A Program Calculus

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

September 25, 2008

Abstract

This document describes a theory of imperative programs, i.e. programs that operate on a system state which is modified by their execution.

For this purpose, we define the syntax and formal semantcs of a small imperative programming language, introduce judgements for reasoning about programs written in this language, and define rules for deriving true judgements. Our treatment includes variable scopes, control flow interruptions, and (also recursive) methods whose contracts are specified in the style of behavioral interface description languages by preconditions, postconditions, and frame conditions. All reasoning is modular, i.e. based on the contracts of methods rather than their implementations.

The core of the calculus on the translation of commands to logical formulas describing the state transitions allowed by the commands; this translation thus removes the "syntactic disguise" of a command and discloses its "semantic essence". The calculus supports reasoning about a program's well-formedness, partial correctness, and termination, as well as the automated construction of preconditions, postconditions, and assertions.

Contents

Chapter 1

Introduction

We present a calculus for reasoning about the properties of imperative programs and verifying their correctness with respect to their specifications. Our presentation consolidates the results presented in [14] where we have elaborated various variants of the calculus in a number of iterations that helped us to clarify our understanding. For more detailed explanations on the rationales behind the concepts presented in this document, we mainly refer the reader to that report.

Nevertheless, to make our presentation more self-contained, we repeat here the essence of the introduction of [14]:

The core idea of the corresponding program reasoning calculus is to lift the "commands as relations" principle from the meta-level (the definition of the semantics) to the object-level (the judgements of the calculus): a command/program implementation is translated to a predicate logic formula I that captures the program semantics; the specification of the command/program given by the user is also such a formula S; the implementation is correct with respect to the specification, if $I \Rightarrow S$ holds.

We believe that, independent of the actual verification, the translation of commands to logical formulas may give (after appropriate simplification) crucial insight into the behavior of a program by pushing through "the syntactic surface" of a program and disclosing its "semantic essence"; this is similar to Schmidt's approach to denotational program semantics [13] (which however uses a functional model rooted in Scott's domain theory). For this purpose, the calculus is settled in classical predicate logic (in contrast to other approaches based on e.g. dynamic logic [2]); this is the logic that (if any) most software developers are familiar with.

The idea of programs as state relations is not new: it is the core idea of the Lamport's "Temporal Logical of Actions" [10] where the individual actions of *a process are described by formulas relating pre- to post-states; Boute's "Calculational Semantics" [3] defines program behavior by program equations; related approaches are Hehner's "Practical Theory of Programming" [5] and Hoare and Jifeng's "Unifying Theories of Programming" [6]. Calculi for program refinement [1, 12, 11, 4] allow specifications as first-order language constructs at the same level as program commands with which they may be freely intermixed.*

While the present report builds upon these ideas, it has a different focus. Most of the calculi described above work on simple "while languages" that have clean and elegant calculi but neglect "messy" constructs that would complicate the calculus. Our goal, however, is to model the full richness of program structures including

- *local variable declarations (and thus commands with different scopes),*
- *commands that break the control flow (`continue`, `break`, `return`),*
- *commands that raise and handle exceptions (`throw`, `try` ... `catch`),*
- *expressions that raise exceptions (`1/0`).*

Furthermore, our calculus includes program procedures ("methods") with static scoping; it supports modular reasoning about programs on the basis of method specifications (rather than on the basis of method implementations). A core motivation of our work was to understand in depth the semantics of modern "behavioral interface specification languages" such as JML [7] or Spec# [15] (which build upon earlier specification languages such as VDM [8]) as the basis of software systems for specifying and verifying computer programs; the method specifications in the present paper are derived from these. The current version of the calculus handles most aspects of imperative programming languages with the major exception of datatypes and pointer/reference semantics semantics (programs operate on mathematical values). It does also not address object-oriented features (object methods, inheritance, overriding) or concurrency.

Since our language model is much closer to real programming languages, the rules are frequently considerably more complicated than those in the calculi presented above. However, we wanted to deal with the current programming reality "as it is" in contrast to what one might think it "should be". We also wanted to stay as close to the source language as possible and avoid translations to simple core languages (such as performed in ESC/Java2 [9]) since these tend to obfuscate the relationship between the program text accessible to the user and the ultimately constructed semantic interpretation which is used for reasoning/verification.

While relational frameworks are good for modeling "partial correctness" (no terminating computation exhibits a wrong result), they have problems with modeling "total correctness" (every computation terminates). There have been various *attempts to embed "termination" into the relational structure, e.g. including "non-termination states" ($\bot$) into the domains of the relations or by simply demanding that for correct programs all computations must terminate. We find these approaches not satisfactory and therefore treat "termination" as an issue orthogonal to (partial) correctness: every program/command is, in addition to a state relation R, specified by an accompanying state condition C: only if C is satisfied in a pre-state, the command is required to terminate (in some post-state allowed by R); the only connection between C and R is that that for every prestate on which C holds, R must allow some post-state (otherwise, the specification is inconsistent).*

While we took great effort in [14] to prove the soundness of the rules in a reasonable level of detail, the goal of the present document is to give a consolidated "reference manual" for further work (in particular the implementation of the calculus); no proofs are given at all.

With respect to the soundness of the calculus presented in this paper, intellectual integrity demands to clearly state that

1. already the results of [14] depend on a number of unproved lemmas and results from simpler models have been transferred to more general ones without complete formal re-checking,
2. moreover, the semantic framework in this paper has been further generalized by a more comprehensive treatment of recursion,
3. also references to global variables have been introduced to the programming language (in addition to the formula language),
4. furthermore, the rules have been consolidated and modified for a clearer and more concise presentation, and
5. the soundness of the translation of specification formulas to classical formulas (with respect to the preservation of validity) has not been proved.

Therefore, while our previous work gives us reason to believe that the calculus presented in this document is "essentially" sound, it may nevertheless have "small" technical errors arising from the various generalizations and changes (notwithstanding any errors that may have already plagued our previous results). Nevertheless, we feel that the foundation is now stable enough for an actual implementation in a software environment for "program exploration" by which our efforts are spent better than by starting a new round of soundness proofs; with little doubt, by this implementation technical and pragmatical problems will be detected.

Chapter 2

Programs

In this chapter, we define the abstract syntax and formal semantics of the programming language about which we are going to reason.

2.1 Example Programs

Figures 2.1 gives an example of a program written in the imperative programming language which we are going to formalize in this document. A *program* written in this language consists of a sequence of method declarations and a "main" command that is executed in the scope of the declarations; the methods and the main command are annotated by *specifications* that describe their respective behaviors.

The example program sets the global variable z to the sum of the values of the global variables x and y; it also sets x to 0 and leaves y unchanged. If the global variable x is shadowed (such as by the parameter declaration in method `add`), we still may refer to the variable by the special syntax `?`x.

The specifications of methods and of the main command consist of

- a *frame condition* "`writesonly` *Rs*" which lists all the global variables *Rs* that are changed by the execution of the method/command,
- an *exception condition* "`throwsonly` *Is*" which lists the names of all the exceptions *Is* that may be thrown by the method/command,
- a *precondition* "`requires` F" which describes by the formula F a condition which must be true in that state in which the method/command is executed (the *prestate*),

```
method add(x)
  writesonly ?x throwsonly ␣
  requires   isnat(?x) and isnat(x)
  ensures    next.executes and ?x' = 0 and next.value = ?x+x
  decreases  dummy
{
  var r = x;
  invariant  next.executes and ?x + r = ?x' + r'
  decreases  ?x
  while (?x > 0)
  (
    r = r+1;
    ?x = ?x-1
  )
  return r;
}

writesonly x,z throwsonly ␣
requires   isnat(x) and isnat(y)
ensures    next.executes and x' = 0 and z' = x+y
decreases  dummy
{
  z = add(y)
}
```

Figure 2.1: A Program

```
method prod(x,y)
  writesonly ␣ throwsonly ␣
  requires   isnat(x) and isnat(y)
  ensures    next.executes and next.value = x*y
  decreases  dummy
{ return x*y }

recursive (
  method fact0(x)
    writesonly ␣ throwsonly ␣
    requires isnat(x) ensures ...
    decreases x
  {
    if (x = 0)
      return 1
    else
      var y; var r;
      ( y=fact(x-1); r=prod(x,y); return r; )
  }

  method fact(x)
    writesonly ␣ throwsonly ␣
    requires isnat(x) ensures ...
    decreases x
  {
    if (x = 0)
      return 1
    else
      var y; var r;
      ( y=fact0(x-1); r=prod(x,y); return r; )
  }
)

writesonly y throwsonly ␣
requires   isnat(x)
ensures    next.executes and y = ...
decreases  dummy
{ y = fact(x) }
```

Figure 2.2: A Recursive Program

- a *postcondition* "`ensures` *F*" which describes by the formula *F* a relation that must hold between the prestate of the method/command and the state in which the execution of the method/command terminates (the *poststate*),
- a *termination measure* "`decreases` *T*" with a term *T* which denotes a natural number that is decreased by every recursive invocation of a method (see below, *T* is not used, if the method is not recursive).

In a postcondition, plain references *x* respectively ?*x* refer to the values of the corresponding variables in the prestate, while primed reference *x*' respectively ?*x*' refer to their values in the poststate. The formula next.executes says that the poststate of the method/command is "executing" (i.e. e.g. no exception is raised), the term next.value refers to the return value of a method after the execution of a `return` command.

Also loops may be annotated by specifications that consist of

- an *invariant* "`invariant` *F*" where the formula "F" describes a relationship between the loop's prestate and the state after every execution of the loop body.
- a *termination measure* "`decreases` *T*" where the term *T* denotes a natural number that is decreased by every iteration of the loop.

Without provision, every method may only call those methods that appear earlier in the list of method declarations (which prevents recursion). However, as shown in Figure 2.2, the construct "`recursive` *Ms*" may be used to introduce a "recursive method set" *Ms*, i.e. a set of methods that may call each member in the set, also (directly or indirectly) recursively. In order to guarantee the termination of recursive method invocations, every method in *Ms* must specify by "`decreases` *T*" a termination measure, i.e. a term *T* that denotes a natural number. If method *a* with measure T_a calls method *b* with measure T_b, the value of T_b (in the state in which *b* is called) must be less than the value of T_a (in the state in which *a* was called)[1]. The methods in the recursion set may nevertheless call those methods that were declared prior to the set without this provision.

[1]This condition is actually stronger than required. By the analysis of the concrete method call dependencies in *Ms*, infinite "recursion cycles" can be ruled out, even if not all method calls decrease the respective measures. All that is needed is that in every cycle there exists some measure that is decreased while no other measure is increased. For simplicity, the calculus presented in this paper does not make use of this generalization.

2.2 Syntax

Figure 2.3 gives the (abstract) syntax of the programming language which is used to write programs while Figure 2.4 gives the (abstract) syntax of the formula language which is used to write the corresponding program specifications.

The *programming language* models the core features of classical imperative programming languages but omits type declarations: all variables range over an unspecified domain of "values". For simplicity, there are no global variable declarations, instead we assume that every identifier denotes a distinct variable; local variable declarations (introduced by the keyword `var`) may shadow global variables. A variable reference *I* refers to the value of *I* in the current context (where *I* may denote a global variable or a local one) while `?`*I* always denotes the global variable. A method call always returns a value and may appear only as separate command that assigns the return value to a variable (i.e. no method calls are allowed within expressions). The keyword `assert` introduces an assertion; a failed assertion "blocks" the program i.e. prevents the generation of any poststate.

The *specification language* is essentially the language of first order predicate logic. Specification formulas are interpreted over *pairs* of states, typically the prestate of a command and its poststate, denoted by the constants now and next, respectively.

In formulas, references to *program variables* may appear as value constants where an unprimed reference *R* denotes the value of the program variable in the prestate and a primed reference *R*' denotes its value in the postate. If a formula shall express a condition on a single state, it is simply interpreted over a pair of identical states, i.e. *R* and *R*' are then considered as synonyms. The formula "readsonly" states that the prestate and the poststate have the same values in all variables denoted by some identifier; the formula "writesonly *Rs*" allows the prestate and the poststate to hold different values for those variables that are referenced by *Rs*.

Formulas may refer to two kinds of *logical variables*:

- Value variables of form $*I* which are bound to values and may be quantified by forall, exists, and let.
- State variables of form #*I* which are bound to states and may be quantified by allstate and exstate.

Program values may be compared by the predicates = and /=. The formulas language includes value predicates isnat and < and an unspecified set of other value predicates and value functions.

Method Language: Abstract Syntax

$P \in$ Program
$RMs \in$ RecMethods
$RM \in$ RecMethod
$Ms \in$ Methods
$M \in$ Method
$S \in$ Specification
$LS \in$ LoopSpec
$C \in$ Command
$E \in$ Expression
$R \in$ Reference
$F, G, H \in$ Formula
$T \in$ Term
$I, J, K, L \in$ Identifier

$P ::= RMs\ S\ \{C\}.$

$RMs ::= \textvisiblespace \mid RMs\ RM.$

$RM ::= M \mid$ `recursive` $Ms.$

$Ms ::= \textvisiblespace \mid Ms\ M.$

$M ::=$ `method` I `(`$I_1, \ldots, I_p$`)` S `{`C`}`.

$S ::=$ `writesonly` $R_1, \ldots, R_n$
`throwsonly` $K_1, \ldots, K_m$
`requires` F_C `ensures` F_R `decreases` $T.$

$LS ::=$ `invariant` F `decreases` $T.$

$C ::= R$ `=` $E \mid$ `var` I`;` $C \mid$ `var` I`=`E`;` $C \mid C_1$`;`C_2
$\mid$ `if (`E`)` $C \mid$ `if (`E`)` C_1 `else` C_2
$\mid$ `while (`E`)` $C \mid LS$ `while (`E`)` C
$\mid$ `continue` $\mid$ `break` $\mid$ `return` $E \mid$ `throw` $I\ E$
$\mid$ `try` C_1 `catch (`$I_k\ I_v$`)` C_2
$\mid R$ `=` I `(`$E_1, \ldots, E_p$`)` $\mid$ `assert` $F.$

$E ::= R \mid \ldots$

$R ::= I \mid ?I.$

$I ::= \ldots$

… (continued in Figure 2.4)

Figure 2.3: The Programming Language

Formula Language: Abstract Syntax

… (continued from Figure 2.3)

$F, G \in$ Formula
$T \in$ Term
$U \in$ StateTerm
$p \in$ Predicate
$f \in$ Function

$F ::=$ true | false
| $p(T_1, \ldots, T_n)$ | T_1 = T_2 | T_1 /= T_2
| readsonly | writesonly $R_1, \ldots, R_n$
| !F | F_1 and F_2 | F_1 or F_2 | F_1 => F_2 | F_1 <=> F_2
| F_1 xor F_2 | if F then F_1 else F_2 |
| forall \$$I_1, \ldots,$ \$$I_n$: F | exists \$$I_1, \ldots,$ \$$I_n$: F
| let \$$I_1$=$T_1, \ldots,$ \$$I_n$=$T_n$ in F |
| allstate #$I_1, \ldots,$ #I_n: F | exstate #$I_1, \ldots,$ #I_n: F
| U_1 == U_2
| U.executes | U.continues | U.breaks
| U.returns | U.throws | U.throws I

$T ::= R$ | R' | \$$I$ | $f(T_1, \ldots, T_n)$
| if F then T_1 else T_2
| let \$$I_1$=$T_1, \ldots,$ \$$I_n$=$T_n$ in T |
| U.value

$U ::=$ now | next | #I

$p ::=$ isnat | < | …

$f ::=$ …

Figure 2.4: The Formula Language

Program states may be compared by the predicate ==. Every state has a specific status which is indicated by the predicates .executes (normal execution), .continues (after execution of the command `continue`), .breaks (after execution of the command `break`), .returns (after execution of the command `return`), .throws (after execution of the command `throw`). After the execution of `throw` with exception type I, the state satisfies the predicate .throws I. After the execution of `return` or `throw`, a state carries a (return/exception) value which can be queried by the state function `.value`.

2.3 Semantics Overview

Before going into the details of the formal semantics of the programming language, we present in this section an overview of the general ideas.

States and Stores Our goal is to describe the behavior of programs that execute commands which modify the *state* of a system. The most important part of a state is the *store* which holds for every *variable* a *value* which may be read and updated by the command. The other part of the state are *control* data which indicate the status of the program after the execution of the command (see Figure 2.5). The corresponding domains are defined as

$$\begin{aligned}&\mathit{Store} := \mathit{Variable} \rightarrow \mathit{Value}\\&\mathit{State} := \mathit{Store} \times \mathit{Control}\end{aligned}$$

The control data of a state consist of

- a *flag* indicating the execution status (“executing” normally, “continuing” a loop body, “breaking” from a loop, “returning” from a method, or “throwing” an exception),
- a *key*, i.e. an identifier indicating the exception type (only used if the execution status is “throwing”),
- a *value* indicating the return value (if the execution status is “returning”) or the exception value (if the execution status is “throwing”).

A command always starts its execution in status “executing” but it may terminate in a state which has any execution status indicated above. The corresponding domains are defined as

$$\begin{aligned}&\mathit{Control} := \mathit{Flag} \times \mathit{Key} \times \mathit{Value}\\&\mathit{Flag} := \{\mathrm{E},\mathrm{C},\mathrm{B},\mathrm{R},\mathrm{T}\}\\&\mathit{Key} := \mathrm{Identifier}\end{aligned}$$

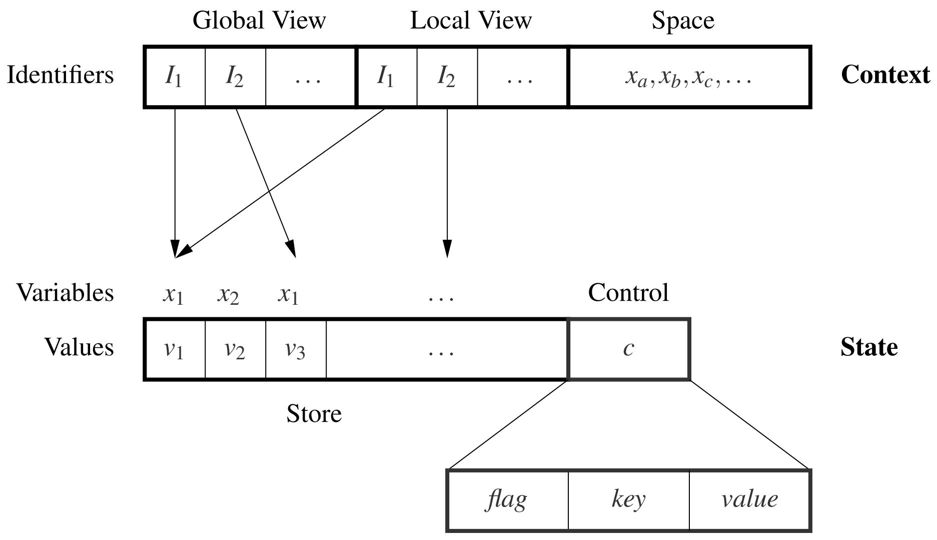

Figure 2.5: States, Stores and Contexts

State Relations and State Conditions A command consists of two parts:

- a *state relation* which describes those transitions from a prestate to a poststate that *may* be performed by the command,
- a *state condition* which describes those prestates for which the command *must* make a transition and yield a poststate.

The state relation must for every prestate included in the state condition allow at least one poststate.

The corresponding domains are

$$\begin{aligned}&\mathit{StateRelation} := \mathbb{P}(\mathit{State} \times \mathit{State})\\&\mathit{StateCondition} := \mathbb{P}(\mathit{State})\\&\mathit{Command} := \mathit{StateRelation} \times \mathit{StateCondition}\end{aligned}$$

The state relation of a command C is denoted by the expression $[\![C]\!]_d^{c,me}$ while its state condition is denoted by $\langle\!\langle C \rangle\!\rangle_d^{c,me}$; here c denotes the command's *context*, *me* its *method environment* (both see below) and $[\![\llcorner\!\lrcorner]\!]_d$ and $\langle\!\langle \llcorner\!\lrcorner \rangle\!\rangle_d$ are the *denotation functions* with signatures

$$\begin{aligned}&[\![\llcorner\!\lrcorner]\!]_d : \text{Command} \to\\&\quad(\mathit{Context} \times \mathit{MethodEnvs}) \to \mathit{StateRelation}\\&\langle\!\langle \llcorner\!\lrcorner \rangle\!\rangle_d : \text{Command} \to\\&\quad(\mathit{Context} \times \mathit{MethodEnvs}) \to \mathit{StateCondition}\end{aligned}$$

The subscript d parameterizes the semantics with respect to the behavior of "undefined" program expressions; if set to TRUE, the result are simply "undefined" values, if set to FALSE they cause the program to raise an "expression evaluation" exception.

Contexts A method's *context* determines to which variables a command has access. A context consists of (see Figure 2.5)

- a *global view* which assigns to every global identifier $?I$ a variable in the store,
- a *local view* which assigns to every local identifier I a variable in the store,
- a *space* which is a pool of variables that are not assigned to any identifiers and may be used for allocation of local variables.

Identifiers $?I$ and I may be assigned the same variables (the local view coincides with the global view) but, due to the declaration of a local variable I, they may also differ. The space does not share any variables with either the global or the local view. The corresponding domains are defined as

$$\begin{aligned}&\mathit{View} = \text{Identifier} \to \mathit{Variable}\\&\mathit{Space} = \mathbb{P}^{\infty}(\mathit{Variable})\\&\mathit{Context} = \mathit{View} \times \mathit{View} \times \mathit{Space}\end{aligned}$$

Methods A command's *method environment* determines to which methods a command has access by assigning to every identifier a *method* that can be called by the command. A method essentially is a mapping from values (the method's *arguments*) to a command (the method's *body*); the arguments are provided by the caller of the method.

Actually, a method not only depends on arguments but, for execution of its body, also on a context and a method environment. The method environment is the one which is active at the point where the method is declared while the context of the method is constructed as follows:

- the context's global view is the one which is active when the method is declared,
- the method's local view is identical to the global view,

- the method's space is the one from the context of method's caller and provided by the caller.

Since a method needs access to both the global view and the method environment active at the point of its declaration, a method environment assigns to an identifier actually not only a method but also global view and a method environment. We therefore construct the domains of method environments and method behaviors in "stages": a method environment of type $\mathit{MethodEnv}^{i+1}$ assigns to an identifier a behavior of type $\mathit{Behavior}^{i+1}$, a global view, and an environment of type $\mathit{MethodEnv}^{i}$. The corresponding declarations are (the gaps "..." are filled later):

$$\mathit{Method} := \mathit{Value}^* \to \mathit{Command}$$

$$\mathit{MethodEnvs} := \textstyle\bigcup_{i\in\mathbb{N}} \mathit{MethodEnv}^i \cup \ldots$$

$$\mathit{MethodEnv}^0 := \text{Identifier} \to \mathit{View} \times \mathit{Behavior}^0$$
$$\mathit{MethodEnv}^{i+1} := \mathit{DirMethodEnv}^i \cup \ldots$$

$$\mathit{DirMethodEnv}^i := \text{Identifier} \to \mathit{View} \times \mathit{MethodEnv}^i \times \mathit{Behavior}^{i+1}$$

$$\mathit{Behavior}^0 := \mathit{Context} \to \mathit{Method}$$
$$\mathit{Behavior}^{i+1} := \mathit{Context} \times \mathit{MethodEnv}^i \to \mathit{Method}$$

Method environments are constructed by method declarations; given a method environment of type $\mathit{MethodEnv}^i$, a method declaration M constructs an environment of type $\mathit{MethodEnv}^{i+1}$.

Recursion The situation gets more complicated, if also recursive method sets are considered. The behavior b of a recursive method is essentially modelled by an infinite sequence $b_0, b_1, \ldots$ of non-recursive method behaviors where each behavior b_{j+1} may make use of behavior b_j, i.e. b_{j+1} denotes the behavior of the recursive method with not more than j recursive invocations. The execution of a recursive method in prestate s terminates in a poststate s' if there exists *some* j such that the execution of b_j in s terminates in s'.

Given a method environment of type $\mathit{MethodEnv}^i$, the declaration of a recursive method set `recursive` *Ms* correspondingly constructs an *infinite* sequence of method environments of types $\mathit{RecMethodEnv}^i_j$ (for each $j \in \mathbb{N}$) where an environment of type $\mathit{RecMethodEnv}^i_j$ holds methods of type $\mathit{RecBehavior}^i_j$ that allow less than j recursive invocations. The corresponding definitions are as follows:

Definitions: Variables and Values

$Variable := \ldots$
$Value := \mathbb{B} \cup \mathbb{N} \cup \ldots$

$Predicate := \mathbb{P}(Value^*)$
$Function := Value^* \rightarrow Value$

$\perp := \textsc{such}\ v : v \notin Value$
$Value_\perp := Value \cup \{\perp\}$

Figure 2.6: Variables and Values

$$MethodEnvs := \bigcup_{i \in \mathbb{N}} MethodEnv^i \cup \bigcup_{j \in \mathbb{N}} RecMethodEnv^i_j$$

$$MethodEnv^{i+1} := DirMethodEnv^i \ \cup \ RecMethodEnv^i$$

$RecMethodEnv^i_0 :=$
  $\text{Identifier} \rightarrow View \times MethodEnv^i \times RecBehavior^i_0$
$RecMethodEnv^i_{j+1} :=$
  $\text{Identifier} \rightarrow View \times RecMethodEnv^i_j \times RecBehavior^i_{j+1}$

$RecBehavior^i_0 := Context \times MethodEnv^i \rightarrow Method$
$RecBehavior^i_{j+1} := Context \times RecMethodEnv^i_j \rightarrow Method$

In the following section, we will give the full definitions of the semantic algebras implementing the ideas sketched above.

2.4 Semantic Algebras

Figures 2.6–2.17 introduce the semantic algebras, i.e. the domains (aka “types”) and associated operations, on which the formal model of the programming language depend. See Appendix A for information on the mathematical language in which these definitions are written.

