A Verifying Compiler for a Multi-threaded Object-Oriented Language

K. Rustan M. Leino and Wolfram Schulte
Microsoft Research,Redmond, WA, USA
e-mail: {leino,schulte}@microsoft.com

Abstract. A verifying compiler automatically verifies the correctness of a source program before compiling it. Founded on the definition of the source language and a set of rules (a methodology) for using the language, the program's correctness criteria and correctness argument are provided in the program text by interface specifications and invariants.

This paper describes the program-verifier component of a verifying compiler for a core multi-threaded object-oriented language. The verifier takes as input a program written in the source language and generates, via a translation into an intermediate verification language, a set of verification conditions. The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas can be analyzed automatically by a satisfiability-modulo-theory (SMT) solver.

The paper defines the source language and intermediate language, the translation from the former into the latter, and the generation of verification conditions from the latter. The paper also builds a methodology for writing and verifying single- and multi-threaded code with object invariants, and encodes the methodology into the intermediate-language program.

The paper is intended as a student's guide to understanding automatic program verification. It includes enough detailed information that students can build their own basic program verifier.

0. Introduction

A *verifying compiler* is a compiler that establishes that a program is correct before allowing it to be run. Verifying compilers can come in many flavors, from systems that generate provably correct code from specifications to systems that ask users to guide an interactive theorem prover to produce a replay-able proof script. In this paper, we consider a verifying compiler that automatically generates logical proof obligations, *verification conditions* (VCs), from a given program, its embedded specifications, and a set of rules (a *methodology*) that guides the use of the language. The validity of the VCs implies the correctness of the program. The VCs are passed to a satisfiability-modulo-theory (SMT) solver to be discharged automatically, if possible. Failed proof attempts are presented to users as error messages, to which a user responds by fixing errors or omissions in the program and its specifications.

The Spec$^\sharp$ programming system [6] is a modern research prototype of such a verifying compiler. It consists of an object-oriented programming language (also called Spec$^\sharp$) designed as a superset of the .NET programming language C$^\sharp$, enriching the type system (for example with non-null types) and adding specifications (like pre- and postconditions) as a part of the language, a methodology for using the language, a compiler that produces executable code for the .NET virtual machine, an integration into the Microsoft Visual Studio integrated development environment, and a static program verifier.

Generating verification conditions for high-level source programs is nontrivial and involves a large number of details and design decisions. Therefore, the Spec$^\sharp$ static program verifier (which is known as Boogie [4]) splits the task into two: it first translates the Spec$^\sharp$ program into an intermediate verification language (called BoogiePL [11]) and then generates VCs from it. This lets the tool designer make modeling decisions in terms of the intermediate language, which thus provides a level of abstraction above the actual formulas passed to the SMT solver.

In this paper, we want to convey the design of the program-verifier component of a verifying compiler. Doing so for Spec$^\sharp$ and BoogiePL is too large of a task for the paper, so we instead define a core object-oriented source language (which we shall call Spec$^\flat$) and an imperative intermediate verification language (which we shall call BoogiePL$^\flat$). As their names suggest, these languages are representative of Spec$^\sharp$ and BoogiePL, respectively. The Spec$^\flat$ language features classes and single-inheritance subclasses, object references, dynamic method dispatch, co-variant arrays, multi-threading, and mutual-exclusion locks.

Outline We start from the bottom up. In Section 1, we define BoogiePL$^\flat$ and its VC generation. We define Spec$^\flat$ in Section 2, where Section 2.3 defines a translation from Spec$^\flat$ into BoogiePL$^\flat$. We then take on some hard questions of how to write specifications in such a way that one can reason about programs modularly—to scale program verification, it must be possible to specify and verify each part (say, each class) of a program separately, in such a way that the separate verification of each part implies the correctness of the entire program. In Section 3, we introduce a methodology for *object invariants*, which specify the data consistency conditions of class instances, and define a translation of the new features and rules into BoogiePL$^\flat$. In Section 4, we also add features for writing multi-threaded code, a methodology for those features, and a corresponding translation into BoogiePL$^\flat$. Throughout, we give enough details to build a basic program verifier for Spec$^\flat$. Concepts and typical design issues carry over to other languages as well.

Foundational Work Program verification has a long history. The foundation for today's verification research was laid down by Floyd's inductive assertion method [25], Hoare's axiomatic basis for programming [30], and Dijkstra's characterization of semantics [18]. Early program verifiers include the systems of King [40,39], Deutsch [15], Good *et al.* [28], and German [27], the Stanford Pascal Verifier [56], and the Ford Pascal-F Verifier [63]. Two verifying compilers for procedural languages are SPARK [2] and B [0].

```
const $K$: int ;
function $f$(int) returns (int) ;
axiom $( \exists k\colon \mathbf{int} \bullet f(k) = K )$ ;
procedure $Find(a\colon \mathbf{int},\ b\colon \mathbf{int})$ returns $(k\colon \mathbf{int})$ ;
  requires $a \leqslant b \wedge (\forall j\colon \mathbf{int} \bullet a < j \wedge j < b \Rightarrow f(j) \neq K )$ ;
  ensures $f(k) = K$ ;
implementation $Find(a\colon \mathbf{int},\ b\colon \mathbf{int})$
{  assume $f(a) = K$ ;  $k := a$
[] assume $f(b) = K$ ;  $k := b$
[] assume $f(a) \neq K \wedge f(b) \neq K$ ;  call $k := Find(a-1, b+1)$
}
```

Figure 0. An example BoogiePL$^\flat$ program, showing the declaration of a constant K, a function f, an axiom that says f has a K element, a procedure $Find$ that finds the K element of f, and a recursive implementation of $Find$. The call statement **call** $x := Find(0, 0)$ will set x to some K element of f.

1. An Intermediate Imperative Verification Language

This section defines BoogiePL$^\flat$, an intermediate language for program verification. BoogiePL$^\flat$ is essentially BoogiePL [11], but without some of the more advanced features of BoogiePL. A BoogiePL$^\flat$ program consists of two parts:

- a mathematical part to define a logical basis for the terms used in the program, described by constants, functions, and axioms, and
- an imperative part to define the behavior of the program, described by procedure specifications, mutable variables, and code.

Figure 0 shows a simple BoogiePL$^\flat$ program. The mathematical part of this program are the declarations of K, f, and the axiom. The imperative part of the program is given by the specification and implementation of $Find$.

The semantics of a BoogiePL$^\flat$ program is defined as a logical formula, consisting of the theory induced by the mathematical part and the semantics induced by each procedure implementation in the imperative part. The program is considered correct if the logical formula is valid.

The next subsections introduce BoogiePL$^\flat$: its type system, the syntax of its mathematical and imperative parts, and the semantics of the code.

1.0. Basic Concepts

Backus Naur Form We use the common Backus Naur form to specify syntax. Nonterminals are written in italics. Terminals are keywords (written in bold), symbols (written as themselves), and a set Id of identifiers. For any nonterminal a, the suffixes $a^?$ denotes either the empty word or a, a^+ denotes one or more repetitions of a, and a^* denotes either the empty word or a^+. Depending on the context, repetitions are separated by commas (*e.g.*, in argument lists) or by white space (*e.g.*, in a sequence of declarations); this is not further specified.

Program Structure At the top level, a BoogiePL$^\flat$ program is a set of declarations.

$$\begin{array}{rcl} \mathit{Program} & ::= & \mathit{Decl}^* \\ \mathit{Decl} & ::= & \mathit{Constant} \mid \mathit{Function} \mid \mathit{Axiom} \\ & \mid & \mathit{Variable} \mid \mathit{Procedure} \mid \mathit{Implementation} \end{array}$$

Type System Value-holding entities in BoogiePL$^\flat$ are typed, despite the fact that a theorem prover used on BoogiePL$^\flat$ programs may be untyped. The purpose of semantic-less types in BoogiePL$^\flat$, like the purpose of explicit declarations of variables and functions, is to guard against certain easy-to-make mistakes in the input.

There are four built-in basic types, map types, and the supertype $\mathbf{any}$:

$$\mathit{Type} \; ::= \; \mathbf{bool} \mid \mathbf{int} \mid \mathbf{ref} \mid \mathbf{name} \mid \; [\; \mathit{Type}^+ \;] \; \mathit{Type} \mid \mathbf{any}$$

The type $\mathbf{bool}$ represents the boolean values $\mathbf{false}$ and $\mathbf{true}$. The type $\mathbf{int}$ represents the mathematical integers. The type $\mathbf{ref}$ represents object references. One of its values is the built-in literal $\mathbf{null}$. The only operations defined by the language on $\mathbf{ref}$ are equality and dis-equality tests. The type $\mathbf{name}$ represents various kinds of defined names (like types and field names). The only operations defined by the language on $\mathbf{name}$ are equality and dis-equality tests and the partial order $<:$. In a map type, the domain (that is, the types of the arguments) is given first, followed by the range type.

Type $\mathbf{any}$ represents the un-tagged union of the other types. Every type can implicitly be converted to and from the type $\mathbf{any}$. Because types in BoogiePL$^\flat$ are semantic-less, we use the identity for these conversions. But note that the implicit conversion from $\mathbf{any}$ to a type T is "unsafe" (since $\mathbf{any}$ is not a tagged union with checked tags). It is our responsibility to guarantee correct usage of expressions of type $\mathbf{any}$.

We say a type T is *assignable* to a type U if T is U or if either T or U is $\mathbf{any}$.

Scope Rules BoogiePL$^\flat$ supports nested lexicographic scoping, which means that (0) all identifiers introduced by top-level declarations must be distinct, and (1) identifiers introduced in inner scopes hide identifiers in outer scopes. During name resolution, an identifier is first looked up in the innermost scope, then the enclosing scope, and so on. It is an error if an identifier can't be found in the scope of its use.

1.1. Theories

The mathematical part of the language (constants, functions, axioms) is similar to other specification languages, including Larch [29] or the input language of the theorem prover Simplify [12].

Constants and *functions* are identifiers that, throughout the interpretation of a program, have a fixed, but possibly unknown, meaning.

$$\begin{array}{rcl} \mathit{Constant} & ::= & \mathbf{const}\; \mathit{Id} \;:\; \mathit{Type} \;; \\ \mathit{Function} & ::= & \mathbf{function}\; \mathit{Id} \;(\; \mathit{Type}^* \;)\; \mathbf{returns} \;(\; \mathit{Type} \;); \end{array}$$

Both can be used in expressions and commands.

To constrain the values of constants and functions, one uses *axioms*:

$$\mathit{Axiom} \; ::= \; \mathbf{axiom}\; \mathit{Expr} \;;$$

The given expression must be of type $\mathbf{bool}$ and must not have any free variables. An axiom that comes from free is that all constants of type $\mathbf{name}$ have distinct values.

1.2. Variables and Procedures

The *state space* of a BoogiePL$^\flat$ program is defined by variables. A *global variable* is a variable that is accessible to all procedures.

$$\mathit{Variable} \quad ::= \quad \textbf{var}\ \mathit{Id}\ :\ \mathit{Type}\ ;$$

A *procedure* is a name for a parameterized operation on the state space.

$$\begin{array}{rcl} \mathit{Procedure} & ::= & \textbf{procedure}\ \mathit{Id}\ \mathit{Signature}\ ;\ \mathit{Specification}^* \\ \mathit{Signature} & ::= & (\mathit{IdType}^*)\ \textbf{returns}\ (\mathit{IdType}^*) \\ \mathit{IdType} & ::= & \mathit{Id}\ :\ \mathit{Type} \\ \mathit{Specification} & ::= & \textbf{requires}\ \mathit{Expr}\ ; \\ & | & \textbf{modifies}\ \mathit{Id}^*\ ; \\ & | & \textbf{ensures}\ \mathit{Expr}\ ; \end{array}$$

The signature defines the list of in-parameters and then the list of out-parameters.

The procedure specification consists of a number of **requires**, **modifies**, and **ensures** clauses. The expressions given by the **requires** and **ensures** clauses must be of type **bool**. Every *Id* mentioned in a **modifies** clause must name a global variable. The in-parameters are in scope in the **requires** clause, and both in- and out-parameters are in scope in the **ensures** clause.

Each **requires** clause specifies a *precondition*, which must hold at each call to the procedure (we shall see calls later). An implementation of the procedure is allowed to assign to a global variable only if it is listed in a **modifies** clause of the procedure's specification. Each **ensures** clause specifies a *postcondition*, which must hold on exit from any implementation of the procedure. The expression in an **ensures** clause is a *two-state* predicate, which means that it can refer to both the initial and final states of the procedure (using **old** expressions for the initial state, as we shall see later). The **ensures** condition thus specifies a relation between the initial and final states of the procedure.

Procedures can be given implementations.

$$\mathit{Implementation} \quad ::= \quad \textbf{implementation}\ \mathit{Id}\ \mathit{Signature}\ \mathit{Block}$$

Here, *Id* must refer to a declared procedure and *Signature* must be identical to that of the declared procedure. There are no restrictions on the number of implementations that one procedure can have; each implementation is verified to obey the same specification.

Variables come in five flavors: global variables, in-parameters, out-parameters, local variables, and quantifier-bound variables. We say that a variable is *writable* in an implementation if it is a local variable, out-parameter, or a global variable mentioned in the modifies clause.

1.3. Motivation for Choice of Commands

BoogiePL$^\flat$ commands have been designed to be simple and primitive. The design makes heavy use of three useful, but perhaps less known, commands: **assert**, **assume**, and **havoc**. Before we define these and other commands in the next subsection, we give a couple of examples to develop an intuitive understanding of these commands.

