowtail (\text{Loc} \times \text{oType}),\\ \text{locindex}\colon (\text{Integer} \times \text{Loc} \times \text{aType}) &\rightarrowtail (\text{Loc} \times \text{oType}).\end{aligned}$$

Intuitively, when defined, these functions map a record (resp., array) typed location to the typed location of one of its record fields (resp., array elements). Hence, for each $\mu \in \text{Map}$, the extension $\mu\colon (\text{Loc} \times \text{sType}) \rightarrowtail \text{sVal}$ can be recursively obtained, for each $l \in \text{Loc}$ and $\text{rT} = \textbf{record}\,\text{id}\,\textbf{of}\ \text{id}_1 : \text{oT}_1; \ldots; \text{id}_j : \text{oT}_j$, as follows and under the following conditions:

$$\mu(l, \text{rT}) \stackrel{\text{def}}{=} \Big\langle \mu\big(\text{locfield}(\text{id}_1, l, \text{rT})\big), \ldots, \mu\big(\text{locfield}(\text{id}_j, l, \text{rT})\big) \Big\rangle,$$

where, for each $l \in \text{Loc}$ and $\text{aT} = \textbf{array}\,m\,\textbf{of}\ \text{oT} \in \text{aType}$,

$$\mu(l, \text{aT}) \stackrel{\text{def}}{=} \Big[\mu\big(\text{locindex}(0, l, \text{aT})\big), \ldots, \mu\big(\text{locindex}(m-1, l, \text{aT})\big)\Big].$$

A similar extension is required for the memory update operator. Note that we will still use υ as a syntactic meta-variable for $\text{ValState} = \text{sVal} \times \text{Mem}$, but now its first component can be either a constant, or an absolute location, or a record value.

Pointer and array indexing errors are modeled via RTS exceptions. It is assumed there exists a special location $l_{\text{null}} \in \text{Loc}$ (the *null pointer* value) such that $(l_{\text{null}}, \text{mT}) \notin \text{dom}(\sigma)$ for all $\sigma \in \text{Mem}$ and $\text{mT} \in \text{mType}$; this also implies that l_{null} cannot be returned by the memory allocation operators. Hence, any attempt to read from or write to memory through this location will result in an exception state. Suitable operators on memory structures are required to check the constraints regarding pointer arithmetics (e.g., out-of-bounds array accesses), pointer comparisons (where $\circledast$ ranges over $\{=, \neq, <, \leq, \geq, >\}$) and to perform "array-to-pointer decay" conversions or record field selections:

$$\begin{aligned}\text{ptrmove}\colon (\text{Integer} \times \text{Loc} \times \text{Mem}) &\to \text{ValState} \uplus \text{ExceptState},\\ \text{ptrdiff}\colon (\text{Loc} \times \text{Loc} \times \text{Mem}) &\to \text{ValState} \uplus \text{ExceptState},\\ \text{ptrcmp}_{\circledast}\colon (\text{Loc} \times \text{Loc} \times \text{Mem}) &\to \text{ValState} \uplus \text{ExceptState},\\ \text{firstof}\colon (\text{Loc} \times \text{Mem}) &\to \text{ValState} \uplus \text{ExceptState},\\ \text{field}\colon (\text{Id} \times \text{Loc} \times \text{Mem}) &\to \text{ValState} \uplus \text{ExceptState}.\end{aligned}$$

Note that array indexing is semantically equivalent to a suitable combination of type decay, pointer arithmetics and pointer indirection. Nonetheless, for the sake of clarity and also to simplify the application of pointer and array dependence analyses [EGH94], we keep the distinction of the two constructs and, to simplify notation, we define[13]

$$\text{index}\colon (\text{Loc} \times \text{ValState}) \rightarrowtail \text{ValState} \uplus \text{ExceptState}$$

as follows:

$$\text{index}\big(l, (m, \sigma)\big) \stackrel{\text{def}}{=} \begin{cases} \varepsilon, & \text{if } \text{firstof}(l, \sigma) = \varepsilon;\\ \text{ptrmove}(m, l_0, \sigma_0), & \text{if } \text{firstof}(l, \sigma) = (l_0, \sigma_0).\end{cases}$$

[13]Functions 'field' and 'index' are similar to 'locfield' and 'locindex', but they are also meant to check their arguments against the memory structure, possibly returning an RTS exception.

Non-terminal and terminal configurations are extended so as to allow for the syntactic categories of offsets and lvalues, whose non-exceptional evaluation leads to a location:

$$\Gamma_{\mathrm{o}}^{\beta} \stackrel{\mathrm{def}}{=} \left\{ \langle o, l, \sigma \rangle \in \mathrm{Offset} \times \mathrm{Loc} \times \mathrm{Mem} \;\middle|\; \begin{array}{l} \exists \mathrm{dT}_0, \mathrm{dT}_1 \in \mathrm{dType} \,. \\ \beta, \mathrm{dT}_0 \vdash_I o : \mathrm{dT}_1 \end{array} \right\},$$

$$\Gamma_{\mathrm{l}}^{\beta} \stackrel{\mathrm{def}}{=} \left\{ \langle \mathrm{lval}, \sigma \rangle \in \mathrm{LValue} \times \mathrm{Mem} \;\middle|\; \exists \mathrm{dT} \in \mathrm{dType} \,.\, \beta \vdash_I \mathrm{lval} : \mathrm{dT} \right\},$$

$$T_{\mathrm{o}} \stackrel{\mathrm{def}}{=} T_{\mathrm{l}} \stackrel{\mathrm{def}}{=} (\mathrm{Loc} \times \mathrm{Mem}) \uplus \mathrm{ExceptState},$$

The dynamic concrete semantics is extended with the following rule schemata.