Definitions: Contexts

$View = \text{Identifier} \rightarrow Variable$
$Space = \mathbb{P}^{\infty}(Variable)$
$Context = View \times View \times Space$

$gview : Context \rightarrow View, gview(v_g, v_l, s) = v_g$
$lview : Context \rightarrow View, lview(v_g, v_l, s) = v_l$
$space : Context \rightarrow Space, space(v_g, v_l, s) = s$

$context : View \times View \times Space \rightarrow Context$
$context(v_g, v_l, s) = \langle v_g, v_l, s \rangle$

$call : View \times Space \rightarrow Context$
$call(v, s) = context(v, v, s)$

$range : Context \rightarrow \mathbb{P}(Variable)$
$range(c) = \text{range}(gview(c)) \cup \text{range}(lview(c)) \cup space(c)$

$take : Space \rightarrow (Variable \times Space)$
$take(s) = \text{LET } x = \text{SUCH } x : x \in s \text{ IN } \langle x, s \backslash \{x\} \rangle$

$push : Context \times \text{Identifier} \rightarrow Context$
$push(c, I) =$
    $\text{LET } \langle x, s' \rangle = take(space(c)) \text{ IN}$
    $context(gview(c), lview(c)[I \mapsto x], s')$

$push(c, I_1, \ldots, I_n) \equiv push(\ldots push(c, I_1) \ldots, I_n)$

Figure 2.7: Contexts

Definitions: States

$State := Store \times Control$

$store : State \to Store,\ store(s,c) = s$
$control : State \to Control,\ control(s,c) = c$
$state : Store \times Control \to State,\ state(s,c) = \langle s,c \rangle$

$StateFunction := State \to Value$
$StateFunction_\perp := State \to Value_\perp$
$BinaryStateFunction := State \times State \to Value$
$ControlFunction := State \times State \to Control$

Figure 2.8: States

Definitions: Stores

$Store := Variable \to Value$

$read : State \times Variable \to Value$
$read(s,x) = store(s)(x)$

$write : State \times Variable \times Value \to State$
$write(s,x,v) = state(store(s)[x \mapsto v], control(s))$

$writes(s,x_1,v_1,\ldots,x_n,v_n) \equiv write(\ldots write(s,x_1,v_1)\ldots,x_n,v_n)$

$s = s'$ EXCEPT $V \equiv$
$\quad \forall v \in Variable : read(s,v) \neq read(s',v) \Rightarrow v \in V$

s EQUALS^c $s' \equiv$
$\quad s = s'$ EXCEPT $space(c)$

$s = s'$ EXCEPT^c $Rs \equiv$
$\quad s = s'$ EXCEPT $space(c) \cup \{[\![R]\!]^c : R \in Rs\}$

Figure 2.9: Stores

Definitions: Control Data

$Flag := \{\mathrm{E}, \mathrm{C}, \mathrm{B}, \mathrm{R}, \mathrm{T}\}$
$Key :=$ Identifier
$Control := Flag \times Key \times Value$

$flag : Control \rightarrow Flag, flag(f,k,v) = f$
$key : Control \rightarrow Key, key(f,k,v) = k$
$value : Control \rightarrow Value, value(f,k,v) = v$

$cont : Flag \times Key \times Value \rightarrow Control$
$cont(f,k,v) = \langle f,k,v \rangle$

$execute : State \rightarrow State$
$execute(s) =$
  $\text{LET } c = control(s) \text{ IN } state(store(s), cont(\mathrm{E}, key(c), value(c)))$
$continue : State \rightarrow State$
$continue(s) =$
  $\text{LET } c = control(s) \text{ IN } state(store(s), cont(\mathrm{C}, key(c), value(c)))$
$break : State \rightarrow State$
$break(s) =$
  $\text{LET } c = control(s) \text{ IN } state(store(s), cont(\mathrm{B}, key(c), value(c)))$
$return : State \times Value \rightarrow State$
$return(s,v) =$
  $\text{LET } c = control(s) \text{ IN } state(store(s), cont(\mathrm{R}, key(c), v))$
$throw : State \times Key \times Value \rightarrow State$
$throw(s,k,v) =$
  $\text{LET } c = control(s) \text{ IN } state(store(s), cont(\mathrm{T}, k, v))$

$executes : \mathbb{P}(Control), executes(c) \Leftrightarrow flag(c) = \mathrm{E}$
$continues : \mathbb{P}(Control), continues(c) \Leftrightarrow flag(c) = \mathrm{C}$
$breaks : \mathbb{P}(Control), breaks(c) \Leftrightarrow flag(c) = \mathrm{B}$
$returns : \mathbb{P}(Control), returns(c) \Leftrightarrow flag(c) = \mathrm{R}$
$throws : \mathbb{P}(Control), throws(c) \Leftrightarrow flag(c) = \mathrm{T}$

Figure 2.10: Control Data

Definitions: Expression Evaluation Exceptions

$EXP := \text{SUCH } k : \ k \in Key$
$VAL := \text{SUCH } v : \ v \in Value$

$expthrow : State \to State$
$expthrow(s) = throw(s, EXP, VAL)$

$expthrows : \mathbb{P}(Control)$
$expthrows(c) \Leftrightarrow throws(c) \wedge key(c) = EXP$

Figure 2.11: Expression Evaluation Exceptions

Definitions: Commands and Methods

$StateRelation := \mathbb{P}(State \times State)$
$StateCondition := \mathbb{P}(State)$
$Command := StateRelation \times StateCondition$

$rel : Command \to StateRelation, rel(r, c) = r$
$cond : Command \to StateCondition, cond(r, c) = c$

$command : StateRelation \times StateCondition \to Command$
$command(r, c) = \langle r, c \rangle$

$Method := Value^* \to Command$

Figure 2.12: Commands and Methods

Definitions: Iteration

$$\begin{aligned}
&\mathit{iterate} \subseteq \mathbb{N} \times \mathit{State}^{\infty} \times \mathit{State}^{\infty} \times \times \mathit{StateFunction}_{\perp} \times \mathit{StateRelation}\\
&\mathit{iterate}(i,t,u,E,C) \Leftrightarrow\\
&\quad \neg \mathit{breaks}(\mathit{control}(u(i))) \wedge \mathit{executes}(\mathit{control}(t(i))) \wedge\\
&\quad E(t(i)) = \text{TRUE} \wedge C(t(i),u(i+1)) \wedge\\
&\quad \text{IF } \mathit{continues}(\mathit{control}(u(i+1))) \vee \mathit{breaks}(\mathit{control}(u(i+1)))\\
&\qquad \text{THEN } t(i+1) = \mathit{execute}(u(i+1))\\
&\qquad \text{ELSE } t(i+1) = u(i+1)
\end{aligned}$$

$$\begin{aligned}
&\mathit{leaves} : \mathbb{P}(\mathit{State})\\
&\mathit{leaves}(s) \Leftrightarrow \neg \mathit{executes}(s) \wedge \neg \mathit{continues}(s)
\end{aligned}$$

Figure 2.13: Iteration

Definitions: Environments

$$\begin{aligned}
&\mathit{ValueEnv} := \text{Identifier} \rightarrow \mathit{Value}\\
&\mathit{ControlEnv} := \text{Identifier} \rightarrow \mathit{Control}\\
&\mathit{Environment} := \mathit{ValueEnv} \times \mathit{ControlEnv}
\end{aligned}$$

$$\begin{aligned}
&\mathit{venv} : \mathit{Environment} \rightarrow \mathit{ValueEnv},\ \mathit{venv}(v,c) = v\\
&\mathit{cenv} : \mathit{Environment} \rightarrow \mathit{ControlEnv},\ \mathit{cenv}(v,c) = c\\
&\mathit{env} : \mathit{ValueEnv} \times \mathit{ControlEnv} \rightarrow \mathit{Environment},\ \mathit{env}(v,c) = \langle v,c \rangle
\end{aligned}$$

$$\begin{aligned}
&e[I_1 \mapsto v_1, \ldots, I_n \mapsto v_n]_{\mathrm{v}} \equiv\\
&\quad \mathit{env}(\mathit{venv}(e)[I_1 \mapsto v_1, \ldots, I_n \mapsto v_n], \mathit{cenv}(e))\\
&e[I_1 \mapsto c_1, \ldots, I_n \mapsto c_n]_{\mathrm{c}} \equiv\\
&\quad \mathit{env}(\mathit{venv}(e), \mathit{cenv}(e)[I_1 \mapsto c_1, \ldots, I_n \mapsto c_n])
\end{aligned}$$

Figure 2.14: Environments

Definitions: Method Environments and Behaviors

$MethodEnvs := \bigcup_{i\in\mathbb{N}} MethodEnv^i \cup \bigcup_{j\in\mathbb{N}} RecMethodEnv^i_j$

$DirMethodEnvs := \bigcup_{i\in\mathbb{N}} DirMethodEnv^i$
$RecMethodEnvs := \bigcup_{i\in\mathbb{N}} RecMethodEnv^i$

$MethodEnv^0 := \text{Identifier} \rightarrow View \times Behavior^0$
$MethodEnv^{i+1} := DirMethodEnv^i \;\cup\; RecMethodEnv^i$

$DirMethodEnv^i := \text{Identifier} \rightarrow View \times MethodEnv^i \times Behavior^{i+1}$

$RecMethodEnv^i_0 :=$
$\quad \text{Identifier} \rightarrow View \times MethodEnv^i \times RecBehavior^i_0$
$RecMethodEnv^i_{j+1} :=$
$\quad \text{Identifier} \rightarrow View \times RecMethodEnv^i_j \times RecBehavior^i_{j+1}$

$Behavior^0 := Context \rightarrow Method$
$Behavior^{i+1} := Context \times MethodEnv^i \rightarrow Method$

$RecBehavior^i_0 := Context \times MethodEnv^i \rightarrow Method$
$RecBehavior^i_{j+1} := Context \times RecMethodEnv^i_j \rightarrow Method$

Figure 2.15: Method Environments and Behaviors

Definitions: Method Access

```
$recmethod_{i,j}$ :
    $RecMethodEnv^i_j \times \text{Identifier} \times Context \rightarrow Method$
$remethod_{i,j}(me, I_m, c) =$
    LET $\langle v, me', b\rangle = me(I_m), c' = call(v, space(c))$ IN $b^{c',me'}$

$recmethod_i : RecMethodEnv^i \times \text{Identifier} \times Context \rightarrow Method$
$recmethod_i(me, I_m, c)(v_1, \ldots, v_p) =$
    LET
        $R : StateRelation$
        $R(s, s') \Leftrightarrow$
            $\exists j \in \mathbb{N}$ :
                LET $com = recmethod_{i,j}(me_j, I_m, c)$ IN
                $rel(com)(v_1, \ldots, v_p)(s, s')$
        $C : StateCondition$
        $C(s) \Leftrightarrow$
            $\exists j \in \mathbb{N}$ :
                LET $com = recmethod_{i,j}(me_j, I_m, c)$ IN
                $cond(com)(v_1, \ldots, v_p)(s)$
    IN $command(R, C)$

$method : MethodEnvs \times \text{Identifier} \times Context \rightarrow Method$
$method(me, I_m, c) \Leftrightarrow$
    LET $sp = space(c)$ IN
    IF $me \in MethodEnv^0$ THEN
        LET $\langle v, b\rangle = me(I_m), c' = call(v, sp)$ IN $b^{c'}$
    ELSE IF $me \in DirMethodEnvs$ THEN
        LET $\langle v, me', b\rangle = me(I_m), c' = call(v, sp)$ IN $b^{c',me'}$
    ELSE IF $me \in RecMethodEnvs$ THEN
        LET $i :=$ SUCH $i \in \mathbb{N} : me \in RecMethodEnv^i$ IN
        $recmethod_i(me, I_m, c)$
    ELSE
        LET $i, j :=$ SUCH $i, j \in \mathbb{N} : me \in RecMethodEnv^i_j$ IN
        $recmethod_{i,j}(me, I_m, c)$
```

Figure 2.16: Method Access

Definitions: Method Environment Construction

```
$behavior_i : MethodEnv_i \times \text{Identifier} \rightarrow Behavior_{i+1}$
$behavior_i(me, I_m) =$
    LET
        $b : Context \times MethodEnv^i \rightarrow Method$
        $b^{c,me'} = method(me, I_m, c)$
    IN $b$

$ebase_i : View \times MethodEnv^i \times \mathbb{P}(\text{Identifier}) \rightarrow RecMethodEnv^i_0$
$ebase_i(v, me, Is)(I_m) =$
    LET
        $b : RecBehavior^i_0$
        $b^{c,me}(v_1, \ldots, v_p) =$
            IF $I_m \in Is$
                THEN $\langle \emptyset, \emptyset \rangle$
                ELSE $method(me, I_m, c)(v_1, \ldots, v_p)$
    IN $\langle v, me, b \rangle$

$enext_{i,j} : View \times RecMethodEnv^i_j \times \rightarrow RecMethodEnv^i_{j+1}$
$enext_{i,j}(v, me)(I_m) =$
    LET
        $b : RecBehavior^i_{j+1}$
        $b^{c,me} = recmethod_{i,j}(me, I_m, c)$
    IN $\langle v, me, b \rangle$
```

Figure 2.17: Method Environment Construction

2.5 Program Semantics

Figure 2.18 defines the semantics associated to the top-level syntactic domains of the programming language:

Program The valuation function takes a global view v and a space sp and constructs from this a context c in which the method definitions RMs are evaluated. These definitions take a method environment me of type $MethodEnv^i$ and return a method environment me' of type $MethodEnv^j$. The state relation $[\![C]\!]_d^{c,me'}$ and state condition $\langle\!\langle C \rangle\!\rangle_d^{c,me'}$ are used to construct the semantics of the method body.

RecMethods The valuation function takes a method environment me of type $MethodEnv^i$ and return a method environment of type $MethodEnv^j$ (provided that the "RecMethods" argument consists of $j-i$ "RecMethod" components).

RecMethod Given a non-recursive method M, a method environment me of type $MethodEnv^i$, and a global view v, the valuation function constructs a "direct method environment" of type $DirMethodEnv^i$ which maps a method identifier I_m,

- if I_m denotes M, to a triple of v, me, and a "direct semantics" $[\![M]\!]_{d,i}$ of M, and
- if I_m denotes another method, to a triple of v, me, and the behavior of the method, which is looked up in me as an object of type $Behavior_i$ and "lifted" to type $Behavior_{i+1}$.

Given a "recursive method set" Ms and me respectively v as above, the valuation function constructs a "recursive method environment" of type $RecMethodEnv^i$; this environment actually represents an infinite sequence of environments:

- the element at position 0 in this sequence is constructed by the function *ebase* such that it looks up the behavior of every method which is *not* in Ms in me and lifts it to type $RecBehavior_0^i$ and maps every method which is in Ms to the empty behavior (representing a non-terminating recursive call),
- the element at position $j+1$ in this sequence is constructed from the element me' at position j by

Program Semantics

$[\![\llcorner]\!]_{d,i,j}$: **Program** $\rightarrow$ ***View*** $\times$ ***Space*** $\times$ ***MethodEnv***^{i} $\rightarrow$ ***Command***

$[\![RMs\ S\ \{C\}]\!]^{v,sp}_{d,i,j}(me) =$
$\quad \text{LET } c = call(v, sp), me' = [\![RMs]\!]^v_{d,i,j}(me) \text{ IN}$
$\quad command([\![C]\!]^{c,me'}_d, \langle\!\langle C \rangle\!\rangle^{c,me'}_d)$

$[\![\llcorner]\!]_{d,i,j}$: **RecMethods** $\rightarrow$ ***View*** $\times$ ***MethodEnv***^{i} $\rightarrow$ ***MethodEnv***^{j}

$[\![\llcorner]\!]^v_{d,i,i}(me) = me$

$[\![RMs\ RM]\!]^v_{d,i,j+1}(me) = [\![RM]\!]^v_{d,j}([\![RMs]\!]^v_{d,i,j}(me))$

$[\![\llcorner]\!]_{d,i}$: **RecMethod** $\rightarrow$ ***View*** $\times$ ***MethodEnv***^{i} $\rightarrow$ ***MethodEnv***$^{i+1}$

```
$[\![M]\!]^v_{d,i}(me)(I_m) =$
    IF $I_m = [\![M]\!]_{\mathrm{I}}$
        THEN $\langle v, me, [\![M]\!]_{d,i} \rangle$
        ELSE $\langle v, me, behavior_i(me, I_m) \rangle$
```

$[\![\texttt{recursive}\ Ms]\!]^v_{d,i}(me)_0 = ebase_i(v, me, [\![Ms]\!]_{\mathrm{I}})$

```
$[\![\texttt{recursive}\ Ms]\!]^v_{d,i}(me)_{j+1} =$
    LET $me' = [\![\texttt{recursive}\ Ms]\!]^v_{d,i}(me)_j$ IN
    $[\![Ms]\!]^v_{d,i,j}(me', enext_{d,i,j}(v, me'))$
```

$[\![\llcorner]\!]_{d,i,j}$: **Methods** $\rightarrow$
$\quad$ ***View*** $\times$ (***RecMethodEnv***^i_j $\times$ ***RecMethodEnv***$^i_{j+1}$) $\rightarrow$
$\quad$ ***RecMethodEnv***$^i_{j+1}$

$[\![\llcorner]\!]^v_{d,i,j}(me, me') = me'$

$[\![Ms\ M]\!]^v_{d,i,j}(me, me') = [\![M]\!]^v_{d,i,j}(me, me')(me, [\![Ms]\!]^v_{d,i,j}(me, me'))$

$[\![\llcorner]\!]_{d,i,j}$: **Method** $\rightarrow$
$\quad$ ***View*** $\times$ (***RecMethodEnv***^i_j $\times$ ***RecMethodEnv***$^i_{j+1}$) $\rightarrow$
$\quad$ ***RecMethodEnv***$^i_{j+1}$

$[\![M]\!]^v_{d,i,j}(me, me') = me'[[\![M]\!]_{\mathrm{I}} \mapsto \langle v, me, [\![M]\!]_{d,i,j} \rangle]$

Figure 2.18: Program Semantics

Method Semantics

$\llbracket \_ \rrbracket_{\mathrm{I}}$: **Methods** $\rightarrow \mathbb{P}($**Identifier**$)$

$\llbracket \_ \rrbracket_{\mathrm{I}} := \emptyset$

$\llbracket Ms\; M \rrbracket_{\mathrm{I}} := \llbracket Ms \rrbracket_{\mathrm{I}} \cup \{\llbracket M \rrbracket_{\mathrm{I}}\}$

$\llbracket \_ \rrbracket_{\mathrm{I}}$: **Method** $\rightarrow$ **Identifier**

$\llbracket \texttt{method}\, I_m\, \texttt{(}J_1,\ldots,J_p\texttt{)}\;\; S\; \texttt{\{}C\texttt{\}} \rrbracket_{\mathrm{I}} := I_m$

$\llbracket \_ \rrbracket_{d,i}$: **Method** $\rightarrow$ ***Behavior***$^{i+1}$

```
$\llbracket \texttt{method}\, I_m\, \texttt{(}J_1,\ldots,J_p\texttt{)}\;\; S\; \texttt{\{}C\texttt{\}} \rrbracket^{c,me}_{d,i,j}(v_1,\ldots,v_p) =$
  LET
    $c' = push(c, J_1, \ldots, J_p)$
    $r \in StateRelation$
    $r(s,s') \Leftrightarrow$
      $\exists s_0, s_1 : State :$
        $s_0 = writes(s, \llbracket J_1 \rrbracket^{c'}, v_1, \ldots, \llbracket J_p \rrbracket^{c'}, v_p)\; \wedge$
        $\llbracket C \rrbracket^{c',me}_{d}(s_0, s_1)\; \wedge$
        $s' =$ IF $throws(control(s_1))$
              THEN $s_1$ ELSE $executes(s_1)$
    $t \in StateCondition$
    $t(s) \Leftrightarrow$
      LET $s_0 = writes(s, \llbracket J_1 \rrbracket^{c'}, v_1, \ldots, \llbracket J_p \rrbracket^{c'}, v_p)$ IN
      $\langle\!\langle C \rangle\!\rangle^{c',me}_{d}(s_0)$
  IN $command(r,t)$
```

$\llbracket \_ \rrbracket_{d,i,j}$: **Method** $\rightarrow$ ***RecBehavior***$^{i}_{j+1}$

$\llbracket \texttt{method}\, I_m\, \texttt{(}J_1,\ldots,J_p\texttt{)}\;\; S\; \texttt{\{}C\texttt{\}} \rrbracket^{c,me}_{d,i,j}(v_1,\ldots,v_p) = \ldots$ (as above)

Figure 2.19: Method Semantics

1. first building a "start environment" by the application of function *enext* which looks up the behavior of every method in me' as an object of type $RecBehavior^i_j$ and lifts it to a behavior of type $RecBehavior^i_{j+1}$, and
2. then updating this environment by every declaration in Ms.

Methods The valuation function updates the method environment me' of type $RecMethodEnv^i_{j+1}$ by every method which is declared in environment me of type $RecMethodEnv^i_j$.

Method The valuation function updates environment me' by a mapping of the identifier of method M to a triple of the global view v, method environment me, and the behavior of M.

The core of Figure 2.19 are the two last valuation functions which construct the direct respectively recursive behavior of a method:

- the first constructs a direct behavior of type $Behavior^{i+1}$ which takes a context and a method environment me of type $MethodEnv^i$ and returns a method,
- the second constructs a recursive behavior of type $Behavior^i_{j+1}$ which takes a context and a method environment me of type $RecMethodEnv^i_j$ and returns a method.

Both valuation functions are defined in a syntactically identical way, since the command valuation functions $[\![\_]\!]$ and $\langle\!\langle \_ \rangle\!\rangle$ accept method environments of type *MethodEnvs* which encompasses both cases.

2.6 Command Semantics

Figures 2.20–2.25 describe the core of the program semantics: the formalization of a command as a pair of a state relation and and a state condition which represent the possible state transitions and the prestates with guaranteed termination, respectively. The various kinds of commands are formalized based on the following intuitions:

Assignment First, the expression E is evaluated. If this yields an undefined value ($\perp$), the command raises an exception, otherwise, the variable denoted by the identifier is updated by the value. The command always terminates.

Command Semantics

```
$[\![ \_ ]\!]_d$ : **Command** $\rightarrow$
    $(\textit{Context} \times \textit{MethodEnvs}) \rightarrow \textit{StateRelation}$
$\langle\!\langle \_ \rangle\!\rangle_d$ : **Command** $\rightarrow$
    $(\textit{Context} \times \textit{MethodEnvs}) \rightarrow \textit{StateCondition}$

$[\![ R = E ]\!]_d^{c,me}(s, s') \Leftrightarrow$
    LET $v = [\![ E ]\!]_d^c(s)$ IN
    IF $v = \perp$
        THEN $s' = \textit{expthrow}(s)$
        ELSE $s' = \textit{write}(s, [\![ R ]\!]^c, \textit{val})$
$\langle\!\langle R = E \rangle\!\rangle_d^{c,me}(s) \Leftrightarrow$ TRUE

$[\![ \texttt{var } I\texttt{; } C ]\!]_d^{c,me}(s, s') \Leftrightarrow$
    LET $c' = \textit{push}(c, I)$ IN $[\![ C ]\!]_d^{c',me}(s, s')$
$\langle\!\langle \texttt{var } I\texttt{; } C \rangle\!\rangle_d^{c,me}(s) \Leftrightarrow$
    LET $c' = \textit{push}(c, I)$ IN $\langle\!\langle C \rangle\!\rangle_d^{c',me}(s)$

$[\![ \texttt{var } I\texttt{=}E\texttt{; } C ]\!]_d^{c,me}(s, s') \Leftrightarrow$
    LET $v = [\![ E ]\!]_d^c(s)$ IN
    IF $v = \perp$ THEN
        $s' = \textit{expthrow}(s)$
    ELSE
        LET $c_0 = \textit{push}(c, I), s_0 = \textit{write}(s, [\![ I ]\!]^{c_0}, v)$
        IN $[\![ C ]\!]_d^{c_0,me}(s_0, s')$
$\langle\!\langle \texttt{var } I\texttt{=}E\texttt{; } C \rangle\!\rangle_d^{c,me}(s) \Leftrightarrow$
    LET $v = [\![ E ]\!]_d^c(s)$ IN
    IF $v = \perp$ THEN
        TRUE
    ELSE
        LET $c_0 = \textit{push}(c, I), s_0 = \textit{write}(s, [\![ I ]\!]^{c_0}, v)$
        IN $\langle\!\langle C \rangle\!\rangle_d^{c_0,me}(s_0)$
```

Figure 2.20: Command Semantics (1/5)

Command Semantics (Contd)

```
$[\![ C_1 \texttt{;} C_2 ]\!]_d^{c,me}(s,s') \Leftrightarrow$
    $\exists s_0 \in State:$
        $[\![ C_1 ]\!]_d^{c,me}(s,s_0) \wedge$
        IF $executes(control(s_0))$
            THEN $[\![ C_2 ]\!]_d^{c,me}(s_0,s')$
            ELSE $s' = s_0$
$\langle\!\langle C_1 \texttt{;} C_2 \rangle\!\rangle_d^{c,me}(s) \Leftrightarrow$
    $\langle\!\langle C_1 \rangle\!\rangle_d^{c,me}(s) \wedge$
    $\forall s_0 \in State:$
        $[\![ C_1 ]\!]_d^{c,me}(s,s_0) \wedge executes(control(s_0)) \Rightarrow \langle\!\langle C_2 \rangle\!\rangle_d^{c,me}(s_0)$

$[\![ \texttt{if (}E\texttt{) } C ]\!]_d^{c,me}(s,s') \Leftrightarrow$
    LET $v = [\![ E ]\!]_d^c(s)$ IN
    IF $v = \perp$ THEN
        $s' = expthrow(s)$
    ELSE IF $v =$ TRUE THEN
        $[\![ C ]\!]_d^{c,me}(s,s')$
    ELSE
        $s' = s$
$\langle\!\langle \texttt{if (}E\texttt{) } C \rangle\!\rangle_d^{c,me}(s) \Leftrightarrow$
    $[\![ E ]\!]_d^c(s) =$ TRUE $\Rightarrow \langle\!\langle C \rangle\!\rangle_d^{c,me}(s)$

$[\![ \texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 ]\!]_d^{c,me}(s,s') \Leftrightarrow$
    LET $v = [\![ E ]\!]_d^c(s)$ IN
    IF $v = \perp$ THEN
        $s' = expthrow(s)$
    ELSE IF $v =$ TRUE THEN
        $[\![ C_1 ]\!]_d^{c,me}(s,s')$
    ELSE
        $[\![ C_2 ]\!]_d^{c,me}(s,s')$
$\langle\!\langle \texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 \rangle\!\rangle_d^{c,me}(s) \Leftrightarrow$
    LET $v = [\![ E ]\!]_d^c(s)$ IN
    IF $v = \perp$
        THEN TRUE
        ELSE IF $v =$ TRUE THEN $\langle\!\langle C_1 \rangle\!\rangle_d^{c,me}(s)$ ELSE $\langle\!\langle C_2 \rangle\!\rangle_d^{c,me}(s)$
```

Figure 2.21: Command Semantics (2/5)

Variable Declaration A new context c' is created by introducing a local variable I. The declaration body C is executed in c'. The command terminates, if the body does.

Variable Definition First, the expression E is evaluated. If this yields an undefined value ($\perp$), the command raises an exception. Otherwise, a new context c' is created by introducing a local variable I and the variable denoted by I is updated by the value. The definition body C is executed in c'. The command terminates, if the body does.

Command Sequence The first command C_1 is executed yielding an intermediate state s_0. If this state is not executing, it immediately represents the sequence's poststate. Otherwise, the second command C_2 is executed with s_0 as its prestate which yields the sequence's poststate. The command terminates, if C_1 terminates and if C_2 terminates in every executing poststate of C_1.

One-Sided Conditional First, the expression E is evaluated. If this yields an undefined value ($\perp$), the command raises an exception. Otherwise, if the expression value is TRUE, the branch C is executed. The command terminates, if C terminates in every state in which E yields TRUE.