Let us look at how we translate Spec$^\flat$'s conditional and while loop into BoogiePL$^\flat$. Spec$^\flat$ has the usual conditional statement, written as **if** (E) S **else** T. It goes wrong if E is not defined; otherwise, if E evaluates to **true**, then the conditional statement executes S, else the conditional statement executes T. The translation of the conditional statement into BoogiePL$^\flat$ is defined as follows:

```
$Tr[\![\textbf{if}\ (E)\ S\ \textbf{else}\ T]\!] =$
    $\textbf{assert}\ Df[\![E]\!]\ ;$
    $\{\ \textbf{assume}\ Tr[\![E]\!]\ ;\ Tr[\![S]\!]$
    $[]\ \textbf{assume}\ \neg Tr[\![E]\!]\ ;\ Tr[\![T]\!]$
    $\}$
```

The translation uses three functions (cf. Section 2 for their full definitions). The function $Tr[\![s]\!] = c$ translates a Spec$^\flat$ statement s into the BoogiePL$^\flat$ command c. The functions $Df[\![e]\!] = e'$ and $Tr[\![e]\!] = e''$ return two BoogiePL$^\flat$ expressions for one Spec$^\flat$ expression: e' says whether e is defined and, if so, e'' denotes its translated value.

The translation uses an **assert** command to check that E is defined. If E is not defined, that is, if $Df[\![E]\!]$ evaluates to **false**, then the **assert** command will cause the program to halt with an error.

The rest of the translation consists of a nondeterministic choice, as denoted by $[]$. Each choice begins with an **assume** command, which indicates under which condition the remainder of that path of the program is analyzed. That is, the translation of S is analyzed only if $Tr[\![E]\!]$ evaluates to **true**, and analogously for T.

Spec$^\flat$'s while loop **while** (E) **invariant** J; $\{S\}$ proceeds as follows. The while loop goes wrong if the loop invariant J is not defined or evaluates to **false**, and it goes wrong if the loop condition E is not defined. Otherwise, if E evaluates to **true**, the body $\{S\}$ is executed, after which the entire while loop is executed again. If E evaluates to **false**, the while loop terminates. We translate the while loop into BoogiePL$^\flat$ as follows:

```
$Tr[\![\textbf{while}\ (E)\ \textbf{invariant}\ J;\ \{S\}]\!] =$
    $\textbf{assert}\ Df[\![J]\!]\ ;\ \textbf{assert}\ Tr[\![J]\!]\ ;$
    $\textbf{havoc}\ Md[\![S]\!]\ ;$
    $\textbf{assume}\ Df[\![J]\!] \wedge Tr[\![J]\!]\ ;$
    $\textbf{assert}\ Df[\![E]\!]\ ;$
    $\{\ \textbf{assume}\ Tr[\![E]\!]$
        $Tr[\![\{S\}]\!]\ ;$
        $\textbf{assert}\ Df[\![J]\!]\ ;\ \textbf{assert}\ Tr[\![J]\!]\ ;$
        $\textbf{assume false}$
    $[]\ \textbf{assume}\ \neg Tr[\![E]\!]$
    $\}$
```

The translation uses a function $Md[\![S]\!]$, which returns the list of variables possibly modified by S, also known as the *syntactic targets* of the loop.

The translation can be understood as follows. First, the loop invariant is checked on entry to the loop. Then, we want to look at just one iteration of the loop, but we want it to be an arbitrary iteration. The translation thus "fast forwards" to an arbitrary iteration by setting the variables to arbitrary values. More precisely, the translation sets the syntactic targets of the loop to arbitrary values (**havoc** $Md[\![S]\!]$) satisfying the loop invariant (**assume** $Df[\![J]\!] \wedge Tr[\![J]\!]$). In that arbitrary iteration, the translation checks that the loop condition is defined, and then either performs one more iteration or terminates the loop. After executing S, the translation checks that the invariant J still holds—this is essentially the inductive step of the loop verification.

One thing remains to be explained: The **assume false** command at the end of the first choice branch indicates that the remainder of the program is not analyzed immediately after an arbitrary loop iteration. The analysis proceeds under the assumption of successful termination, which happens through the second choice branch.

1.4. Commands

Now we are ready to introduce the BoogiePL$^\flat$ commands, which follow this grammar:

$$
\begin{array}{rcl}
\mathit{Command} & ::= & \textbf{assert}\ \mathit{Expr} \\
 & \mid & \textbf{assume}\ \mathit{Expr} \\
 & \mid & \textbf{havoc}\ \mathit{Id}^{+} \\
 & \mid & \mathit{Designator}\ :=\ \mathit{Expr} \\
 & \mid & \textbf{call}\ \mathit{Id}^{*}\ :=\ \mathit{Id}\ (\ \mathit{Expr}^{*}\ ) \\
 & \mid & \mathit{Command}\ ;\ \mathit{Command} \\
 & \mid & \mathit{Command}\ []\ \mathit{Command} \\
 & \mid & \mathit{Block} \\
\mathit{Block} & ::= & \{\ \mathit{Variable}^{*}\ \mathit{Command}\} \\
\mathit{Designator} & ::= & \mathit{Id} \\
 & \mid & \mathit{Designator}\ [\ \mathit{Expr}^{+}\ ]
\end{array}
$$

The command **assert** E evaluates E, which must be of type **bool**. If E evaluates to **true**, then the command terminates. If E evaluates to **false**, then the command *goes wrong*, which indicates a non-recoverable error.

The command **assume** E evaluates E, which must be of type **bool**. If E evaluates to **true**, then the command terminates. If E evaluates to **false**, then the execution of the program stalls forever, which entails that this program path no longer has any chance of going wrong.

In the command **havoc** xx, each identifier in the list xx must refer to a writable variable. The command assigns an arbitrary value to every variable in xx.

The assignment command uses a *designator*. In general, a designator expression has one of two forms. If it is an identifier x, then x must refer to a variable or constant. The type of such an expression is the type of x. A designator expression of the form $A[EE]$ requires the type of A to be a map type. The number of expressions in the list EE must equal the number of argument types of A, and the types of the expressions in EE must be assignable to the types of the corresponding argument types of A. The type of the expression $A[EE]$ is the range type of A.

The designator expression used as the left-hand side d of an assignment command $d := E$ must be either a writable variable or an expression $a[EE]$ where a is a writable map variable. The type of E must be assignable to the type of d. If the left-hand side is a variable, the assignment command changes the value of that variable to E. If the left-hand side is an expression $a[EE]$, the assignment command overwrites a with a new map that is like the old except that it maps EE to E.

In the call command **call** $w := P(EE)$, P must refer to a procedure and w must refer to distinct writable variables that are not mentioned in P's **modifies** clauses. The length of the list w must equal the number of out-parameters of P, and the types of the out-parameters of P must be assignable to the types of the corresponding variables in w. The length of the list EE of expressions must equal the number of in-parameters of P, and the types of the expressions in EE must be the assignable to the types of the corresponding in-parameters of P.

The call command evaluates the expressions in EE and binds the resulting values to the in-parameters of P. It also binds w to the out-parameters of P. The call command goes wrong if any of P's declared preconditions is not satisfied. Otherwise, the call command sets w and the variables in P's **modifies** clauses to arbitrary values satisfying P's postconditions. Note that the meaning of the call command is given by the procedure's specification alone; the procedure's implementations are separately checked against the specification, thus enabling modular reasoning.

The sequential composition of two commands S and T is written $S\ ;\ T$, and its behaviors are defined by the behaviors of S followed by the behaviors of T. The choice composition of two commands is written $S\ []\ T$, and its behaviors are defined as the union of the behaviors of S and T. That is, $S\ []\ T$ can behave as either S or T. We let ; bind stronger than [].

The block command $\{VV\ \ S\}$ introduces local variables VV for use in S. The behavior of the block command is the behavior of S started with arbitrary values for VV.

1.5. Expressions

Expressions are fairly standard and follow this grammar, where $\oplus$ denotes any binary operator shown in Fig. 1:

$$\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Expr}\ \oplus\ \mathit{Expr} \\
& | & \neg\ \mathit{Expr} \\
& | & \mathit{Atom} \\
\mathit{Atom} & ::= & \mathit{Literal} \\
& | & \mathit{Designator} \\
& | & \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
& | & \textbf{old}\ (\ \mathit{Expr}\ ) \\
& | & \mathit{Quantification} \\
\mathit{Literal} & ::= & \textbf{false} \mid \textbf{true} \mid \textbf{null} \mid 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Quantification} & ::= & (\ \mathit{Quantor}\ \mathit{IdType}^+\ \bullet\ \mathit{Expr}\ ) \\
\mathit{Quantor} & ::= & \forall \mid \exists
\end{array}$$

Unary and binary operators are given in Fig. 1. Each line shows the supported type signatures of the operators and common names for the operations. The figure also describes BoogiePL$^\flat$'s precedence rules. Each box holds operators with the same precedence. Operators in higher boxes have higher precedence than operators in lower boxes. For example, $a + b * c$ means $(a + (b * c))$, as usual. Implication is right associative. The other logical operators are associative, but associate only with themselves. All other operators are left associative. Although we do not show them in the grammar, we also allow expressions to contain parentheses, which can be used to override operator precedence.

The literals **false** and **true** have type **bool**, the literal **null** has type **ref**, and the integer literals have type **int**.

$\neg$: $\mathbf{bool} \to \mathbf{bool}$	logical negation
$*$: $\mathbf{int} \times \mathbf{int} \to \mathbf{int}$	multiplication
$/$: $\mathbf{int} \times \mathbf{int} \to \mathbf{int}$	integer division
$\%$: $\mathbf{int} \times \mathbf{int} \to \mathbf{int}$	integer modulo
$+$: $\mathbf{int} \times \mathbf{int} \to \mathbf{int}$	addition
$-$: $\mathbf{int} \times \mathbf{int} \to \mathbf{int}$	subtraction
$<$: $\mathbf{int} \times \mathbf{int} \to \mathbf{bool}$	arithmetic less-than
$\leqslant$: $\mathbf{int} \times \mathbf{int} \to \mathbf{bool}$	arithmetic at-most
$\geqslant$: $\mathbf{int} \times \mathbf{int} \to \mathbf{bool}$	arithmetic at-least
$>$: $\mathbf{int} \times \mathbf{int} \to \mathbf{bool}$	arithmetic greater-than
$<:$: $\mathbf{name} \times \mathbf{name} \to \mathbf{bool}$	partial order on names
$=$: $T \times T \to \mathbf{bool}$	equality
$\neq$: $T \times T \to \mathbf{bool}$	disequality
$\wedge$: $\mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$	logical conjunction
$\vee$: $\mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$	logical disjunction
$\Rightarrow$: $\mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$	logical implication
$\Leftrightarrow$: $\mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}$	logical equivalence

Figure 1. BoogiePL$^\flat$ operators, their types, and syntactic precedence.

In the function-application expression $f(EE)$, f must refer to a function. The number of expressions in the list EE must equal the number of arguments of f, and the types of the expressions in EE must be assignable to the types of the corresponding arguments of f. The function-application expression has the same type as the result type of f.

The expression $\mathbf{old}(E)$ is allowed to appear only in $\mathbf{ensures}$ clauses and procedure implementations. If it appears in code, E must only refer to variables that are in scope in the procedure's preconditions; more precisely, global variables, in-parameters, and quantifier-bound variables can be mentioned, but out-parameters and the implementation's local variables cannot. The expression denotes the value of E on entry to the procedure.

The quantifier expression $(\ \mathcal{Q}\, w \bullet\ E\ )$, where $\mathcal{Q}$ is either $\forall$ or $\exists$, defines the identifiers in w as bound variables that can be used in E. The type of a quantifier expression is $\mathbf{bool}$. The expression denotes the corresponding logical quantifier.

1.6. Weakest Preconditions

The semantics of the commands in our simple language is defined by weakest preconditions [18,65].

The *weakest precondition* of a command S and predicate Q on the post-state of S, denoted by $wp[\![S, Q]\!]$, is a predicate on the pre-state of S that characterizes the set of all states such that execution of S begun in any of those states does not go wrong, and if it terminates successfully, terminates in Q.

We define the following:

$$\begin{array}{lcl}
wp[\![\mathbf{assert}\; E,\; Q]\!] & = & E \wedge Q \\
wp[\![\mathbf{assume}\; E,\; Q]\!] & = & E \Rightarrow Q \\
wp[\![\mathbf{havoc}\; xx,\; Q]\!] & = & (\forall\, xx \bullet\; Q\,) \\
wp[\![x := E,\; Q]\!] & = & Q[E/x] \\
wp[\![S\; ;\; T,\; Q]\!] & = & wp[\![S,\; wp[\![T, Q]\!]]\!]) \\
wp[\![S\; [\,]\; T,\; Q]\!] & = & wp[\![S, Q]\!] \wedge wp[\![T, Q]\!] \\
wp[\![\{\, VV\; S\},\; Q]\!] & = & (\forall\, vv \bullet\; wp[\![S, Q]\!]\,)
\end{array}$$

where vv denotes the list of variables declared in VV, and where we understand a quantification with an empty list of bound variables to be just the body of the quantification. The semantics of map assignment is defined in Section 1.8 and the semantics of procedure calls is defined below.

The translation function $Q[E/x]$ denotes the capture-avoiding substitution of E for x in Q that keeps all **old** subexpressions intact. For example, if Q is

$$x < \mathbf{old}(x) \wedge (\forall\, x \bullet\; 0 \leqslant g(x)\,) \wedge (\forall\, y \bullet\; f(x, y) \Rightarrow g(x)\,)$$

then $Q[y + 1/x]$ is

$$y + 1 < \mathbf{old}(x) \wedge (\forall\, x \bullet\; 0 \leqslant g(x)\,) \wedge (\forall\, z \bullet\; f(y + 1, z) \Rightarrow g(y + 1)\,)$$

where, to avoid variable capture, the substitution operation renamed the bound variable y to a fresh variable z.