Offset.

$$\frac{}{\rho \vdash_\beta \langle \square, l, \sigma \rangle \to \langle l, \sigma \rangle}$$

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \bigl\langle [e] \cdot o, l, \sigma \bigr\rangle \to \varepsilon} \qquad \frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \upsilon}{\rho \vdash_\beta \bigl\langle [e] \cdot o, l, \sigma \bigr\rangle \to \varepsilon} \quad \text{if } \mathrm{index}(l, \upsilon) = \varepsilon$$

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \upsilon \quad \rho \vdash_\beta \langle o, l_0, \sigma_0 \rangle \to \eta}{\rho \vdash_\beta \bigl\langle [e] \cdot o, l, \sigma \bigr\rangle \to \eta} \quad \text{if } \mathrm{index}(l, \upsilon) = (l_0, \sigma_0)$$

$$\frac{}{\rho \vdash_\beta \langle .\, \mathrm{id}_i \cdot o, l, \sigma \rangle \to \varepsilon} \quad \text{if } \mathrm{field}(\mathrm{id}_i, l, \sigma) = \varepsilon$$

$$\frac{\rho \vdash_\beta \langle o, l_0, \sigma_0 \rangle \to \eta}{\rho \vdash_\beta \langle .\, \mathrm{id}_i \cdot o, l, \sigma \rangle \to \eta} \quad \text{if } \mathrm{field}(\mathrm{id}_i, l, \sigma) = (l_0, \sigma_0)$$

Lvalue.

$$\frac{\rho \vdash_\beta \langle o, \sigma \mathbin{@} a, \sigma \rangle \to \eta}{\rho \vdash_\beta \langle \mathrm{id} \cdot o, \sigma \rangle \to \eta} \quad \text{if } \rho(\mathrm{id}) = (a, \mathrm{mT})$$

$$\frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \bigl\langle (*\, e) \cdot o, \sigma \bigr\rangle \to \varepsilon} \qquad \frac{\rho \vdash_\beta \langle e, \sigma \rangle \to \langle l_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle o, l_0, \sigma_0 \rangle \to \eta}{\rho \vdash_\beta \bigl\langle (*\, e) \cdot o, \sigma \bigr\rangle \to \eta}$$

Null pointer and address-of operator.

$$\frac{}{\rho \vdash_\beta \bigl\langle (\mathrm{pT})\, 0, \sigma \bigr\rangle \to \langle l_{\mathrm{null}}, \sigma \rangle} \qquad \frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to \eta}{\rho \vdash_\beta \langle \&\, \mathrm{lval}, \sigma \rangle \to \eta}$$

Type decay.

$$\frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{val}\, \mathrm{lval}, \sigma \rangle \to \varepsilon}$$

$$\frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to \langle l, \sigma_0 \rangle}{\rho \vdash_\beta \langle \mathbf{val}\, \mathrm{lval}, \sigma \rangle \to \sigma_0[l, \mathrm{sT}]} \quad \text{if } \beta \vdash_{\mathrm{FI(lval)}} \mathrm{lval} : \mathrm{sT}\ \mathrm{loc}$$

$$\frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to \upsilon}{\rho \vdash_\beta \langle \mathbf{val}\, \mathrm{lval}, \sigma \rangle \to \mathrm{firstof}(\upsilon)} \quad \text{if } \beta \vdash_{\mathrm{FI(lval)}} \mathrm{lval} : \mathrm{aT}\ \mathrm{loc}$$

$$\frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to \upsilon}{\rho \vdash_\beta \langle \mathbf{val}\, \mathrm{lval}, \sigma \rangle \to \upsilon} \quad \text{if } \beta \vdash_{\mathrm{FI(lval)}} \mathrm{lval} : \mathrm{fT\ loc}$$

Pointer arithmetics. Let $\boxdot$ denote a binary abstract syntax operator in $\{+, -\}$, as well as the corresponding unary operation on integers. Then, the following are added to rule schemata (6)–(9).

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle l, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \varepsilon}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle l, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle m, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 \boxdot e_1, \sigma \rangle \to \mathrm{ptrmove}(m_0, l, \sigma_1)} \quad \text{if } m_0 = \boxdot\, m$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle m, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle l, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 + e_1, \sigma \rangle \to \mathrm{ptrmove}(m, l, \sigma_1)}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle l_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle l_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 - e_1, \sigma \rangle \to \mathrm{ptrdiff}(l_0, l_1, \sigma_1)}$$

Pointer comparison. Let $\circledast$ denote a binary abstract syntax operator in the set $\{=, \neq, <, \leq, \geq, >\}$. Then, the following are added to rule schemata (10)–(12).

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle l, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma \rangle \to \varepsilon}$$

$$\frac{\rho \vdash_\beta \langle e_0, \sigma \rangle \to \langle l_0, \sigma_0 \rangle \quad \rho \vdash_\beta \langle e_1, \sigma_0 \rangle \to \langle l_1, \sigma_1 \rangle}{\rho \vdash_\beta \langle e_0 \circledast e_1, \sigma \rangle \to \mathrm{ptrcmp}_{\circledast}(l_0, l_1, \sigma_1)}$$

Assignment.

$$\frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathrm{lval} := e, \sigma \rangle \to \varepsilon} \qquad \frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to (l, \sigma_0) \quad \rho \vdash_\beta \langle e, \sigma_0 \rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathrm{lval} := e, \sigma \rangle \to \varepsilon}$$

$$\frac{\rho \vdash_\beta \langle \mathrm{lval}, \sigma \rangle \to (l, \sigma_0) \quad \rho \vdash_\beta \langle e, \sigma_0 \rangle \to \langle \mathrm{sval}, \sigma_1 \rangle}{\rho \vdash_\beta \langle \mathrm{lval} := e, \sigma \rangle \to \sigma_1\big[(l, \mathrm{sT}) := \mathrm{sval}\big]} \quad \text{if } \beta \vdash_{\mathrm{FI}(e)} e : \mathrm{sT}$$

Similar changes are required for the case of a function call. First, the lvalue is evaluated so as to obtain the target location where the result of the function call will be stored; then, the function designator (an expression) is evaluated to obtain a location having function type; this location is fed to the memory structure so as to obtain the function abstract. All the other computation steps, including parameter passing, are performed as before. On exit from the function call, the return value is stored at the location computed in the first step. Exceptions are eventually detected and propagated as usual. Also note that, thanks to the rules for type decay, arrays and functions can be passed to and returned from function calls.