Two-Sided Conditional First, the expression E is evaluated. If this yields an undefined value ($\perp$), the command raises an exception. Otherwise, if the value is TRUE, the first branch C_1 is executed, and else the second branch C_2 is executed. The command terminates, if E yields $\perp$, or if E yields TRUE and C_1 terminates, or if E yields neither $\perp$ nor TRUE and C_2 terminates.

While Loop The behavior of a while loop is determined by two sequences of states t and u both starting with the loop's prestate s (see Figure 2.23). The execution of the loop body C transforms every $t(i)$ to $u(i+1)$ from which the prestate $t(i+1)$ of the next iteration is constructed by setting the status from "continuing" or "breaking" to "executing". The loop is terminated with state $t(k)$ if $u(k)$ indicates a "leaving" (not executing and not continuing) state or if the loop expression E does not yield TRUE in $t(k)$; if the loop expression yields an undefined value, the poststate s' is constructed from $t(k)$ by throwing an exception, otherwise s' equals $t(k)$. The loop terminates if for every prestate $t(k)$ of the loop body C the execution of C terminates and if no infinite sequence of iterations arises.

Continue The statement sets the poststate to "continuing" and terminates.

Break The statement sets the poststate to "breaking" and terminates.

Command Semantics (Contd)

$$
\begin{array}{l}
[\![\texttt{while (}E\texttt{)}\ C]\!]_d^{c,me}(s,s') \Leftrightarrow \\
\quad \exists k \in \mathbb{N}, t,u \in State^\infty : \\
\qquad t(0) = s \wedge u(0) = s \wedge \\
\qquad (\forall i \in \mathbb{N}_k : iterate(i,t,u,s,[\![E]\!]_d^c,[\![C]\!]_d^{c,me})) \wedge \\
\qquad (leaves(u(k)) \vee [\![E]\!]_d^c(t(k)) \neq \text{TRUE}) \wedge \\
\qquad \text{IF } leaves(u(k)) \vee [\![E]\!]_d^c(t(k)) \neq \bot \\
\qquad\quad \text{THEN } s' = t(k) \\
\qquad\quad \text{ELSE } s' = expthrow(t(k)) \\
\langle\!\langle \texttt{while (}E\texttt{)}\ C\rangle\!\rangle_d^{c,me}(s) \Leftrightarrow \\
\quad \forall t,u \in State^\infty : t(0) = s \wedge u(0) = s \Rightarrow \\
\qquad (\forall k \in \mathbb{N} : \\
\qquad\quad (\forall i \in \mathbb{N}_k : iterate(i,t,u,s,[\![E]\!]_d^c,[\![C]\!]_d^{c,me})) \wedge \\
\qquad\quad \neg leaves(u(k)) \wedge [\![E]\!]_d^c(t(k)) = \text{TRUE} \Rightarrow \\
\qquad\qquad \langle\!\langle C\rangle\!\rangle_d^{c,me}(t(k))) \wedge \\
\qquad \neg(\forall i \in \mathbb{N} : iterate(i,t,u,s,[\![E]\!]_d^c,[\![C]\!]_d^{c,me})) \wedge
\end{array}
$$

$[\![LS\ \texttt{while (}E\texttt{)}\ C]\!]_d^{c,me}(s,s') \Leftrightarrow \ldots$ (as above)

$\langle\!\langle LS\ \texttt{while (}E\texttt{)}\ C\rangle\!\rangle_d^{c,me}(s) \Leftrightarrow \ldots$ (as above)

Figure 2.22: Command Semantics (3/5)

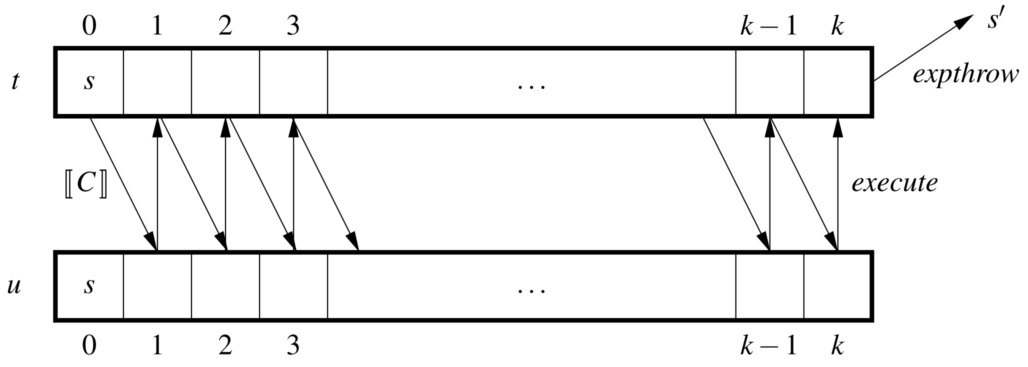

Figure 2.23: While Loop

Command Semantics (Contd)

$[\![\texttt{continue}]\!]^{c,me}_{d}(s,s') \Leftrightarrow s' = \mathit{continue}(s)$
$\langle\!\langle\texttt{continue}\rangle\!\rangle^{c,me}_{d}(s) \Leftrightarrow \text{TRUE}$

$[\![\texttt{break}]\!]^{c,me}_{d}(s,s') \Leftrightarrow s' = \mathit{break}(s)$
$\langle\!\langle\texttt{break}\rangle\!\rangle^{c,me}_{d}(s) \Leftrightarrow \text{TRUE}$

```
$[\![\texttt{return}\ E]\!]^{c,me}_{d}(s,s') \Leftrightarrow$
    LET $v = [\![E]\!]^{c}_{d}(s)$ IN
    IF $v = \perp$
        THEN $s' = \mathit{expthrow}(s)$
        ELSE $s' = \mathit{return}(s,v)$
$\langle\!\langle\texttt{return}\ E\rangle\!\rangle^{c,me}_{d}(s) \Leftrightarrow \text{TRUE}$

$[\![\texttt{throw}\ I\ E]\!]^{c,me}_{d}(s,s') \Leftrightarrow$
    LET $v = [\![E]\!]^{c}_{d}(s)$ IN
    IF $v = \perp$
        THEN $s' = \mathit{expthrow}(s)$
        ELSE $s' = \mathit{throw}(s,I,c)$
$\langle\!\langle\texttt{throw}\ I\ E\rangle\!\rangle^{c,me}_{d}(s) \Leftrightarrow \text{TRUE}$

$[\![\texttt{try}\ C_1\ \texttt{catch}\ \texttt{(}I_k\ I_v\texttt{)}\ C_2]\!]^{c,me}_{d}(s,s') \Leftrightarrow$
    $\exists s_0, s_1 \in \mathit{State}:$
        $[\![C_1]\!]^{c,me}_{d}(s,s_0)\ \wedge$
        IF $\mathit{throws}(\mathit{control}(s_0)) \wedge \mathit{key}(\mathit{control}(s_0)) = I_k$ THEN
            LET $c_0 = \mathit{push}(c, I_v)$ IN
            $s_1 = \mathit{write}(\mathit{execute}(s_0), [\![I_v]\!]^{c_0}, \mathit{value}(\mathit{control}(s_0)))\ \wedge$
            $[\![C_2]\!]^{c_0,me}_{d}(s_1,s')$
        ELSE $s' = s_0$
$\langle\!\langle\texttt{try}\ C_1\ \texttt{catch}\ \texttt{(}I_k\ I_v\texttt{)}\ C_2\rangle\!\rangle^{c,me}_{d}(s) \Leftrightarrow$
    $\langle\!\langle C_1\rangle\!\rangle^{c,me}_{d}(s) \wedge$
    $\forall s_0 \in \mathit{State}:$
        $[\![C_1]\!]^{c,me}_{d}(s,s_0)\ \wedge$
        $\mathit{executes}(\mathit{control}(s_0)) \wedge \mathit{key}(\mathit{control}(s_0)) = I_k\ \Rightarrow$
            LET $c_0 = \mathit{push}(c, I_v)$ IN
            $s_1 = \mathit{write}(\mathit{execute}(s_0), [\![I_v]\!]^{c_0}, \mathit{value}(\mathit{control}(s_0)))\ \wedge$
            $\langle\!\langle C_2\rangle\!\rangle^{c_0,me}_{d}(s_1)$
```

Figure 2.24: Command Semantics (4/5)

Command Semantics (Contd)

```
$[\![ R = I_m\ (E_1, \ldots, E_p)\ ]\!]^{c,me}_d(s, s') \Leftrightarrow$
    LET $v_1 = [\![ E_1 ]\!]^c_d(s), \ldots, v_p = [\![ E_p ]\!]^c_d(s)$ IN
    IF $v_1 = \bot \vee \ldots \vee v_p = \bot$ THEN
        $s' = expthrow(s)$
    ELSE
        LET $r = rel(method(me, I_m, c)(v_1, \ldots, v_p))$ IN
        $\exists s_0 \in State :$
            $r(s, s_0) \wedge$
            IF $throws(control(s_0))$
                THEN $s' = s_0$
                ELSE $s' = write(s_0, [\![ R ]\!]^c, value(control(s_0)))$
$\langle\!\langle R = I_m\ (E_1, \ldots, E_p)\ \rangle\!\rangle^{c,me}_d(s) \Leftrightarrow$
    LET $v_1 = [\![ E_1 ]\!]^c_d(s), \ldots, v_p = [\![ E_p ]\!]^c_d(s)$ IN
    IF $v_1 = \bot \vee \ldots \vee v_p = \bot$ THEN
        $s' = expthrow(s)$
    ELSE
        LET $t = cond(method(me, I_m, c)(v_1, \ldots, v_p))$ IN $t(s)$
```

$[\![\texttt{assert}\ F]\!]^{c,me}_d(s, s') \Leftrightarrow [\![F]\!]^c_e(s, s) \wedge s' = s$

$\langle\!\langle \texttt{assert}\ F \rangle\!\rangle^{c,me}_d(s) \Leftrightarrow [\![F]\!]^c_e(s, s)$

Figure 2.25: Command Semantics (5/5)

Return First, the expression E is evaluated. If this yields an undefined value ($\perp$), the command raises an exception. Otherwise, the statement sets the poststate to “continuing” and the control value to the value of E and terminates.

Throw First, the expression E is evaluated. If this yields an undefined value ($\perp$), the command raises an exception. Otherwise, the statement sets the poststate to “throwing”, the control data’s key to the indicated identifier, and the control data’s value to the expression value.

Exception Handler The first command C_1 is executed yielding an intermediate state s_0. If s_0 does not throw an exception of type I_k, it immediately represents the handler’s poststate. Otherwise, a new context c_0 is created with the handler’s parameter I_v as a local variable. A new executing state s_1 is created from s_0 by updating this variable with the exception value of s_0. Then C_2 is executed with s_1 as prestate which yields the handler’s poststate. The command terminates, if C_1 terminates and if C_2 terminates in every prestate s_1 that can be derived from any poststate s_0 of C_1 as indicated above.

Method Call First, the argument expressions $E_1, \ldots, E_p$ are evaluated. If any of the evaluations yields an undefined value ($\perp$), the command raises an exception. Otherwise, the method environment *me* is looked up for the method identified by I_m. This method is applied to the argument values which yields a command behavior from which the transition relation is extracted. The application of this transition relation to the prestate gives a poststate s_0; if s_0 throws an exception, it already represents the postate of the command. Otherwise s_0 is updated by writing into the variable referenced by R the method’s return value. The method terminates, if the evaluation of some argument expression yields an undefined value, or if the command behavior’s termination condition indicates termination.

Assertion If the condition holds, the command does not change the state. Otherwise, the command blocks.

2.7 Expression Semantics

The core semantics of an expression E is defined in two parts (see Figure 2.26): Given a context c and a state s,

- the value semantics $[\![E]\!]_{\mathrm{V}}$ maps E to a value.
- the definedness condition $[\![E]\!]_{\mathrm{D}}$ tells, whether this value makes sense or not.

Expression Semantics

$[\![\_]\!]$: **Expression** $\rightarrow$ ***Context*** $\rightarrow \mathbb{B} \rightarrow$ ***StateFunction***$_\perp$
$[\![E]\!]^c_d(s) = \text{IF } d \vee [\![E]\!]^c_{\mathrm{D}}(s) \text{ THEN } [\![E]\!]^c_{\mathrm{V}}(s) \text{ ELSE } \perp$

$[\![\_]\!]_{\mathrm{D}}$: **Expression** $\rightarrow$ ***Context*** $\rightarrow$ ***StateCondition***
$[\![R]\!]^c_{\mathrm{D}}(s) \Leftrightarrow \text{TRUE}$
$[\![\ldots]\!]^c_{\mathrm{D}}(s) \Leftrightarrow \ldots$

$[\![\_]\!]_{\mathrm{V}}$: **Expression** $\rightarrow$ ***Context*** $\rightarrow$ ***StateFunction***
$[\![R]\!]^c_{\mathrm{V}}(s) = \mathit{read}(s, [\![R]\!]^c)$
$[\![\ldots]\!]^c_{\mathrm{V}}(s) = \ldots$

$[\![\_]\!]$: **Reference** $\rightarrow$ ***Context*** $\rightarrow$ ***Variable***
$[\![I]\!]^c = \mathit{lview}(c)(I)$
$[\![?I]\!]^c = \mathit{gview}(c)(I)$

Figure 2.26: Expression Semantics

The overall semantics $[\![E]\!]_d$ then depends on a boolean value d:

- If d is true, the value of $[\![E]\!]_{\mathrm{V}}$ is unconditionally forwarded;
- If d is false, then $[\![E]\!]_{\mathrm{V}}$ is only returned if $[\![E]\!]_{\mathrm{D}}$ says that this makes sense; otherwise, the special value $\perp$ (“undefined value”) is returned.

In the second case, a command that evaluates the expression transforms an undefined value to an “expression evaluation exception” in the command’s postate (see the previous section).

2.8 Formula and Term Semantics

Figures 2.27 and 2.28 introduce the semantics of specification formulas while Figure 2.29 defines the semantics of terms within these formulas. Here we should especially note the differences in the semantics of references to program variables in the prestate respectively poststate (*R* and *R*'), the semantics of logical value variables ($*I*), and the semantics of logical state variables (#*I*).

Formula Semantics

$[\![\ \_\ ]\!]$: **Formula** $\rightarrow$ ***Context*** $\times$ ***Environment*** $\rightarrow$ ***StateRelation***

$[\![\mathsf{true}]\!]^c_e(s,s') \Leftrightarrow \text{TRUE}$

$[\![\mathsf{false}]\!]^c_e(s,s') \Leftrightarrow \text{FALSE}$

$[\![p(T_1,\ldots,T_n)]\!]^c_e(s,s') \Leftrightarrow [\![p]\!]([\![T_1]\!]^c_e(s,s'),\ldots,[\![T_n]\!]^c_e(s,s'))$

$[\![T_1 \mathsf{=} T_2]\!]^c_e(s,s') \Leftrightarrow [\![T_1]\!]^c_e(s,s') = [\![T_2]\!]^c_e(s,s')$

$[\![T_1 \mathsf{/=} T_2]\!]^c_e(s,s') \Leftrightarrow [\![T_1]\!]^c_e(s,s') \neq [\![T_2]\!]^c_e(s,s')$

$[\![\mathsf{readsonly}]\!]^c_e(s,s') \Leftrightarrow s \text{ EQUALS}^c\ s'$

$[\![\mathsf{writesonly}\ R_1,\ldots,R_n]\!]^c_e(s,s') \Leftrightarrow$
$\quad s = s' \text{ EXCEPT}^c\ R_1,\ldots,R_n$

$[\![\mathsf{!}F]\!]^c_e(s,s') \Leftrightarrow \neg[\![F]\!]^c_e(s,s')$

$[\![F_1\ \mathsf{and}\ F_2]\!]^c_e(s,s') \Leftrightarrow [\![F_1]\!]^c_e(s,s') \wedge [\![F_2]\!]^c_e(s,s')$

$[\![F_1\ \mathsf{or}\ F_2]\!]^c_e(s,s') \Leftrightarrow [\![F_1]\!]^c_e(s,s') \vee [\![F_2]\!]^c_e(s,s')$

$[\![F_1\ \mathsf{=>}\ F_2]\!]^c_e(s,s') \Leftrightarrow [\![F_1]\!]^c_e(s,s') \Rightarrow [\![F_2]\!]^c_e(s,s')$

$[\![F_1\ \mathsf{<=>}\ F_2]\!]^c_e(s,s') \Leftrightarrow [\![F_1]\!]^c_e(s,s') \Leftrightarrow [\![F_2]\!]^c_e(s,s')$

$[\![F_1\ \mathsf{xor}\ F_2]\!]^c_e(s,s') \Leftrightarrow [\![F_1]\!]^c_e(s,s') \not\Leftrightarrow [\![F_2]\!]^c_e(s,s')$

$[\![\mathsf{if}\ F\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2]\!]^c_e(s,s') \Leftrightarrow$
$\quad \text{IF } [\![F]\!]^c_e(s,s') \text{ THEN } [\![F_1]\!]^c_e(s,s') \text{ ELSE } [\![F_2]\!]^c_e(s,s')$

$[\![\mathsf{forall}\ \$I_1,\ldots,\$I_n\mathsf{:}\ F]\!]^c_e(s,s') \Leftrightarrow$
$\quad \forall v_1,\ldots,v_n \in Value : [\![F]\!]^c_{e[I_1\mapsto v_1,\ldots,I_n\mapsto v_n]_\mathrm{v}}(s,s')$

$[\![\mathsf{exists}\ \$I_1,\ldots,\$I_n\mathsf{:}\ F]\!]^c_e(s,s') \Leftrightarrow$
$\quad \exists v_1,\ldots,v_n \in Value : [\![F]\!]^c_{e[I_1\mapsto v_1,\ldots,I_n\mapsto v_n]_\mathrm{v}}(s,s')$

$[\![\mathsf{let}\ \$I_1\mathsf{=}T_1,\ldots,\$I_n\mathsf{=}T_n\ \mathsf{in}\ F]\!]^c_e(s,s') \Leftrightarrow$
$\quad \text{LET}$
$\qquad e_1 = e[I_1 \mapsto [\![T_1]\!]^c_e(s,s')]_\mathrm{v}$
$\qquad \ldots$
$\qquad e_n = e_{n-1}[I_n \mapsto [\![T_n]\!]^c_{e_{n-1}}(s,s')]_\mathrm{v}$
$\quad \text{IN } [\![F]\!]^c_{e_n}(s,s')$

... (continued in Figure 2.28)

Figure 2.27: Formula Semantics (1/2)

Formula Semantics (Contd)

$[\![\,\llcorner\!\lrcorner\,]\!]$: **Formula** $\rightarrow$ ***Context*** $\times$ ***Environment*** $\rightarrow$ ***StateRelation***

… (continued from Figure 2.27)

$[\![\,\text{allstate}\ \#I_1,\ldots,\#I_n\colon F\,]\!]^c_e(s,s') \Leftrightarrow$
$\quad \forall c_1,\ldots,c_n \in Control : [\![\,F\,]\!]^c_{e[I_1\mapsto c_1,\ldots,I_n\mapsto c_n]_c}(s,s')$
$[\![\,\text{exstate}\ \#I_1,\ldots,\#I_n\colon F\,]\!]^c_e(s,s') \Leftrightarrow$
$\quad \exists c_1,\ldots,c_n \in Control : [\![\,F\,]\!]^c_{e[I_1\mapsto c_1,\ldots,I_n\mapsto c_n]_c}(s,s')$
$[\![\,U_1 \texttt{==} S_2\,]\!]^c_e(s,s') \Leftrightarrow [\![\,U_1\,]\!]^c_e(s,s') = [\![\,U_2\,]\!]^c_e(s,s')$
$[\![\,U.\text{executes}\,]\!]^c_e(s,s') \Leftrightarrow executes([\![\,U\,]\!]^c_e(s,s'))$
$[\![\,U.\text{continues}\,]\!]^c_e(s,s') \Leftrightarrow continues([\![\,U\,]\!]^c_e(s,s'))$
$[\![\,U.\text{breaks}\,]\!]^c_e(s,s') \Leftrightarrow breaks([\![\,U\,]\!]^c_e(s,s'))$
$[\![\,U.\text{returns}\,]\!]^c_e(s,s') \Leftrightarrow returns([\![\,U\,]\!]^c_e(s,s'))$
$[\![\,U.\text{throws}\,]\!]^c_e(s,s') \Leftrightarrow throws([\![\,U\,]\!]^c_e(s,s'))$
$[\![\,U.\text{throws}\ I\,]\!]^c_e(s,s') \Leftrightarrow$
$\quad \text{LET } c' = [\![\,U\,]\!]^c_e(s,s') \text{ IN } throws(c') \wedge key(c') = I$

$[\![\,\llcorner\!\lrcorner\,]\!]$: **Predicate** $\rightarrow$ ***Predicate***

$[\![\,\text{isnat}\,]\!](v) \Leftrightarrow v \in \mathbb{N}$

$[\![\,\texttt{<}\,]\!](v_1,v_2) \Leftrightarrow v_1 < v_2$

$[\![\,\llcorner\!\lrcorner\,]\!](v_1,\ldots,v_n) \Leftrightarrow \ldots$

Figure 2.28: Formula Semantics (2/2)

Term Semantics

$[\![\,\llcorner\!\lrcorner\,]\!]$: **Term** $\rightarrow$ ***Context*** $\times$ ***Environment*** $\rightarrow$ ***BinaryStateFunction***

$[\![R]\!]^c_e(s,s') = read(s,[\![R]\!]^c)$
$[\![R\text{'}]\!]^c_e(s,s') = read(s',[\![R]\!]^c)$
$[\![\$I]\!]^c_e(s,s') = venv(e)(I)$
$[\![f(T_1,\ldots,T_n)]\!]^c_e(s,s') = [\![f]\!]([\![T_1]\!]^c_e(s,s'),\ldots,[\![T_n]\!]^c_e(s,s'))$
$[\![\text{if } F \text{ then } T_1 \text{ else } T_2]\!]^c_e(s,s') =$
    $\text{IF } [\![F]\!]^c_e(s,s') \text{ THEN } [\![T_1]\!]^c_e(s,s') \text{ ELSE } [\![T_2]\!]^c_e(s,s')$
$[\![\text{let } \$I_1\text{=}T_1,\ldots,\$I_n\text{=}T_n \text{ in } T]\!]^c_e(s,s') \Leftrightarrow$

```
LET
    $e_1 = e[I_1 \mapsto [\![T_1]\!]^c_e(s,s')]_{\mathrm{v}}$
    ...
    $e_n = e_{n-1}[I_n \mapsto [\![T_n]\!]^c_{e_{n-1}}(s,s')]_{\mathrm{v}}$
IN $[\![T]\!]^c_{e_n}(s,s')$
```

$[\![U.\text{value}]\!]^c_e(s,s') = value([\![U]\!]^c_e(s,s'))$

$[\![\,\llcorner\!\lrcorner\,]\!]$: **State** $\rightarrow$ ***Context*** $\times$ ***Environment*** $\rightarrow$ ***ControlFunction***

$[\![\text{now}]\!]^c_e(s,s') = control(s)$
$[\![\text{next}]\!]^c_e(s,s') = control(s')$
$[\![\#I]\!]^c_e(s,s') = cenv(e)(I)$

$[\![\,\llcorner\!\lrcorner\,]\!]$: **Function** $\rightarrow$ ***Function***

$[\![\ldots]\!](v_1,\ldots,v_n) = \ldots$

Figure 2.29: Term Semantics

Chapter 3

Judgements

In this chapter, we define the syntax and formal semantics of program judgements which we are going to derive in our calculus.

3.1 Syntax

Figure 3.1 lists the syntax of the judgements that we are going to deal with. Their informal interpretations are:

$se \vdash RMs\ S\ \{C\}$ states that for a given specification environment *se* which describes "predefined" methods, the program is correct with respect to the specifications of its own methods and its main command. In particular, the execution of the program

1. does not encounter the evaluation of "undefined expressions",
2. does not change any variables protected by the frame conditions,
3. does not throw any exceptions prohibited by the exception conditions,
4. does not exhibit any state transitions that violate the postconditions,
5. terminates in those states that are demanded by the preconditions.

$se \vdash RMs$ states that above holds for all methods declared in *RMs*.

$se \vdash RM$ states that above holds for all methods declared in *RM*.

$se, Is \vdash Ms$ states that above holds for all methods declared in method set *Ms* provided that *Ms* is part of a "recursive method set" consisting of those methods named in *Is*.

List of Judgements

$se \vdash RMs\ S\ \{C\}$

$se \vdash RMs$
$se \vdash RM$

$se, Is \vdash Ms$
$se, Is \vdash M$
$se \vdash M$

$se, Is, Vs \vdash C \checkmark F$
$se, Is, Vs \vdash C : F$
$se, Is, Vs \vdash C \downarrow^{I} F$

$se, Is, Vs \vdash \textsc{pre}(C, Q) = P$
$se, Is, Vs \vdash \textsc{post}(C, P) = Q$
$se, Is, Vs \vdash \textsc{assert}(C, P) = C'$

$F\ \S\ {}^{Ks}_{Rs}$
$F\ \S_{\mathrm{c}}\ F_c$
$F\ \S_{\mathrm{b}}\ F_b$
$F\ \S_{\mathrm{s}}\ {}^{Qs}_{Rs}$
$F\ \S_{\mathrm{e}}\ {}^{Ls}_{Ks}$
$\models_{Rs} F$

Figure 3.1: List of Judgements

$se, Is \vdash s$ states that above holds for method M provided that M is contained in a "recursive method set" consisting of those methods named in *Is*.

$se \vdash M$ states that above holds for M provided that M is non-recursive.

$se, Is, Vs \vdash C \checkmark F$ states that in a state that satisfies F the execution of C does not encounter the evaluation of "undefined expressions", provided that

1. *se* represents a correct specification environment for the methods visible to C,
2. *Is* is empty and C occurs in a non-recursive method or *Is* is not empty and C occurs in a method that is part of a recursive method set consisting of those methods named in *Is*,
3. C occurs in the scope of those local variables that are identified by *Vs*.

$se, Is, Vs \vdash C : F$ states that the state relation of command C is described by the formula F, provided that the same constraints concerning se, Is, Vs hold that were described above.

$se, Is, Vs \vdash C \downarrow^I F$ states that in a state that satisfies formula F the execution of command C terminates, provided that the same constraints concerning se, Is, Vs hold that were described above.

$se, Is, Vs \vdash \text{PRE}(C, Q) = P$ and $se, Is, Vs \vdash \text{POST}(C, P) = Q$ state the same[1]: if command C is executed in a prestate in which formula P holds it only gives rise to such a poststate in which formula Q holds.

$se, Is, Vs \vdash \text{ASSERT}(C, P) = C'$ assert states that if command C is executed in a state in which formula P holds, it can be substituted by command C' without difference to the behavior of the program.

$F \S^{Ks}_{Rs}$ states that any transition relation satisfying formula F ensures that only the program variables referenced by *Rs* are changed and that no other exceptions are thrown than those listed in *Ks*.

$F \S_{\mathbf{c}} F_c$ states that any transition relation satisfying formula F ensures that the poststate is "continuing" only if formula F_c is true.

$F \S_{\mathbf{c}} F_b$ states that any transition relation satisfying formula F ensures that the poststate is "breaking" only if formula F_b is true.