The definition of assert says that the legal pre-states of **assert** E are those in which both E and Q hold. Here and in Spec$^\flat$, programmers use **assert** E to claim that the condition E holds, and a program verifier must verify that claim.

The definition of assume says that the legal pre-states of **assume** E are those in which either E does not hold or Q already holds. Here and in Spec$^\flat$, programmers use **assume** E to express the fact that they care only about those executions where E holds, and a program verifier is then allowed to use E as an assumption.

The definition of **havoc** xx says that Q has to hold for all possible values of xx.

The assignment says that in order for Q to hold after the assignment, Q with x replaced by E must hold before it.

Sequential composition and choice correspond to functional composition and conjunction, respectively.

The meaning of the block command $\{\mathbf{var}\; x\colon T;\; C\}$ is defined in terms of the meaning of its embedded command C by universally quantifying over all possible initial values of x. Note that the block command is equivalent to **havoc** $x\;;\;C$, but the block command also introduces the scope of x.

Here are some example derivations for wp, where we have simplified some of the right-hand sides:

$$\begin{array}{lcl}
wp[\![\mathbf{assert}\; 1 < x,\; Q]\!] & = & 1 < x \wedge Q \\
wp[\![\mathbf{assert}\; \mathbf{true},\; Q]\!] & = & Q \\
wp[\![i := i + 1,\; i \leqslant 1]\!] & = & i \leqslant 0 \\
wp[\![\mathbf{assume}\; y = x + 1,\; y = 5]\!] & = & y = x + 1 \Rightarrow y = 5 \\
wp[\![\mathbf{assume}\; \mathbf{false},\; Q]\!] & = & \mathit{true} \\
wp[\![\mathbf{assert}\; P\; ;\; \mathbf{assume}\; P,\; Q]\!] & = & P \wedge (P \Rightarrow Q)
\end{array}$$

Shorthand Notations If a call command has no out-parameters, we write it simply as **call** $P(EE)$.

In the definition of procedure calls below, it will be convenient to use simultaneous assignments, which we write as $xx := EE$, where xx is a list of distinct writable variables, EE is a list of expressions of the same length, and each expression in EE is assignable to the type of the corresponding variable in xx. The values of the variables after executing an assignment command $xx := EE$ equal the values of the corresponding expressions before executing it. Simultaneous assignments can be defined in terms of block commands and assignments: a block command introduces temporary variables for the variables in xx, then assigns, in sequence, each of the expressions in EE to the temporary variables, and finally assigns each of the temporary variables to xx. The wp of simultaneous assignment then becomes:

$$wp[\![xx := EE,\ Q]\!] \ = \ Q[EE/xx]$$

where $[EE/xx]$ denotes simultaneous substitution.

Procedures To define the semantics of procedure calls and implementations, we refer to the names in the following schema:

```
procedure $P(AA)$ returns $(RR)$;
  requires $Pre$;  modifies $gg$;  ensures $Post$;
```

The semantics of a procedure call **call** $xx := P(EE)$ is defined to be the semantics of the following command:

```
{ var $AA$; var $RR$; var $HH$;
  $aa := EE$;
  assert $Pre$;
  $hh := gg$;
  havoc $gg$;                                              (0)
  assume $StripOld[\![ReplaceOld[\![Post, gg, hh]\!]]\!]$;
  $xx := rr$
}
```

where HH is a list of fresh variable declarations corresponding to the global variables gg, and where we use aa, rr, and hh to denote the lists of identifiers introduced by AA, RR, and HH, respectively. Here and in the sequel, we are sloppy with the exact syntax of lists, as in showing just one **var** keyword in front of the list AA.

The definition of procedure call introduces local variables for the formal parameters AA and RR, and introduces a fresh variable in HH for every global variable mentioned in gg. The definition then evaluates the actual in-parameters and assigns these to the formals. The definition then requires that the caller has established the precondition of P. The **havoc** command destroys all knowledge about modified global variables; the assignment $hh := gg$ captures the previous values of gg. The caller can then assume that the postcondition has been established, where in the postcondition we first handle **old** expressions: for each variable g in gg, the translation function $ReplaceOld[\![Post, gg, hh]\!]$ replaces every occurrence of g nested inside an **old** expression within $Post$ by g's corresponding variable in hh; translation function $StripOld[\![Q]\!]$ replaces every subexpres-

$$\begin{array}{l}
\textbf{function}\ f(\textbf{int})\ \textbf{returns}\ (\textbf{int})\ ;\\
\textbf{axiom}\ (\forall\, k{:}\,\textbf{int} \bullet\ 0 \leqslant k\ \Rightarrow\ 2 * k \leqslant f(k)\ )\ ;\\
\textbf{var}\ x{:}\,\textbf{int}\ ;\\
\textbf{procedure}\ \mathit{Inc}(n{:}\,\textbf{int})\ \textbf{returns}\ (r{:}\,\textbf{int})\ ;\\
\quad \textbf{requires}\ 0 \leqslant n\ ;\\
\quad \textbf{modifies}\ x\ ;\\
\quad \textbf{ensures}\ \textbf{old}(x) \leqslant x \land r = \textbf{old}(x)\ ;\\
\textbf{implementation}\ \mathit{Inc}(n{:}\,\textbf{int})\ \textbf{returns}\ (r{:}\,\textbf{int})\\
\{\ r := x\ ;\ x := x + f(n)\ \}
\end{array}$$

Figure 2. An example BoogiePL$^\flat$ program, showing the declaration of a function f, an axiom that constrains f, a global variable x, a procedure Inc that is specified to operate on x, and an implementation of Inc.

sion $\textbf{old}(E)$ in Q by E (we omit the formal definitions of $\mathit{ReplaceOld}$ and $\mathit{StripOld}$). Finally, the definition of the call assigns the formal out-parameters to the actuals.

This definition is correct only if AA and RR do not capture variables used in EE. If they do, we have to introduce fresh variables in AA and RR and consistently rename the uses of the formal in- and out-parameters in Pre and $Post$ before unfolding the definition above.

An implementation

$$\begin{array}{l}\textbf{implementation}\ P(AA)\ \textbf{returns}\ (RR)\\ \quad \mathit{Body}\end{array}$$

of procedure P is *valid* if it obeys the procedure's specification, under the proviso of the mathematical theory (that is, the conjunction of the axioms), here called MT:

$$\begin{array}{l}
\mathit{valid}(P, \mathit{Body}) =\\
\quad \mathit{StripTypes}[\![MT\ \Rightarrow\ \mathit{StripOld}[\![\ \mathit{wp}[\![\{\ \begin{array}[t]{l}\textbf{var}\ AA;\ \textbf{var}\ RR;\\ \textbf{assume}\ \mathit{Pre};\\ \mathit{Body};\\ \textbf{assert}\ \mathit{Post};\end{array}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \},\ \ \mathit{true}\ ]\!]]\!]]\!]
\end{array} \tag{1}$$

The application of wp produces a predicate on the pre-state of the procedure. In that state, $\textbf{old}(E)$ means just E, so we apply the translation function $\mathit{StripOld}$. The translation function $\mathit{StripTypes}$ erases the types of all quantifier-bound variables (we omit the formal definition).

Note that, compared to the call, the roles of the assert and assume commands are reversed here. Also, note that the modifies clause gg need not be verified, since it is already syntactically checked (Body is allowed to assign only to writable variables).

We say that a BoogiePL$^\flat$ program is *correct* if all its procedure implementations are valid. Note that this verification technique is *modular*, since it verifies each implementation separately.

1.7. Example

Consider the BoogiePL$^\flat$ program in Fig. 2. The mathematical theory, MT in (1), of this program is its one axiom. For the implementation of Inc, the command to which wp is applied in (1) is:

$$\begin{array}{l} \{\ \textbf{var}\ n\colon \textbf{int}\ ;\ \textbf{var}\ r\colon \textbf{int}\ ; \\ \quad \textbf{assume}\ 0 \leqslant n\ ; \\ \quad \{\ r := x\ ;\ x := x + f(n)\ \}\ ; \\ \quad \textbf{assert}\ \textbf{old}(x) \leqslant x \wedge r = \textbf{old}(x) \\ \} \end{array}$$

The wp of this command with respect to $true$ is:

$$(\forall n, r \bullet\ 0 \leqslant n \Rightarrow \textbf{old}(x) \leqslant x + f(n) \wedge x = \textbf{old}(x) \wedge true\ )$$

Applying $StripOld$ to this formula yields:

$$(\forall n, r \bullet\ 0 \leqslant n \Rightarrow x \leqslant x + f(n) \wedge x = x \wedge true\ )$$

So, the verification condition generated for the program in Fig. 2 is:

$$\begin{array}{l} (\forall k \bullet\ 0 \leqslant k \Rightarrow 2 * k \leqslant f(k)\ ) \Rightarrow \\ \quad (\forall n, r \bullet\ 0 \leqslant n \Rightarrow x \leqslant x + f(n) \wedge x = x \wedge true\ ) \end{array}$$

This is a valid formula, which an SMT solver like Simplify [12] easily verifies, so the program is correct.

By definition (0), the semantics of a call command:

$$\textbf{call}\ z := Inc(17)$$

is given by the following command:

$$\begin{array}{l} \{\ \textbf{var}\ n\colon \textbf{int}\ ;\ \textbf{var}\ r\colon \textbf{int}\ ;\ \textbf{var}\ old\_x\colon \textbf{int}\ ; \\ \quad n := 17\ ; \\ \quad \textbf{assert}\ 0 \leqslant n\ ; \\ \quad old\_x := x\ ; \\ \quad \textbf{havoc}\ x\ ; \\ \quad \textbf{assume}\ old\_x \leqslant x \wedge r = old\_x\ ; \\ \quad z := r \\ \} \end{array}$$

1.8. Targeting a Theorem Prover

The correctness of an implementation $Body$ of a procedure P is verified by passing $valid(P, Body)$ to an theorem prover, like a Satisfiability Modulo Theories (SMT) solver. Modern SMT solvers, like Simplify [12], are particularly well suited for automatic verification. First, they require no user interaction and can thus be used as a push-button technology. Second, they are refutation based, that is, they may produce counterexamples in case a property can't be satisfied, and those counterexamples can be used for error reporting. Third, their heuristics are tuned for software verification.

These SMT solvers are typically built around Nelson-Oppen cooperating decision procedures [64,66]. They all provide decision procedures for congruence closure (uninterpreted function symbols and equality), linear arithmetic, and quantifiers. Some also provide partial orders, maps, and other theories. In case a targeted theorem prover does not support a BoogiePL$^\flat$ defined operator, we have to add proper axioms. Simplify, for instance, does not have a built-in decision procedure for maps. Consequently, the verification-condition generator for Simplify has to axiomatize operators for map select and map update.

For instance, the verification-condition generator for Simplify replaces every map select expression $A[E]$ by the term $select(A, E)$, and replaces every map update $A[E] := F$ by $A := store(A, E, F)$. Simplify is untyped, so it suffices to add one axiom for $select$ and $store$ to the theorem prover's background axioms:

$$\begin{array}{ll}(\forall\, m, i, j, v \bullet & (i = j \;\Rightarrow\; select(store(m, i, v), j) = v) \;\wedge \\ & (i \neq j \;\Rightarrow\; select(store(m, i, v), j) = select(m, j))\ )\end{array}$$

Of course, arities of function symbols have to be respected in Simplify, that is, we need different function symbols and axioms to support maps of different arities.

2. An Object-oriented Programming Language

This section defines Spec$^\flat$, an object-oriented programming language. Spec$^\flat$ is a core of Spec$^\sharp$ [6]. Like the modern object-oriented languages Java, C$^\sharp$, Eiffel, and Modula-3, Spec$^\flat$ has object references, classes, subclasses with single inheritance, methods with dynamic dispatch, and co-variant arrays. Spec$^\flat$ excludes features like interfaces, multiple inheritance, structs, delegates, generics, static members, once functions, abstract methods, properties, events, iterators, overloading, boxing, and visibility modifiers.

Figure 3 shows an example Spec$^\flat$ program.

We give the semantics of Spec$^\flat$ in terms of a translation into the procedural language BoogiePL$^\flat$.

2.0. Programs, Classes, and Members

At the top level, a Spec$^\flat$ program is a set of classes.

$$Prog \;::=\; Class^*$$

A *class* defines an object type and provides its implementation. A class has a name (an identifier), a superclass, and a set of member declarations.

$$\begin{array}{rcl} Class & ::= & \mathbf{class}\; Id \;:\; ObjectType\;\{\; Member^* \;\} \\ ObjectType & ::= & \mathbf{object} \mid Id \end{array}$$

Each class derives from a single existing class, its *immediate superclass*. The declared classes form a single-inheritance subtype hierarchy rooted at the built-in object type $\mathbf{object}$. As a shorthand, we allow ": $ObjectType$" to be omitted; $ObjectType$ then defaults to $\mathbf{object}$. The values of object types are called *objects* and consist of the special value **null** and of *references* to a suite of class members (fields, invariants, and methods). Every reference has a built-in readonly field called $Type$, which returns the run-time type of the reference, represented as an object.