(Multi-dimensional) Global array declaration. In the following rule schemata, let $n > 0$, $\mathrm{aT} = \mathbf{array}\, m_1\, \mathbf{of}\ (\ldots(\mathbf{array}\, m_n\, \mathbf{of}\ \mathrm{sT})\ldots)$ and $m = m_1 \times \ldots \times m_n$.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{gvar}\, \mathrm{id} : \mathrm{aT} = e, \sigma\rangle \to \mathrm{cleanup_d}(\varepsilon)}$$

if either $\eta = \varepsilon$, or $\eta = \upsilon$ and $\mathrm{newarray_d}(m, \upsilon) = \varepsilon$;

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \upsilon}{\rho \vdash_\beta \langle \mathbf{gvar}\, \mathrm{id} : \mathrm{aT} = e, \sigma\rangle \to \langle \rho_0, \sigma_0\rangle}$$

if $\mathrm{newarray_d}(m, \upsilon) = (\sigma_0, l)$ and $\rho_0 = \bigl\{\mathrm{id} \mapsto (l, \mathrm{aT})\bigr\}$.

The rules for local array declaration are similar. Since function abstracts are now stored in memory structures, a few minor adaptations, omitted for space reasons, are also required for the rule of function declarations (which uses $\mathrm{new_t}$) and the rules for recursive environments and declarations.

9.3 Heap Memory Management

By adding a *heap segment* to memory structures, as well as suitable helper functions ($\mathrm{new_h}$, $\mathrm{delete_h}$ and the corresponding array versions), it is possible to further extend the language to embrace dynamic memory allocation and deallocation.

9.3.1 *Syntax.* We add an allocation expression and a deallocation statement:

$$\mathrm{Exp} \ni e ::= \ldots \mid \mathbf{new}\, \mathrm{sT} = e$$
$$\mathrm{Stmt} \ni s ::= \ldots \mid \mathbf{delete}\, e$$

9.3.2 *Static Semantics.*

$$\frac{\beta \vdash_I e : \mathrm{sT}}{\beta \vdash_I \mathbf{new}\, \mathrm{sT} = e : \mathrm{sT}*} \qquad \frac{\beta \vdash_I e : \mathrm{sT}*}{\beta \vdash_I \mathbf{delete}\, e}$$

9.3.3 *Concrete Dynamic Semantics.* This is extended with the schemata:

New expression.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{new}\, \mathrm{sT} = e, \sigma\rangle \to \varepsilon} \qquad \frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \upsilon}{\rho \vdash_\beta \langle \mathbf{new}\, \mathrm{sT} = e, \sigma\rangle \to \varepsilon} \quad \text{if } \mathrm{new_h}(\upsilon) = \varepsilon$$

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \upsilon}{\rho \vdash_\beta \langle \mathbf{new}\, \mathrm{sT} = e, \sigma\rangle \to \langle l, \sigma_0\rangle} \quad \text{if } \mathrm{new_h}(\upsilon) = (\sigma_0, l)$$

Delete operator.

$$\frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \varepsilon}{\rho \vdash_\beta \langle \mathbf{delete}\, e, \sigma\rangle \to \varepsilon} \qquad \frac{\rho \vdash_\beta \langle e, \sigma\rangle \to \upsilon}{\rho \vdash_\beta \langle \mathbf{delete}\, e, \sigma\rangle \to \mathrm{delete_h}(\upsilon)}$$

Similar rules allow for allocation and deallocation of an array on the heap: note that, contrary to the previous cases, the dimensions of the array can be specified as expressions that will be evaluated dynamically.

Regarding the abstract semantics, the extensions concerning C-like pointers and arrays as well as heap memory management can be obtained along the lines followed in Section 6. In particular, the new memory structure operators described above are provided with safe approximations and a new abstract domain $\mathrm{Loc}^\sharp$ for location-valued expressions has to be defined. By generalizing the abstract memory read and update operators so as to take as input an abstract location, we realize the so-called *weak read* and *weak update* operators, so as to correctly deal with, e.g., assignments or function calls whose target is not statically known. In practice, no fundamentally new issue has to be solved as far as the specification of the abstract interpreter is concerned. This is not to say that these extensions are trivial; rather, the real issues (e.g., the efficient and accurate tracking of aliasing information for pointers [Ema93; EGH94] or the appropriate summarization techniques for large arrays [GRS05] and heap-allocated data [GDD$^+$04; SRW02]) are orthogonal to the current approach and should be addressed elsewhere.

9.4 Non-Structured Control Flow Mechanisms

It turns out that the approach we have chosen to model exceptional behavior of programs can be easily generalized so as to capture all the non-structured control flow mechanisms of languages such as C and C++. To exemplify such a generalization, the abstract syntax of commands is extended with branching and labeled statements:

$$\mathrm{Label} \ni l ::= \mathrm{id} \mid m \mid \mathbf{default}$$
$$\mathrm{Stmt} \ni s ::= \ldots \mid \mathbf{goto}\,\mathrm{id} \mid \mathbf{switch}\,e\,\mathbf{in}\,s \mid \mathbf{break} \mid \mathbf{continue} \mid \mathbf{return}\,e \mid l : s$$

We assume that the static semantics ensures the labels used in a function body are all distinct (if the language supports local labels, then a trivial renaming will be required) and that every goto has access to a corresponding labeled statement, respecting the constraints imposed by the language (concerning, for instance, jumping into and outside blocks).