$F \S_{\mathbf{s}}{}^{Qs}_{Rs}$ ensures that any transition relation satisfying formula F and only changing those variables listed in *Qs* only changes those variables listed in *Rs*.

[1]However, there will be different rules for the two judgements.

$F \ \S_{\mathbf{e}}\, {}^{Ls}_{Ks}$ ensures that any transition relation satisfying formula F and only throwing those exceptions listed in *Ls* only throws those exceptions listed in *Ks*.

$\models_{Rs} F$ ensures that F is true for every prestate and poststate that differ visibly only by those variables referenced in *Rs*.

3.2 Semantic Algebras

Figures 3.2 to 3.11 introduce new semantic domains and operations which are needed to formalize the semantics of the judgements stated in the previous section. The core definitions are those of the following predicates:

- *bspecifies* (Figure 3.7) describes the correctness property for a pre-defined "base" method (i.e. a method with a behavior of type $Behavior^0$),
- $dspecifies_i$ (Figure 3.7) describes the correctness property for a direct (non-recursive) method defined in the program at stage i (i.e. a method with a behavior of type $Behavior^{i+1}$),
- $rspecifies_{i,j}$ (Figure 3.8) describes the correctness property for a recursive method defined in the program at stage i and with a recursion depth bound j (i.e. for a behavior of type $RecBehavior^i_j$).

Based on these definitions, the following predicates express the correctness of method environments with respect to specification environments:

- $specifies_i$ (Figure 3.10) states that a method environment *me* at stage i is well specified with respect to specification environment *se* in context c;
- $rspecifies_{i,j}$ (Figure 3.9) states that a method environment *me* at stage i and with recursion depth bound j (where identifiers *Is* denote the recursive methods) is well specified with respect to specification environment *se* in context c.

Finally, the predicates

- *wellspecified* (Figures 3.9 and 3.10) and
- *iscorrect* (Figure 3.11)

describe the semantics of the judgements as stated in the following section.

Definitions: Contexts

$isvalid \subseteq View \times Space$
$isvalid(v, s) \Leftrightarrow$
$\quad \forall I_1, I_2 \in \text{Identifier} : I_1 \neq I_2 \Rightarrow v(I_1) \neq v(I_2) \wedge$
$\quad \forall I \in \text{Identifier} : v(I) \notin s$

$isvalid \subseteq Context$
$isvalid(c) \Leftrightarrow$
$\quad \text{LET } s = space(c) \text{ IN}$
$\quad isvalid(gview(c), s) \wedge isvalid(lview(c), s)$

$global \subseteq Context$
$global(c) \Leftrightarrow isvalid(c) \wedge gview(c) = lview(c)$

$c_0 = c_1 \text{ EXCEPT } I_1, \ldots, I_n \equiv$
$\quad \forall I \in \text{Identifier} : I \neq I_1 \wedge \ldots \wedge I \neq I_n \Rightarrow [\![I]\!]^{c_0} = [\![I]\!]^{c_1}$

$c_0 = c_1 \text{ AT } I_1, \ldots, I_n \ :\equiv$
$\quad \forall I \in \text{Identifier} : I = I_1 \vee \ldots \vee I = I_n \Rightarrow [\![I]\!]^{c_0} = [\![I]\!]^{c_1}$

$c_0 \text{ EQUALS } c_1 \equiv$
$\quad \forall I \in \text{Identifier} : [\![I]\!]^{c_0} = [\![I]\!]^{c_1}$

$contexts \subseteq Context \times Context$
$contexts(c_g, c_l) \Leftrightarrow$
$\quad global(c_g) \wedge isvalid(c_l) \wedge gview(c_g) = gview(c_l) \wedge$
$\quad space(c_l) \subseteq space(c_g)$

$contexts \subseteq Context \times Context \times \text{Identifier}^*$
$contexts(c_g, c_l, Is) \Leftrightarrow$
$\quad contexts(c_g, c_l) \wedge$
$\quad c_g = c_l \text{ EXCEPT } Is \wedge \{[\![I]\!]^{c_l} : I \in Is\} \subseteq space(c_g)$

Figure 3.2: Definitions: Contexts

Definitions: Formulas and Terms

F is closed $\equiv$
$$\forall c \in \mathit{Context}, e_0, e_1 \in \mathit{Environment}, s, s' \in \mathit{State} : \llbracket F \rrbracket^c_{e_0}(s, s') \Leftrightarrow \llbracket F \rrbracket^c_{e_1}(s, s')$$

F does not depend on the poststate $\equiv$
$$\forall c \in \mathit{Context}, e \in \mathit{Environment}, s, s_0, s_1 \in \mathit{State} : \llbracket F \rrbracket^c_{e}(s, s_0) \Leftrightarrow \llbracket F \rrbracket^c_{e}(s, s_1)$$

T is closed $\equiv$
$$\forall c \in \mathit{Context}, e_0, e_1 \in \mathit{Environment}, s, s' \in \mathit{State} : \llbracket T \rrbracket^c_{e_0}(s, s') = \llbracket T \rrbracket^c_{e_1}(s, s')$$

T does not depend on the poststate $\equiv$
$$\forall c \in \mathit{Context}, e \in \mathit{Environment}, s, s_0, s_1 \in \mathit{State} : \llbracket T \rrbracket^c_{e}(s, s_0) = \llbracket T \rrbracket^c_{e}(s, s_1)$$

F makes $\$I$ a natural number $\equiv$
$$\forall e \in \mathit{Environment}, c \in \mathit{Context}, s, s' \in \mathit{State} : \llbracket F \rrbracket^c_{e}(s, s') \Rightarrow e(I) \in \mathbb{N}$$

F makes T a natural number $\equiv$
$$\forall e \in \mathit{Environment}, c \in \mathit{Context}, s, s' \in \mathit{State} : \llbracket F \rrbracket^c_{e}(s, s') \Rightarrow \llbracket T \rrbracket^c_{e}(s, s') \in \mathbb{N}$$

Figure 3.3: Definitions: Formulas and Terms

Definitions: Method Specifications

$\mathit{Spec} := \text{Reference}^* \times \text{Identifier}^* \times \text{Formula} \times \text{Formula} \times \text{Term}$
$\mathit{MethodSpec} := \text{Identifier}^* \times \mathit{Spec}$
$\mathit{SpecEnv} := \text{Identifier} \to \mathit{MethodSpec}$

$\mathit{spec}:$
$\quad \text{Reference}^* \times \text{Identifier}^* \times \text{Formula} \times \text{Formula} \times \text{Term} \to \mathit{Spec}$
$\mathit{spec}(Rs, Ks, F_C, F_R, T) = \langle Rs, Ks, F_C, F_R, T \rangle$

$\mathit{pre} : \mathit{Spec} \to \mathit{Formula}$
$\mathit{pre}(Rs, Ks, F_C, F_R, T) = F_C$

$\mathit{mspec} : \text{Identifier}^* \times \mathit{Spec} \to \mathit{MethodSpec}$
$\mathit{mspec}(Is, S) = \langle Is, S \rangle$

$\mathit{wellformed} \subseteq \mathit{MethodSpec}$
$\mathit{wellformed}(\langle (J_1, \ldots, J_p), \langle Rs, Ks, F_C, F_R, T \rangle \rangle) \Leftrightarrow$
$\quad |\{J_1, \ldots, J_p\}| = p \wedge$
$\quad \{J_1, \ldots, J_p\} \cap Rs = \emptyset \wedge$
$\quad F_C, F_R$ and T are closed $\wedge$
$\quad F_C$ and T do not depend on the poststate $\wedge$
$\quad F_C$ makes T a natural number

$\mathit{wellformed} \subseteq \text{LoopSpec}$
$\mathit{wellformed}(\texttt{invariant}\ F\ \texttt{decreases}\ T) \Leftrightarrow$
$\quad F$ and T are closed $\wedge$
$\quad T$ does not depend on the poststate $\wedge$
$\quad F$ makes T a natural number

Figure 3.4: Definitions: Method Specifications (1/2)

Definitions: Method Specifications (Contd)

$$
\begin{aligned}
&[\![\,\llcorner\!\lrcorner\,]\!]^{*} : \textbf{RecMethods} \rightarrow \\
&\qquad (\textbf{RecMethod} \times \textbf{SpecEnv})^{*} \times \textbf{SpecEnv} \rightarrow \\
&\qquad (\textbf{RecMethod} \times \textbf{SpecEnv})^{*} \times \textbf{SpecEnv}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\,\llcorner\!\lrcorner\,]\!]^{*}(s, se) = \langle s, se\rangle \\
&[\![RMs\ RM]\!]^{*}(s, se) = \\
&\quad \text{LET } \langle s', se'\rangle = [\![RMs]\!]^{*}(s, se), se'' = [\![RM]\!]_{\mathrm{S}}(se') \text{ IN} \\
&\quad \langle s'[\,\text{LENGTH}(s') \mapsto \langle RM, se'\rangle\,], se''\rangle
\end{aligned}
$$

$$
[\![\,\llcorner\!\lrcorner\,]\!]_{\mathrm{S}} : \textbf{RecMethods} \rightarrow \textbf{\textit{SpecEnv}} \rightarrow \textbf{\textit{SpecEnv}}
$$

$$
\begin{aligned}
&[\![\,\llcorner\!\lrcorner\,]\!]_{\mathrm{S}}(se) = se \\
&[\![RMs\ RM]\!]_{\mathrm{S}}(se) = [\![RM]\!]_{\mathrm{S}}([\![RMs]\!]_{\mathrm{S}}(se))
\end{aligned}
$$

$$
[\![\,\llcorner\!\lrcorner\,]\!]_{\mathrm{S}} : \textbf{RecMethod} \rightarrow \textbf{\textit{SpecEnv}} \rightarrow \textbf{\textit{SpecEnv}}
$$

$$
\begin{aligned}
&[\![M]\!]_{\mathrm{S}}(se) = [\![M]\!]_{\mathrm{S}}(se) \\
&[\![\texttt{recursive}\ Ms]\!]_{\mathrm{S}}(se) = [\![Ms]\!]_{\mathrm{S}}(se)
\end{aligned}
$$

$$
[\![\,\llcorner\!\lrcorner\,]\!]_{\mathrm{I}} : \textbf{RecMethod} \rightarrow \mathbb{P}(\textbf{Identifier})
$$

$$
\begin{aligned}
&[\![M]\!]_{\mathrm{I}} = \{[\![M]\!]_{\mathrm{I}}\} \\
&[\![\texttt{recursive}\ Ms]\!]_{\mathrm{I}} = [\![Ms]\!]_{\mathrm{I}}
\end{aligned}
$$

Figure 3.5: Definitions: Method Specifications (2/3)

Definitions: Method Specifications (Contd)

$[\![\llcorner\!\lrcorner]\!]_{\mathrm{S}}$: **Methods** $\rightarrow$ ***SpecEnv*** $\rightarrow$ ***SpecEnv***

$[\![\llcorner\!\lrcorner]\!]_{\mathrm{S}}(se) = se$

$[\![Ms\ M]\!]_{\mathrm{S}}(se) = [\![M]\!]_{\mathrm{S}}([\![Ms]\!]_{\mathrm{S}}(se))$

$[\![\llcorner\!\lrcorner]\!]_{\mathrm{S}}$: **Method** $\rightarrow$ ***SpecEnv*** $\rightarrow$ ***SpecEnv***

$[\![\texttt{method}\ I_m\ \texttt{(}J_1,\ldots,J_p\texttt{)}\ \ S\ \texttt{\{}C\texttt{\}}]\!]_{\mathrm{S}}(se) =$
$\quad se[I_m \mapsto [\![\texttt{method}\ I_m\ \texttt{(}J_1,\ldots,J_p\texttt{)}\ \ S\ \texttt{\{}C\texttt{\}}]\!]_{\mathrm{S}}]$

$[\![\llcorner\!\lrcorner]\!]_{\mathrm{S}}$: **Method** $\rightarrow$ ***MethodSpec***

$[\![\texttt{method}\ I_m\ \texttt{(}J_1,\ldots,J_p\texttt{)}\ \ S\ \texttt{\{}C\texttt{\}}]\!]_{\mathrm{S}} = mspec((J_1,\ldots,J_p),[\![S]\!])$

$[\![\llcorner\!\lrcorner]\!]$: **Specification** $\rightarrow$ ***Spec***

$[\![\texttt{writesonly}\ R_1,\ldots,R_n\texttt{,throwsonly}\ K_1,\ldots,K_m$
$\quad \texttt{requires}\ F_C\ \texttt{ensures}\ F_R\ \texttt{decreases}\ T]\!] =$
$\quad spec((R_1,\ldots,R_n),(K_1,\ldots,K_m),F_C,F_R,T)$

Figure 3.6: Definitions: Method Specifications (3/3)

Definitions: Method Correctness

$bspecifies \subseteq SpecEnv \times MethodEnv^0 \times Context$
$bspecifies(se,me,c) \Leftrightarrow \forall I_m \in \text{Identifier} : bspecifies(se,me,c,I_m)$

$bspecifies \subseteq SpecEnv \times MethodEnv^0 \times Context \times \text{Identifier}$
$bspecifies(se,me,c,I_m) \Leftrightarrow$
$\quad wellformed(se(I_m)) \wedge bspecifies(se(I_m),me,c,I_m)$

$bspecifies \subseteq MethodSpec \times MethodEnv^0 \times Context \times \text{Identifier}$
$bspecifies(\langle(J_1,\ldots,J_p),\langle Rs,Ks,F_C,F_R,T\rangle\rangle,me,c,I_m) \Leftrightarrow$

```
    $\forall c' \in Context, s, s' \in State, v_1,\ldots,v_p \in Value, e \in Environment :$
        $contexts(c,c') \wedge executes(control(s)) \;\Rightarrow$
            LET
                $com = method(me,I_m,c')(v_1,\ldots,v_p),$
                $c'' = push(c',J_1,\ldots,J_p),$
                $s_1 = writes(s,J_1,v_1,\ldots,J_p,v_p)^{c''}$
            IN
                $(rel(com)(s,s') \;\Rightarrow$
                    $([\![F_C]\!]^{c''}_e(s_1,s') \Rightarrow [\![F_R]\!]^{c''}_e(s_1,s')) \wedge$
                    $s = s'$ EXCEPT $range(c') \cup \text{range}(gview(c)) \wedge$
                    $s = s'$ EXCEPT$^c$ $Rs \wedge$
                    $(executes(control(s')) \vee throws(control(s'))) \wedge$
                    $(throws(control(s')) \Rightarrow key(control(s')) \in Ks)) \wedge$
                $([\![F_C]\!]^{c''}_e(s_1,s_1) \Rightarrow cond(com)(s))$
```

$dspecifies_i \subseteq SpecEnv \times DirMethodEnv^i \times Context$
$dspecifies_i(se,me,c) \Leftrightarrow \forall I_m \in \text{Identifier} : dspecifies_i(se,me,c,I_m)$

$dspecifies_i \subseteq SpecEnv \times DirMethodEnv^i \times Context \times \text{Identifier}$
$dspecifies_i(se,me,c,I_m) \Leftrightarrow$
$\quad wellformed(se(I_m)) \wedge dspecifies_i(se(I_m),me,c,I_m)$

$dspecifies_i \subseteq MethodSpec \times DirMethodEnv^i \times Context \times \text{Identifier}$
$dspecifies_i(\langle(J_1,\ldots,J_p),\langle Rs,Ks,F_C,F_R,T\rangle\rangle,me,c,I_m) \Leftrightarrow$
$\quad \ldots$(like above)

Figure 3.7: Definitions: Method Correctness (1/4)

Definitions: Method Correctness (Contd)

$$
\begin{array}{l}
\mathit{rspecifies}_{i,j} \subseteq \\
\quad \mathit{SpecEnv} \times \mathit{RecMethodEnv}^{i}_{j} \times \mathit{Context} \times \text{Identifier} \times \mathbb{B} \\
\mathit{rspecifies}_{i}(se, me, c, I_m, isrec) \Leftrightarrow \\
\quad \mathit{wellformed}(se(I_m)) \wedge \mathit{rspecifies}_{i,j}(se(I), me, c, I_m, isrec)
\end{array}
$$

$$
\begin{array}{l}
\mathit{rspecifies}_{i,j} \subseteq \\
\quad \mathit{MethodSpec} \times \mathit{RecMethodEnv}^{i}_{j} \times \mathit{Context} \times \text{Identifier} \times \mathbb{B} \\
\mathit{rspecifies}_{i,j}(\langle (J_1, \ldots, J_p), \langle Rs, Ks, F_C, F_R, T \rangle \rangle, me, c, I_m, isrec) \Leftrightarrow \\
\quad \forall c' \in \mathit{Context}, s, s' \in \mathit{State}, v_1, \ldots, v_p \in \mathit{Value}, e \in \mathit{Environment} : \\
\qquad \mathit{contexts}(c, c') \wedge \mathit{executes}(\mathit{control}(s)) \ \Rightarrow \\
\qquad\quad \text{LET} \\
\qquad\qquad com = \mathit{recmethod}_{i,j}(me, I_m, c')(v_1, \ldots, v_p), \\
\qquad\qquad c'' = \mathit{push}(c', J_1, \ldots, J_p), \\
\qquad\qquad s_1 = \mathit{writes}(s, J_1, v_1, \ldots, J_p, v_p)^{c''} \\
\qquad\quad \text{IN} \\
\qquad\qquad (\mathit{rel}(com)(s, s') \ \Rightarrow \\
\qquad\qquad\quad (\llbracket F_C \rrbracket^{c''}_{e}(s_1, s') \Rightarrow \llbracket F_R \rrbracket^{c''}_{e}(s_1, s')) \wedge \\
\qquad\qquad\quad s = s' \ \text{EXCEPT} \ \mathit{range}(c') \cup \text{range}(\mathit{gview}(c)) \wedge \\
\qquad\qquad\quad s = s' \ \text{EXCEPT}^{c} \ Rs \wedge \\
\qquad\qquad\quad (\mathit{executes}(\mathit{control}(s')) \vee \mathit{throws}(\mathit{control}(s'))) \wedge \\
\qquad\qquad\quad (\mathit{throws}(\mathit{control}(s')) \Rightarrow \mathit{key}(\mathit{control}(s')) \in Ks)) \wedge \\
\qquad\qquad (\llbracket F_C \rrbracket^{c''}_{e}(s_1, s_1) \Rightarrow \\
\qquad\qquad\quad (\neg isrec \vee j > \llbracket T \rrbracket^{c''}_{e}(s_1, s_1)) \Rightarrow \mathit{cond}(com)(s))
\end{array}
$$

Figure 3.8: Definitions: Method Correctness (2/4)

Definitions: Method Correctness (Contd)

$$\begin{array}{l}
\mathit{rspecifies}_{i,j} \subseteq \mathit{SpecEnv} \times \mathit{MethodEnv}^{i}_{j} \times \mathit{Context} \times \\
\quad \mathbb{P}(\text{Identifier}) \times \mathbb{P}(\text{Identifier}) \\
\mathit{rspecifies}_{i,j}(se, me, c, \mathit{Is}, \mathit{Js}) \Leftrightarrow \\
\quad \forall I_m \in \mathit{Js} : \mathit{rspecifies}_{i,j}(se, me, c, I_m, I_m \in \mathit{Is})
\end{array}$$

$$\begin{array}{l}
\mathit{rspecifies}_{i,j} \subseteq \mathit{SpecEnv} \times \mathit{MethodEnv}^{i}_{j} \times \mathit{Context} \times \mathbb{P}(\text{Identifier}) \\
\mathit{rspecifies}_{i,j}(se, me, c, \mathit{Is}) \Leftrightarrow \mathit{rspecifies}_{i,j}(se, me, c, \mathit{Is}, \text{Identifier})
\end{array}$$

$$\begin{array}{l}
\mathit{wellspecified} \subseteq \mathit{Method} \times \mathit{SpecEnv} \times \mathbb{P}(\text{Identifier}) \\
\mathit{wellspecified}(M, se, \mathit{Is}) \Leftrightarrow \\
\quad \forall c \in \mathit{Context}, d \in \mathbb{B} : \\
\quad \forall i, j \in \mathbb{N}, me \in \mathit{MethodEnv}^{i}_{j}, me' \in \mathit{MethodEnv}^{i}_{j+1} : \\
\qquad \mathit{rspecifies}_{i,j}(se, me, c, \mathit{Is}) \Rightarrow \\
\qquad\quad \mathit{rspecifies}_{i,j+1}(se, [\![M]\!]^{\mathit{gview}(c)}_{d,i,j}(me, me'), c, \mathit{Is}, \{[\![M]\!]_{\mathrm{I}}\})
\end{array}$$

$$\begin{array}{l}
\mathit{wellspecified} \subseteq \mathit{Methods} \times \mathit{SpecEnv} \times \mathbb{P}(\text{Identifier}) \\
\mathit{wellspecified}(\mathit{Ms}, se, \mathit{Is}) \Leftrightarrow \\
\quad \forall c \in \mathit{Context}, d \in \mathbb{B} : \\
\quad \forall i, j \in \mathbb{N}, me \in \mathit{MethodEnv}^{i}_{j}, me' \in \mathit{MethodEnv}^{i}_{j+1} : \\
\qquad \mathit{rspecifies}_{i,j}(se, me, c, \mathit{Is}) \Rightarrow \\
\qquad\quad \mathit{rspecifies}_{i,j+1}(se, [\![\mathit{Ms}]\!]^{\mathit{gview}(c)}_{d,i,j}(me, me'), c, \mathit{Is}, [\![\mathit{Ms}]\!]_{\mathrm{I}})
\end{array}$$

Figure 3.9: Definitions: Method Correctness (3/4)

Definitions: Method Correctness (Contd)

```
$specifies_i \subseteq SpecEnv \times MethodEnv^i \times Context \times \mathbb{P}(\text{Identifier})$
$specifies_i(se, me, c, Is) \Leftrightarrow$
    IF $i = 0$ THEN
        $bspecifies(se, me, c)$
    ELSE IF $me \in DirMethodEnv^{i-1}$ THEN
        $dspecifies_{i-1}(se, me, c)$
    ELSE
        $\forall j \in \mathbb{N} : rspecifies_{i-1,j}(se, me_j, c, Is)$
```

$specifies_i \subseteq SpecEnv \times MethodEnv^i \times Context \times \mathbb{P}(\text{Identifier})$
$specifies_i(se, me, c) \Leftrightarrow specifies_i(se, me, c, \emptyset)$

```
$wellspecified \subseteq Method \times SpecEnv$
$wellspecified(M, se) \Leftrightarrow$
    $\forall c \in Context, d \in \mathbb{B} :$
    $\forall i \in \mathbb{N}, me \in MethodEnv^i :$
        $specifies_i(se, me, c) \Rightarrow specifies_{i+1}(se, env_{d,i}^{gview(c),me,M}, c)$
```

```
$wellspecified \subseteq RecMethod \times SpecEnv$
$wellspecified(RM, se) \Leftrightarrow$
    $\forall c \in Context, i \in \mathbb{N}, me \in MethodEnv^i, d \in \mathbb{B} :$
        $specifies_i(se, me, c) \Rightarrow$
            $specifies_{i+1}([\![ RM ]\!]_S(se), [\![ RM ]\!]_{d,i}^{gview(c)}(me), c)$
```

```
$wellspecified \subseteq RecMethods \times SpecEnv$
$wellspecified(RMs, se) \Leftrightarrow$
    LET $r = [\![ RMs ]\!]^*(\emptyset, se).1$ IN
    $\forall i \in \mathbb{N}_{length(r)} : wellspecified(r(i).1, r(i).2)$
```

Figure 3.10: Definitions: Method Correctness (4/4)

Definitions: Program Correctness

$$\begin{array}{l}
\mathit{iscorrect} \subseteq \mathit{Spec} \times \mathit{Context} \times \mathit{Environment} \times \mathit{State} \times \mathit{State} \\
\mathit{iscorrect}(\langle Rs, Ks, F_C, F_R, T\rangle, c, e, s, s') \Leftrightarrow \\
\quad (\llbracket F_C \rrbracket_e^c(s,s) \Rightarrow \llbracket F_R \rrbracket_e^c(s,s')) \wedge \\
\quad s = s' \text{ EXCEPT}^c\ Rs \wedge \\
\quad \neg \mathit{continues}(\mathit{control}(s')) \wedge \neg \mathit{breaks}(\mathit{control}(s')) \wedge \\
\quad (\mathit{throws}(\mathit{control}(s')) \Rightarrow \mathit{key}(\mathit{control}(s')) \in Ks)
\end{array}$$

Figure 3.11: Definitions: Program Correctness

3.3 Judgement Semantics

Figures 3.12 to 3.16 introduce the semantic interpretation of the judgements based on the predicates introduced in the previous sections. The definitions are intended to formalize the informal semantics sketched at the beginning of this chapter. As for the formalization of "does not encounter the evaluation of undefined expressions", we chose the strategy to state that the execution of the program does not depend on the "definedness flag" d which decides whether an unevaluated expression gives rise to an "evaluation exception"; we may thus chose the (simpler) semantics without such exceptions when reasoning about these programs.