In addition to object types, we consider the Spec$^\flat$ types booleans, and integers, and one-dimensional arrays.

```
class Cell {
  int x;

  Cell(int i)
    ensures x == i;
  { x = i; }

  virtual int Get()
    ensures result == x;
  { return x; }

  virtual void Set(int i)
    modifies this.*;
    ensures x == i;
  { x = i; }

  void IncBy(int i)
    modifies this.*;
    ensures x == old(x) + i;
  { int t = Get(); Set(t + i); }
}
```

```
class BackupCell : Cell {
  int b;

  BackupCell(int i)
    ensures b == i && x == i;
  { base(i); b = i; }

  override void Set(int i)
    ensures b == old(x);
  { b = x; base.Set(i); }

  virtual int GetBackup()
    ensures result == b;
  { return b; }

  void Rollback()
    modifies x;
    ensures x == b;
  { x = b; }
}
```

Figure 3. Two example classes written in Spec$^\flat$. Class *Cell* represents a single storage location. The subclass *BackupCell* additionally maintains a recent history of the contents of that storage location.

$$Type \quad ::= \quad \mathbf{bool} \mid \mathbf{int} \mid ObjectType \mid Type\,[\,]$$

Arrays are *references* to sequences of values. Each array type is a subtype of **object**. We refer to object types and array types as *reference types*. The types respect *polymorphic subtyping*, that is, if T is a subtype of S, then an expression of type T can be assigned to a designator of type S (but not vice versa, unless T and S are the same type). Our array types are *co-variant* in their element type. For example, the type $Point[\,]$ is a subtype of **object**$[\,]$, provided $Point$ is a subtype of **object**. Arrays support the usual indexing lookup and update operations; they also have a built-in readonly field called $Length$.

Class members can be fields and methods.

$$Member \quad ::= \quad Field \mid Method$$

A *field* is an instance variable, that is, each instance of the class has its own copy of the variable.

$$Field \quad ::= \quad FieldModifier^{?}\ Type\ Id\ ;$$

Modifiers for fields will be introduced later.

Spec$^\flat$ supports usual scoping rules. For simplicity, we assume here that the fields declared in a class are distinct from other fields declared in the class and its superclasses.

A *method* is a name for a parameterized operation on the state space. A method can be invoked, passing a fixed number of values as parameters. Every method declaration belongs to some class. Syntactically, Spec$^\flat$ distinguishes three kinds of methods: *con-*

structors, *non-virtual methods*, and *virtual methods*. When a method includes a **virtual** or **override** modifier, that method is said to be a virtual method; otherwise, the method is said to be a non-virtual method. A constructor declaration looks like a non-virtual method declaration, but it is named with the name of the class in which it is defined and it has no result type.

A non-virtual method must be distinct from all methods declared in the class and its superclasses. The implementation of a non-virtual method is the same whether the method is invoked on an instance of the class in which it is declared or on an instance of a derived class.

A virtual method declared with **virtual** must be distinct from all other methods declared in the class and its superclasses. In contrast to non-virtual methods, the implementation of a virtual method can be overridden in derived classes. The declaration of the method override, indicated by the **override** keyword, must have the same signature as the overridden method; it provides a new implementation of the overridden method.

Every class has a constructor. It is only used in the creation of an object of the class. Constructors are implicitly called by class instance creation expressions (**new**) and by base calls inside constructors. For simplicity, we restrict classes to have just one constructor. If a class declares no constructor, then a default constructor with no parameters is provided, which simply calls the superclass constructor with no parameters (which is an error if the superclass does have a parameterless constructor). If a constructor is given, the first statement of its body must be a call to the superclass constructor.

$$
\begin{array}{rcl}
\mathit{Method} & ::= & \mathit{NonConstrPrefix}^{?} \\
 & & \mathit{Id}\ (\mathit{TypeId}^{*})\ \mathit{Specification}^{*}\ \mathit{Block} \\
\mathit{NonConstrPrefix} & ::= & \mathit{MethodModifier}^{?}\ \mathit{ReturnType} \\
\mathit{MethodModifier} & ::= & \mathbf{virtual} \mid \mathbf{override} \\
\mathit{ReturnType} & ::= & \mathit{Type} \mid \mathbf{void} \\
\mathit{TypeId} & ::= & \mathit{Type}\ \mathit{Id} \\
\mathit{Specification} & ::= & \mathbf{requires}\ \mathit{Expr}\ ; \\
 & \mid & \mathbf{modifies}\ \mathit{ModDesignator}^{+}\ ; \\
 & \mid & \mathbf{ensures}\ \mathit{Expr}\ ; \\
\mathit{ModDesignator} & ::= & \mathit{ODesignator}\ \mathit{ModSuffix} \\
\mathit{ODesignator} & ::= & \mathbf{this} \mid \mathit{Designator} \\
\mathit{ModSuffix} & ::= & .\ \mathit{Id} \mid\ .\,* \ \mid [\,*\,]
\end{array}
$$

In addition to the explicitly declared parameters, each method takes an implicit *receiver* parameter referred to by the keyword **this**. If a method has no return value, its return type is specified as **void**.

The procedure specification consists of a number of **requires**, **modifies**, and **ensures** clauses, which like in BoogiePL$^\flat$ introduce preconditions, modifies clauses, and postconditions. The expressions in the pre- and postconditions must be of type **bool**. The parameters, including **this**, are in scope in the specification, except that **this** is not available in the precondition of constructors. Postconditions can also mention **old** and **fresh** expressions (explained below) and, for non-void, non-constructor methods, the keyword **result**, which denotes the return value. The pre- and postconditions are not allowed to contain allocation and call expressions (explained below). The modifies clause must not list the designator expression $E.\mathit{Type}$ or $E.\mathit{Length}$, for any expression E, at the top level.

Each method has one implementation, consisting of a block statement. Unlike in BoogiePL$^\flat$, we cannot enforce modifies clauses in Spec$^\flat$ by syntactic restrictions. Instead, method implementations in Spec$^\flat$ need to be verified to satisfy their modifies clauses. The declared modifies clauses indicate a set of heap locations, which one gets by evaluating every modifies designator expression on entry to the method. A modifies designator of the form $E.f$ gives the license to modify the f field of object E, $E.*$ gives the license to modify any field of object E, and $E[*]$ gives the license to modify array E at any index. A method implementation also gets a blanket license to modify some other things, as we explain later.

Spec$^\flat$ supports *behavioral subtyping* [42,55,16], that is, whenever an object of static type S is expected, any object of a subtype T of S can be used without invalidating the program's verification. A necessary condition for behavioral subtyping is the following: Consider a virtual method m defined in S (written $S^\circ m$) and an override of m defined in a subclass T (written $T^\circ m$); then, $T^\circ m$ can only weaken $S^\circ m$'s precondition and only strengthen $S^\circ m$'s postcondition. In this paper, we consider only the strengthening of postconditions, which override $T^\circ m$ can specify by providing additional **ensures** clauses. These are then conjoined with the postconditions of the overridden method, as the translation into BoogiePL$^\flat$ will make explicit.

2.1. Statements

Statements in Spec$^\flat$ follow this grammar:

$$\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathit{Block} \\
 & | & \mathit{Type}\ \mathit{Id}\ ; \\
 & | & \mathit{Type}\ \mathit{Id}\ =\ \mathit{Expr}\ ; \\
 & | & \mathbf{assert}\ \mathit{Expr}\ ; \\
 & | & \mathbf{assume}\ \mathit{Expr}\ ; \\
 & | & \mathit{Designator}\ =\ \mathit{Expr}\ ; \\
 & | & \mathit{Call} \\
 & | & \mathit{IfStmt} \\
 & | & \mathit{WhileStmt} \\
 & | & \mathbf{return}\ \mathit{Expr}\ ; \\
\mathit{Block} & ::= & \{\ \mathit{Stmt}^*\ \} \\
\mathit{IfStmt} & ::= & \mathbf{if}\ (\ \mathit{Expr}\ )\ \mathit{Stmt}\ \mathit{ElseStmt}^? \\
\mathit{ElseStmt} & ::= & \mathbf{else}\ \mathit{Stmt} \\
\mathit{WhileStmt} & ::= & \mathbf{while}\ (\ \mathit{Expr}\ )\ \mathit{Invariant}^?\ \mathit{Block} \\
\mathit{Invariant} & ::= & \mathbf{invariant}\ \mathit{Expr}\ ;
\end{array}$$

If a statement attempts to evaluate an undefined expression (like x/y when y evaluates to 0), the statement *goes wrong*, which is an unrecoverable error.

The block statement consists of a sequence of statements, which are executed in order. The declaration statement $T\ x$; introduces a local variable x whose scope goes from the declaration until the end of the enclosing block. As usual, x must be distinct from any other variable introduced among the statements, but, unlike in BoogiePL$^\flat$, a variable in Spec$^\flat$ must not hide another local variable x in an outer scope. If a local variable hides a field in scope, then the field has to be accessed via **this** explicitly.

The statement $T\ x =\ E$; is simply a shorthand for $T\ x;\ x =\ E$;.

Syntax and semantics of $\mathbf{assert}$ and $\mathbf{assume}$ are the same as for BoogiePL$^\flat$, except that the Spec$^\flat$ statements also check that their expressions are defined. Embedded expressions must not contain call or allocation expressions. In Spec$^\flat$, the assume statement introduces an assumption that is used, but not validated, by the program verifier; the assumption can instead be validated at run time.

In the assignment statement $d = E;$, the type of E must be a subtype of the type of d. If d is of the form $O.f$, then f must not be the built-in readonly fields Type or Length. The statement goes wrong if d or E is not defined; otherwise, it assigns the value of E to d. More specifically, if d is of the form $O.f$, the statement updates the heap so that field f of object O becomes E; if d is of the form $A[F]$, the statement updates the heap so that element F of array A becomes E.

The call statement is explained in Section 2.2. Only constructors (called via Base) and methods with a void return type can be used as Call statements.

In the conditional statement $\mathbf{if}\ (E)\ S\ \mathbf{else}\ T$, expression E must be of type $\mathbf{bool}$. The statement goes wrong if E is not defined. Otherwise, if E evaluates to $\mathbf{true}$, the statement executes S, and if E evaluates to $\mathbf{false}$, the statement executes T. If "$\mathbf{else}\ S$" is omitted, S defaults to $\{\}$. Parsing of conditional statements is ambiguous; we resolve any ambiguity of parsing $\mathbf{else}$ statements by associating them to the rightmost (innermost) $\mathbf{if}$.

In the while loop $\mathbf{while}\ (E)\ \mathbf{invariant}\ J;\ \mathit{Body}$, the condition E and loop invariant J must be of type $\mathbf{bool}$. Furthermore, J must not contain call or allocation expressions. As we explained in Section 1.3, the statement goes wrong if J is not defined, if J does not evaluate to $\mathbf{true}$, or if E is not defined. Otherwise, if E evaluates to $\mathbf{true}$, then Body is executed, after which the entire while loop is executed again (including the evaluation of the loop invariant). If E evaluates to $\mathbf{false}$, the execution of the while loop terminates.

In the return statement $\mathbf{return}\ E;$, the type of E must be a subtype of the method's return type. The return statement is allowed only as the last statement of a method implementation. The statement goes wrong if E is not defined; otherwise, it returns the value of E to the method's caller.

2.2. Expressions

Spec$^\flat$ expressions follow the grammar in Fig. 4.

Spec$^\flat$ shares most of its operators with BoogiePL$^\flat$. Except as noted here, the shared operators have the same typing, precedence, and meaning. In Spec$^\flat$, division and modulo are defined only for non-zero divisors. For the relational operators $=$ and $\neq$, the types of the operands must be *compatible*; that is, the type of one operand must be a subtype of the type of the other.

Instead of the logical operators $\Rightarrow$, $\wedge$, and $\vee$ in BoogiePL$^\flat$, Spec$^\flat$ defines the corresponding *short-circuit* versions of these operators, written $\overset{.}{\Longrightarrow}$, &&, and $||$, respectively. Short-circuiting means that if the left-hand operand is defined and evaluates to a value that determines the result of the operator ($\mathbf{false}$, $\mathbf{false}$, and $\mathbf{true}$, respectively), then the expressions are defined regardless of whether or not the right-hand operand is defined.

Spec$^\flat$ also adds the binary $\mathbf{is}$ operator, whose precedence lies between those of $=$ and &&. The precedence of cast expressions, where the prefix "(T)" is like a unary operator, is just higher than $\otimes$. Although we do not show them in the grammar, we

$$\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Expr} \otimes \mathit{Expr} \\
& \mid & \neg\, \mathit{Expr} \\
& \mid & \mathit{Atom} \\
\otimes & ::= & * \mid / \mid \% \mid + \mid - \mid < \mid \leqslant \mid \geqslant \mid > \mid = \mid \neq \\
& \mid & \&\& \mid \,||\, \mid \dot{\Longrightarrow} \\
\mathit{Atom} & ::= & \mathbf{this} \mid \mathbf{result} \\
& \mid & \mathit{Literal} \\
& \mid & \mathit{Designator} \\
& \mid & \mathit{Call} \\
& \mid & \mathit{Allocation} \\
& \mid & \mathit{Expr}\ \mathbf{is}\ \mathit{Type} \\
& \mid & (\ \mathit{Type}\ )\ \mathit{Expr} \\
& \mid & \mathbf{old}\ (\ \mathit{Expr}\ ) \\
& \mid & \mathbf{fresh}\ (\ \mathit{Expr}\ ) \\
& \mid & \mathit{Quantification} \\
\mathit{Literal} & ::= & \mathbf{false} \mid \mathbf{true} \mid \mathbf{null} \mid 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Designator} & ::= & \mathit{Id} \\
& \mid & \mathit{Expr}\ .\ \mathit{Id} \\
& \mid & \mathit{Expr}\ [\ \mathit{Expr}\ ] \\
\mathit{Call} & ::= & \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
& \mid & \mathit{Expr}\ .\ \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
& \mid & \mathbf{base}\ (\ \mathit{Expr}^*\ ) \\
& \mid & \mathbf{base}\ .\ \mathit{Id}\ (\ \mathit{Expr}^*\ ) \\
\mathit{Allocation} & ::= & \mathbf{new}\ \mathit{ObjectType}\ (\ \mathit{Expr}^*\ ) \\
& \mid & \mathbf{new}\ \mathit{Type}\ [\ \mathit{Expr}\ ] \\
\mathit{Quantification} & ::= & \mathit{Quantor}\ \{\ \mathit{Binding}\ ;\ \mathit{Expr}\ \} \\
\mathit{Quantor} & ::= & \mathbf{forall} \mid \mathbf{exists} \\
\mathit{Binding} & ::= & \mathbf{int}\ \mathit{Id}\ \mathbf{in}\ (\ \mathit{Expr}\ :\ \mathit{Expr}\ )
\end{array}$$

Figure 4. The grammar of Spec$^\flat$ expressions.

also allow expressions to contain parentheses, which can be used to override operator precedence.