The state of a computation is captured, besides the current program point, by a *control mode* and a memory structure, which together constitute what we call a *control state*. A control state is classified by the corresponding control mode in either a plain execution state or an *exception state*; a plain execution state can be further distinguished in either a *normal execution state*, or a *branching state*, or a *value state* (for computations yielding a proper value), or an *environment state* (for computations yielding an execution environment).

Definition 9.1. (GotoMode, SwitchMode, ValMode, EnvMode, ExceptMode, CtrlMode, CtrlState.) *The sets of* goto*,* switch*,* value*,* environment*,* exception *and all* control modes *are given, respectively, by*

$$\begin{aligned}
\mathrm{GotoMode} &\stackrel{\mathrm{def}}{=} \bigl\{\, \mathrm{goto}(\mathrm{id}) \bigm| \mathrm{id} \in \mathrm{Id} \,\bigr\},\\
\mathrm{SwitchMode} &\stackrel{\mathrm{def}}{=} \bigl\{\, \mathrm{switch}(\mathrm{sval}) \bigm| \mathrm{sval} \in \mathrm{sVal} \,\bigr\},\\
\mathrm{ValMode} &\stackrel{\mathrm{def}}{=} \bigl\{\, \mathrm{value}(\mathrm{sval}) \bigm| \mathrm{sval} \in \mathrm{sVal} \,\bigr\},\\
\mathrm{EnvMode} &\stackrel{\mathrm{def}}{=} \bigl\{\, \mathrm{env}(\rho) \bigm| \rho \in \mathrm{Env} \,\bigr\},\\
\mathrm{ExceptMode} &\stackrel{\mathrm{def}}{=} \bigl\{\, \mathrm{except}(\xi) \bigm| \xi \in \mathrm{Except} \,\bigr\},\\
\mathrm{CtrlMode} &\stackrel{\mathrm{def}}{=} \mathrm{GotoMode} \uplus \mathrm{SwitchMode} \uplus \mathrm{ValMode} \uplus \mathrm{EnvMode}
\end{aligned}$$

$$\uplus \mathrm{ExceptMode} \uplus \{\mathrm{continue}, \mathrm{break}, \mathrm{return}, \mathrm{exec}\},$$

where continue, break *and* return *are the* exit modes *and* exec *is the* plain execution mode. *Control modes are denoted by* cm, cm_0, cm_1 *and so forth.*

A control state *is an element of* $\mathrm{CtrlState} \stackrel{\mathrm{def}}{=} \mathrm{CtrlMode} \times \mathrm{Mem}$. *Control states are denoted by* cs, cs_0, cs_1 *and so forth.*

The concrete semantics of the goto statement can now be expressed by

$$\frac{}{\rho \vdash_\beta \big\langle \mathbf{goto}\, \mathrm{id}, (\mathrm{cm}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma\rangle}$$

if $\mathrm{cm} = \mathrm{exec}$ and $\mathrm{cm}_0 = \mathrm{goto(id)}$ or $\mathrm{cm} \neq \mathrm{exec}$ and $\mathrm{cm}_0 = \mathrm{cm}$.

The semantics of labeled statements is given by

$$\frac{\rho \vdash_\beta \big\langle s, (\mathrm{cm}_0, \sigma)\big\rangle \to \eta}{\rho \vdash_\beta \big\langle l : s, (\mathrm{cm}, \sigma)\big\rangle \to \eta}$$

where $\mathrm{cm}_0 = \mathrm{exec}$ if $\mathrm{cm} = \mathrm{exec}$, or $\mathrm{cm} = \mathrm{goto(id)}$ and $l = \mathrm{id}$, or $\mathrm{cm} = \mathrm{switch(sval)}$ and $l \in \{\mathbf{default}, \mathrm{sval}\}$; otherwise $\mathrm{cm}_0 = \mathrm{cm}$.

Of course, the semantics of all statements must be suitably modified. For instance, the assignment should behave like a nop unless the control mode is the normal execution one. Statements with non trivial control flow need more work. For example, the semantics of the conditional statement can be captured by[14]

$$\frac{\rho \vdash_\beta \big\langle e, (\mathrm{exec}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{exec}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle} \quad \text{if } \mathrm{cm}_0 \in \mathrm{ExceptMode}$$

$$\frac{\begin{array}{ll}\rho \vdash_\beta \big\langle e, (\mathrm{exec}, \sigma)\big\rangle \to \big\langle \mathrm{value(tt)}, \sigma_0\big\rangle & \rho \vdash_\beta \big\langle s_0, (\mathrm{exec}, \sigma_0)\big\rangle \to \langle \mathrm{cm}_1, \sigma_1\rangle \\ & \rho \vdash_\beta \big\langle s_1, (\mathrm{cm}_1, \sigma_1)\big\rangle \to \eta\end{array}}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{exec}, \sigma)\big\rangle \to \eta} \tag{124}$$

if $\mathrm{cm}_1 \in \mathrm{GotoMode}$;

$$\frac{\rho \vdash_\beta \big\langle e, (\mathrm{exec}, \sigma)\big\rangle \to \big\langle \mathrm{value(tt)}, \sigma_0\big\rangle \quad \rho \vdash_\beta \big\langle s_0, (\mathrm{exec}, \sigma_0)\big\rangle \to \langle \mathrm{cm}_1, \sigma_1\rangle}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{exec}, \sigma)\big\rangle \to \langle \mathrm{cm}_1, \sigma_1\rangle}$$

if $\mathrm{cm}_1 \notin \mathrm{GotoMode}$;