Method Judgements

$$
\begin{array}{l}
se \vdash RMs\; S\; \{C\} \;\Leftrightarrow \\
\quad \mathit{specifies}(RMs, se) \wedge \\
\quad \text{LET } se' = [\![RMs]\!]_{\mathrm{S}}(se) \text{ IN} \\
\quad \forall c \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\qquad \mathit{global}(c) \wedge \mathit{specifies}_i(se', me, c) \;\Rightarrow \\
\qquad\quad \forall s, s' \in \mathit{State}, e \in \mathit{Environment} : \\
\qquad\qquad \mathit{executes}(\mathit{control}(s)) \Rightarrow \\
\qquad\qquad\quad ([\![C]\!]^{c,me}_{\mathrm{TRUE}}(s,s') \Leftrightarrow [\![C]\!]^{c,me}_{\mathrm{FALSE}}(s,s')) \wedge \\
\qquad\qquad\quad (\langle\!\langle C \rangle\!\rangle^{c,me}_{\mathrm{TRUE}}(s) \Leftrightarrow \langle\!\langle C \rangle\!\rangle^{c,me}_{\mathrm{FALSE}}(s)) \wedge \\
\qquad\qquad\quad ([\![C]\!]^{c,me}_{\mathrm{TRUE}}(s,s') \Rightarrow \mathit{iscorrect}([\![S]\!]_{\mathrm{S}}, c, e, s, s')) \wedge \\
\qquad\qquad\quad ([\![\mathit{pre}(S)]\!]^{c}_{e}(s,s) \Rightarrow \langle\!\langle C \rangle\!\rangle^{c,me}_{\mathrm{TRUE}}(s))
\end{array}
$$

$$se \vdash RMs \;\Leftrightarrow \mathit{wellspecified}(RMs, se)$$

$$se \vdash RM \;\Leftrightarrow\; \mathit{wellspecified}(RM, se)$$

$$se, Is \vdash Ms \;\Leftrightarrow \mathit{wellspecified}(Ms, se, Is)$$

$$se, Is \vdash M \;\Leftrightarrow \mathit{wellspecified}(M, se, Is)$$

$$se \vdash M \;\Leftrightarrow \mathit{wellspecified}(M, se)$$

Figure 3.12: Method Judgements

Command Judgements

$$
\begin{array}{l}
se, Is, Vs \vdash C \checkmark F \;\Leftrightarrow \\
\quad F \text{ is closed } \wedge\; F \text{ does not depend on the poststate } \;\Rightarrow \\
\qquad \forall c_g, c_l \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\qquad\quad \mathit{contexts}(c_g, c_l, Vs) \wedge \mathit{specifies}_i(se, me, c_g, Is) \Rightarrow \\
\qquad\qquad \forall e \in \mathit{Environment}, s, s' \in \mathit{State} : \\
\qquad\qquad\quad \mathit{executes}(\mathit{control}(s)) \wedge [\![F]\!]_e^{c_l}(s,s) \Rightarrow \\
\qquad\qquad\qquad ([\![C]\!]_{\mathrm{TRUE}}^{c_l,me}(s,s') \Leftrightarrow [\![C]\!]_{\mathrm{FALSE}}^{c_l,me}(s,s')) \wedge \\
\qquad\qquad\qquad (\langle\!\langle C\rangle\!\rangle_{\mathrm{TRUE}}^{c_l,me}(s) \Leftrightarrow \langle\!\langle C\rangle\!\rangle_{\mathrm{FALSE}}^{c_l,me}(s))
\end{array}
$$

$$
\begin{array}{l}
se, Is, Vs \vdash C : F \;\Leftrightarrow \\
\quad F \text{ is closed } \wedge \\
\quad \forall c_g, c_l \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\qquad \mathit{contexts}(c_g, c_l, Vs) \wedge \mathit{specifies}_i(se, me, c_g, Is) \Rightarrow \\
\qquad\quad \forall e \in \mathit{Environment}, s, s' \in \mathit{State} : \\
\qquad\qquad \mathit{executes}(\mathit{control}(s)) \wedge [\![C]\!]_{\mathrm{TRUE}}^{c_l,me}(s,s') \Rightarrow \\
\qquad\qquad\quad [\![F]\!]_e^{c_l}(s,s')
\end{array}
$$

$$
\begin{array}{l}
se, Is, Vs \vdash C \downarrow^I F \;\Leftrightarrow \\
\quad F \text{ is closed } \wedge\; F \text{ does not depend on the poststate } \wedge \\
\quad F \text{ makes } \$I \text{ a natural number} \Rightarrow \\
\qquad \forall c_g, c_l \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\qquad\quad \mathit{contexts}(c_g, c_l, Vs) \wedge \mathit{specifies}_i(se, me, c_g, Is) \Rightarrow \\
\qquad\qquad \forall e \in \mathit{Environment}, s \in \mathit{State} : \\
\qquad\qquad\quad \mathit{executes}(\mathit{control}(s)) \wedge [\![F]\!]_e^{c_l}(s,s) \wedge i \geq e(I) \Rightarrow \\
\qquad\qquad\qquad \langle\!\langle C\rangle\!\rangle_{\mathrm{TRUE}}^{c_l,me}(s)
\end{array}
$$

Figure 3.13: Command Judgements

Transformation Judgements

$$
\begin{array}{l}
se, Is, Vs \vdash \textsc{pre}(C, Q) = P \Leftrightarrow \\
\quad Q \text{ does not depend on the poststate} \Rightarrow \\
\quad\quad P \text{ does not depend on the poststate} \land \\
\quad\quad \forall c_g, c_l \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\quad\quad\quad \mathit{contexts}(c_g, c_l, Vs) \land \mathit{specifies}_i(se, me, c_g, Is) \Rightarrow \\
\quad\quad\quad\quad \forall e \in \mathit{Environment}, s, s' \in \mathit{State} : \\
\quad\quad\quad\quad\quad \mathit{executes}(\mathit{control}(s)) \land [\![P]\!]_e^{c_l}(s, s) \Rightarrow \\
\quad\quad\quad\quad\quad\quad ([\![C]\!]_{\mathrm{TRUE}}^{c_l, me}(s, s') \Rightarrow [\![Q]\!]_e^{c_l}(s', s'))
\end{array}
$$

$$
\begin{array}{l}
se, Is, Vs \vdash \textsc{post}(C, P) = Q \Leftrightarrow \\
\quad P \text{ does not depend on the poststate} \Rightarrow \\
\quad\quad Q \text{ does not depend on the poststate} \land \\
\quad\quad \forall c_g, c_l \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\quad\quad\quad \mathit{contexts}(c_g, c_l, Vs) \land \mathit{specifies}_i(se, me, c_g, Is) \Rightarrow \\
\quad\quad\quad\quad \forall e \in \mathit{Environment}, s, s' \in \mathit{State} : \\
\quad\quad\quad\quad\quad \mathit{executes}(\mathit{control}(s)) \land [\![P]\!]_e^{c_l}(s, s) \Rightarrow \\
\quad\quad\quad\quad\quad\quad ([\![C]\!]_{\mathrm{TRUE}}^{c_l, me}(s, s') \Rightarrow [\![Q]\!]_e^{c_l}(s', s'))
\end{array}
$$

$$
\begin{array}{l}
se, Is, Vs \vdash \textsc{assert}(C, P) = C' \Leftrightarrow \\
\quad P \text{ does not depend on the poststate} \Rightarrow \\
\quad\quad \forall c_g, c_l \in \mathit{Context}, i \in \mathbb{N}, me \in \mathit{MethodEnv}^i : \\
\quad\quad\quad \mathit{contexts}(c_g, c_l, Vs) \ \land \mathit{specifies}_i(se, me, c_g, Is) \Rightarrow \\
\quad\quad\quad\quad \forall e \in \mathit{Environment}, s, s' \in \mathit{State} : \\
\quad\quad\quad\quad\quad \mathit{executes}(\mathit{control}(s)) \Rightarrow \\
\quad\quad\quad\quad\quad\quad ([\![C']\!]_{\mathrm{TRUE}}^{c_l, me}(s, s') \Leftrightarrow \\
\quad\quad\quad\quad\quad\quad\quad [\![P]\!]_e^{c_l}(s) \land [\![C]\!]_{\mathrm{TRUE}}^{c_l, me}(s, s')) \land \\
\quad\quad\quad\quad\quad\quad (\langle\!\langle C' \rangle\!\rangle_{\mathrm{TRUE}}^{c_l, me}(s) \Leftrightarrow \\
\quad\quad\quad\quad\quad\quad\quad [\![P]\!]_e^{c_l}(s) \land \langle\!\langle C \rangle\!\rangle_{\mathrm{TRUE}}^{c_l, me}(s))
\end{array}
$$

Figure 3.14: Transformation Judgements

Frame Judgements

$$
\begin{array}{l}
F \;\S\,{}^{Ks}_{Rs} \;\Leftrightarrow\\
\quad \forall c \in Context, s, s' \in State, e \in Environment :\\
\qquad global(c) \wedge executes(control(s)) \wedge\\
\qquad [\![F]\!]^c_e(s, s') \Rightarrow\\
\qquad\quad \neg continues(control(s')) \wedge \neg breaks(control(s')) \wedge\\
\qquad\quad s = s' \text{ EXCEPT}^c\; Rs \wedge\\
\qquad\quad (throws(control(s')) \Rightarrow key(control(s')) \in Ks)
\end{array}
$$

$$
\begin{array}{l}
F \;\S_{\mathrm{c}}\; F_c \Leftrightarrow\\
\quad \forall c \in Context, s, s' \in State, e \in Environment :\\
\qquad global(c) \wedge executes(control(s)) \wedge\\
\qquad [\![F]\!]^c_e(s, s') \wedge [\![F_c]\!]^c_e(s, s) \Rightarrow\\
\qquad\quad \neg continues(control(s'))
\end{array}
$$

$$
\begin{array}{l}
F \;\S_{\mathrm{b}}\; F_b \Leftrightarrow\\
\quad \forall c \in Context, s, s' \in State, e \in Environment :\\
\qquad global(c) \wedge executes(control(s)) \wedge\\
\qquad [\![F]\!]^c_e(s, s') \wedge [\![F_b]\!]^c_e(s, s) \Rightarrow\\
\qquad\quad \neg breaks(control(s'))
\end{array}
$$

$$
\begin{array}{l}
F \;\S_{\mathrm{s}}\,{}^{Qs}_{Rs} \Leftrightarrow\\
\quad \forall c \in Context, s, s' \in State, e \in Environment :\\
\qquad global(c) \wedge executes(control(s)) \wedge\\
\qquad [\![F]\!]^c_e(s, s') \wedge s = s' \text{ EXCEPT } Rs \;\Rightarrow\\
\qquad\quad s = s' \text{ EXCEPT } Qs
\end{array}
$$

$$
\begin{array}{l}
F \;\S_{\mathrm{e}}\,{}^{Ls}_{Ks} \Leftrightarrow\\
\quad \forall c \in Context, s, s' \in State, e \in Environment :\\
\qquad global(c) \wedge executes(control(s)) \wedge\\
\qquad [\![F]\!]^c_e(s, s') \wedge throws(control(s')) \wedge\\
\qquad key(control(s')) \in Ks \Rightarrow\\
\qquad\quad key(control(s')) \in Ls
\end{array}
$$

Figure 3.15: Frame Judgements

Formula Judgements

$$
\begin{array}{l}
\models_{Rs} F \Leftrightarrow \\
\quad \forall c \in \mathit{Context}, e \in \mathit{Environment}, s, s' \in \mathit{State} : \\
\qquad s = s' \text{ EXCEPT}^{c}\ Rs \ \Rightarrow \\
\qquad\quad [\![F]\!]^{c}_{e}(s, s')
\end{array}
$$

Figure 3.16: Formula Judgements

Chapter 4

Rules

In this chapter we give rules for deriving judgements of the kinds introduced in Chapter 3. We claim that the rules are sound, i.e. that by their application only judgements can be derived that are true with respect to their formal semantics.

4.1 Definitions

Figures 4.1 and 4.2 give auxiliary definitions used in the formulation of the rules:

$[F]_{Rs}^{F_c,F_b,F_r,\{K_1,\ldots,K_n\}}$ denotes a transition relation by a particular pattern of a specification formula: its form makes explicit which variables may be changed by the transition, which exceptions may be thrown, and whether the transition may result in a continuing, breaking, or returning poststate.

$\simeq$ and $\overset{\mathrm{D}}{\simeq}$ translate program expressions to formulas respectively terms: $\simeq$ translates an expression to an equivalent formula respectively term while $\overset{\mathrm{D}}{\simeq}$ generates from an expression a formula that characterizes those states in which the expression is defined[1].

$[\![R]\!]_{\mathrm{R}}^{Vs}$ determines a canonical version of reference R in the scope of the local variables names Vs: this version is identical to R, except that if R is a global reference which is not shadowed by the declarations in Vs, then it is replaced by the local version of R (which refers to the same variable).

[1]The translation depends on the expression language and the predicates/functions of the formula language; we therefore give no rules for the translation in our calculus.

$[F]_{Rs}^{F_c,F_b,F_r,\{K_1,\ldots,K_n\}} \equiv$
F and writesonly Rs and
(next.continues => F_c) and
(next.breaks => F_b) and
(next.returns => F_r) and
(next.throws =>
(next.throws K_1 or $\ldots$ or next.throws K_n))

$\llcorner \simeq \llcorner \subseteq \text{Expression} \times \text{Formula}$
$E \simeq F \Leftrightarrow$
F has no primed program variables $\wedge$
F has no occurrence of next $\wedge$
$\forall c \in \mathit{Context}, e \in \mathit{Environment}, s, s' \in \mathit{State}:$
$[\![E]\!]_{\mathrm{V}}^{c}(s) = \text{TRUE} \Leftrightarrow [\![F]\!]_{e}^{c}(s,s')$

$\llcorner \simeq \llcorner \subseteq \text{Expression} \times \text{Term}$
$E \simeq T \Leftrightarrow$
T has no primed program variables $\wedge$
T has no occurrence of next $\wedge$
$\forall c \in \mathit{Context}, e \in \mathit{Environment}, s, s' \in \mathit{State}:$
$[\![E]\!]_{\mathrm{V}}^{c}(s) = [\![T]\!]_{e}^{c}(s,s')$

$\llcorner \overset{\mathrm{D}}{\simeq} \llcorner \subseteq \text{Expression} \times \text{Formula}$
$E \overset{\mathrm{D}}{\simeq} F_D \Leftrightarrow$
$\forall c \in \mathit{Context}, s \in \mathit{State}, s' \in \mathit{State}:$
$\forall s \in \mathit{State}: [\![E]\!]_{\mathrm{D}}^{c}(s) \Leftrightarrow [\![F_D]\!]_{e}^{c}(s,s')$

Figure 4.1: Definitions (1/2)

$[\![\llcorner\!\lrcorner]\!]_{\mathrm{R}}$: Reference $\rightarrow \mathbb{P}$(Identifier) $\rightarrow$ Reference
$[\![I]\!]_{\mathrm{R}}^{Vs} = I$
$[\![\,?I]\!]_{\mathrm{R}}^{Vs} = \text{IF } I \in Vs \text{ THEN } ?I \text{ ELSE } I$

```
$Invariant(G,H,F)_{\{R_1,\ldots,R_n\}} \equiv$
    $G$ is closed $\wedge$
    $\$I_1,\ldots,\$I_n,\#I_s,\#I_t$ do not occur in $G,H,$ and $F$ $\wedge$
    $\models_{\{R_1,\ldots,R_n\}}$
        forall $\$I_1,\ldots,\$I_n$: allstate $\#I_s$,$\#I_t$:
            $(G[\#I_s/\text{next}][\$I_1/R_1\text{'},\ldots,\$I_n/R_n\text{'}]$
                and ($\#I_s$.executes or $\#I_s$.continues)
                and $\#I_t$.executes
                and $H[\#I_t/\text{now}][\$I_1/R_1,\ldots,\$I_n/R_n]$
                and $F[\#I_t/\text{now}][\$I_1/R_1,\ldots,\$I_n/R_n])$
            => $G)$
```

Figure 4.2: Definitions (2/2)

$Invariant(G,H,F)_{\{R_1,\ldots,R_n\}}$ essentially describes the obligation of proving that G is an invariant for a loop whose condition is denoted by formula H and where the effect of every iteration (which changes only variables identified by $\{R_1,\ldots,R_n\}$) is described by formula F.

Based on these definitions, we give in the following section rules for each kind of judgement.

$$\frac{\begin{array}{l} se \vdash RMs \\ wellformed(S) \\ S = \textsf{writesonly}\ Rs\ \textsf{throwsonly}\ Ks \\ \quad \textsf{requires}\ F_C\ \textsf{ensures}\ F_R\ \textsf{decreases}\ T \\ se' = [\![RMs]\!]_S(se) \\ se',\emptyset,\emptyset \vdash C : [F]^{F_c,F_b,F_r,Ls}_{Ms} \\ [F]^{F_c,F_b,F_r,Ls}_{Ms} \S^{Ks}_{Rs} \\ se',\emptyset,\emptyset \vdash C \checkmark F_C \\ \models_{Rs} (\textsf{now.executes and}\ F) \texttt{=>} (F_C \texttt{=>} F_R) \\ se',\emptyset,\emptyset \vdash C \downarrow^J F_C\ \textsf{and}\ \$J = T \end{array}}{se \vdash RMs\ S\ \{C\}}$$

Figure 4.3: Rules for $se \vdash RMs\ S\ \{C\}$

4.2 Judgement $se \vdash RMs\ S\ \{C\}$

Based on the rule depicted in Figure 4.3, to verify the correctness of a program with respect to its specification, we have to

- verify correctness of the methods *RMs* with respect to their specifications,
- check well-formedness of the specification *S* of the main command *C*,
- compute the specification environment se' established by the methods for use in the further steps,
- compute the formula $[F]^{F_c,F_b,F_r,Ls}_{Ms}$ denoting the transition relation of *C*,
- check that this formula preserves several general constraints (command does not result in a continuing or breaking postate, only modifies variables and throws exceptions allowed by *S*),
- check that the execution of *C* does not encounter the evaluation of any undefined expression in those states that are allowed by precondition F_c,
- verify that *F* implies the core specification F_c => F_R,
- verify that *C* terminates in every state allowed by F_C.

The correctness of the last judgement does actually not depend on the correctness of th main command's termination measure T, because the command does not appear in a recursive method set; the measure can therefore be skipped from the specification and the clause "and $J = T" can be omitted from the last judgement.

$$se \vdash \textvisiblespace$$

$$\frac{\begin{array}{l} se \vdash RMs \\ se' = [\![RMs]\!]_{\mathrm{S}}(se) \\ se' \vdash RM \end{array}}{se \vdash RMs\ RM}$$

Figure 4.4: Rules for $se \vdash RMs$

4.3 Judgement $se \vdash RMs$

This judgement verifies the correctness of the method declarations *RMs* by verifying each component of *RMs* in the specification environment set up by the previous components.

$$\frac{se \vdash M}{se \vdash M}$$

$$\frac{\begin{array}{l} Is = [\![Ms]\!]_{\mathrm{I}} \\ se, Is \vdash Ms \end{array}}{se \vdash \texttt{recursive}\, Ms}$$

Figure 4.5: Rules for $se \vdash RM$

4.4 Judgement $se \vdash RM$

This judgement distinguishes between two kinds of method definitions, the definition of a non-recursive method M and the definition of a recursive method set Ms. The verification tasks are correspondingly delegated to the respective judgements.

$$se, Is \vdash \textvisiblespace$$

$$\frac{se, Is \vdash M \qquad se, Is \vdash Ms}{se, Is \vdash M\ Ms}$$

Figure 4.6: Rules for $se, Is \vdash Ms$

4.5 Judgement $se, Is \vdash Ms$

To verify the correctness of a recursive method set, the correctness of each element of the set can be independently verified.

$$
\begin{array}{l}
\mathit{wellformed}((J_1,\ldots,J_p),S) \\
S = \texttt{writesonly}\ \mathit{Rs}\ \texttt{throwsonly}\ \mathit{Ks} \\
\quad \texttt{requires}\ F_C\ \texttt{ensures}\ F_R\ \texttt{decreases}\ T \\
\mathit{se},\mathit{Is},\{J_1,\ldots,J_p\} \ \vdash\ C : [F]^{F_c,F_b,F_r,Ls}_{Ms} \\
\$J_1,\ldots,\$J_p,\$L_1,\ldots,\$L_p \text{ do not occur in } F \\
[\text{exists } \$J_1,\ldots,\$J_p,\$L_1,\ldots,\$L_p\text{:} \\
\quad F[\$J_1/J_1,\ldots,\$J_p/J_p,\$L_1/J_1\text{'},\ldots,\$L_p/J_p\text{'}] \\
\quad]^{F_c,F_b,F_r,Ls}_{Ms\backslash\{J_1,\ldots,J_p\}}\ \S\ ^{Ks}_{Rs} \\
\mathit{se},\mathit{Is},\{J_1,\ldots,J_p\} \ \vdash\ C \checkmark F_C \\
\#I_s \text{ does not occur in } F \\
\models_{Rs} \\
\quad (\text{now.executes and} \\
\qquad \text{exstate } \#I_s\text{:} \\
\qquad\quad F[\#I_s/\text{next}] \text{ and} \\
\qquad\quad \text{if } \#I_s\text{.throws then} \\
\qquad\qquad \text{next == } \#I_s \\
\qquad\quad \text{else} \\
\qquad\qquad \text{next.executes and} \\
\qquad\qquad \text{next.value } = \#I_s\text{.value}) \texttt{ => } (F_C \texttt{ => } F_R) \\
\mathit{se},\mathit{Is},\{J_1,\ldots,J_p\} \ \vdash\ C \downarrow^J F_C \text{ and } \$J = T \\
\hline
\mathit{se},\mathit{Is} \ \vdash\ \texttt{method}\ I_m\,\texttt{(}J_1,\ldots,J_p\texttt{)}\ S\ \texttt{\{}C\texttt{\}}
\end{array}
$$

Figure 4.7: Rules for $se,Is \vdash M$

4.6 Judgement $se,Is \vdash M$

This judgement verifies the correctness of a recursive method that occurs in a recursive method set that consists of those methods named in *Is*. The overall verification steps are similar as for establishing the correctness of the program's main command:

- check wellformedness of the specification S of the method,
- compute the formula $[F]^{F_c,F_b,F_r,Ls}_{Ms}$ denoting the transition relation of method body C,
- check that this formula preserves several general constraints (the method body does not result in a continuing or breaking postate, and only modifies those variables and throws those exceptions that are allowed by S),

- check that the execution of C does not encounter the evaluation of any undefined expression in those states that are allowed by precondition F_c,
- verify that a modified version of F implies the core specification F_c => F_R,
- verify that C terminates in every state allowed by precondition F_C using not more recursive invocations than denoted by the termination measure T.

The modification of F in the last but one step reflects the fact that a "returning" poststate of the method body C is set to "executing" on return of the method.

$$\frac{se, \emptyset \vdash M}{se \vdash M}$$

Figure 4.8: Rules for $se \vdash M$

4.7 Judgement $se \vdash M$

The verification of a non-recursive method M is simply delegated to the rule for the verification of a recursive method with, however, an empty set of recursive methods. The only difference in the core verification will be the proof of termination where it will be not necessary to consider the termination measure in the specification of M (which thus can be omitted from the specification of M).

4.8 Judgement $se, Is, Vs \vdash C \checkmark F$

The rules for this judgement verify that, for given specification environment *se*, recursive method names *Is*, local variable names *Vs*, the execution of command C in a state for which condition F holds does not encounter the evaluation of "undefined expressions". By the application of the rules, for every expression E with "definedness condition" F_D, a proof obligation of form $\models_{Rs} \ldots$ => F_D is generated. Some points to be highlighted are:

- For all local variables (introduced in variable declarations, variable definitions, and exception handler parameters), it is distinguished whether the declaration shadows a global variable I (then I has to be replaced by ?I in the respective state condition) or just another local variable I (then I has to be replaced by a new mathematical variable $\$J$).

- In the rules for command sequences and exception handlers, it is necessary to generate the postconditions holding after the execution of the first command C_1.

- The handling of loops profits from the existence of a loop invariant G which can be used to strengthen the condition P that holds before the execution of the loop body C.

- The rule for method calls only verifies the well-definedness of the method's arguments. It could be, however, also easily strengthened to verify the correctness of the method's precondition (since the violation of a precondition typically indicates an error situation rather than a lack of interest in the properties of the poststate).

$$\frac{E \overset{\mathrm{D}}{\simeq} F_D \qquad \models_{\emptyset} (\textsf{now.executes and } F) \texttt{ => } F_D}{se, Is, Vs \vdash I\texttt{=}E \checkmark F}$$

$$\frac{I \notin Vs \qquad se, Is, Vs \cup \{I\} \vdash C \checkmark F[?I/I]}{se, Is, Vs \vdash \texttt{var } I\texttt{; } C \checkmark F}$$

$$\frac{I \in Vs \qquad \$J \text{ does not occur in } F \qquad se, Is, Vs \vdash C \checkmark \textsf{exists } \$J : F[\$J/I]}{se, Is, Vs \vdash \texttt{var } I\texttt{; } C \checkmark F}$$

$$\frac{I \notin Vs \qquad E \overset{\mathrm{D}}{\simeq} F_D \qquad \models_{\emptyset} (\textsf{now.executes and } F) \texttt{ => } F_D \qquad E \simeq T \qquad se, Is, Vs \cup \{I\} \vdash C \checkmark F[?I/I] \textsf{ and } I = T[?I/I]}{se, Is, Vs \vdash \texttt{var } I\texttt{=}E\texttt{; } C \checkmark F}$$

$$\frac{I \in Vs \qquad E \overset{\mathrm{D}}{\simeq} F_D \qquad \models_{\emptyset} (\textsf{now.executes and } F) \texttt{ => } F_D \qquad E \simeq T \qquad \$J \text{ does not occur in } F \text{ and } T \qquad se, Is, Vs \vdash C \checkmark \textsf{exists } \$J : F[\$J/I] \textsf{ and } I = T[\$J/I]}{se, Is, Vs \vdash \texttt{var } I\texttt{=}E\texttt{; } C \checkmark F}$$

Figure 4.9: Rules for $se, Is, Vs \vdash C \checkmark F$ (1/4)

$$\frac{\begin{array}{l} se, Is, Vs \vdash \textsc{post}(C_1, F) = G \\ se, Is, Vs \vdash C_1 \checkmark F \\ se, Is, Vs \vdash C_2 \checkmark G \end{array}}{se, Is, Vs \vdash C_1\,\texttt{;}\,C_2 \checkmark F}$$

$$\frac{\begin{array}{l} E \overset{\mathrm{D}}{\simeq} F_D \\ \models_{\emptyset} (\textsf{now.executes and } F) \;\texttt{=>}\; F_D \\ E \simeq G \\ se, Is, Vs \vdash C \checkmark F \textsf{ and } G \end{array}}{se, Is, Vs \vdash \texttt{if (}E\texttt{)}\ C \checkmark F}$$

$$\frac{\begin{array}{l} E \overset{\mathrm{D}}{\simeq} F_D \\ \models_{\emptyset} (\textsf{now.executes and } F) \;\texttt{=>}\; F_D \\ E \simeq G \\ se, Is, Vs \vdash C_1 \checkmark F \textsf{ and } G \\ se, Is, Vs \vdash C_1 \checkmark G \textsf{ and !}G \end{array}}{se, Is, Vs \vdash \texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2 \checkmark F}$$

Figure 4.10: Rules for $se, Is, Vs \vdash C \checkmark F$ (2/4)

$$\frac{\begin{array}{l}E \overset{\mathrm{D}}{\simeq} F_D \\ \models_\emptyset \text{(now.executes and } F\text{) => } F_D \\ E \simeq H \\ se, Is, Vs \vdash C : [F_R]^{F_c, F_b, F_r, Ks}_{R_1,\ldots,R_n} \\ \$I_1, \ldots, \$I_n, \#I_s \text{ do not occur in } F \\ P \;=\; H \text{ and} \\ \qquad\quad \text{exists } \$I_1, \ldots, \$I_n\text{: exstate } \#I_s\text{:} \\ \qquad\qquad F[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \\ se, Is, Vs \vdash C \checkmark P \\ se, Is, Vs \vdash \textsc{post}(C, P) = Q \\ \models_\emptyset \text{((now.executes or now.continues) and } Q\text{) => } F_D\end{array}}{se, Is, Vs \vdash \texttt{while(}E\texttt{)}\ C \checkmark F}$$

$$\frac{\begin{array}{l}E \overset{\mathrm{D}}{\simeq} F_D \\ \models_\emptyset \text{(now.executes and } F\text{) => } F_D \\ E \simeq H \\ se, Is, Vs \vdash C : [F_R]^{F_c, F_b, F_r, Ks}_{R_1,\ldots,R_n} \\ \mathit{Invariant}(G, H, F_R)_{\{R_1,\ldots,R_n\}} \\ \$I_1, \ldots, \$I_n, \#I_s, \#I_t \text{ do not occur in } F \text{ and } G \\ P \;=\; H \text{ and} \\ \quad \text{exists } \$I_1, \ldots, \$I_n\text{: exstate } \#I_s, \#I_t\text{:} \\ \qquad F[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \text{ and} \\ \qquad (G[\text{now}/\text{next}][R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \\ \qquad\quad [\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \text{ =>} \\ \qquad\quad G[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \\ \qquad\qquad [\#I_t/\text{next}][R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \text{ and} \\ \qquad\quad (!(\#I_t\text{.continues or } \#I_t\text{.breaks) =>} \\ \qquad\qquad \text{now == } \#I_t)) \\ se, Is, Vs \vdash C \checkmark P \\ se, Is, Vs \vdash \textsc{post}(C, P) = Q \\ \models_\emptyset \text{((now.executes or now.continues) and } Q\text{) => } F_D\end{array}}{\begin{array}{l}se, Is, Vs \vdash \\ \quad \texttt{invariant}\ G\ \texttt{decreases}\ T\ \texttt{while(}E\texttt{)}\ C \checkmark F\end{array}}$$