The type of the expression **this** is the enclosing class. The keyword **result** is allowed to appear only in **ensures** clauses of non-void, non-constructor methods; its type is the method's return type, and its value is the method's return value. The type of **null** is any reference type. The type of boolean literals is **bool**, the type of integer literals is **int**.

Three forms of *designators* are distinguished. In the first form, if Id does not mention a variable, then it is a synonym for **this**.Id. The type of the expression Id is the type of the variable Id. In the second form, $E.f$, E must be of a reference type, call it T, and f must be Type, Length (if E is of an array type), or a field declared in class T or a superclass thereof. The expression is defined only if E evaluates to a non-null value. The type of $E.\mathit{Type}$ is **object**, the type of $E.\mathit{Length}$ is **int**, and otherwise the type of $E.f$ is the type of f. In the third form, $E[F]$, E must be of an array type and F of type integer. The expression is defined only if E evaluates to a non-null value, F evaluates to a non-negative integer that is less than $E.Length$. The type of the expression is the element type of E.

Spec$^\flat$ supports four forms of *call expressions*. All legal call expressions resolve to some method. The types of the expressions in list EE must be subtypes of the types of the respective formal parameters of the callee. If the callee is a constructor or has a void return type, then the call is allowed only as a statement; otherwise, the type of the call expression is the return type of the callee.

The first form of the call expression, $m(EE)$, is a shorthand for $\mathbf{this}.m(EE)$.

In the second form, $E.m(EE)$, E must be of an object type, call it T, and m must name a non-constructor method in T or a superclass thereof. The call expression is defined only if E evaluates to a non-null value. The expression binds the formal receiver parameter **this** of m to the value of E and binds the formal parameters of m to the values of EE. Then, if any precondition of m evaluates to **false**, the call statement goes wrong. The evaluation of the call expression proceeds by transferring control to the method's implementation, upon return of which the result value becomes the value of the call expression. If m is a virtual method, then the implementation invoked is the one found in the most derived supertype of the run-time type of E.

The third form, $\mathbf{base}.m(EE)$ where m must denote a virtual method, is allowed only in overrides of m. It is treated like the call $\mathbf{this}.m(EE)$, except that control transfers to the implementation found in the most derived supertype of the immediate superclass of the enclosing class.

The fourth form, $\mathbf{base}(EE)$, is allowed only as the first statement in constructors. It calls the constructor of the immediate superclass.

Two forms of *allocation expressions* are supported. The expression $\mathbf{new}\ C(EE)$, where C must name a class, has type C. It allocates a new object c of run-time type C with all of c's fields set to zero-equivalent values. Next, it calls $C's$ constructor with c as the receiver and EE as its actual parameters. All constraints of method calls have to be obeyed. Upon return of the constructor, c is the result of the expression. The expression $\mathbf{new}\ T[F]$, where T must be a type and F must be of type **int**, has type $T[\,]$. It allocates and returns a new array a of run-time type $T[\,]$ and of length F. All array elements of a have zero-equivalent values. The statement goes wrong if either F is not defined or if F evaluates to a negative integer.

For the type test expression $E\ \mathbf{is}\ T$ and cast expression $(T)E$, T must be a reference type and the type of E must be compatible with T. The type of $E\ \mathbf{is}\ T$ is **bool**, the type of $(T)E$ is T. The type test expression evaluates to **true** if E evaluates to a non-null reference whose run-time type is a subtype of T. The cast expression is defined only if E evaluates to **null** or if $E\ \mathbf{is}\ T$ would evaluate to **true**, and it returns the value of E.

The expressions $\mathbf{old}(E)$ and $\mathbf{fresh}(E)$ are allowed to appear only in **ensures** clauses. (Note, unlike BoogiePL$^\flat$, Spec$^\flat$ does not allow **old** expressions in code.) The type of $\mathbf{old}(E)$ is the type of E and the type of $\mathbf{fresh}(E)$ is **bool**. The former returns the value of E evaluated on entry to the method; the latter returns **true** if E denotes an object that was not yet allocated on method entry.

In a quantifying expression $\mathbf{forall}\{\mathbf{int}\ x\ \mathbf{in}\ (M : N);\ E\}$, M and N must have type **int** and the expression E must have type **bool**. The newly introduced x is in scope in E, but not in M or N. Expressions M, N, and E must not include call or allocation expressions. The quantifying expression has type **bool**. It is defined and returns

true, respectively, if E is defined and evaluates to **true**, respectively, for every value of x satisfying $N \leqslant x < M$. Existential quantification is defined in terms of universal quantification in the usual way.

2.3. Translating Spec$^\flat$ into BoogiePL$^\flat$

We give the semantics of Spec$^\flat$ in terms of a translation into BoogiePL$^\flat$.

2.3.0. Prelude

The translation into BoogiePL$^\flat$ begins with the prelude described in this subsection. The prelude is specific to Spec$^\flat$, but independent of the particular program being translated.

Axiomatizing the Type System We map the Spec$^\flat$ types **bool** and **int** to the corresponding BoogiePL$^\flat$ types (we ignore the fact that Spec$^\flat$'s integers have a fixed size). We map all reference types in Spec$^\flat$ to the BoogiePL$^\flat$ type **ref**.

We introduce a name for each Spec$^\flat$ type. The names of the built-in types are:

$\textbf{const}\ \_bool\colon \textbf{name}\ ;$
$\textbf{const}\ \_int\colon \textbf{name}\ ;$
$\textbf{const}\ object\colon \textbf{name}\ ;$

Since the names are declared as constants of type **name**, BoogiePL$^\flat$ provides the implicit axiom that the type names are distinct.

Spec$^\flat$'s subtyping relation is captured by BoogiePL$^\flat$'s partial-order operator $<:$ and is specified via axioms. To tie $<:$ to type names, we introduce a function *superclass*:

$\textbf{function}\ superclass(\textbf{name})\ \textbf{returns}\ (\textbf{name})\ ;$
$\textbf{axiom}\ (\forall\, T\colon \textbf{name} \bullet\ T <: superclass(T)\ )\ ;$

The function *array* maps a type name T to the name of an array type. Given the name of such an array type, function *elemType* gives back T.

$\textbf{function}\ array(\textbf{name})\ \textbf{returns}\ (\textbf{name})\ ;$
$\textbf{function}\ elemType(\textbf{name})\ \textbf{returns}\ (\textbf{name})\ ;$
$\textbf{axiom}\ (\forall\, T\colon \textbf{name} \bullet\ elemType(array(T)) = T\ )\ ;$

Array types are distinct and co-variant:

$\textbf{axiom}\ (\forall\, T\colon \textbf{name} \bullet\ array(T) <: object\ )\ ;$
$\textbf{axiom}\ (\forall\, T\colon \textbf{name},\ U\colon \textbf{name} \bullet\ array(T) <: U\ \Rightarrow$
$\quad U = object \lor (U = array(elemType(U)) \land T <: elemType(U))\ )\ ;$

Fields are also declared to be of type **name**. We introduce a function that maps names of fields to the names of their declared types:

$\textbf{function}\ fieldType(\textbf{name})\ \textbf{returns}\ (\textbf{name})\ ;$

Function *type* returns the name of the run-time type of a non-null reference.

$\textbf{function}\ type(\textbf{ref})\ \textbf{returns}\ (\textbf{name})\ ;$

If o has static type T for a reference type T, then the static type system guarantees that $type(o) <: T$ holds.

Storage Model We model the heap as a map from references and field names to values.

$\mathbf{var}\ \mathcal{H}\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}\ ;$

Our heap variable includes all references, allocated or not. We introduce a field $alloc$ to track whether or not a reference has been allocated. We refer to such a field as a *ghost field*, meaning that it is not explicitly represented in the Spec$^\flat$ program.

$\mathbf{const}\ alloc\colon \mathbf{name}\ ;$

$\mathcal{H}[o, alloc]$ says that o is allocated in $\mathcal{H}$.

Not all mathematical maps are heaps reachable in a Spec$^\flat$ program. We introduce a function $wellFormed(h)$ to describe that h is a reachable heap.

$\mathbf{function}\ wellFormed([\mathbf{ref}, \mathbf{name}]\mathbf{any})\ \mathbf{returns}\ (\mathbf{bool})\ ;$

In a well-formed heap, reference-valued fields map allocated references to allocated references of the appropriate type.

```
$\mathbf{axiom}\ (\forall\, h\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any} \bullet\ wellFormed(h)\ \Rightarrow$
  $(\forall\, r\colon \mathbf{ref},\ f\colon \mathbf{name} \bullet\ r \neq \mathbf{null} \wedge h[r, alloc] \wedge fieldType(f) <: object\ \Rightarrow$
    $h[r, f] = \mathbf{null} \vee (h[h[r, f], alloc] \wedge type(h[r, f]) <: fieldType(f))\ ))\ ;$
```

We introduce a function that relates the heap at two successive program points. It says that the new heap is well-formed and that every reference allocated in the old heap is also allocated in the new heap.

```
$\mathbf{function}\ successor([\mathbf{ref}, \mathbf{name}]\mathbf{any},\ [\mathbf{ref}, \mathbf{name}]\mathbf{any})\ \mathbf{returns}\ (\mathbf{bool})\ ;$
$\mathbf{axiom}\ (\forall\, \_old\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any},\ \_new\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any} \bullet$
  $successor(\_old, \_new)\ \Rightarrow$
    $wellFormed(\_new)\ \wedge$
    $(\forall\, r\colon \mathbf{ref} \bullet\ \_old[r, alloc]\ \Rightarrow\ \_new[r, alloc]\ ))$
```

The elements of an array are stored as one "big" value in a ghost field called $elems$:

$\mathbf{const}\ elems\colon \mathbf{name}\ ;$

For example, the Spec$^\flat$ array dereference expression $a[j]$ is translated into BoogiePL$^\flat$ as $Select(\mathcal{H}[a, elems],\ j)$. The length of an array is modeled as a function:

$\mathbf{function}\ length(\mathbf{ref})\ \mathbf{returns}\ (\mathbf{int})\ ;$

Array elements are assigned and updated using the following functions.

```
$\mathbf{function}\ Select(\mathbf{any}, \mathbf{int})\ \mathbf{returns}\ (\mathbf{any})\ ;$
$\mathbf{function}\ Store(\mathbf{any}, \mathbf{int}, \mathbf{any})\ \mathbf{returns}\ (\mathbf{any})\ ;$
```

These functions are related as follows:

```
$\mathbf{axiom}\ (\forall\, e\colon \mathbf{any},\ i\colon \mathbf{int},\ j\colon \mathbf{int},\ v\colon \mathbf{any} \bullet$
  $(i = j\ \Rightarrow\ Select(Store(e, i, v), j) = v)\ \wedge$
  $(i \neq j\ \Rightarrow\ Select(Store(e, i, v), j) = Select(e, j))\ )\ ;$
```

In a well-formed heap, reference values stored in $elems$ fields are allocated and of the appropriate type:

$$
\begin{array}{l}
\textbf{axiom}\ (\forall\, h{:}\,[\textbf{ref},\textbf{name}]\textbf{any} \bullet\ \mathit{wellFormed}(h)\ \Rightarrow \\
\quad (\forall\, r{:}\,\textbf{ref},\ i{:}\,\textbf{int} \bullet \\
\qquad r \neq \textbf{null} \wedge h[r, \mathit{alloc}] \wedge \mathit{elemType}(\mathit{type}(r)) <: \mathit{object} \wedge \\
\qquad 0 \leqslant i \wedge i < \mathit{length}(r)\ \Rightarrow \\
\qquad\quad \mathit{Select}(h[r, \mathit{elems}], i) = \textbf{null} \vee \\
\qquad\quad (h[\mathit{Select}(h[r, \mathit{elems}], i), \mathit{alloc}] \wedge \\
\qquad\qquad \mathit{type}(\mathit{Select}(h[r, \mathit{elems}], i)) <: \mathit{elemType}(\mathit{type}(r)))\ ))\ ;
\end{array}
$$

Object constructor Class **object** is built into the language, so we predefine the specification of its constructor:

$$\textbf{procedure}\ \mathit{object}^{\circ}\mathit{object}(\mathit{this}{:}\,\textbf{ref})\ \textbf{returns}\ (\,)\ ;$$

2.3.1. Classes and Fields

In the sequel, we think of the translation as producing a stream of BoogiePL$^\flat$ program text. The translation is described formally using the function Tr, which takes a Spec$^\flat$ fragment and produces a BoogiePL$^\flat$ fragment.

For the translation of a *program*, which consists of a list classes, we have:

```
$Tr[\![classes]\!] =$
    for each $c \in classes$ do
        $Tr[\![c]\!]$
```

The prescription of the translation requires control structures, which we introduce as meta-syntax, such as the "for each ... do..." construct here. Note that meta-syntax is written in a Roman font.