$$\frac{\rho \vdash_\beta \big\langle e, (\mathrm{exec}, \sigma)\big\rangle \to \big\langle \mathrm{value(ff)}, \sigma_0\big\rangle \quad \rho \vdash_\beta \big\langle s_1, (\mathrm{exec}, \sigma_0)\big\rangle \to \eta}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{exec}, \sigma)\big\rangle \to \eta}$$

$$\frac{\rho \vdash_\beta \big\langle s_0, (\mathrm{cm}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{cm}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle}$$

if $\mathrm{cm} \in \mathrm{GotoMode} \uplus \mathrm{SwitchMode}$ and $\mathrm{cm}_0 \notin \mathrm{GotoMode} \uplus \mathrm{SwitchMode}$;

$$\frac{\rho \vdash_\beta \big\langle s_0, (\mathrm{cm}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle \quad \rho \vdash_\beta \big\langle s_1, (\mathrm{cm}_0, \sigma_0)\big\rangle \to \eta}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{cm}, \sigma)\big\rangle \to \eta}$$

[14] Recall that, in C, it is perfectly legal to jump into the "else branch" from the "then branch."

if $\mathrm{cm} \in \mathrm{GotoMode} \uplus \mathrm{SwitchMode}$ and $\mathrm{cm}_0 \in \mathrm{GotoMode} \uplus \mathrm{SwitchMode}$;

$$\frac{}{\rho \vdash_\beta \big\langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, (\mathrm{cm}, \sigma)\big\rangle \to \langle \mathrm{cm}, \sigma\rangle}$$

if $\mathrm{cm} \notin \mathrm{GotoMode} \uplus \mathrm{SwitchMode} \uplus \{\mathrm{exec}\}$.

Likewise, the semantics of the **switch** statement can be captured by:

$$\frac{\rho \vdash_\beta \big\langle e, (\mathrm{exec}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle}{\rho \vdash_\beta \big\langle \mathbf{switch}\, e\, \mathbf{in}\, s, (\mathrm{exec}, \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle} \quad \text{if } \mathrm{cm}_0 \in \mathrm{ExceptMode}$$

$$\frac{\begin{array}{c}\rho \vdash_\beta \big\langle e, (\mathrm{exec}, \sigma)\big\rangle \to \big\langle \mathrm{value}(\mathrm{sval}_0), \sigma_0\big\rangle \\ \rho \vdash_\beta \big\langle s, (\mathrm{switch}(\mathrm{sval}_0), \sigma_0)\big\rangle \to \langle \mathrm{cm}_1, \sigma_1\rangle\end{array}}{\rho \vdash_\beta \big\langle \mathbf{switch}\, e\, \mathbf{in}\, s, (\mathrm{exec}, \sigma)\big\rangle \to \langle \mathrm{cm}_2, \sigma_1\rangle}$$

$$\text{if } \mathrm{cm}_2 = \begin{cases} \mathrm{exec}, & \text{if } \mathrm{cm}_1 \in \mathrm{SwitchMode} \uplus \{\mathrm{break}\}, \\ \mathrm{cm}_1, & \text{otherwise;} \end{cases}$$

$$\frac{\rho \vdash_\beta \big\langle s, (\mathrm{goto}(\mathrm{id}), \sigma)\big\rangle \to \langle \mathrm{cm}_0, \sigma_0\rangle}{\rho \vdash_\beta \big\langle \mathbf{switch}\, e\, \mathbf{in}\, s, (\mathrm{goto}(\mathrm{id}), \sigma)\big\rangle \to \langle \mathrm{cm}_1, \sigma_0\rangle}$$

$$\text{if } \mathrm{cm}_1 = \begin{cases} \mathrm{exec}, & \text{if } \mathrm{cm}_0 = \mathrm{break}, \\ \mathrm{cm}_0, & \text{otherwise;} \end{cases}$$

$$\frac{}{\rho \vdash_\beta \big\langle \mathbf{switch}\, e\, \mathbf{in}\, s, (\mathrm{cm}, \sigma)\big\rangle \to \langle \mathrm{cm}, \sigma\rangle} \quad \text{if } \mathrm{cm} \notin \mathrm{GotoMode} \uplus \{\mathrm{exec}\}.$$

While such a semantic treatment captures all forward jumps, for backward jumps something more is required. One simple possibility (which is not the only one) is to explicitly introduce a looping construct that is (only) available in the abstract syntax. That is, we extend Stmt once again as

$$\mathrm{Stmt} \ni s ::= \ldots \mid \mathbf{loop}\, s$$

and assume that a set of such loops has been inserted so that all backward jumps are enclosed in at least one loop (notice that at most one such loop per function body suffices, but more can be used as a matter of optimization). For $s \in \mathrm{Stmt}$, let $\mathrm{SL}(s)$ denote the set of statement labels in s. The concrete semantics of this looping construct is now given by

$$\frac{\rho \vdash_\beta \langle s, \mathrm{cs}\rangle \to \langle \mathrm{cm}, \sigma\rangle}{\rho \vdash_\beta \langle \mathbf{loop}\, s, \mathrm{cs}\rangle \to \langle \mathrm{cm}, \sigma\rangle} \quad \text{if } \mathrm{cm} \neq \mathrm{goto}(\mathrm{id}) \text{ for each } \mathrm{id} \in \mathrm{SL}(s)$$

$$\frac{\rho \vdash_\beta \langle s, \mathrm{cs}\rangle \to \big\langle \mathrm{goto}(\mathrm{id}), \sigma\big\rangle \quad \rho \vdash_\beta \big\langle \mathbf{loop}\, s, \big(\mathrm{goto}(\mathrm{id}), \sigma\big)\big\rangle \to \eta}{\rho \vdash_\beta \langle \mathbf{loop}\, s, \mathrm{cs}\rangle \to \eta} \quad \text{if } \mathrm{id} \in \mathrm{SL}(s)$$

Observe that the systematic use of the looping construct can make rule schema (124) redundant.