Figure 4.11: Rules for $se, Is, Vs \vdash C \checkmark F$ (3/4)

$$se, Is, Vs \vdash \texttt{continue} \checkmark F$$

$$se, Is, Vs \vdash \texttt{break} \checkmark F$$

$$\frac{\begin{array}{l} E \simeq F_D \\ \models_{\emptyset} (\textsf{now.executes and } F) \textsf{ => } F_D \end{array}}{se, Is, Vs \vdash \texttt{return}\, E \checkmark F}$$

$$\frac{\begin{array}{l} E \simeq F_D \\ \models_{\emptyset} (\textsf{now.executes and } F) \textsf{ => } F_D \end{array}}{se, Is, Vs \vdash \texttt{throw}\, I_k\, E \checkmark F}$$

$$\frac{\begin{array}{l} I_v \notin Vs \\ se, Is, Vs \vdash C_1 \checkmark F \\ se, Is, Vs \vdash \textsc{post}(C_1, F) = G \\ \#I_s \text{ does not occur in } G \\ se, Is, Vs \cup \{I_v\} \vdash C_2 \checkmark \\ \quad \textsf{exstate } \#I_s\textsf{:} \\ \qquad G[\#I_s/\texttt{now}][?I_v/I_v] \textsf{ and} \\ \qquad \#I_s\textsf{.throws } I_k \textsf{ and } \#I_s\textsf{.value = } I_v \end{array}}{se, Is, Vs \vdash \texttt{try}\, C_1\, \texttt{catch}\,(I_k\, I_v)\; C_2 \checkmark F}$$

$$\frac{\begin{array}{l} I_v \in Vs \\ se, Is, Vs \vdash C_1 \checkmark F \\ se, Is, Vs \vdash \textsc{post}(C_1, F) = G \\ \$I, \#I_s \text{ do not occur in } G \\ se, Is, Vs \vdash C_2 \checkmark \\ \quad \textsf{exists } \$I\textsf{: exstate } \#I_s\textsf{:} \\ \qquad G[\#I_s/\texttt{now}][\$I/I_v] \textsf{ and} \\ \qquad \#I_s\textsf{.throws } I_k \textsf{ and } \#I_s\textsf{.value = } I_v \end{array}}{se, Is, Vs \vdash \texttt{try}\, C_1\, \texttt{catch}\,(I_k\, I_v)\; C_2 \checkmark F}$$

$$\frac{\begin{array}{l} E_1 \overset{\mathrm{D}}{\simeq} F_1, \ldots, E_P \overset{\mathrm{D}}{\simeq} F_p \\ \models_{\emptyset} (\textsf{now.executes and } F) \textsf{ => } (F_1 \textsf{ and } \ldots \textsf{ and } F_p) \end{array}}{se, Is, Vs \vdash I_r = I_m\,(E_1, \ldots, E_p)\; \checkmark F}$$

$$se, Is, Vs \vdash \texttt{assert}\, G \checkmark F$$

Figure 4.12: Rules for $se, Is, Vs \vdash C \checkmark F$ (4/4)

4.9 Judgement $se, Is, Vs \vdash C : F$

The rules for this judgement are constructed in such a way that they compute for given specification environment *se*, recursive method names *Is*, and local variable names *Vs*, from a command C a formula F which captures the transition relation of C; the formula is actually computed in the form

$$[F]_{Rs}^{F_c,F_b,F_r,\{K_1,\ldots,K_n\}}$$

which makes the "external effects" of the transition explicit.

We highlight the following items:

- The first two rules are generic i.e. the do not depend on the kind of command: the first one allows to widen the frame *Rs* by another reference *R*, while the second one allows to shrink the frame, if it can be proved that a variable does not change from pre- to poststate. The first rule has to be especially applied in connection with the rules for the two-sided conditional, command sequence, and exception handler, since these involve two commands C_1 and C_2 whose transition relations have to be extended to a common frame (typically the smallest possible one).
- The rules for variable declarations, variable definitions, and exception handlers come in three variants depending on whether the declared identifier *I* shadows a global variable or not, and if not, whether the global variable is changed by the command body or not.
- The first rule for command sequences handles the special case that the first command C_1 always results in an "executing" poststate while the second rule handles the general case.
- There are two rules for each version of the while loop (with and without invariants) that depend on the case whether the loop body may result in a "breaking" state or not: only in the second case an "executing" poststate of the loop indicates that the loop condition does not hold any more.
- If a loop is provided with an invariant *G*, a corresponding proof obligation is generated (see the definition of *Invariant*); *G* may be used to strengthen the generated transition formula (which is otherwise pretty weak).
- The rule for method calls comes in two variants depending on the fact whether the reference I_r denotes a global variable or not. If the method raises an exception, it may have changed I_r in the first case while I_r remains unchanged in the second case.

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{Rs} \\ R \notin Rs \end{array}}{se, Is, Vs \vdash C : [F \text{ and } R\text{'}= R]^{F_c, F_b, F_r, Ks}_{Rs \cup \{R\}}}$$

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{Rs} \\ \models_{Rs} F \text{ => } R\text{'}= R \end{array}}{se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{Rs \setminus \{R\}}}$$

$$\frac{E \simeq T}{se, Is, Vs \vdash R = E : [R\text{'}{=}T \text{ and next.executes}]^{\text{false}, \text{false}, \text{false}, \emptyset}_{\{[\![R]\!]^{Vs}_{\mathrm{R}}\}}}$$

$$\frac{\begin{array}{l} I \notin Vs \\ se, Is, Vs \cup \{I\} \vdash C : [F]^{F_c, F_b, F_r, Ks}_{Rs} \\ ?I \in Rs \\ \$I_a, \$I_b \text{ do not occur in } F \end{array}}{\begin{array}{l} se, Is, Vs \vdash \texttt{var}\ I\texttt{;}\ C : \\ \quad [\text{exists } \$I_a, \$I_b\text{: } F[\$I_a/I, \$I_b/I\text{'}][I/?I, I\text{'}/?I\text{'}]]^{F_c, F_b, F_r, Ks}_{(Rs \setminus \{?I\}) \cup \{I\}} \end{array}}$$

$$\frac{\begin{array}{l} I \notin Vs \\ se, Is, Vs \cup \{I\} \vdash C : [F]^{F_c, F_b, F_r, Ks}_{Rs} \\ ?I \notin Rs \\ \$I_a, \$I_b \text{ do not occur in } F \end{array}}{\begin{array}{l} se, Is, Vs \vdash \texttt{var}\ I\texttt{;}\ C : \\ \quad [\text{exists } \$I_a, \$I_b\text{: } F[\$I_a/I, \$I_b/I\text{'}][I/?I, I\text{'}/?I\text{'}]]^{F_c, F_b, F_r, Ks}_{Rs \setminus \{I\}} \end{array}}$$

$$\frac{\begin{array}{l} I \in Vs \\ se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{Rs} \\ \$I_a, \$I_b \text{ do not occur in } F \end{array}}{\begin{array}{l} se, Is, Vs \vdash \texttt{var}\ I\texttt{;}\ C : \\ \quad [\text{exists } \$I_a, \$I_b\text{: } F[\$I_a/I, \$I_b/I\text{'}]]^{F_c, F_b, F_r, Ks}_{Rs \setminus \{I\}} \end{array}}$$

Figure 4.13: Rules for $se, Is, Vs \vdash C : F$ (1/8)

$$\begin{array}{l}
I \notin Vs \\
se, Is, Vs \cup \{I\} \;\vdash\; C : [\,F\,]^{F_c, F_b, F_r, Ks}_{Rs} \\
?I \in Rs \\
\$I_a, \$I_b \text{ do not occur in } F \\
E \simeq T \\
\hline
se, Is, Vs \;\vdash\; \mathtt{var}\ I\mathtt{=}E\mathtt{;}\ C : \\
\quad [\,\textsf{exists } \$I_a, \$I_b\textsf{: } \$I_a = T \textsf{ and} \\
\qquad F[\$I_a/I, \$I_b/I'][I/?I, I'/?I']\,]^{F_c, F_b, F_r, Ks}_{(Rs \setminus \{?I\}) \cup \{I\}}
\end{array}$$

$$\begin{array}{l}
I \notin Vs \\
se, Is, Vs \cup \{I\} \;\vdash\; C : [\,F\,]^{F_c, F_b, F_r, Ks}_{Rs} \\
?I \notin Rs \\
\$I_a, \$I_b \text{ do not occur in } F \\
E \simeq T \\
\hline
se, Is, Vs \;\vdash\; \mathtt{var}\ I\mathtt{=}E\mathtt{;}\ C : \\
\quad [\,\textsf{exists } \$I_a, \$I_b\textsf{: } \$I_a = T \textsf{ and} \\
\qquad F[\$I_a/I, \$I_b/I'][I/?I, I'/?I']\,]^{F_c, F_b, F_r, Ks}_{Rs \setminus \{I\}}
\end{array}$$

$$\begin{array}{l}
I \in Vs \\
se, Is, Vs \;\vdash\; C : [\,F\,]^{F_c, F_b, F_r, Ks}_{Rs} \\
\$I_a, \$I_b \text{ do not occur in } F \\
\hline
E \simeq T \\
se, Is, Vs \;\vdash\; \mathtt{var}\ I\mathtt{=}E\mathtt{;}\ C : \\
\quad [\,\textsf{exists } \$I_a, \$I_b\textsf{: } \$I_a = T \textsf{ and} \\
\qquad F[\$I_a/I, \$I_b/I']\,]^{F_c, F_b, F_r, Ks}_{Rs \setminus \{I\}}
\end{array}$$

Figure 4.14: Rules for $se, Is, Vs \;\vdash\; C : F$ (2/8)

$$\frac{\begin{array}{l} se, Is, Vs \vdash C_1 : [F_1]^{\text{false},\text{false},\text{false},\emptyset}_{\{R_1,\ldots,R_n\}} \\ se, Is, Vs \vdash C_2 : [F_2]^{F_c,F_b,F_r,Ks}_{\{R_1,\ldots,R_n\}} \\ \$I_1,\ldots,\$I_n,\#I_s \text{ do not occur in } F_1 \text{ and } F_2 \end{array}}{\begin{array}{l} se, Is, Vs \vdash C_1\,\texttt{;}\,C_2 : \\ \quad [\text{exists } \$I_1,\ldots,\$I_n\text{: exstate } \#I_s\text{:} \\ \qquad F_1[\#I_s/\text{next}][\$I_1/R_1\text{'},\ldots,\$I_n/R_n\text{'}] \text{ and} \\ \qquad F_2[\#I_s/\text{now}][\$I_1/R_1,\ldots,\$I_n/R_n]]^{F_c,F_b,F_r,Ks}_{\{R_1,\ldots,R_n\}} \end{array}}$$

$$\frac{\begin{array}{l} se, Is, Vs \vdash C_1 : [F_1]^{F_{c1},F_{b1},F_{r1},Ks}_{\{R_1,\ldots,R_n\}} \\ se, Is, Vs \vdash C_2 : [F_2]^{F_{c2},F_{b2},F_{r2},Ls}_{\{R_1,\ldots,R_n\}} \\ \$I_1,\ldots,\$I_n,\#I_s \text{ do not occur in } F_1 \text{ and } F_2 \end{array}}{\begin{array}{l} se, Is, Vs \vdash C_1\,\texttt{;}\,C_2 : \\ \quad [\text{exists } \$I_1,\ldots,\$I_n\text{: exstate } \#I_s\text{:} \\ \qquad F_1[\#I_s/\text{next}][\$I_1/R_1\text{'},\ldots,\$I_n/R_n\text{'}] \text{ and} \\ \qquad \text{if } \#I_s\text{.executes then} \\ \qquad\quad F_2[\#I_s/\text{now}][\$I_1/R_1,\ldots,\$I_n/R_n] \\ \qquad \text{else} \\ \qquad\quad \text{next==}\#I_s \text{ and } R_1\text{'=}\$I_1 \text{ and } \ldots \text{ and } R_n\text{'=}\$I_n \\ \quad]^{F_{c1} \text{ or } F_{c2},F_{b1} \text{ or } F_{b2},F_{r1} \text{ or } F_{r2},Ks\cup Ls}_{\{R_1,\ldots,R_n\}} \end{array}}$$

$$\frac{\begin{array}{l} C : [F]^{F_c,F_b,F_r,Ks}_{Rs} \\ E \simeq F_0 \end{array}}{\texttt{if (}E\texttt{)}\ C : [\text{if } F_0 \text{ then } F \text{ else readsonly}]^{F_c,F_b,F_r,Ks}_{Rs}}$$

$$\frac{\begin{array}{l} C_1 : [F_1]^{F_{c1},F_{b1},F_{r1},Ks}_{Rs} \\ C_2 : [F_2]^{F_{c2},F_{b2},F_{r2},Ls}_{Rs} \\ E \simeq F_0 \end{array}}{\begin{array}{l} \texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2 : \\ \quad [\text{if } F_0 \text{ then } F_1 \text{ else } F_2]^{F_{c1} \text{ or } F_{c2},F_{b1} \text{ or } F_{b2},F_{r1} \text{ or } F_{r2},Ks\cup Ls}_{Rs} \end{array}}$$

Figure 4.15: Rules for $se, Is, Vs \vdash C : F$ (3/8)

$$\frac{se, Is, Vs \vdash C : [\,F\,]_{Rs}^{F_c, F_b, F_r, Ks}}{\begin{array}{l} se, Is, Vs \vdash \texttt{while (}E\texttt{)}\ C : \\ \quad [\,\text{!next.continues and !next.breaks}\,]_{Rs}^{\text{false}, \text{false}, F_r, Ks} \end{array}}$$

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [\,F\,]_{\{R_1, \ldots, R_n\}}^{F_c, \text{false}, F_r, Ks} \\ E \simeq H \end{array}}{\begin{array}{l} se, Is, Vs \vdash \texttt{while (}E\texttt{)}\ C : \\ \quad [\,\text{!next.continues and !next.breaks and} \\ \qquad (\text{next.executes => !}H[\text{next/now}][R_1\text{'}/R_1, \ldots, R_n\text{'}/R_n]) \\ \quad]_{\{R_1, \ldots, R_n\}}^{\text{false}, \text{false}, F_r, Ks} \end{array}}$$

Figure 4.16: Rules for $se, Is, Vs \vdash C : F$ (4/8)

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{\{R_1,\ldots,R_n\}} \\ E \simeq H \\ \mathit{Invariant}(G, H, F)_{\{R_1,\ldots,R_n\}} \\ \#I_s \text{ does not occur in } G \end{array}}{\begin{array}{l} se, Is, Vs \vdash \texttt{invariant}\ G\ \texttt{decreases}\ T\ \texttt{while (}E\texttt{)}\ C : \\ \quad [\,\text{!next.continues and !next.breaks and} \\ \qquad (G[\text{now}/\text{next}][R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \Rightarrow \\ \qquad\quad \text{exists } \#I_s : G[\#I_s/\text{next}] \text{ and} \\ \qquad\qquad \text{if } \#I_s\text{.continues or } \#I_s\text{.breaks} \\ \qquad\qquad\quad \text{then next.executes} \\ \qquad\qquad\quad \text{else next == } \#I_s) \\ \qquad]^{\text{false},\text{false},F_r,Ks}_{\{R_1,\ldots,R_n\}} \end{array}}$$

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [F]^{F_c, \text{false}, F_r, Ks}_{\{R_1,\ldots,R_n\}} \\ E \simeq H \\ \mathit{Invariant}(G, H, F)_{\{R_1,\ldots,R_n\}} \\ \#I_s \text{ does not occur in } G \end{array}}{\begin{array}{l} se, Is, Vs \vdash \texttt{invariant}\ G\ \texttt{decreases}\ T\ \texttt{while (}E\texttt{)}\ C : \\ \quad [\,\text{!next.continues and !next.breaks and} \\ \qquad (\text{next.executes} \Rightarrow \\ \qquad\quad !H[\text{next}/\text{now}][R_1\text{'}/R_1, \ldots, R_n\text{'}/R_n]) \text{ and} \\ \qquad (G[\text{now}/\text{next}][R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \Rightarrow \\ \qquad\quad \text{exists } \#I_s : G[\#I_s/\text{next}] \text{ and} \\ \qquad\qquad \text{if } \#I_s\text{.continues or } \#I_s\text{.breaks} \\ \qquad\qquad\quad \text{then next.executes} \\ \qquad\qquad\quad \text{else next == } \#I_s) \\ \qquad]^{\text{false},\text{false},F_r,Ks}_{\{R_1,\ldots,R_n\}} \end{array}}$$

Figure 4.17: Rules for $se, Is, Vs \vdash C : F$ (5/8)

$$\texttt{continue} : [\text{next.continues}]^{\text{true},\text{false},\text{false},\emptyset}_{\emptyset}$$

$$\texttt{break} : [\text{next.breaks}]^{\text{false},\text{true},\text{false},\emptyset}_{\emptyset}$$

$$\frac{T \simeq E}{\begin{array}{l}\texttt{return}\, E : \\ \quad [\text{next.returns and next.value=}T\,]^{\text{false},\text{false},\text{true},\emptyset}_{\emptyset}\end{array}}$$

$$\frac{T \simeq E}{\begin{array}{l}\texttt{throw}\, I\, E : \\ \quad [\text{next.throws } I \text{ and next.value=}T\,]^{\text{false},\text{false},\text{false},\{I\}}_{\emptyset}\end{array}}$$

$$\frac{\begin{array}{l}I_v \notin Vs \\ se, Is, Vs \vdash C_1 : [F_1]^{F_{c1},F_{b1},F_{r1},Ks}_{\{R_1,\ldots,R_n\}} \\ se, Vs \cup \{I_v\} \vdash C_2 : [F_2]^{F_{c2},F_{b2},F_{r2},Ls}_{\{R_1,\ldots,R_n,I_v\}} \\ ?I_v \in \{R_1,\ldots,R_n\} \\ \$I_1,\ldots,\$I_n,\#I_s \text{ do not occur in } F_1 \text{ and } F_2 \\ \$I_a,\$I_b,\#I_t \text{ do not occur in } F_2 \\ \{\$I_a,\$I_b,\#I_s\} \cap \{\$I_1,\ldots,\$I_n,\#I_s\} = \emptyset\end{array}}{\begin{array}{l}se, Is, Vs \vdash \text{try } C_1 \text{ catch}(I_k\ I_v)\ C_2 : \\ \quad [\text{exists } \$I_1,\ldots,\$I_n\text{: exstate } \#I_s\text{:} \\ \qquad F_1[\#I_s/\text{next}][\$I_1/R_1\text{'},\ldots,\$I_n/R_n\text{'}][I_v/?I_v, I_v\text{'}/?I_v\text{'}] \text{ and} \\ \qquad \text{if } \#I_s\text{.throws } I_k \text{ then} \\ \qquad\quad \text{exists } \$I_a,\$I_b\text{: exstate } \#I_t\text{:} \\ \qquad\qquad \$I_a = \#I_s\text{.value and } \#I_t\text{.executes and} \\ \qquad\qquad F_2[\#I_t/\text{now}][\$I_a/I_v][\$I_1/R_1,\ldots,\$I_n/R_n][\$I_b/I_v\text{'}] \\ \qquad\qquad\quad [I_v/?I_v, I_v\text{'}/?I_v\text{'}] \\ \qquad \text{else} \\ \qquad\quad R_1\text{'=}\$I_1 \text{ and } \ldots \text{ and } R_n\text{'=}\$I_n \text{ and next==}\#I_s \\ \quad]^{F_{c1} \text{ or } F_{c2}, F_{b1} \text{ or } F_{b2}, F_{r1} \text{ or } F_{r2}, (Ks\setminus\{I_k\})\cup Ls}_{(\{R_1,\ldots,R_n\}\setminus\{?I_v\})\cup\{I_v\}}\end{array}}$$

Figure 4.18: Rules for $se, Is, Vs \vdash C : F$ (6/8)

$$\frac{\begin{array}{l}
I_v \notin Vs \\
se, Is, Vs \vdash C_1 : [F_1]^{F_{c1}, F_{b1}, F_{r1}, Ks}_{\{R_1, \ldots, R_n\}} \\
se, Vs \cup \{I_v\} \vdash C_2 : [F_2]^{F_{c2}, F_{b2}, F_{r2}, Ls}_{\{R_1, \ldots, R_n, I_v\}} \\
?I_v \notin \{R_1, \ldots, R_n\} \\
\$I_1, \ldots, \$I_n, \#I_s \text{ do not occur in } F_1 \text{ and } F_2 \\
\$I_a, \$I_b, \#I_t \text{ do not occur in } F_2 \\
\{\$I_a, \$I_b, \#I_s\} \cap \{\$I_1, \ldots, \$I_n, \#I_s\} = \emptyset
\end{array}}{\begin{array}{l}
se, Is, Vs \vdash \textsf{try } C_1 \textsf{ catch(}I_k\ I_v\textsf{) } C_2 : \\
\quad [\textsf{exists } \$I_1, \ldots, \$I_n\textsf{: exstate } \#I_s\textsf{:} \\
\quad\quad F_1[\#I_s/\textsf{next}][\$I_1/R_1\textsf{'}, \ldots, \$I_n/R_n\textsf{'}][I_v/?I_v, I_v\textsf{'}/?I_v\textsf{'}] \textsf{ and} \\
\quad\quad \textsf{if } \#I_s\textsf{.throws } I_k \textsf{ then} \\
\quad\quad\quad \textsf{exists } \$I_a, \$I_b\textsf{: exstate } \#I_t\textsf{:} \\
\quad\quad\quad\quad \$I_a \textsf{ = } \#I_s\textsf{.value and } \#I_t\textsf{.executes and} \\
\quad\quad\quad\quad F_2[\#I_t/\textsf{now}][\$I_a/I_v][\$I_1/R_1, \ldots, \$I_n/R_n][\$I_b/I_v\textsf{'}] \\
\quad\quad\quad\quad\quad [I_v/?I_v, I_v\textsf{'}/?I_v\textsf{'}] \\
\quad\quad \textsf{else} \\
\quad\quad\quad R_1\textsf{'=}\$I_1 \textsf{ and } \ldots \textsf{ and } R_n\textsf{'=}\$I_n \textsf{ and next==}\#I_s \\
\quad]^{F_{c1} \textsf{ or } F_{c2}, F_{b1} \textsf{ or } F_{b2}, F_{r1} \textsf{ or } F_{r2}, (Ks \setminus \{I_k\}) \cup Ls}_{\{R_1, \ldots, R_n\}}
\end{array}}$$

$$\frac{\begin{array}{l}
I_v \in Vs \\
se, Is, Vs \vdash C_1 : [F_1]^{F_{c1}, F_{b1}, F_{r1}, Ks}_{\{R_1, \ldots, R_n\}} \\
se, Vs \cup \{I_v\} \vdash C_2 : [F_2]^{F_{c2}, F_{b2}, F_{r2}, Ls}_{\{R_1, \ldots, R_n, I_v\}} \\
\$I_1, \ldots, \$I_n, \#I_s \text{ do not occur in } F_1 \text{ and } F_2 \\
\$I_a, \$I_b, \#I_t \text{ do not occur in } F_2 \\
\{\$I_a, \$I_b, \#I_s\} \cap \{\$I_1, \ldots, \$I_n, \#I_s\} = \emptyset
\end{array}}{\begin{array}{l}
se, Is, Vs \vdash \textsf{try } C_1 \textsf{ catch(}I_k\ I_v\textsf{) } C_2 : \\
\quad [\textsf{exists } \$I_1, \ldots, \$I_n\textsf{: exstate } \#I_s\textsf{:} \\
\quad\quad F_1[\#I_s/\textsf{next}][\$I_1/R_1\textsf{'}, \ldots, \$I_n/R_n\textsf{'}] \textsf{ and} \\
\quad\quad \textsf{if } \#I_s\textsf{.throws } I_k \textsf{ then} \\
\quad\quad\quad \textsf{exists } \$I_a, \$I_b\textsf{: exstate } \#I_t\textsf{:} \\
\quad\quad\quad\quad \$I_a \textsf{ = } \#I_s\textsf{.value and } \#I_t\textsf{.executes and} \\
\quad\quad\quad\quad F_2[\#I_t/\textsf{now}][\$I_a/I_v][\$I_1/R_1, \ldots, \$I_n/R_n][\$I_b/I_v\textsf{'}] \\
\quad\quad \textsf{else} \\
\quad\quad\quad R_1\textsf{'=}\$I_1 \textsf{ and } \ldots \textsf{ and } R_n\textsf{'=}\$I_n \textsf{ and next==}\#I_s \\
\quad]^{F_{c1} \textsf{ or } F_{c2}, F_{b1} \textsf{ or } F_{b2}, F_{r1} \textsf{ or } F_{r2}, (Ks \setminus \{I_k\}) \cup Ls}_{\{R_1, \ldots, R_n\}}
\end{array}}$$

Figure 4.19: Rules for $se, Is, Vs \vdash C : F$ (7/8)

$$\frac{\begin{array}{l} se(I_m) = \langle (J_1,\ldots,J_p), \langle Rs, Ks, F_C, F_R, T\rangle\rangle \\ E_1 \simeq T_1, \ldots, E_p \simeq T_p \\ I_r \in \{V_1,\ldots,V_o\} \vee I_r \notin \{I_1,\ldots,I_n\} \\ \$I_1,\ldots,\$I_p,\$L_1,\ldots,\$L_p,\$R \text{ do not occur in } F_C \text{ and } F_R \end{array}}{\begin{array}{l} se, Is, \{V_1,\ldots,V_o\} \vdash I_r = I_m\,(E_1,\ldots,E_p) : \\ \quad [\text{exists } \$I_1,\ldots,\$I_p,\$L_1,\ldots,\$L_p,\$R\text{:} \\ \qquad \$I_1\text{=}T_1 \text{ and } \ldots \text{ and } \$I_p\text{=}T_p \text{ and} \\ \qquad (F_C \text{ => } F_R) \\ \qquad\quad [\$I_1/J_1,\ldots,\$I_p/J_p,\$L_1/J_1',\ldots,\$L_p/J_p'][\$R/I_r'] \\ \qquad\quad [?V_1/V_1,\ldots,?V_o/V_o,?V_1'/V_1',\ldots,?V_o'/V_o'] \text{ and} \\ \qquad \text{if next.executes} \\ \qquad\quad \text{then } I_r' = \text{next.value} \\ \qquad\quad \text{else } I_r' = I_r \\ \quad]^{\text{false},\text{false},\text{false},Ks}_{(Is\backslash\{V_1,\ldots,V_o\})\cup\{?I:I\in Is\cap\{V_1,\ldots,V_o\}\}\cup\{I_r\}} \end{array}}$$

$$\frac{\begin{array}{l} se(I_m) = \langle (J_1,\ldots,J_p), \langle Rs, Ks, F_C, F_R, T\rangle\rangle \\ E_1 \simeq T_1, \ldots, E_p \simeq T_p \\ I_r \in Is\backslash\{V_1,\ldots,V_o\} \\ \$I_1,\ldots,\$I_p,\$L_1,\ldots,\$L_p,\$R \text{ do not occur in } F_C \text{ and } F_R \end{array}}{\begin{array}{l} se, Is, \{V_1,\ldots,V_o\} \vdash I_r = I_m\,(E_1,\ldots,E_p) : \\ \quad [\text{exists } \$I_1,\ldots,\$I_p,\$L_1,\ldots,\$L_p,\$R\text{:} \\ \qquad \$I_1\text{=}T_1 \text{ and } \ldots \text{ and } \$I_p\text{=}T_p \text{ and} \\ \qquad (F_C \text{ => } F_R) \\ \qquad\quad [\$I_1/J_1,\ldots,\$I_p/J_p,\$L_1/J_1',\ldots,\$L_p/J_p'][\$R/I_r'] \\ \qquad\quad [?V_1/V_1,\ldots,?V_o/V_o,?V_1'/V_1',\ldots,?V_o'/V_o'] \text{ and} \\ \qquad \text{if next.executes} \\ \qquad\quad \text{then } I_r' = \text{next.value} \\ \qquad\quad \text{else } I_r' = \$R \\ \quad]^{\text{false},\text{false},\text{false},Ks}_{(Is\backslash\{V_1,\ldots,V_o\})\cup\{?I:I\in Is\cap\{V_1,\ldots,V_o\}\}} \end{array}}$$

$$\begin{array}{l} se, Is, Vs \vdash \texttt{assert}\ F : \\ \quad [\text{if } F \text{ then readsonly else false}]^{\text{false},\text{false},\text{false},\emptyset}_{\emptyset} \end{array}$$

Figure 4.20: Rules for $se, Is, Vs \vdash C : F$ (8/8)

4.10 Judgement $se, Is, Vs \vdash C \downarrow^J F$

The rules for this judgement are constructed in such a way that they verify for given specification environment *se*, recursive method names *Is*, and local variable names *Vs*, whether command C terminates in every state in which formula F holds provided that $\$J$ denotes a bound for the depth of method calls allowed by the execution of C (if the bound is exceeded, the method does not terminate).