We translate a *class declaration* as follows:

```
$Tr[\![\textbf{class}\ T : S\ \{\ members\ \}]\!] =$
    $\textbf{const}\ T{:}\,\textbf{name}\ ;$
    $\textbf{axiom}\ superclass(T) = S\ ;$
    for each $m \in members$ do
        $Tr[\![m]\!]$
```

For brevity, we use a star to map a translation function over a list of fragments. In this notation, we write the last two lines as just:

$$Tr^{*}[\![members]\!]$$

As usual, we are sloppy with the connectives between the translated fragments; the implicit connectives are conjunction, some punctuation, or white space.

In the rest of this subsection, we use $\mathbb{C}$ to denote the name of the current class.

Field declarations are translated as follows:

```
$Tr[\![T\ f;]\!] =$
    $\textbf{const}\ \mathbb{C}^{\circ}f{:}\,\textbf{name}\ ;$
    $\textbf{axiom}\ fieldType(\mathbb{C}^{\circ}f) = Type[\![T]\!]\ ;$
```

We use "$\circ$" as just another character that can appear as part of identifier names in BoogiePL$^\flat$, but that cannot be used in Spec$^\flat$. Translation function *Type* gives the BoogiePL$^\flat$ term for Spec$^\flat$ types:

$$\begin{array}{lcll}
\mathit{Type}[\![\mathbf{bool}]\!] & = & \_\mathit{bool} & \\
\mathit{Type}[\![\mathbf{int}]\!] & = & \_\mathit{int} & \\
\mathit{Type}[\![T]\!] & = & T & \text{for any object type } T \\
\mathit{Type}[\![T[\,]]\!] & = & \mathit{array}(\mathit{Type}[\![T]\!]) &
\end{array}$$

2.3.2. Methods

The translation of *method declarations* is more involved. Recall that BoogiePL$^\flat$ only has procedures, no instance methods, so we add **this** as an explicit parameter to the generated procedure. Furthermore, since BoogiePL$^\flat$ types are semantic-less, we instead preserve Spec$^\flat$ types via specifications. Also, BoogiePL$^\flat$ has no built-in notion of heap properties, so we preserve properties like allocatedness of references via specifications. BoogiePL$^\flat$ has no notion of inheritance, so we translate overriding and strengthening of postconditions using multiple procedures. Finally, BoogiePL$^\flat$ syntactically distinguishes calls with possible side effect from side-effect free expressions; thus, we flatten Spec$^\flat$ method bodies as part of the translation into BoogiePL$^\flat$ expressions and commands.

New Methods The declaration of a new non-virtual or virtual method in a class $\mathbb{C}$ is translated into BoogiePL$^\flat$ as follows:

```
$\mathit{Tr}[\![\mathbf{virtual}^?\ T\ m\ (\mathit{Args})\ \ \mathit{Spec}\ \mathit{Body}]\!] =$
    $\mathbf{procedure}\ \mathbb{C}{\circ}m\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\mathit{Tr}[\![T\ \_\mathit{result}]\!])\ ;$
        $\mathit{Tr}^*[\![\mathit{Spec}]\!]$
        $\mathit{TrMod}[\![\mathit{Spec}]\!]$
    $\mathbf{implementation}\ \mathbb{C}{\circ}m\ (\mathit{Tr}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!])\ \mathbf{returns}\ (\mathit{Tr}[\![T\ \_\mathit{result}]\!])$
        $\{\ \mathbf{assume}\ \mathit{wellFormed}(\mathcal{H})\ ;$
          $\mathbf{assume}\ \mathit{this} \neq \mathbf{null}\ ;$
          $\mathbf{assume}\ \mathit{TypeConstraint}^*[\![\mathbb{C}\ \mathit{this}, \mathit{Args}]\!]\ ;$
          $\mathit{Tr}[\![\mathit{Body}]\!]$
        $\}$
```

where formal parameters are translated as follows:

$$\begin{array}{lcll}
\mathit{Tr}[\![\mathbf{bool}\ x]\!] & = & x\!:\mathbf{bool} & \\
\mathit{Tr}[\![\mathbf{int}\ x]\!] & = & x\!:\mathbf{int} & \\
\mathit{Tr}[\![T\ x]\!] & = & x\!:\mathbf{ref} & \text{for any reference type } T \\
\mathit{Tr}[\![\mathbf{void}\ x]\!] & = & &
\end{array}$$

The last case is intentionally left blank; it is used only for method return types, and a method with a void return type gives rise to no out-parameter is the translation.

The types of the formal parameters we just described are there only to please BoogiePL$^\flat$. The run-time types guaranteed by the static type system of Spec$^\flat$ give rise to assumptions:

$$\begin{array}{lll} \mathit{TypeConstraint}[\![\mathbf{bool}\ x]\!] & = & \\ \mathit{TypeConstraint}[\![\mathbf{int}\ x]\!] & = & \\ \mathit{TypeConstraint}[\![T\ x]\!] & = & \text{for any reference type } T \\ \quad x = \mathbf{null} \lor (\mathcal{H}[x, \mathit{alloc}] \land \mathit{type}(x) <: \mathit{Type}[\![T]\!]) & & \end{array}$$

For convenience later, we also define:

$$\mathit{TypeConstraint}[\![\mathcal{H}]\!] \ =$$

Overriding Methods If a method overrides a virtual method, then any new postconditions are added to those previously declared.

```
$Tr[\![\mathbf{override}\ T\ m\ (Args)\ \ Spec\ Body]\!] =$
    $\mathbf{procedure}\ \mathbb{C}°m\ (Tr[\![\mathbb{C}\ this, Args]\!])\ \mathbf{returns}\ (Tr[\![T\ \_result]\!])$ ;
      for the “$\mathbf{virtual}\ T\ m\ (Args)\ \ Spec'\ Body'$” in a superclass of $\mathbb{C}$ do
        $Tr^*[\![Spec']\!]$
        $TrMod[\![Spec']\!]$
      for each “$\mathbf{override}\ T\ m\ (Args)\ \ Spec'\ Body'$” in $\mathbb{C}$ or a superclass thereof do
        $Tr^*[\![Spec']\!]$
    $\mathbf{implementation}\ \mathbb{C}°m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \mathbf{returns}\ (Tr[\![T\ \_result]\!])$
      { $\mathbf{assume}\ wellFormed(\mathcal{H})$ ;
        $\mathbf{assume}\ this \neq \mathbf{null}$ ;
        $\mathbf{assume}\ TypeConstraint^*[\![\mathbb{C}\ this, Args]\!]$ ;
        $Tr[\![Body]\!]$
      }
```

Constructors For constructors, we automatically grant the license to modify all fields of the object being constructed. The implementation initializes the fields before translating the given constructor body.

```
$Tr[\![\mathbb{C}\ (Args)\ Spec\ Body]\!] =$
    $\mathbf{procedure}\ \mathbb{C}°\mathbb{C}\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \mathbf{returns}\ (\,)$ ;
      $Tr^*[\![Spec]\!]$
      $TrMod[\![\mathbf{modifies\ this}.*;\ Spec]\!]$
    $\mathbf{implementation}\ \mathbb{C}°\mathbb{C}\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \mathbf{returns}\ (\,)$
      { $\mathbf{assume}\ wellFormed(\mathcal{H})$ ;
        $\mathbf{assume}\ this \neq \mathbf{null}$ ;
        $\mathbf{assume}\ TypeConstraint^*[\![\mathbb{C}\ this, Args]\!]$ ;
        for each field “$F\ f$;” defined in $\mathbb{C}$ do
          $\mathbf{assume}\ \mathcal{H}[this, \mathbb{C}°f] = Zero[\![F]\!]$ ;
        $Tr[\![Body]\!]$
      }
```

where

$$\begin{array}{lll} \mathit{Zero}[\![\mathbf{bool}]\!] & = & \mathbf{false} \\ \mathit{Zero}[\![\mathbf{int}]\!] & = & 0 \\ \mathit{Zero}[\![R]\!] & = & \mathbf{null} \qquad \text{for any reference type } R \end{array}$$

Method Specifications Translating pre- and postconditions is straightforward:

$$\begin{array}{lll} Tr[\![\mathbf{requires}\ E;]\!] & = & \mathbf{requires}\ Df[\![E]\!] \land Tr[\![E]\!]\ ; \\ Tr[\![\mathbf{modifies}\ W;]\!] & = & \\ Tr[\![\mathbf{ensures}\ E;]\!] & = & \mathbf{ensures}\ Df[\![E]\!] \land Tr[\![E]\!]\ ; \end{array}$$

Here, we have opted for the simple design of putting the burden of establishing the definedness of the precondition on callers and the burden of establishing the definedness of the postcondition on the implementation.

To translate the modifies clauses of a method, we first collect all of them and then add the contribution of the modifies list to the method's postcondition. This is described by the following function:

```
$TrMod[\![Spec]\!] =$
    $\mathbf{modifies}\ \mathcal{H}\ ;$
    $\mathbf{ensures}\ (\forall\, o\!:\!\mathbf{ref},\ f\!:\!\mathbf{name} \bullet$
      $o \neq \mathbf{null} \land \mathbf{old}(\mathcal{H})[o, alloc] \Rightarrow$
        $ModAllowed[\![Spec, o, f]\!] \lor$
        $\mathcal{H}[o, f] = \mathbf{old}(\mathcal{H})[o, f]\ )$
```

where *ModAllowed* generates a disjunction of the translated modifies-clause terms:

```
$ModAllowed[\![Spec, o, f]\!] =$
    for each "$\mathbf{modifies}\ W;$" in $Spec$ do
      for each "$desig\ suffix$" in $W$ do
        case $suffix$ of
          $(.g)$ :     $(o = \mathbf{old}(Tr[\![desig]\!]) \land f = g) \lor$
          $(.*)$ :     $(o = \mathbf{old}(Tr[\![desig]\!])) \lor$
          $([*])$ :    $(o = \mathbf{old}(Tr[\![desig]\!]) \land f = elems) \lor$
```

2.3.3. Statements

The translation of statements needs a preprocessing step, which we call normalizing.

Normalization A Spec$^\flat$ body is in normal form, if (i) it is context extended, *i.e.*, all its names are properly resolved; (ii) local variable declarations appear only at the beginning of a block; and (iii) allocations and non-void returning calls appear only as right-hand sides of the designator form Id.

We establish (i) as follows: We add **this** as the target expression to each designator Id that references a field or method in scope. We prefix each application of an Id that denotes a method or field in scope (except the built-in fields *Type* and *Length*), with the most derived class of its definition. For example, if a class A declares a method M, a subclass B overrides M, and C is a subclass of B that does not declare a further override, then $c.M(EE)$ where c has static type C is normalized into $c.B{\circ}M(EE)$. We normalize each call of the form $\mathbf{base}.M(EE)$ into a call $\mathbf{this}.S{\circ}M(EE)$, where S is the most derived class of M's definition among superclasses of the enclosing class. Finally, each call of the form $\mathbf{base}(EE)$ is normalized into a call $\mathbf{this}.S{\circ}S(EE)$ where S is the immediate superclass of the enclosing class.

We establish (ii) by moving each local variable declaration to the beginning of its immediately enclosing block. Spec$^\flat$'s context conditions guarantee that this preserves the meaning of the program.

We establish (iii) by repeatedly applying the following normalizing transformations:

- Let S be an assignment, call, or return statement that contains an allocation or call subexpression. Then, select the leftmost innermost subexpression e in S that is a non-Id designator, a call expression, an allocation expression, or a quantifier, and is not the entire left-hand side of an assignment or the entire call statement; we write $S[e]$ to single out that occurrence of e. If such an e exists, then $S[e]$ is normalized into

  $$\{T\ x;\ x =\ e;\ S[x]\}$$

  where x is a fresh identifier and T is the type of e.
- If the guard expression E of a conditional statement **if** $(E)\ S$ **else** T contains an allocation or call expression, then the conditional statement is normalized into

  $$\{\mathbf{bool}\ x;\ x =\ E;\ \mathbf{if}\ (x)\ S\ \mathbf{else}\ T\}$$

- If the guard expression E of a while loop $\mathbf{while}\ (E)\ \mathbf{invariant}\ J;\ \{S\}$ contains an allocation or call expression, then the while loop is normalized into

  $$\{\mathbf{bool}\ x;\ x =\ E;\ \mathbf{while}\ (x)\ \mathbf{invariant}\ J;\ \{\{S\}\ x =\ E;\}\}$$

This preserves the meaning of the program, since it reflects Spec$^\flat$'s leftmost-innermost evaluation order.

Translation We now define the translation of normalized statements.