Other rules are omitted for space reasons. However, there are no additional difficulties besides the ones just addressed: the rules for **break** and **continue** are straightforward; **return** e can be modeled as the assignment to the reserved identifier $\underline{\mathrm{x}}_0$ (see concrete rule (67)), followed by the setting of the control mode; the rules for the **while** loop are a bit involved as they must support the 'break' and 'continue' control modes in addition to 'goto' and 'switch'.

The proposed approach handles non-structured control flow mechanisms essentially by adding a sort of control register to the rule-based interpreter of the language. As far as the abstract semantics is concerned, a first choice to be made concerns the approximation of the values that the control register can take. As usual, there is a complexity/precision trade-off to be faced: the simple solution is to approximate $\wp(\mathrm{CtrlMode})$ by some (simple) abstract domain $\mathrm{CtrlMode}^\sharp$ and then approximate $\mathrm{CtrlState} = \mathrm{CtrlMode} \times \mathrm{Mem}$ by $\mathrm{CtrlMode}^\sharp \otimes \mathrm{Mem}^\sharp$; a more precise solution is to approximate $\wp(\mathrm{CtrlState})$ by an abstract domain $\mathrm{CtrlState}^\sharp$ that captures relational information connecting the control modes to the memory structures they can be coupled with. The abstract rules schemata must of course be modified to match the concrete world. For instance, the abstract rule for the conditional statement becomes:

$$\frac{\rho \vdash_\beta \langle e, \mathrm{cs}^\sharp_{\mathrm{cond}} \rangle \to \mathrm{cs}^\sharp_0 \quad \rho \vdash_\beta \langle s_0, \mathrm{cs}^\sharp_{\mathrm{then}} \rangle \to \mathrm{cs}^\sharp_1 \quad \rho \vdash_\beta \langle s_1, \mathrm{cs}^\sharp_{\mathrm{else}} \rangle \to \mathrm{cs}^\sharp_2}{\rho \vdash_\beta \langle \mathbf{if}\, e\, \mathbf{then}\, s_0\, \mathbf{else}\, s_1, \mathrm{cs}^\sharp \rangle \rightsquigarrow \mathrm{cs}^\sharp_3}$$

where

$$\begin{aligned}
\mathrm{cs}^\sharp_{\mathrm{cond}} &= \Phi_{\mathrm{e}}(\rho, \mathrm{cs}^\sharp, \mathrm{tt}),\\
\mathrm{cs}^\sharp_{\mathrm{then}} &= \Phi_{\mathrm{e}}(\rho, \mathrm{cs}^\sharp, e) \sqcup \Phi_{\mathrm{m}}(\mathrm{cs}^\sharp, \mathrm{GotoMode} \uplus \mathrm{SwitchMode}),\\
\mathrm{cs}^\sharp_{\mathrm{else}} &= \Phi_{\mathrm{e}}(\rho, \mathrm{cs}^\sharp, \mathbf{not}\, e) \sqcup \Phi_{\mathrm{m}}(\mathrm{cs}^\sharp_1, \mathrm{GotoMode}) \sqcup \mathrm{cs}^\sharp_{\mathrm{jump}},\\
\mathrm{cs}^\sharp_{\mathrm{jump}} &= \begin{cases} \bot, & \text{if } \Phi_{\mathrm{m}}(\mathrm{cs}^\sharp, \mathrm{GotoMode} \uplus \mathrm{SwitchMode}) = \bot,\\ \Phi_{\mathrm{m}}(\mathrm{cs}^\sharp_1, C_{\mathrm{jump}}), & \text{otherwise,} \end{cases}\\
C_{\mathrm{jump}} &= \mathrm{GotoMode} \cup \bigl\{\, \mathrm{cm} \in \mathrm{CtrlMode} \bigm| \exists \sigma \in \mathrm{Mem} \,.\, \gamma(\mathrm{cs}^\sharp) = (\mathrm{cm}, \sigma) \,\bigr\},\\
\mathrm{cs}^\sharp_3 &= \Phi_{\mathrm{m}}\Bigl(\mathrm{cs}^\sharp, \mathrm{CtrlMode} \setminus \bigl(\{\mathrm{exec}\} \uplus \mathrm{GotoMode} \uplus \mathrm{SwitchMode}\bigr)\Bigr)\\
&\qquad \sqcup \Phi_{\mathrm{m}}\Bigl(\mathrm{cs}^\sharp_0, \mathrm{CtrlMode} \setminus \mathrm{ValMode}\Bigr) \sqcup \mathrm{cs}^\sharp_1 \sqcup \mathrm{cs}^\sharp_2,
\end{aligned}$$

and the two computable filter functions $\Phi_{\mathrm{e}} \colon (\mathrm{Env} \times \mathrm{CtrlState}^\sharp \times \mathrm{Exp}) \to \mathrm{CtrlState}^\sharp$ and $\Phi_{\mathrm{m}} \colon \bigl(\mathrm{CtrlState}^\sharp \times \wp(\mathrm{CtrlMode})\bigr) \to \mathrm{CtrlState}^\sharp$ are defined as follows, for each $\rho \in \mathrm{Env}$, $\mathrm{cs}^\sharp \in \mathrm{CtrlState}^\sharp$, $e \in \mathrm{Exp}$ and $C \subseteq \mathrm{CtrlMode}$ such that, for some $\beta \in \mathrm{TEnv}$, $\beta : I$ with $\mathrm{FI}(e) \subseteq I$ and $\beta \vdash_I e : \mathrm{boolean}$:

$$\begin{aligned}
\gamma\bigl(\Phi_{\mathrm{e}}(\rho, \mathrm{cs}^\sharp, e)\bigr) &\supseteq \left\{ \mathrm{cs} \in \gamma(\mathrm{cs}^\sharp) \;\middle|\; \begin{array}{l} \exists \sigma \in \mathrm{Mem} \,.\, \mathrm{cs} = (\mathrm{exec}, \sigma),\\ \exists \sigma' \in \mathrm{Mem}\\ \qquad .\, \bigl(\rho \vdash_\beta \langle e, \mathrm{cs} \rangle \to \bigl\langle \mathrm{value}(\mathrm{tt}), \sigma' \bigr\rangle\bigr) \end{array} \right\},\\
\gamma\bigl(\Phi_{\mathrm{m}}(\mathrm{cs}^\sharp, C)\bigr) &\supseteq \bigl\{\, \mathrm{cs} \in \gamma(\mathrm{cs}^\sharp) \bigm| \exists \sigma \in \mathrm{Mem} \,.\, \mathrm{cs} = (\mathrm{cm}, \sigma), \mathrm{cm} \in C \,\bigr\}.
\end{aligned}$$