We highlight the following items:

- The overall structure of the rules resemble those for verifying the "well-definedness" of expression evaluations; both kinds of judgements could be correspondingly combined to a single "top-down checking" judgement.
- The only potential sources of non-termination are loops and method calls; our focus is therefore on the rules for these constructs.
- Loops without a loop specification can only be verified in trivial cases (the loop condition does already not hold at the very beginning).
- For loops with a specification, we have to verify
  1. that formula G indeed represents an invariant and that G holds in the initial state,
  2. that the loop body C terminates whenever the loop is entered (i.e. it holds in those states that are characterized by the loop expression E and the invariant G),
  3. that the loop terminates in a finite number of iterations.

  The proof of the last item involves the loop termination measure T which must denote a natural number that is decreased by every iteration.
- For method calls, we have to verify that the method's precondition holds and, if we call a method in the recursion set *Is* (which is only possible if the current method is a member of this set) that the termination measure of that method denotes a natural number that is smaller than the one denoted by $\$J$.

Rather than natural numbers, any well-founded ordering might be chosen as the domain of the termination measure; furthermore, different recursive method sets in the same program may chose different well-founded orderings. Since a typical well-founded ordering are tuples of natural numbers under lexicographical ordering, every recursive method set might choose a domain of tuples of a certain length n with $n = 1$ being the case presented in this section.

$se, Is, Vs \vdash I = E \downarrow^J F$

$$\frac{I \notin Vs \qquad se, Is, Vs \cup \{I\} \vdash C \downarrow^J F[?I/I]}{se, Is, Vs \vdash \texttt{var}\ I\texttt{;}\ C \downarrow^J F}$$

$$\frac{I \in Vs \qquad \$K \text{ does not occur in } F \qquad se, Is, Vs \vdash C \downarrow^J \textsf{exists}\ \$K\textsf{:}\ F[\$K/I]}{se, Is, Vs \vdash \texttt{var}\ I\texttt{;}\ C \downarrow^J F}$$

$$\frac{I \notin Vs \qquad E \simeq T \qquad se, Is, Vs \cup \{I\} \vdash C \downarrow^J I\textsf{=}T[?I/I]\ \textsf{and}\ F[?I/I]}{se, Is, Vs \vdash \texttt{var}\ I\texttt{=}E\texttt{;}\ C \downarrow^J F}$$

$$\frac{I \in Vs \qquad E \simeq T \qquad \$K \text{ does not occur in } T \text{ and } F \qquad se, Is, Vs \vdash C \downarrow^J \textsf{exists}\ \$K\textsf{:}\ I = T[\$K/I]\ \textsf{and}\ F[\$K/I]}{se, Is, Vs \vdash \texttt{var}\ I\texttt{=}E\texttt{;}\ C \downarrow^J F}$$

$$\frac{se, Is, Vs \vdash \textsc{post}(C_1, F) = G \qquad se, Is, Vs \vdash C_1 \downarrow^J F \qquad se, Is, Vs \vdash C_2 \downarrow^J G}{se, Is, Vs \vdash C_1\texttt{;}C_2 \downarrow^J F}$$

Figure 4.21: Rules for $se, Is, Vs \vdash C \downarrow^J F$ (1/4))

$$\frac{\begin{array}{l} E \simeq G \\ se, Is, Vs \vdash C \downarrow^J F \text{ and } G \end{array}}{se, Is, Vs \vdash \texttt{if (}E\texttt{)}\ C \downarrow^J F}$$

$$\frac{\begin{array}{l} E \simeq G \\ se, Is, Vs \vdash C_1 \downarrow^J F \text{ and } G \\ se, Is, Vs \vdash C_2 \downarrow^J F \text{ and } !G \end{array}}{se, Is, Vs \vdash \texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2 \downarrow^J F}$$

$$\frac{\begin{array}{l} E \simeq G \\ \models_\emptyset F \text{ => } !G \end{array}}{se, Is, Vs \vdash \texttt{while (}E\texttt{)}\ C \downarrow^J F}$$

$$\frac{\begin{array}{l} \mathit{wellformed}(LS) \\ LS = \texttt{invariant}\ G\ \texttt{decreases}\ T \\ E \simeq H \\ se, Is, Vs \vdash C : [F_R]^{F_c, F_b, F_r, Ks}_{\{R_1, \ldots, R_n\}} \\ \mathit{Invariant}(G, H, F_R)_{\{R_1, \ldots, R_n\}} \\ \models_\emptyset F \text{ => } G[\text{now/next}][I_1/I_1', \ldots, I_n/I_n'] \\ \$I_1, \ldots, \$I_n, \#I_s, \#I_t \text{ do not occur in } F, G, H \\ P = \\ \quad \#I_s\text{.executes and} \\ \quad F[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \text{ and} \\ \quad G[\#I_s/\text{now}, \#I_t/\text{next}] \\ \quad\quad [\$I_1/R_1, \ldots, \$I_n/R_n, R_1/R_1', \ldots, R_n/R_n'] \text{ and } H \text{ and} \\ \quad (\#I_t\text{.executes or } \#I_t\text{.continues}) \\ C \downarrow^J \text{ exists } \$I_1, \ldots, \$I_n\text{: exstate } \#I_s, \#I_t\text{: } P \\ \models_{\{R_1, \ldots, R_n\}} \\ \quad \text{forall } \$I_1, \ldots, \$I_n\text{: allstate } \#I_s, \#I_t\text{:} \\ \quad\quad P \text{ and now.executes and } F_R \text{ and} \\ \quad\quad \text{(next.executes or next.continues) =>} \\ \quad\quad\quad \text{let } \$N\text{=}T, \$M = T[R_1'/R_1, \ldots, R_n'/R_n] \\ \quad\quad\quad \text{in nat(}\$M\text{) and } \$M < \$N \end{array}}{se, Is, Vs \vdash LS\ \texttt{while (}E\texttt{)}\ C \downarrow^J F}$$

Figure 4.22: Rules for $se, Is, Vs \vdash C \downarrow^J F$ (2/4))

$se, Is, Vs \vdash \texttt{continue} \downharpoonleft^{J} F$

$se, Is, Vs \vdash \texttt{break} \downharpoonleft^{J} F$

$se, Is, Vs \vdash \texttt{return}\, E \downharpoonleft^{J} F$

$se, Is, Vs \vdash \texttt{throw}\, I_k\; E \downharpoonleft^{J} F$

$$\frac{\begin{array}{l} I_v \notin Vs \\ se, Is, Vs \vdash C_1 \downharpoonleft^{J} F \\ se, Is, Vs \vdash \textsc{post}(C_1, F) = G \\ \#I_s \text{ does not occur in } G \\ se, Is, Vs \vdash C_2 \;\downharpoonleft^{J} \\ \quad \textsf{exstate } \#I_s\textsf{:} \\ \quad\quad G[\#I_s/\texttt{now}][?I_v/I_v] \textsf{ and} \\ \quad\quad \#I_s\textsf{.throws } I_k \textsf{ and } \#I_s\textsf{.value = } I_v \end{array}}{se, Is, Vs \vdash \texttt{try}\, C_1\; \texttt{catch}\,\texttt{(}I_k\; I_v\texttt{)}\; C_2 \downharpoonleft^{J} F}$$

$$\frac{\begin{array}{l} I_v \in Vs \\ se, Is, Vs \vdash C_1 \downharpoonleft^{J} F \\ se, Is, Vs \vdash \textsc{post}(C_1, F) = G \\ \$I, \#I_s \text{ do not occur in } G \\ se, Is, Vs \vdash C_2 \;\downharpoonleft^{J} \\ \quad \textsf{exists } \$I\textsf{: exstate } \#I_s\textsf{:} \\ \quad\quad G[\#I_s/\texttt{now}][\$I/I_v] \textsf{ and} \\ \quad\quad \#I_s\textsf{.throws } I_k \textsf{ and } \#I_s\textsf{.value = } I_v \end{array}}{se, Is, Vs \vdash \texttt{try}\, C_1\; \texttt{catch}\,\texttt{(}I_k\; I_v\texttt{)}\; C_2 \downharpoonleft^{J} F}$$

Figure 4.23: Rules for $se, Is, Vs \vdash C \downharpoonleft^{J} F$ (3/4))

$$\dfrac{\begin{array}{l}
I_m \notin Is \\
se(I_m) = \langle (J_1,\ldots,J_p), \langle Rs, Ks, F_C, F_R, T\rangle\rangle \\
E_1 \simeq T_1, \ldots, E_p \simeq T_p \\
\$I_1,\ldots,\$I_p \text{ do not occur in } F_C \\
\models_\emptyset \textsf{ now.executes and (exists } \$J : F\textsf{) =>} \\
\qquad \textsf{forall } \$I_1,\ldots,\$I_p \textsf{ :} \\
\qquad\quad \$I_1\textsf{=}T_1 \textsf{ and } \ldots \textsf{ and } \$I_p\textsf{=}T_p \textsf{ =>} \\
\qquad\quad F_C[\$I_1/J_1,\ldots,\$I_p/J_p,\$I_1/J_1\textsf{'},\ldots,\$I_p/J_p\textsf{'}] \\
\qquad\qquad [?V_1/V_1,\ldots,?V_o/V_o,?V_1\textsf{'}/V_1\textsf{'},\ldots,?V_o\textsf{'}/V_o\textsf{'}]
\end{array}}{se, Is, \{V_1,\ldots,V_o\} \vdash I_r = I_m\,(E_1,\ldots,E_p) \downarrow^J F}$$

$$\dfrac{\begin{array}{l}
I_m \in Is \\
se(I_m) = \langle (J_1,\ldots,J_p), \langle Rs, Ks, F_C, F_R, T\rangle\rangle \\
E_1 \simeq T_1, \ldots, E_p \simeq T_p \\
\$I_1,\ldots,\$I_p \text{ do not occur in } F_C \text{ and } T \\
\models_\emptyset \textsf{ forall } \$I_1,\ldots,\$I_p,\$J\textsf{:} \\
\qquad \textsf{now.executes and } F \textsf{ and} \\
\qquad \$I_1\textsf{=}T_1 \textsf{ and } \ldots \textsf{ and } \$I_p\textsf{=}T_p \textsf{ =>} \\
\qquad\quad F_C[\$I_1/J_1,\ldots,\$I_p/J_p,\$I_1/J_1\textsf{'},\ldots,\$I_p/J_p\textsf{'}] \\
\qquad\qquad [?V_1/V_1,\ldots,?V_o/V_o,?V_1\textsf{'}/V_1\textsf{'},\ldots,?V_o\textsf{'}/V_o\textsf{'}] \textsf{ and} \\
\qquad\quad \textsf{let } \$M\textsf{=}T[\$I_1/J_1,\ldots,\$I_p/J_p,\$I_1/J_1\textsf{'},\ldots,\$I_p/J_p\textsf{'}] \\
\qquad\qquad\qquad [?V_1/V_1,\ldots,?V_o/V_o,?V_1\textsf{'}/V_1\textsf{'},\ldots,?V_o\textsf{'}/V_o\textsf{'}] \\
\qquad\quad \textsf{in nat(}\$M\textsf{) and } \$M \textsf{ < } \$J
\end{array}}{se, Is, \{V_1,\ldots,V_o\} \vdash I_r = I_m\,(E_1,\ldots,E_p) \downarrow^J F}$$

$$se, Is, Vs \vdash \texttt{assert}\ G \downarrow^J F$$

Figure 4.24: Rules for $se, Is, Vs \vdash C \downarrow^J F$ (4/4))

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{\{R_1, \ldots, R_n\}} \\ \$I_1, \ldots, \$I_n, \#I_s \text{ do not occur in } Q \end{array}}{\begin{array}{l} se, Is, Vs \vdash \text{PRE}(C, Q) = \\ \quad \textsf{forall } \$I_1, \ldots, \$I_n\textsf{: allstate } \#I_s\textsf{:} \\ \qquad (\textsf{now.executes and } F[\#I_s/\textsf{next}][\$I_1/R_1\textsf{'}, \ldots, \$I_n/R_n\textsf{'}]) \textsf{ =>} \\ \qquad Q[\#I_s/\textsf{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \end{array}}$$

Figure 4.25: Rules for $se, Is, Vs \vdash \text{PRE}(C, Q) = P$

4.11 Judgement $se, Is, Vs \vdash \textbf{PRE}(C, Q) = P$

The rule for this judgement constructs (for given specification environment *se*, recursive method names *Is*, and local variable names *Vs*), from a command *C* and a condition *Q* on its poststate a suitable condition *P* on its prestate. It is interesting to note that the precondition can be computed generically from the formula denoting the command's state transition relation.

The precondition is not necessarily the weakest one (it indeed is, if the command does not contain loops).

This judgement is actually not used by any rule of the calculus; it might nevertheless be provided to the user on demand for exploring the properties of a program.

$$\frac{\begin{array}{l} se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{\{R_1, \ldots, R_n\}} \\ \$I_1, \ldots, \$I_n, \#I_s \text{ do not occur in } P \end{array}}{\begin{array}{l} se, Is, Vs \vdash \textsc{post}(C,P) = \\ \quad \text{exists } \$I_1, \ldots, \$I_n\text{: exstate } \#I_s\text{:} \\ \quad\quad \#I_s\text{.executes and} \\ \quad\quad P[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \text{ and} \\ \quad\quad F[\#I_s/\text{now}, \text{now}/\text{next}] \\ \quad\quad\quad [\$I_1/R_1, \ldots, \$I_n/R_n, R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \end{array}}$$

Figure 4.26: Rules for $se, Is, Vs \vdash \textsc{post}(C,P) = Q$

4.12 Judgement $se, Is, Vs \vdash \textsc{post}(C,P) = Q$

The generic rule for this judgement constructs (for given specification environment *se*, recursive method names *Is*, and local variable names *Vs*), from a command *C* and a condition *P* on its prestate a suitable condition *Q* on its poststate. It is interesting to note that the postcondition can be computed generically from the formula denoting the command's state transition relation.

The postcondition is not necessarily the strongest one (but if the command does not contain loops it indeed is).

The judgement is used in the rules for verifying the well-definedness, for computing the state transition relation, and for verifying the termination of a command.

4.13 Judgement $se, Is, Vs \vdash \textbf{ASSERT}(C, P) = C'$

The rules for this judgement construct (for given specification environment *se*, recursive method names *Is*, and local variable names *Vs*), from command *C* and a condition *P* on its prestate a version *C′* of *C* in which every subcommand (including *C* itself) is preceded by an annotation expressing all the information available on the prestate of the subcommand.

The overall structure of the rule set closely resembles (a non-generic version of) the rule set for the computation of a command's postcondition.

This judgement is actually not used by any rule of the calculus; it might nevertheless be provided to the user on demand for exploring the properties of a program.

$$se, Is, Vs \vdash \textsc{assert}(R\texttt{=}E, P) = \texttt{assert}\ P\texttt{;}\ R\texttt{=}E$$

$$\frac{\begin{array}{l} I \notin Vs \\ se, Is, Vs \cup \{I\} \vdash \textsc{assert}(C, P[?I/I]) = C' \end{array}}{se, Is, Vs \vdash \textsc{assert}(\texttt{var}\ I\texttt{;}C, P) = \texttt{assert}\ P\texttt{;}\ \texttt{var}\ I\texttt{;}C'}$$

$$\frac{\begin{array}{l} I \in Vs \\ \$J \text{ does not occur in } P \\ se, Is, Vs \vdash \textsc{assert}(C, \mathsf{exists}\ \$J : P[\$J/I]) = C' \end{array}}{se, Is, Vs \vdash \textsc{assert}(\texttt{var}\ I\texttt{;}C, P) = \texttt{assert}\ P\texttt{;}\ \texttt{var}\ I\texttt{;}C'}$$

$$\frac{\begin{array}{l} I \notin Vs \\ E \simeq T \\ se, Is, Vs \cup \{I\} \vdash \textsc{assert}(C, P[?I/I]\ \mathsf{and}\ I = T[?I/I]) = C' \end{array}}{\begin{array}{l} se, Is, Vs \vdash \textsc{assert}(\texttt{var}\ I\texttt{=}E\texttt{;}C, P) = \\ \quad \texttt{assert}\ P\texttt{;}\ \texttt{var}\ I\texttt{=}E\texttt{;}C' \end{array}}$$

$$\frac{\begin{array}{l} I \in Vs \\ E \simeq T \\ \$J \text{ does not occur in } P \text{ and in } T \\ se, Is, Vs \vdash \textsc{assert}(C, \mathsf{exists}\ \$J : P[\$J/I]\ \mathsf{and}\ I = T[\$J/I]) = C' \end{array}}{\begin{array}{l} se, Is, Vs \vdash \textsc{assert}(\texttt{var}\ I\texttt{=}E\texttt{;}C, P) = \\ \quad \texttt{assert}\ P\texttt{;}\ \texttt{var}\ I\texttt{=}E\texttt{;}C' \end{array}}$$

$$\frac{\begin{array}{l} se, Is, Vs \vdash \textsc{post}(C_1, P) = Q \\ se, Is, Vs \vdash \textsc{assert}(C_1, P) = C_1' \\ se, Is, Vs \vdash \textsc{assert}(C_2, Q) = C_2' \end{array}}{se, Is, Vs \vdash \textsc{assert}(C_1\texttt{;}C_2, P) = \texttt{assert}\ P\texttt{;}\ C_1'\texttt{;}C_2'}$$

$$\frac{\begin{array}{l} E \simeq F \\ se, Is, Vs \vdash \textsc{assert}(C, P\ \mathsf{and}\ F) = C' \end{array}}{se, Is, Vs \vdash \textsc{assert}(\texttt{if (}E\texttt{)}\ C, P) = \texttt{assert}\ P\texttt{;}\ \texttt{if (}E\texttt{)}\ C'}$$

$$\frac{\begin{array}{l} E \simeq F \\ se, Is, Vs \vdash \textsc{assert}(C_1, P\ \mathsf{and}\ F) = C_1' \\ se, Is, Vs \vdash \textsc{assert}(C_2, P\ \mathsf{and}\ !F) = C_2' \end{array}}{\begin{array}{l} se, Is, Vs \vdash \textsc{assert}(\texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2, P) = \\ \quad \texttt{assert}\ P\texttt{;}\ \texttt{if (}E\texttt{)}\ C_1'\ \texttt{else}\ C_2' \end{array}}$$

Figure 4.27: Rules for $se, Is, Vs \vdash \textsc{assert}(C, P) = C'$ (1/3)

$$\dfrac{\begin{array}{l}
E \simeq H \\
se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{\{R_1, \ldots, R_n\}} \\
\$I_1, \ldots, \$I_n, \#I_s \text{ do not occur in } P \\
se, Is, Vs \vdash \textsc{assert}(C, \\
\quad H \text{ and} \\
\quad \text{exists } \$I_1, \ldots, \$I_n\text{: exists } \#I_s\text{:} \\
\qquad P[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n]) = C'
\end{array}}{\begin{array}{l}
se, Is, Vs \vdash \textsc{assert}(\texttt{while(}E\texttt{)}\ C, P) = \\
\quad \texttt{assert}\ P\texttt{;}\ \texttt{while(}E\texttt{)}\ C'
\end{array}}$$

$$\dfrac{\begin{array}{l}
E \simeq H \\
se, Is, Vs \vdash C : [F]^{F_c, F_b, F_r, Ks}_{\{R_1, \ldots, R_n\}} \\
Invariant(G, H, F)_{\{R_1, \ldots, R_n\}} \\
\$I_1, \ldots, \$I_n, \#I_s, \#I_t \text{ do not occur in } P \\
se, Is, Vs \vdash \textsc{assert}(C, \\
\quad H \text{ and} \\
\quad \text{exists } \$I_1, \ldots, \$I_n\text{: exstate } \#I_s, \#I_t\text{:} \\
\qquad P[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \text{ and} \\
\qquad (G[\text{now}/\text{next}][R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \\
\qquad\quad [\#I_s/\text{now}][\$I_1/R_1, \ldots, \$I_n/R_n] \Rightarrow \\
\qquad\quad G[\#I_s/\text{now}][\$I_1/R_1, \ldots, \$J_n/R_n] \\
\qquad\qquad [\#I_t/\text{next}][R_1/R_1\text{'}, \ldots, R_n/R_n\text{'}] \text{ and} \\
\qquad\quad (!(\#I_t\text{.continues or } \#I_t\text{.breaks}) \Rightarrow \\
\qquad\qquad \text{now == } \#I_t))) = C'
\end{array}}{\begin{array}{l}
se, Is, Vs \vdash \\
\quad \textsc{assert}(\texttt{invariant}\ G\ \texttt{decreases}\ T\ \texttt{while}\,(E)\ C, P) = \\
\quad \texttt{assert}\ P\texttt{;}\ \texttt{while(}E\texttt{)}\ C'
\end{array}}$$

Figure 4.28: Rules for $se, Is, Vs \vdash \textsc{assert}(C, P) = C'$ (2/3)

$se, Is, Vs \vdash \textsc{assert}(\texttt{continue}, P) = \texttt{assert}\ P\texttt{;}\ \texttt{continue}$

$se, Is, Vs \vdash \textsc{assert}(\texttt{break}, P) = \texttt{assert}\ P\texttt{;}\ \texttt{break}$

$se, Is, Vs \vdash \textsc{assert}(\texttt{return}\ E, P) = \texttt{assert}\ P\texttt{;}\ \texttt{return}\ E$

$se, Is, Vs \vdash \textsc{assert}(\texttt{throw}\ I_k\ E, P) = \texttt{assert}\ P\texttt{;}\ \texttt{throw}\ I_k\ E$

$$\frac{\begin{array}{l} I_v \notin Vs \\ \#I_s \text{ does not occur in } Q \\ se, Is, Vs \vdash \textsc{post}(C_1, P) = Q \\ se, Is, Vs \vdash \textsc{assert}(C_1, P) = C_1' \\ se, Is, Vs \cup \{I_v\} \vdash \textsc{assert}(C_2, \\ \quad \textsf{exstate}\ \#I_s\textsf{:} \\ \quad\quad Q[\#I_s/\textsf{now}][?I_v/I_v]\ \textsf{and} \\ \quad\quad \#I_s\textsf{.throws}\ I_k\ \textsf{and}\ \#I_s\textsf{.value = }I_v) = C_2' \end{array}}{\begin{array}{l} se, Is, Vs \vdash \textsc{assert}(\texttt{try}\ C_1\ \texttt{catch}\ \texttt{(}I_k\ I_v\texttt{)}\ C_2, P) = \\ \quad \texttt{assert}\ P\texttt{;}\ \texttt{try}\ C_1'\ \texttt{catch}\ \texttt{(}I_k\ I_v\texttt{)}\ C_2' \end{array}}$$

$$\frac{\begin{array}{l} I_v \in Vs \\ \$J, \#I_s \text{ do not occur in } Q \\ se, Is, Vs \vdash \textsc{post}(C_1, P) = Q \\ se, Is, Vs \vdash \textsc{assert}(C_1, P) = C_1' \\ se, Is, Vs \vdash \textsc{assert}(C_2, \\ \quad \textsf{exists}\ \$J\textsf{: exstate}\ \#I_s\textsf{:} \\ \quad\quad Q[\#I_s/\textsf{now}][\$J/I_v]\ \textsf{and} \\ \quad\quad \#I_s\textsf{.throws}\ I_k\ \textsf{and}\ \#I_s\textsf{.value = }I_v) = C_2' \end{array}}{\begin{array}{l} se, Is, Vs \vdash \textsc{assert}(\texttt{try}\ C_1\ \texttt{catch}\ \texttt{(}I_k\ I_v\texttt{)}\ C_2, P) = \\ \quad \texttt{assert}\ P\texttt{;}\ \texttt{try}\ C_1'\ \texttt{catch}\ \texttt{(}I_k\ I_v\texttt{)}\ C_2' \end{array}}$$

$se, Is, Vs \vdash \textsc{assert}(I_r\ \texttt{=}\ I_m\ \texttt{(}E_1, \ldots, E_p\texttt{)}, P) =$
$\quad \texttt{assert}\ P\texttt{;}\ I_r\ \texttt{=}\ I_m\ \texttt{(}E_1, \ldots, E_p\texttt{)}$

$se, Is, Vs \vdash \textsc{assert}(\texttt{assert}\ Q, P) = \texttt{assert}\ P\texttt{;}\ \texttt{assert}\ Q$

Figure 4.29: Rules for $se, Is, Vs \vdash \textsc{assert}(C, P) = C'$ (3/3)

$$\frac{F\ \S_{\mathrm{c}}\ F_c \quad F\ \S_{\mathrm{b}}\ F_b \quad F\ \S_{\mathrm{s}}\ {}^{Qs}_{Rs} \quad F\ \S_{\mathrm{e}}\ {}^{Ls}_{Ks}}{[F]^{F_c,F_b,F_r,Ls}_{Qs}\ \S\ {}^{Ks}_{Rs}}$$

Figure 4.30: Rules for $F\ \S\ {}^{Ks}_{Rs}$

4.14 Judgement $F\ \S\ {}^{Ks}_{Rs}$

This judgement verifies that the state transition formula $[F]^{F_c,F_b,F_r,Ls}_{Qs}$ derived from a command (the program's main command or a method's body) conforms to the overall constraints of the command's behavior and its specification. This task is delegated to four "subjudgements".