The translation of blocks is straightforward: translate each variable declaration followed by the translation of the individual statements.

$$\begin{array}{l} Tr[\![\{\mathit{typeIds}\ \mathit{stmts}\}]\!] = \\ \quad \{\ Tr^*[\![\mathit{typeIds}]\!]\ Tr^*[\![\mathit{stmts}]\!]\ \} \end{array}$$

The translations of assert, assume, and return statements check that everything is defined before the corresponding BoogiePL$^\flat$ command is generated:

$$\begin{array}{lcl} Tr[\![\mathbf{assert}\ E;]\!] & = & \mathbf{assert}\ \mathit{Df}[\![E]\!]\ ;\mathbf{assert}\ Tr[\![E]\!] \\ Tr[\![\mathbf{assume}\ E;]\!] & = & \mathbf{assert}\ \mathit{Df}[\![E]\!]\ ;\mathbf{assume}\ Tr[\![E]\!] \\ Tr[\![\mathbf{return}\ E;]\!] & = & \mathbf{assert}\ \mathit{Df}[\![E]\!]\ ;\_\mathit{result} := Tr[\![E]\!] \end{array}$$

The bulk of the remaining translation is translating assignments. Field update is translated as follows:

$$\begin{array}{l} Tr[\![E.f =\ F;]\!] = \\ \quad \mathbf{assert}\ \mathit{Df}[\![E]\!]\ ;\mathbf{assert}\ Tr[\![E]\!] \neq \mathbf{null}\ ; \\ \quad \mathbf{assert}\ \mathit{Df}[\![F]\!]\ ; \\ \quad \mathcal{H}[Tr[\![E]\!], f] := Tr[\![F]\!]) \end{array}$$

Array update needs to check that the array is non-null, that the index is within the bounds of the array, and that run-time type of G is a subtype of the element type of the run-time type of the array (that is, we check for co-variance).

```
$Tr[\![E[F] = \ G;]\!] =$
    $\mathbf{assert}\ Df[\![E]\!]$ ; $\mathbf{assert}\ Tr[\![E]\!] \neq \mathbf{null}$ ;
    $\mathbf{assert}\ Df[\![F]\!]$ ; $\mathbf{assert}\ 0 \leqslant Tr[\![F]\!] \wedge Tr[\![F]\!] < length(Tr[\![E]\!])$ ;
    $\mathbf{assert}\ Df[\![G]\!]$ ; $\mathbf{assert}\ type(Tr[\![G]\!]) <: elemType(type(Tr[\![E]\!]))$ ;
    $\mathcal{H}[Tr[\![E]\!], elems] := Store(\mathcal{H}[Tr[\![E]\!], elems], Tr[\![F]\!], Tr[\![G]\!])$
```

Since the statements we translate are normalized, there are only four cases of local-variable assignments to consider. When the right-hand side is an allocation of an object, then the assignment is translated as follows, where o and $oldHeap$ denote fresh variables:

```
$Tr[\![x = \ \mathbf{new}\ C(EE);]\!] =$
    $\{$ $\mathbf{var}\ o\colon \mathbf{ref}$ ; $\mathbf{var}\ oldHeap\colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}$ ;
      $\mathbf{assume}\ o \neq \mathbf{null} \wedge type(o) = C$ ;
      $\mathbf{assume}\ \neg\mathcal{H}[o, alloc]$ ;
      $\mathcal{H}[o, alloc] := \mathbf{true}$ ;
      $\mathbf{assert}\ Df^*[\![EE]\!]$ ;
      $oldHeap := \mathcal{H}$ ;
      $\mathbf{call}\ C^\circ C(o, Tr^*[\![EE]\!])$ ;
      $\mathbf{assume}\ successor(oldHeap, \mathcal{H})$ ;
      $x := o$
    $\}$
```

This translation picks an arbitrary o with the properties that it is non-null, has the appropriate run-time type, and is not yet allocated. The translation then allocates the object o by setting its $alloc$ field to $\mathbf{true}$. Finally, it calls the C constructor, adds the assumption that the post-call heap is a well-formed successor of the pre-call heap, and assigns o to the local variable in the assignment statement.

Array allocation is similar:

```
$Tr[\![x = \ \mathbf{new}\ T[E];]\!] =$
    $\{$ $\mathbf{var}\ o\colon \mathbf{ref}$ ;
      $\mathbf{assume}\ o \neq \mathbf{null} \wedge type(o) = Type[\![T[\,]\,]\!]$ ;
      $\mathbf{assume}\ \neg\mathcal{H}[o, alloc]$ ;
      $\mathbf{assert}\ Df[\![E]\!]$ ; $\mathbf{assert}\ 0 \leqslant Tr[\![E]\!]$ ;
      $\mathbf{assume}\ length(o) = Tr[\![E]\!]$ ;
      $\mathbf{assume}\ (\forall\, i\colon \mathbf{int}\ \bullet$
        $0 \leqslant i \wedge i < Tr[\![E]\!] \Rightarrow Select(\mathcal{H}[o, elems],\ i) = Zero[\![T]\!]\ )$ ;
      $\mathcal{H}[o, alloc] := \mathbf{true}$ ;
      $x := o$
    $\}$
```

Here, there are two additional assumptions about the reference o: that o has the specified length E, which we check to be non-negative, and that the elements of o all have zero-equivalent values.

When the right-hand side is a call to a method $T^\circ m$ with return type R, then the assignment is translated as follows:

```
$Tr[\![x = \ E.T{\circ}m(EE);]\!] =$
    $\{\ \mathbf{var}\ oldHeap \colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}\ ;$
      $\mathbf{assert}\ Df[\![E]\!]\ ; \mathbf{assert}\ Tr[\![E]\!] \neq \mathbf{null}\ ;$
      $\mathbf{assert}\ Df^{*}[\![EE]\!]\ ;$
      $oldHeap := \mathcal{H}\ ;$
      $\mathbf{call}\ x := T{\circ}m(\,Tr^{*}[\![E, EE]\!])\ ;$
      $\mathbf{assume}\ successor(oldHeap, \mathcal{H})\ ;$
      $\mathbf{assume}\ TypeConstraint[\![R\ x]\!]$
    $\}$
```

The last assumption states properties that are guaranteed by the Spec$^\flat$ type system.

For all other assignments to local variables, the translation is:

```
$Tr[\![x = \ E;]\!] =$
    $\mathbf{assert}\ Df[\![E]\!]\ ;$
    $x := Tr[\![E]\!]$
```

Call statements are like the calls in local-variable assignments, but they use void methods and have no result value:

```
$Tr[\![E.T{\circ}m(EE);]\!] =$
    $\{\ \mathbf{var}\ oldHeap \colon [\mathbf{ref}, \mathbf{name}]\mathbf{any}\ ;$
      $\mathbf{assert}\ Df[\![E]\!]\ ; \mathbf{assert}\ Tr[\![E]\!] \neq \mathbf{null}\ ;$
      $\mathbf{assert}\ Df^{*}[\![EE]\!]\ ;$
      $oldHeap := \mathcal{H}\ ;$
      $\mathbf{call}\ T{\circ}m(\,Tr^{*}[\![E, EE]\!])\ ;$
      $\mathbf{assume}\ successor(oldHeap, \mathcal{H})$
    $\}$
```

The translation of the conditional statement is the one we showed in Section 1.3.

The translation of the while loop in Fig. 5 is almost like we showed in Section 1.3, but we also assume well-formedness properties of the syntactic targets of the loop, and we check and assume some "modifies clauses" on the loop. In particular, we conjoin the postcondition contribution of the enclosing method's modifies clause to the loop invariant. The strengthened loop invariant makes it possible to prove the method's modifies clause at the end of the implementation body. In the definition in Fig. 5, we use *Spec* to denote the declared specification of the enclosing method, prepended with "**modifies this**.∗;" if the enclosing method is a constructor. Note that the expansion of $LoopMod$ produces a predicate that refers to the heap in three different states: $\mathcal{H}$ refers to the current value of the heap, which in this context means the value of the heap on loop-iteration boundaries; $oldHeap$ refers to the value of the heap upon entry to the loop, before any of its iterations; and $\mathbf{old}(\mathcal{H})$, which occurs in the antecedent and may arise in the expansion of $ModAllowed$, refers to the heap on entry to the enclosing method, which is where the method's modifies clause gets its meaning. Note also that, since $LoopMod$ becomes part of the loop invariant, we should in principle check it on entry to the loop, but since by construction it is idempotent, we can omit the check.

Finally, we define Md to return a list of syntactic targets, each of whose form is either "$\mathcal{H}$" or a type-id pair "$T\ x$". From such a list L, the translation function $StripTypes[\![L]\!]$ that we used above returns L with the type of each type-id pair removed

```
$Tr[\![\textbf{while}\ (E)\ \textbf{invariant}\ J;\ \{S\}]\!] =$
    $\{\ \textbf{var}\ oldHeap\colon [\textbf{ref}, \textbf{name}]\textbf{any}\ ;$
       $oldHeap := \mathcal{H}\ ;$
       $\textbf{assert}\ Df[\![J]\!]\ ;\ \textbf{assert}\ Tr[\![J]\!]\ ;$
       /* $\textbf{assert}\ LoopMod[\![Spec, oldHeap]\!]\ ;$ */
       $\textbf{havoc}\ StripTypes[\![Md[\![S]\!]]\!]\ ;$
       $\textbf{assume}\ TypeConstraint^*[\![Md[\![S]\!]]\!]\ ;$
       $\textbf{assume}\ successor(oldHeap, \mathcal{H})\ ;$
       $\textbf{assume}\ Df[\![J]\!] \wedge Tr[\![J]\!] \wedge LoopMod[\![Spec, oldHeap]\!]\ ;$
       $\textbf{assert}\ Df[\![E]\!]\ ;$
       $\{\ \textbf{assume}\ Tr[\![E]\!]$
          $Tr[\![\{S\}]\!]\ ;$
          $\textbf{assert}\ Df[\![J]\!]\ ;\ \textbf{assert}\ Tr[\![J]\!]\ ;\ \textbf{assert}\ LoopMod[\![Spec, oldHeap]\!]\ ;$
          $\textbf{assume false}$
       $[]\ \textbf{assume}\ \neg Tr[\![E]\!]$
       $\}$

$LoopMod[\![Spec, oldHeap]\!] =$
    $(\forall\ o\colon \textbf{ref},\ f\colon \textbf{name}\ \bullet$
       $o \neq \textbf{null} \wedge \textbf{old}(\mathcal{H})[o, alloc]\ \Rightarrow$
          $ModAllowed[\![Spec, o, f]\!]\ \vee$
          $\mathcal{H}[o, f] = oldHeap[o, f]\ )$
```

Figure 5. The translation of while loops.

(we omit the formal definition). In the following, E denotes an expression other than an allocation or call expression, and x denotes an identifier of a type X.

$$
\begin{array}{lcl}
Md[\![\{typeIds\ stmts\}]\!] & = & Md^*[\![stmts]\!] \setminus typeIds \\
Md[\![\textbf{assert}\ E;]\!] & = & \\
Md[\![\textbf{assume}\ E;]\!] & = & \\
Md[\![x =\ E;]\!] & = & X\ x \\
Md[\![x =\ E.m(EE);]\!] & = & X\ x,\ \mathcal{H} \\
Md[\![x =\ \textbf{new}\ C(EE);]\!] & = & X\ x,\ \mathcal{H} \\
Md[\![x =\ \textbf{new}\ T[E];]\!] & = & X\ x,\ \mathcal{H} \\
Md[\![E.f =\ F;]\!] & = & \mathcal{H} \\
Md[\![E[F] =\ G;]\!] & = & \mathcal{H} \\
Md[\![\textbf{if}\ (E)\ S\ \textbf{else}\ T]\!] & = & Md[\![S]\!], Md[\![T]\!] \\
Md[\![\textbf{while}\ (E)\ \textbf{invariant}\ J;\ \{S\}]\!] & = & Md[\![\{S\}]\!] \\
Md[\![\textbf{return}\ E;]\!] & = &
\end{array}
$$

2.3.4. Expressions

The well-definedness of an expression E is defined by translation function $Df[\![E]\!]$.

$$
\begin{array}{lcll}
\mathit{Df}\llbracket E \overset{\cdot}{\Longrightarrow} F\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge (\mathit{Tr}\llbracket E\rrbracket \Rightarrow \mathit{Df}\llbracket F\rrbracket) & \\
\mathit{Df}\llbracket E \mathrel{||} F\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge (\mathit{Tr}\llbracket E\rrbracket \vee \mathit{Df}\llbracket F\rrbracket) & \\
\mathit{Df}\llbracket E \mathbin{\&\&} F\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge (\mathit{Tr}\llbracket E\rrbracket \Rightarrow \mathit{Df}\llbracket F\rrbracket) & \\
\mathit{Df}\llbracket E \otimes F\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge \mathit{Df}\llbracket F\rrbracket & \text{with } \otimes \text{ being } +, -, \text{ or } * \\
\mathit{Df}\llbracket E \otimes F\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge \mathit{Df}\llbracket F\rrbracket \wedge \mathit{Tr}\llbracket F\rrbracket \neq 0 & \text{with } \otimes \text{ being } / \text{ or } \% \\
\mathit{Df}\llbracket \neg E\rrbracket & = & \mathit{Df}\llbracket E\rrbracket & \\
\mathit{Df}\llbracket \mathbf{this}\rrbracket & = & & \\
\mathit{Df}\llbracket \mathbf{result}\rrbracket & = & & \\
\mathit{Df}\llbracket \lambda\rrbracket & = & & \text{with } \lambda \text{ being any literal} \\
\mathit{Df}\llbracket x\rrbracket & = & & \\
\mathit{Df}\llbracket E.f\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge \mathit{Tr}\llbracket E\rrbracket \neq \mathbf{null} & \\
\mathit{Df}\llbracket E[F]\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge \mathit{Tr}\llbracket E\rrbracket \neq \mathbf{null} \wedge & \\
 & & \mathit{Df}\llbracket F\rrbracket \wedge 0 \leqslant \mathit{Tr}\llbracket F\rrbracket \wedge \mathit{Tr}\llbracket F\rrbracket < \mathit{length}(\mathit{Tr}\llbracket E\rrbracket) & \\
\mathit{Df}\llbracket E \mathbf{\ is\ } T\rrbracket & = & \mathit{Df}\llbracket E\rrbracket & \\
\mathit{Df}\llbracket (T)E\rrbracket & = & \mathit{Df}\llbracket E\rrbracket \wedge (\mathit{Tr}\llbracket E\rrbracket = \mathbf{null} \vee \mathit{type}(\mathit{Tr}\llbracket E\rrbracket) <: \mathit{Type}\llbracket T\rrbracket) & \\
\mathit{Df}\llbracket \mathbf{old}(E)\rrbracket & = & \mathbf{old}(\mathit{Df}\llbracket E\rrbracket) & \\
\mathit{Df}\llbracket \mathbf{fresh}(E)\rrbracket & = & \mathit{Df}\llbracket E\rrbracket &
\end{array}
$$