10. CONCLUSION

In this paper, we have confronted the problem of defining an analysis framework for the specification and realization of precise static analyzers for mainstream imperative programming languages, tools in very short supply that, however, ought to become part of the current programming practice. A proposal put forward by Schmidt twelve years ago [Sch95] held, in our eyes, considerable promise, despite the fact it had not been fully developed and applied in realistic contexts. It was therefore natural to question whether the promise could be fulfilled. To investigate Schmidt's approach, which is based on structured operational semantics and abstract interpretation, we have defined an imperative language, CPM, that embodies all the "problematic features" of single-threaded imperative languages now in widespread use. We have presented a concrete semantics of CPM that is suitable for abstraction while retaining all the nice features of SOS descriptions. For a subset of the language we have formally defined an abstract semantics that can fully exploit the precision offered by relational abstract domains, and proved its soundness with respect to the concrete one. We have also shown how approximations of the abstract semantics can be effectively computed. In order to provide an experimental evaluation of the ideas presented in this paper, both the concrete and the abstract semantics —instantiated over sophisticated numeric domains and together with a suitable fixpoint computation engine— have been incorporated into the ECLAIR system. This work allows us to conclude that the proposal of Schmidt can play a crucial role in the development of reliable and precise analyzers. The key features of this approach are:

—a fairly concise concrete semantics that experts can easily read (and modify as needed) and that everyone can execute on non-trivial examples in order to check its agreement with the applicable language standards;

—a fairly concise abstract semantics that is fully parametric with respect to the abstract domain, that is not difficult to prove correct with respect to the concrete one (to the point that automatizing the proof seems to be a reasonable goal), and that directly leads to the implementation of static analyzers.

Of course, the story does not end here. For instance, our analysis framework is parametric on abstract memory structures. While the literature seems to provide all that is necessary to realize very sophisticated ones, it is not difficult to predict that, among all the code out there waiting to be analyzed, some will greatly exacerbate the complexity/precision trade-off. However, these are research problems for the future — now that we have, as given here, a formal design on which analyzers can be built, our next goal is to complete the build and make the technology described here truly available and deployable.

ACKNOWLEDGMENTS

Anna Dolma Alonso, Irene Bacchi, Danilo Bonardi, Andrea Cimino, Enrico Franchi, Davide Masi and Alessandro Vincenzi (all students of the course on "Analysis and Verification of Software" taught by Roberto Bagnara at the University of Parma) and Vajirapan Panumong (University of Leeds) collaborated on previous, much more restricted versions of this work. We are also grateful to David Merchat (formerly at the University of Parma) and Katy Dobson (University of Leeds) for the discussions we have had on the subject of this paper.

REFERENCES

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, *Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software*, The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday] (T. Æ. Mogensen, D. A. Schmidt, and I. Hal Sudborough, eds.), Lecture Notes in Computer Science, vol. 2566, Springer-Verlag, Berlin, 2002, pp. 85–108.

———, *A static analyzer for large safety-critical software*, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03) (San Diego, California, USA), ACM Press, 2003, pp. 196–207.

R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella, *Precise widening operators for convex polyhedra*, Science of Computer Programming **58** (2005), no. 1–2, 28–56.

R. Bagnara, P. M. Hill, and E. Zaffanella, *Not necessarily closed convex polyhedra and the double description method*, Formal Aspects of Computing **17** (2005), no. 2, 222–257.

———, *The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems*, Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006, Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

M. Bruynooghe, *A practical framework for the abstract interpretations of logic programs*, Journal of Logic Programming **10** (1991), 91–124.

P. Cousot and R. Cousot, *Static determination of dynamic properties of programs*, Proceedings of the Second International Symposium on Programming (Paris, France) (B. Robinet, ed.), Dunod, Paris, France, 1976, pp. 106–130.

———, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages (New York), ACM Press, 1977, pp. 238–252.

———, *Static determination of dynamic properties of recursive procedures*, IFIP Conference on Formal Description of Programming Concepts (E. J. Neuhold, ed.), North-Holland, 1977, pp. 237–277.

———, *Systematic design of program analysis frameworks*, Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages (New York), ACM Press, 1979, pp. 269–282.

———, *Abstract interpretation frameworks*, Journal of Logic and Computation **2** (1992), no. 4, 511–547.

———, *Comparing the Galois connection and widening/narrowing approaches to abstract interpretation*, Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (Leuven, Belgium) (M. Bruynooghe and M. Wirsing, eds.), Lecture Notes in Computer Science, vol. 631, Springer-Verlag, Berlin, 1992, pp. 269–295.

———, *Inductive definitions, semantics and abstract interpretation*, Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages (Albuquerque, New Mexico, USA), ACM Press, 1992, pp. 83–94.

———, *Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages)*, Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages (Toulouse, France) (H. E. Bal, ed.), IEEE Computer Society Press, 1994, Invited paper, pp. 95–112.