The rationale for this judgement is that is sometimes may not suffice to prove by "static" program analysis the conformance of a command to general constraints but that sometimes a "dynamic" proof might be necessary.

However, in practice only the third judgement might be of relevance (conformance to a variable frame) whose purpose is also served by the "frame shrinking rule" in the computation of command transition relations.

$$F \ \S_{\mathrm{c}}\ \mathsf{false}$$

$$\frac{\models_{\emptyset} (\mathsf{now.executes\ and}\ F) \mathrel{\mathsf{=>}} \mathsf{!next.continues}}{F \ \S_{\mathrm{c}}\ \mathsf{true}}$$

Figure 4.31: Rules for $F\ \S_{\mathrm{c}}\ F_c$

4.15 Judgement $F\ \S_{\mathrm{c}}\ F_c$

This judgement verifies that a command does not result in a “continuing” post-state; the first rule covers the typical situation where the command does not involve a `continue` statement outside a loop body; the second rule handles the general case where it can be proved that such a statement is never executed.

$F\ \S_{\mathrm{b}}$ false

$$\frac{\models_{\emptyset} (\text{now.executes and } F) \text{ => !next.breaks}}{F\ \S_{\mathrm{b}}\ \text{true}}$$

Figure 4.32: Rules for $F\ \S_{\mathrm{b}}\ F_b$

4.16 Judgement $F\ \S_{\mathrm{b}}\ F_b$

This judgement verifies that a command does not result in a "breaking" poststate; the first rule covers the typical situation where the command does not involve a `break` statement outside a loop body; the second rule handles the general case where it can be proved that such a statement is never executed.

$$\frac{Qs \subseteq Rs}{F \ \S_\mathrm{s} \ {}^{Qs}_{Rs}}$$

$$\frac{\begin{array}{l} Qs \backslash Rs = \{Q_1, \ldots, Q_o\} \\ \models_\emptyset \text{(now.executes and } F\text{) => } (Q_1\text{=}Q_o\text{' and } \ldots \text{ and } Q_o\text{=}Q_o\text{')} \end{array}}{F \ \S_\mathrm{s} \ {}^{Rs}_{Qs}}$$

Figure 4.33: Rules for $F \ \S_\mathrm{s} \ {}^{Qs}_{Rs}$

4.17 Judgement $F \ \S_\mathrm{s} \ {}^{Qs}_{Rs}$

This judgement verifies that a command does not change any variables outside the frame *Rs*; the first rule covers the typical situation where the command does not change any variables outside of frame *Qs* which is contained in *Rs*; the second rule handles the general case where it can be proved that all variables outside of *Rs* have the same value before and after the execution of the command.

$$\frac{Ls \subseteq Ks}{F\ \S_{\mathrm{s}}\ {}^{Ks}_{Ls}}$$

$$\frac{\begin{array}{l} Ls \backslash Ks = \{L_1, \ldots, L_r\} \\ \models_{\emptyset} (\texttt{now.executes and } F) \texttt{ =>} \\ \qquad (\texttt{!next.throws } L_1 \texttt{ and } \ldots \texttt{ and !next.throws } L_r) \end{array}}{F\ \S_{\mathrm{s}}\ {}^{Ls}_{Ks}}$$

Figure 4.34: Rules for $F\ \S_{\mathrm{e}}\ {}^{Ls}_{Ks}$

4.18 Judgement $F\ \S_{\mathrm{e}}\ {}^{Ls}_{Ks}$

This judgement verifies that a command does not throw any exceptions outside the set *Ks*; the first rule covers the typical situation where the command does not throw any exceptions outside *Ls* which is contained in *Rs*; the second rule handles the general case where it can be proved no command is executed that throws some exception outside of *Rs*.

4.19 Judgement $\models_{Rs} F$

This judgement verifies the validity of a specification formula F for pairs of pre- and poststates that have the same values in all variables denoted by identifiers except for those listed in Rs (this information is necessary for the handling of the specification formulas writesonly and readsonly, see below).

The validity of this judgement can be proved by

1. translating F to a classical first order predicate logic formula F' and by
2. proving the validity of F' in first order predicate logic with equality over a domain that includes the natural numbers.

The translation is informally sketched below:

- The specification predicates = and /= on values and == on states are translated to the predicates

  $$=, \neq$$

  with the classical interpretation of equality/inequality.

- The specification formulas isnat(T) and T_1<T_2 are translated to the corresponding classical formulas interpreted over the domain of natural numbers.

- The specification constants now and next are translated to the uninterpreted classical constants

  $$now, next$$

- The specification formulas U.executes, U.continues, U.breaks U.returns, U.throws and U.throws I are translated to the classical formulas

  $$executes(U'), breaks(U'), returns(U')$$
  $$throws(U'), throws(U') \wedge key(U') = I_c$$

  where the predicate/function constants remain uninterpreted and I_c is a constant that is uniquely derived from I.

- The specification term U.value is translated to the classical term

  $$value(U')$$

  where the function constant *value* remains uninterpreted.

- The formula readsonly is first translated to

$R_1 = R_1$' and ... and $R_1 = R_1$'

(where $\{R_1, \ldots, R_n\} = Rs$) and is then translated further as described below.

- The formula writesonly $S_1, \ldots, S_m$ is first translated to

  $T_1 = T_1$' and ... and $T_o = T_1$'

  (where $\{T_1, \ldots, T_o\} = Rs \backslash \{S_1, \ldots, S_m\}$) and is then translated further as described below.

- The specification constants true and false and the connectives not, and, or, =>, <=>, xor are translated to the classical constants/connectives

  $\text{TRUE}, \text{FALSE}, \neg, \wedge, , \vee, \Rightarrow, \Leftrightarrow, \not\Leftrightarrow$

- The program variables I,I', ?I, ?I' are translated to the classical constants

  $I_{la}, I_{lb}, I_{ga}, I_{gb}$

- The logical variables \$$I$ and #I are translated to the classical variables

  I_v, I_s

- The specification formulas forall \$$I$: F and exists \$$I$: F are translated to the classical formulas

  $\forall I_v : isvalue(I_v) \Rightarrow F'$
  $\exists I_v : isvalue(I_v) \wedge F'$

  with an uninterpreted unary predicate *isvalue*.

- The specification formulas allstate #I: F and exstate #I: F are translated to to the classical formulas

  $\forall I_s : isstate(I_s) \Rightarrow F'$
  $\exists I_s : isstate(I_s) \wedge F'$

  with an uninterpreted unary predicate *isstate*.

- The specification formula if F_1 then F_2 else F_3 is translated to the logical formula

  $(F_1' \Rightarrow F_2') \wedge (\neg F_1' \Rightarrow F_3')$

- A specification formula F[if F_c then T_1 else T_2] (where F is the innermost formula that includes the term if F_c then T_1 else T_2) is translated to the classical formula

  $\exists I_v : (F_c' \Rightarrow I_v = T_1') \wedge (\neg F_c' \Rightarrow I_v = T_2') \wedge F'[I_v]$

- The specification formula let $\$I_1$=$T_1,\ldots,\$I_n$=$T_n$ in F is translated to the classical formula

$$\exists I_{v1},\ldots,I_{vn} : I_{v1} = T_1' \wedge \ldots \wedge I_{vn} = T_n' \wedge F'$$

- A specification formula $F[$let $\$I_1$=$T_1,\ldots,\$I_n$=$T_n$ in $T]$ (where F is the innermost formula that includes the term let $\$I_1$=$T_1,\ldots,\$I_n$=$T_n$ in T) is translated to the classical formula

$$\exists I_{v1},\ldots,I_{vn} : I_{v1} = T_1' \wedge \ldots \wedge I_{vn} = T_n' \wedge F'[T']$$

Let $\{k_1,k_2,k_3,\ldots,k_{n-1},k_n\}$ be the set of all constants derived from specification formulas U.throws I as explained above. The validity of this formula can be proven by using the axioms depicted in Figure 4.35 which give a meaning to the uninterpreted predicate/function symbols.

$$
\begin{array}{l}
k_1 \neq k_2 \wedge k_1 \neq k_3 \wedge \ldots \wedge k_1 \neq k_{n-1} \wedge k_1 \neq k_n \\
k_2 \neq k_3 \wedge \ldots \wedge k_2 \neq k_{n-1} \wedge k_2 \neq k_n \\
\ldots \\
k_{n-1} \neq k_n \\
\forall I : \mathit{isvalue}(I) \not\Leftrightarrow \mathit{isstate}(I) \\
\forall I : \mathit{isnat}(I) \Rightarrow \mathit{isvalue}(I) \\
\mathit{isstate}(\mathit{now}) \\
\mathit{isstate}(\mathit{next}) \\
\forall I_s : \mathit{isstate}(I_s) \Rightarrow \\
\quad \mathit{isvalue}(\mathit{value}(I_s)) \wedge \\
\quad (\mathit{key}(I_s) = k_1 \vee \ldots \vee \mathit{key}(I_s) = k_n) \wedge \\
\quad (\mathit{executes}(I_s) \vee \mathit{continues}(I_s) \vee \mathit{breaks}(I_s) \vee \\
\qquad \mathit{returns}(I_s) \vee \mathit{throws}(I_s)) \wedge \\
\quad (\mathit{executes}(I_s) \Rightarrow \\
\qquad \neg(\mathit{continues}(I_s) \vee \mathit{breaks}(I_s) \vee \mathit{returns}(I_s) \vee \mathit{throws}(I_s))) \wedge \\
\quad (\mathit{continues}(I_s) \Rightarrow \\
\qquad \neg(\mathit{executes}(I_s) \vee \mathit{breaks}(I_s) \vee \mathit{returns}(I_s) \vee \mathit{throws}(I_s))) \wedge \\
\quad (\mathit{breaks}(I_s) \Rightarrow \\
\qquad \neg(\mathit{executes}(I_s) \vee \mathit{continues}(I_s) \vee \mathit{returns}(I_s) \vee \mathit{throws}(I_s))) \wedge \\
\quad (\mathit{returns}(I_s) \Rightarrow \\
\qquad \neg(\mathit{executes}(I_s) \vee \mathit{continues}(I_s) \vee \mathit{breaks}(I_s) \vee \mathit{throws}(I_s))) \wedge \\
\quad (\mathit{throws}(I_s) \Rightarrow \\
\qquad \neg(\mathit{executes}(I_s) \vee \mathit{continues}(I_s) \vee \mathit{breaks}(I_s) \vee \mathit{returns}(I_s)))
\end{array}
$$

Figure 4.35: Axioms

Bibliography

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, New York, 1998.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, Berlin, Germany, 2007.

[3] Raymond T. Boute. Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus. *ACM Transactions on Programming Languages and Systems*, 28(4):747–793, July 2006.

[4] Mike Gordon. Specification and Verification I. Lecture Notes, http:// www.cl.cam.ac.uk/ mjcg/Teaching/SpecVer1/SpecVer1.html.

[5] Eric C.R. Hehner. *A Practical Theory of Programming*. Springer, New York, 2006. http://www.cs.utoronto.ca/˜hehner/aPToP.

[6] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, London, UK, 1998.

[7] The Java Modeling Language (JML), 2008. http://www.cs.ucf.edu/ leavens/JML.

[8] Cliff B. Jones. *Systematic Software Devleopment Using VDM*. Prentice Hall, 2nd edition, 1990.

[9] K. Rustan M. Leino and James B. Saxe and Raymie Stata. Checking Java Programs via Guarded Commands. Compaq SRC Technical Note 1999-002, Compaq, 1999. http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1999-002.html.

[10] Leslie Lamport. *Specifying Systems; The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. http:// research.microsoft.com/users/lamport/tla/book.html.

[11] Carroll Morgan. *Programming from Specifications*. Prentice Hall, London, UK, 2nd edition, 1998.

[12] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[13] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986. http://people.cis.ksu.edu/~schmidt/text/densem.html.

[14] Wolfgang Schreiner. Understanding Programs. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, July 2008.

[15] Spec#, 2008. http://research.microsoft.com/SpecSharp.

Appendix A

Mathematical Language

This appendix summarizes the mathematical language used in this document. Our theoretical framework is classical Zermelo-Fraenkel set theory (ZF) formalized in first-order predicate logic (FOL), as it is presented in typical introductions to mathematics for computer scientists. We therefore mostly refrain from providing formal definitions of the concepts but focus on a presentation of the notations we use together with informal explanations of their interpretations. The exact definitions can be looked up in various text books on this subject.

In the following descriptions, we use the meta-variables $x, x_1, \ldots$ to denote object variables, $f, f_1, \ldots$ to denote function names, $p, p_1, \ldots$ to denote predicate names, $T, T_1, \ldots$ to denote terms, $F, F_1, \ldots$ to denote formulas, and $P, P_1, \ldots$ to denote generic phrases (terms or formulas).

Terms We use the following kinds of terms to denote values:

- $\mathbf{x}$: the variable x.
- $\mathbf{f}$: the constant (0-ary function) f.
- $\mathbf{f}(\mathbf{T_1}, \ldots, \mathbf{T_n})$: the application of the n-ary function f to $T_1, \ldots, T_n$ ($n \geq 1$).
- **SUCH** $\mathbf{x} : \mathbf{F}$: some x such that F is true for x, if F is true for any value, and x is arbitrary, otherwise.

Formulas We use the following kinds of formulas to denote propositions:

- $\mathbf{p}(\mathbf{T_1}, \ldots, \mathbf{T_n})$: the n-ary predicate p is true for $T_1, \ldots, T_n$ ($n \geq 1$).

- $\mathbf{T_1 = T_2}$: T_1 equals T_2.
- $\neg\mathbf{F}$: F is not true.
- $\mathbf{F_1 \land F_2}$: F_1 is true and F_2 is true.
- $\mathbf{F_1 \lor F_2}$: F_1 is true or F_2 is true (also both may be true).
- $\mathbf{F_1 \Rightarrow F_2}$: if F_1 is true, then F_2 is also true (but F_1 may be false).
- $\mathbf{F_1 \Leftrightarrow F_2}$: both F_1 and F_2 are true or both are false.
- $\forall\mathbf{x : F}$: for every x, F is true.
- $\exists\mathbf{x : F}$: for some x, F is true.

Generic Phrases We use the following generic phrases:

- IF **F** THEN $\mathbf{P_1}$ ELSE $\mathbf{P_2}$: if F is true, then P_1, else P_2.
- LET $\mathbf{x = T}$ IN **P**: P, where the value of x is the value of T (the meaning of this phrase is the same as that of $P[T/x]$, see below).

Free and Bound Variables An occurrence of a variable x in one of the phrases SUCH $x : F$, $\forall x : F$, $\exists x : F$, or LET $x = T$ IN P is called *bound* by that phrase. A non-bound occurrence of a variable in a phrase is *free* in that phrase.

Term Substitutions We introduce the following substitutions of terms by other terms in a phrase P:

- $\mathbf{P[U_1/T_1, \ldots, U_n/T_n]}$: that variant of P where all occurrences of terms $T_1, \ldots, T_n$ (pairwise-different) are replaced by terms $U_1, \ldots, U_n$.
- $\mathbf{P[U_1(x)/T_1(x), \ldots, U_n(x)/T_n(x) : F]}$: that variant of P where, for every term x for which F is true, all occurrences of terms $T_1(x), \ldots, T_n(x)$ (pairwise different and different for different x) are replaced by the corresponding terms $U_1(x), \ldots, U_n(x)$.

Sets We use the following predicates, constants, and functions on sets:

- $\mathbf{T_1 \in T_2}$: T_1 is in T_2.
- $\mathbf{T_1 \subseteq T_2}$: T_1 is a subset of T_2, i.e., every x which is in T_1 is also in T_2.
- $\emptyset$: the empty set.
- $T_1 \cap T_2$,$T_1 \cup T_2$, $T_1 \backslash T_2$: the intersection, union, and difference of T_1 and T_2.
- $\{\mathbf{T_1}, \ldots, \mathbf{T_n}\}$: the set of values $T_1, \ldots, T_n$.
- $\{\mathbf{x \in T}: \mathbf{F}\}$: the set of values x in T for which F is true.
- $\{\mathbf{f(x_1, \ldots, x_n) \in T}: \mathbf{F}\}$: the set of values $f(x_1, \ldots, x_n)$ in T such that F is true for $x_1, \ldots, x_n$.
- $\mathbb{B}$: the set $\{\text{TRUE}, \text{FALSE}\}$ of truth values (Boolean values).
- $\mathbb{N}$: the set $\{0, 1, 2, \ldots\}$ of the natural numbers including 0.
- $\mathbb{N}_\mathbf{n}$: the set $\{0, 1, \ldots, n-1\}$ of the n natural numbers less than n (hence $\mathbb{N}_0 = \emptyset$).
- $\mathbb{Z}$: the set $\{0, 1, -1, 2, -2, \ldots\}$ of the integer numbers including 0 (hence $\mathbb{Z}_0 = \emptyset$).
- $\mathbb{Z}_\mathbf{n}$: the set $\{-n, \ldots, -1, 0, 1, \ldots, n-1\}$ of the $2n$ integer numbers greater than or equal $-n$ and less than n.
- $\mathbb{Q}$: the set of all rational numbers.
- $\mathbb{P}(\mathbf{T})$: the powers et of T (the set of its subsets), also considered as the set of relations on T: for every r in $\mathbb{P}(T)$ (i.e. $r \subseteq T$) and for every x in T, $r(x)$ (i.e. $x \in r$) is true or false.
- $\mathbb{P}^\infty(\mathbf{T})$: the set of all infinite subsets of T.

Tuples The datatype *tuple* (ordered sequence of unnamed values) is introduced as follows:

- $\mathbf{T_1 \times \ldots \times T_n}$: the set of tuples $\langle v_1, \ldots, v_n \rangle$ with $v_1 \in T_1, \ldots, v_n \in T_n$; if $t = \langle v_1, \ldots, v_n \rangle$, then $t.i = v_i$ $(1 \le i \le n)$.
- $\mathbf{T_t[i \mapsto T]}$: the tuple which is identical to tuple T_t except that $T_t.i = T$ (i.e. $T_t[i \mapsto T].i' = T_r.i'$, for all $i' \neq i$).

Records The datatype *record* (ordered sequence of named values) is introduced as follows:

- $\mathbf{t_1} : \mathbf{T_1} \times \ldots \times \mathbf{t_n} : \mathbf{T_n}$: the set of records $\langle t_1 : v_1, \ldots, t_n : v_n \rangle$ $(= \{\langle t_1, v_1\rangle, \ldots, \langle t_n, v_n\rangle\})$ where $t_1, \ldots, t_n$ are disjoint values (tags) and $v_1 \in T_1, \ldots, v_n \in T_n$; if $r = \langle t_1 : v_1, \ldots, t_n : v_n\rangle$, then $r.t_i = v_i$ $(1 \leq i \leq n)$.
- $\mathbf{T_r}[\mathbf{t} \mapsto \mathbf{T}]$: the record which is identical to record T_r except that $T_r.t = T$ (i.e. $T_r[t \mapsto T].t' = T_r.t'$, for all $t' \neq t$).

Maps The datatype *map* (the set-theoretic counterpart of a function) is introduced as follows:

- $\mathbf{T_1} \xrightarrow{\textbf{part.}} \mathbf{T_2}$: the set of partial maps from T_1 to T_2; for every f in $T_1 \xrightarrow{\text{part.}} T_2$ and for every x in domain$(f) \subseteq T_1$, $f(x)$ is in T_2 and $\langle x, f(x)\rangle$ is in f (i.e. $T_1 \xrightarrow{\text{part.}} T_2$ is a subset of $T_1 \times T_2$).
- $\mathbf{T_1} \rightarrow \mathbf{T_2}$: the set of total maps from T_1 to T_2 (a subset of $T_1 \xrightarrow{\text{part.}} T_2$); for every f in $T_1 \rightarrow T_2$ and for every x in T_1, $f(x)$ is in T_2 (i.e. domain$(f) = T_1$).
- $\mathbf{T_m}[\mathbf{T_x} \mapsto \mathbf{T_y}]$: the map which is identical to map T_m except that $T_m[T_x \mapsto T_y](T_x) = T_y$ (i.e. $T_m[T_x \mapsto T_y](x) = T_m(x)$ for every $x \neq T_x$).
- $[\mathbf{T_x} \mapsto \mathbf{T_y}]$: the map m such that domain$(m) = \{T_x\}$ and $m(T_x) = T_y$.

Sequences The domain *sequence* (finite sequence of arbitrary length) is introduced as follows:

- $\mathbf{A^k} := \mathbb{N}_\mathbf{k} \rightarrow \mathbf{A}$: the set of sequences of length $k \in \mathbb{N}$ whose values are in A; for every $s \in A^k, i \in \mathbb{N}_k, v \in A$, we have $s(i) \in A$ and $s[i \mapsto v] \in A^k$.
- $\mathbf{A}^* := \bigcup_{\mathbf{k} \in \mathbb{N}} \mathbf{A^k}$: the set of finite sequences whose values are in A with function $\textsc{length} : A^* \rightarrow \mathbb{N}$: for every $s \in A^k \subseteq A^*$, we have $\textsc{length}(s) = k$.

The domain of infinite sequences is correspondingly introduced:

- $\mathbf{A}^\infty := \mathbb{N} \rightarrow \mathbf{A}$: the set of infinite sequences whose values are in A; for every $s \in A^\infty, i \in \mathbb{N}, v \in A$, we have $s(i) \in A$ and $s[i \mapsto v] \in A^\infty$.

Abbreviations We use the following syntactic abbreviations of terms and formulas ("$P_1 \equiv P_2$" means "P_1 is an abbreviation of P_2"):

- $\textbf{SUCH } \mathbf{x} \in \mathbf{T} : \mathbf{F} \equiv \text{SUCH } x : x \in T \wedge F$
- $\forall \mathbf{x} \in \mathbf{T} : \mathbf{F} \equiv \forall x : x \in T \Rightarrow F$
- $\forall \mathbf{x_1} \in \mathbf{T_1}, \dots, \mathbf{x_n} \in \mathbf{T_n} : \mathbf{F} \equiv \forall x_1 \in T_1 : \dots : \forall x_n \in T_n : F$
- $\forall \mathbf{x_1}, \dots, \mathbf{x_n} \in \mathbf{T} : \mathbf{F} \equiv \forall x_1 \in T, \dots, x_n \in T : F$
- $\exists \mathbf{x} \in \mathbf{T} : \mathbf{F} \equiv \exists x : x \in T \wedge F$
- $\exists \mathbf{x_1} \in \mathbf{T_1}, \dots, \mathbf{x_n} \in \mathbf{T_n} : \mathbf{F} \equiv \exists x_1 \in T_1 : \dots : \exists x_n \in T_n : F$
- $\exists \mathbf{x_1}, \dots, \mathbf{x_n} \in \mathbf{T} : \mathbf{F} \equiv \exists x_1 \in T, \dots, x_n \in T : F$
- $\textbf{LET } \mathbf{x_1} = \mathbf{T_1}, \dots, \mathbf{x_n} = \mathbf{T_n} \textbf{ IN } \mathbf{P} \equiv \text{LET } x_1 = T_1 \text{ IN LET } \dots \text{ IN LET } x_n = T_n \text{ IN } P$
- $\mathbf{T}[\mathbf{T_1} \mapsto \mathbf{T'_1}, \dots, \mathbf{T_n} \mapsto \mathbf{T'_n}] \equiv T[T_1 \mapsto T'_1] \dots [T_n \mapsto T'_n]$
- $[\mathbf{T_1} \mapsto \mathbf{T'_1}, \dots, \mathbf{T_n} \mapsto \mathbf{T'_n}] \equiv [T_1 \mapsto T'_1] \dots [T_n \mapsto T'_n]$
- $\text{MIN } \mathbf{x} \in \mathbf{S} : \mathbf{F} \equiv \text{SUCH } x \in S : (F \wedge \neg \exists y \in S : y < x \wedge F[y/x])$
- $\text{MAX } \mathbf{x} \in \mathbf{S} : \mathbf{F} \equiv \text{SUCH } x \in S : (F \wedge \neg \exists y \in S : y > x \wedge F[y/x])$

Definitions of Relations and Maps We use the following formats to define relations and maps.

- $\mathbf{r} \subseteq \mathbf{T_1} \times \dots \times \mathbf{T_n},\ \mathbf{r}(\mathbf{x_1}, \dots, \mathbf{x_n}) \Leftrightarrow \mathbf{F}$

  This definition introduces a relation r on $T_1 \times \dots \times T_n$ such that, for all $x_1 \in T_1, \dots, x_n \in T_n$, the formula $r(x_1, \dots, x_n)$ is true if and only if F is true.

  The relation $r \in \mathbb{P}(T_1 \times \dots \times T_n)$ is the set

  $$r = \{\langle x_1, \dots, x_n \rangle \in T_1 \times \dots \times T_n : F\}$$

  The formula $r(x_1, \dots, x_n)$ is the syntactic abbreviation

  $$r(x_1, \dots, x_n) \equiv \langle x_1, \dots, x_n \rangle \in r$$

- $\mathbf{f : T_1 \times \ldots \times T_n \rightarrow T_0,\ f(x_1, \ldots, x_n) = T}$

  This definition introduces a total map f from $T_1 \times \ldots \times T_n$ to T_0 such that, for all $x_1 \in T_1, \ldots, x_n \in T_n$, the value of the term $f(x_1, \ldots, x_n)$ is in T_0 and equals the value of T.

  The map $f \in T_1 \times \ldots \times T_n \rightarrow T_0$ is the set

  $$f = \{\langle x_1, \ldots, x_n, y\rangle \in T_1 \times \ldots \times T_n \times T_0 :\ y = T\}$$

  The term $f(x_1, \ldots, x_n)$ is the syntactic abbreviation

  $$f(x_1, \ldots, x_n)\ \equiv\ \text{SUCH}\ y \in T_0 : \langle x_1, \ldots, x_n, y\rangle \in f$$

Since relations and maps are just special sets, they may serve as the values of variables in terms and formulas (in contrast to logical predicates and functions). On the other hand, the abbreviations $r(x_1, \ldots, x_n)$ and $f(x_1, \ldots, x_n)$ make a relation r and a map f usable like a predicate respectively function; we thus use the notions “predicate” and ”relation” respectively “function” and “map” interchangeably.