P. Cousot and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages (Tucson, Arizona), ACM Press, 1978, pp. 84–96.

P. Cousot, *Semantic foundations of program analysis*, Program Flow Analysis: Theory and Applications (S. S. Muchnick and N. D. Jones, eds.), Prentice Hall, Englewood Cliffs, NJ, USA, 1981, pp. 303–342.

———, *The calculational design of a generic abstract interpreter*, Calculational System Design (M. Broy and R. Steinbrüggen, eds.), NATO ASI Series F. IOS Press, Amsterdam, NL, 1999.

———, *The verification grand challenge and abstract interpretation*, Verified Software: Theories, Tools, Experiments (VSTTE) (ETH Zürich, Switzerland), 2005, Position paper.

N. Dor, M. Rodeh, and S. Sagiv, *Cleanness checking of string manipulations in C programs via integer analysis*, Static Analysis: 8th International Symposium, SAS 2001 (Paris, France) (P. Cousot, ed.), Lecture Notes in Computer Science, vol. 2126, Springer-Verlag, Berlin, 2001, pp. 194–212.

M. Emami, R. Ghiya, and L. J. Hendren, *Context-sensitive interprocedural points-to analysis in the presence of function pointers*, Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (Orlando, Florida), vol. 29, ACM SIGPLAN Notices, no. 6, Association for Computing Machinery, 1994, pp. 242–256.

M. Emami, *A practical inter-procedural alias analysis for an optimizing/paralleling C compiler*, Master's thesis, School of Computer Science, McGill University, Montreal, Canada, August 1993.

D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and M. Sagiv, *Numeric domains with summarized dimensions*, Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004 (Barcelona, Spain) (K. Jensen and A. Podelski, eds.), Lecture Notes in Computer Science, vol. 2988, Springer-Verlag, Berlin, 2004, pp. 512–529.

R. Giacobazzi, S. K. Debray, and G. Levi, *A generalized semantics for constraint logic programs*, Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92) (Tokyo, Japan), ICOT, 1992, pp. 581–591.

D. Gopan, T. W. Reps, and M. Sagiv, *A framework for numeric analysis of array operations*, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Long Beach, California, USA), 2005, pp. 338–350.

N. Halbwachs, *Delay analysis in synchronous programs*, Computer Aided Verification: Proceedings of the 5th International Conference (Elounda, Greece) (C. Courcoubetis, ed.), Lecture Notes in Computer Science, vol. 697, Springer-Verlag, Berlin, 1993, pp. 333–346.

C. A. R. Hoare, *The verifying compiler: A grand challenge for computing research*, Journal of the ACM **50** (2003), no. 1, 63–69.

N. Halbwachs, Y.-E. Proy, and P. Roumanoff, *Verification of real-time systems using linear relation analysis*, Formal Methods in System Design **11** (1997), no. 2, 157–185.

B. Jeannet and W. Serwe, *Abstracting call-stacks for interprocedural verification of imperative programs*, Publication interne 1543, IRISA, Campus de Beaulieu, Rennes, France, 2003.

———, *Abstracting call-stacks for interprocedural verification of imperative programs*, Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology (Stirling, Scotland, UK) (C. Rattray, S. Maharaj, and C. Shankland, eds.), Lecture Notes in Computer Science, vol. 3116, Springer-Verlag, Berlin, 2004, pp. 258–273.

G. Kahn, *Natural semantics*, Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (Passau, Germany) (F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds.), Lecture Notes in Computer Science, vol. 247, Springer-Verlag, Berlin, 1987, pp. 22–39.

X. Leroy, *Coinductive big-step operational semantics*, Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming (Vienna, Austria) (P. Sestoft, ed.), Lecture Notes in Computer Science, vol. 3924, Springer-Verlag, Berlin, 2006, pp. 54–68.

A. Miné, *Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics*, Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (Ottawa, Ontario, Canada) (M. J. Irwin and K. De Bosschere, eds.), ACM Press, 2006, pp. 54–63.

G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, *CIL: Intermediate language and tools for analysis and transformation of C programs*, Compiler Construction: Proceedings of the 11th International Conference (CC 2002) (Grenoble, France) (R. N. Horspool, ed.), Lecture Notes in Computer Science, vol. 2304, Springer-Verlag, Berlin, 2002, pp. 213–228.

G. D. Plotkin, *A structural approach to operational semantics*, Journal of Logic and Algebraic Programming **60–61** (2004), 17–139.

D. A. Schmidt, *Natural-semantics-based abstract interpretation (preliminary version)*, Static Analysis: Proceedings of the 2nd International Symposium (Glasgow, UK) (A. Mycroft, ed.), Lecture Notes in Computer Science, vol. 983, Springer-Verlag, Berlin, 1995, pp. 1–18.

———, *Abstract interpretation of small-step semantics*, Analysis and Verification of Multiple-Agent Languages (M. Dam, ed.), Lecture Notes in Computer Science, vol. 1192, Springer-Verlag, Berlin, 1997, 5th LOMAPS Workshop Stockholm, Sweden, June 24–26, 1996, Selected Papers, pp. 76–99.

———, *Trace-based abstract interpretation of operational semantics*, LISP and Symbolic Computation **10** (1998), no. 3, 237–271.

R. Shaham, E. K. Kolodner, and S. Sagiv, *Automatic removal of array memory leaks in Java*, Proceedings of the 9th International Conference on Compiler Construction (CC 2000) (Berlin, Germany) (D. A. Watt, ed.), Lecture Notes in Computer Science, vol. 1781, Springer-Verlag, Berlin, 2000, pp. 50–66.

M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*, Program Flow Analysis: Theory and Applications (S. S. Muchnick and N. D. Jones, eds.), Prentice Hall, Englewood Cliffs, NJ, USA, 1981, pp. 189–233.

S. Sagiv, T. W. Reps, and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, ACM Transactions on Programming Languages and Systems **24** (2002), no. 3, 217–298.