A Spec$^\flat$ expression E is translated into a corresponding BoogiePL$^\flat$ expression by $\mathit{Tr}\llbracket E\rrbracket$. In the following, we use $\oplus$ to denote the BoogiePL$^\flat$ operator corresponding to the Spec$^\flat$ operator $\otimes$; except for short-circuit operators (and type setting differences), $\otimes$ and $\oplus$ are the same.

$$
\begin{array}{lcll}
\mathit{Tr}\llbracket E \otimes F\rrbracket & = & \mathit{Tr}\llbracket E\rrbracket \oplus \mathit{Tr}\llbracket F\rrbracket & \\
\mathit{Tr}\llbracket \neg E\rrbracket & = & \neg \mathit{Tr}\llbracket E\rrbracket & \\
\mathit{Tr}\llbracket \mathbf{this}\rrbracket & = & \mathit{this} & \\
\mathit{Tr}\llbracket \mathbf{result}\rrbracket & = & \mathit{\_result} & \\
\mathit{Tr}\llbracket \lambda\rrbracket & = & \lambda & \\
\mathit{Tr}\llbracket x\rrbracket & = & x & \\
\mathit{Tr}\llbracket E.f\rrbracket & = & \mathcal{H}[\mathit{Tr}\llbracket E\rrbracket, f] & \text{with } f \text{ not } \mathit{Type} \text{ or } \mathit{Length} \\
\mathit{Tr}\llbracket E.\mathit{Type}\rrbracket & = & \mathit{type}(\mathit{Tr}\llbracket E\rrbracket) & \\
\mathit{Tr}\llbracket E.\mathit{Length}\rrbracket & = & \mathit{length}(\mathit{Tr}\llbracket E\rrbracket) & \\
\mathit{Tr}\llbracket E[F]\rrbracket & = & \mathit{Select}([\mathit{Tr}\llbracket E\rrbracket, \mathit{elems}], \mathit{Tr}\llbracket F\rrbracket) & \\
\mathit{Tr}\llbracket E \mathbf{\ is\ } T\rrbracket & = & \mathit{Tr}\llbracket E\rrbracket \neq \mathbf{null} \wedge \mathit{type}(\mathit{Tr}\llbracket E\rrbracket) <: \mathit{Type}\llbracket T\rrbracket & \\
\mathit{Tr}\llbracket (T)E\rrbracket & = & \mathit{Tr}\llbracket E\rrbracket & \\
\mathit{Tr}\llbracket \mathbf{old}(E)\rrbracket & = & \mathbf{old}(\mathit{Tr}\llbracket E\rrbracket) & \\
\mathit{Tr}\llbracket \mathbf{fresh}(E)\rrbracket & = & \neg\mathbf{old}(\mathcal{H})[\mathit{Tr}\llbracket E\rrbracket, \mathit{alloc}] &
\end{array}
$$

Note that we have special cases for the built-in fields Type and Length, which return the run-time type of an object and the length of an array, respectively.

2.4. Example

The Spec$^\flat$ program in Fig. 6 is translated into the BoogiePL$^\flat$ program in Fig. 7.

2.5. Summary

To verify object-oriented programs, one needs to define their semantics. One way to do that, which we have followed here, is to translate them into a simpler language with a

```
class C : object {
  int x;
  C(int y) { base(); x = y; }
  int M(int n)
    modifies x;
    ensures result = old(x);
  { int r = x; x = x/n; return r; }
}
```

Figure 6. An example Spec$^\flat$ program, declaring a class with an integer field, a constructor, and a method.

precisely defined semantics. The simpler language need not be just logical formulas; in fact, there is evidence that including imperative features, closer to the object-oriented language than to logical formulas, makes the encoding of the semantics easier to understand and to implement [53,4]. In the encoding of Spec$^\flat$ that we have presented in this section, we have decided on a storage model, axiomatized types and declarations, and prescribed the translation of statements and expressions. In this translation, we have addressed issues like behavioral subtyping and partiality of operations.

3. Invariants and Ownership

To prove the correctness of a method, it is usually necessary to know that its parameters, including the receiver parameter, reference well-formed data, that is, data that satisfy certain consistency conditions. Many consistency conditions can be described by *object invariants*.

This section introduces object invariant patterns and their proof obligations. Syntactically, an object invariant is declared as a class member:

$$\begin{array}{lcl} \textit{Member} & ::= & \ldots \\ & | & \textit{Invariant} \end{array}$$

The declaration gives an invariant for the enclosing class, written in terms of an arbitrary object denoted by the keyword `this`. We start in Section 3.0 with an example that highlights the central question of where invariants hold. In Section 3.1, we look at *intra-object invariants*, which express semantic constraints on the fields of each object. In Section 3.2, we look at an important form of *inter-object invariants*, which express properties of linked objects, that is, of objects that refer to each other. *Aliasing*, the interaction between object references, complicates the handling of inter-object invariants. We employ an *ownership regime* to control the impact of changes among objects. In Section 3.3, we discuss inheritance and dynamic dispatch.

3.0. Where Do Invariants Hold?

The quintessential idea of object invariants is that an object satisfies its invariant whenever no constructor or method of the object is active (*cf.* [59]). With some restrictions, this idea can be realized by checking that the constructor establishes the object invariant and that every non-constructor method of the class preserves the invariant.

```
const $C$: name ;
axiom $superclass(C) = object$ ;
const $C°x$: name ;
axiom $fieldType(C°x) = \_int$ ;
procedure $C°C(this$: ref, $y$: int) returns ( ) ;
  modifies $\mathcal{H}$ ;
  ensures $(\forall\, o$: ref, $f$: name $\bullet$ $o \neq$ null $\wedge$ old$(\mathcal{H})[o, alloc] \Rightarrow$
    $o =$ old$(this) \vee \mathcal{H}[o,f] =$ old$(\mathcal{H})[o,f]$ ) ;
implementation $C°C(this$: ref, $y$: int) returns ( )
  { assume $wellFormed(\mathcal{H})$ ; assume $this \neq$ null ;
    assume $this =$ null $\vee$ $(\mathcal{H}[this, alloc] \wedge type(this) <: C$ ;
    assume $\mathcal{H}[this, C°x] = 0$ ;
    { var $oldHeap$: [ref, name]any ;                    /* base(); */
      assert $this \neq$ null ;
      $oldHeap := \mathcal{H}$ ;
      call $object°object(this)$ ;
      assume $successor(oldHeap, \mathcal{H})$
    } ;
    assert $this \neq$ null ;                             /* this.$x$ = $y$; */
    $\mathcal{H}[this, C°x] := y$
  }
procedure $C°M(this$: ref, $n$: int) returns ($\_result$: int) ;
  ensures $\_result =$ old$(\mathcal{H}[this, C°x])$ ;
  modifies $\mathcal{H}$ ;
  ensures $(\forall\, o$: ref, $f$: name $\bullet$ $o \neq$ null $\wedge$ old$(\mathcal{H})[o, alloc] \Rightarrow$
    $(o =$ old$(this) \wedge f = C°x) \vee \mathcal{H}[o,f] =$ old$(\mathcal{H})[o,f]$ ) ;
implementation $C°M(this$: ref, $n$: int) returns ($\_result$: int)
  { assume $wellFormed(\mathcal{H})$ ; assume $this \neq$ null ;
    assume $this =$ null $\vee$ $(\mathcal{H}[this, alloc] \wedge type(this) <: C)$ ;
    { var $r$: int ;                                        /* int $r$; */
      assert $this \neq$ null ;                             /* $r$ = this.$x$; */
      $r := \mathcal{H}[this, C°x]$ ;
      assert $this \neq$ null ;                   /* this.$x$ = this.$x/n$; */
      assert $this \neq$ null $\wedge$ $n \neq 0$ ;
      $\mathcal{H}[this, C°x] := \mathcal{H}[this, C°x]/n$ ;
      $\_result := r$                                       /* return $r$; */
    }
  }
```

Figure 7. The BoogiePL$^\flat$ translation of the Spec$^\flat$ program in Fig. 6. This figure omits the prelude of the translation, which is described in Section 2.3.0 the same for all translated Spec$^\flat$ programs.

Class $Subject$ in Fig. 8 illustrates the idea. Its invariant constrains st to be non-zero. The constructor sets st to 1, establishing the invariant, and the other methods leave st with a non-zero value, maintaining the invariant. Given that **this** satisfies its object invariant on entry to method Get, one can prove that the division in the body of Get is defined.

```
class Subject {
  Observer obs;
  int st;
  invariant st ≠ 0;

  Subject(Observer o)
  { st = 1; obs = o; }

  void Update(int y)
    requires y ≠ 0;
    modifies this.∗, obs.∗;
  { st = 0;
    if (obs ≠ null)
      obs.Notify(this);
    st = y;
  }

  int Get()
  { return 1000/st; }
}
```

```
class Observer {
  int cache;

  void Notify(Subject s)
    requires s ≠ null;
    modifies cache;
  { cache = s.Get(); }
}


class Program {
  void Main() {
    Observer o = new Observer();
    Subject s = new Subject(o);
    s.Update(5);
  }
}
```

Figure 8. An example program showing the interaction between two objects, a subject and an observer. Without the invariant declaration, the formalization of Spec$^\flat$ in Section 2 will report one error in this program, namely a division-by-zero error in method *Get*. The **invariant** in *Subject* declares an intention to keep *st* non-zero, but exactly when is the invariant supposed to hold?

The interaction between class *Subject* and class *Observer* illustrates a problem with the basic realization of the quintessential idea of object invariants. Before it calls *Notify* on the observer, method *Update* changes *st* to 0, temporarily violating the invariant. But *Notify* then causes control to *reenter* the subject, which leads to a division-by-zero error. Evidently, there is more to verifying and using object invariants than checking them at the end of methods.

3.1. Intra-object Invariants

We introduce a programming discipline, a specification and verification *methodology*, that makes it possible describe and enforce the program's intended design regarding reentrancy and, more generally, regarding object invariants. The methodology explicitly keeps track of when an invariant is known to hold [3].

For every object and array, we introduce a ghost field *inv*, which can take on the values *valid* and *mutable*. The intended meaning of these two states is that the object invariant holds of objects in the *valid* state, but may or may not hold of objects in the *mutable* state. Newly allocated objects are mutable; that is, on entry to a constructor, the object to be constructed is mutable.

For any class T and object o of class T, we let $Inv_T[\![o]\!]$ denote the invariant declared in class T applied to object o. For an array type T, $Inv_T[\![o]\!]$ is just $\mathbf{true}$.

In order to be able to do modular verification with object invariants, it is necessary to restrict what an object invariant can depend on. For intra-object invariants, we say an invariant (declaration) is *admissible* if it only refers to fields of the object, that is, if each of its field-select subexpressions are of the form **this**.f for some field f (where, as usual, "**this**." can be implicit).

We can now formalize the connection between the inv field and admissible invariants:

Program Invariant 0 *If the invariant a class C is admissible, then*

$$(\forall\, o \bullet\; o.inv = valid \;\Rightarrow\; Inv_C[\![o]\!]\;)$$

where the quantification ranges over non-null, allocated objects of type C, is a program invariant*, that is, it holds in every reachable program state.*

To ensure this program invariant, the methodology restricts updates of inv (which occurs in the antecedent of the quantified formula) and updates of other fields of the object (which may occur in the consequent).

Updates of inv are restricted to two new operations, written **unpack** o; and **pack** o; for any object-valued expression o. The idea is that these operations delineate where an object is mutable: **unpack** o; makes o mutable, and **pack** o; makes o valid after first checking that $Inv_C[\![o]\!]$ holds.

Other field updates are restricted to mutable objects only. That is, we introduce $o.inv = mutable$ as a new precondition of each field update statement $o.f = E$;.

Applying this methodology to the subject-observer example, we change the code in Fig. 8 as shown in Fig. 9.

Defaults and Shorthands This methodology for object invariants uses unpack and pack operations to change inv, and uses pre- and postconditions to specify the value of inv on method boundaries. As is exemplified in Fig. 9, these operations and specifications tend to be used in a highly stylized fashion: the constructor ends with a pack operation, state changes in other methods are bracketed by a unpack and pack, the constructor postcondition says that the object is valid, and the precondition of non-constructor methods requires the object to be valid. To simplify the program text, we introduce some defaults.

First, we add a **pack this**; operation at the end of every constructor.

Second, we introduce a structured statement **expose** (o) $\{S\}$ to stand for the common sequence

$$\textbf{unpack}\ o;\ \{S\}\ \textbf{pack}\ o;$$

where we assume S does not change o. In fact, we only add the **expose** statement, not the **unpack** and **pack** operations, to the Spec$^\flat$ language syntax:

$$\begin{array}{lll} Stmt & ::= & \ldots \\ & | & \textbf{expose}\ (\ Expr\ )\ Block \end{array}$$

where the type of the expression must be a non-**object** reference type.

Third, for every reference-valued method parameter p, including **this** unless the method is a constructor, we add the default precondition $p.inv = valid$.

Fourth, we add the default postcondition $inv = valid$ to constructors.

Fifth, because it tends to be a more common specification pattern, we change the definition of $o.*$ in modifies clauses to exclude the inv field (*cf.* page 26):

```
class Subject {
   Observer obs;
   int st;
   invariant st ≠ 0;

   Subject(Observer o)
      ensures inv = valid;
   {  st = 1; obs = o;
      pack this;
   }

   void Update(int y)
      requires inv = valid;
      requires y ≠ 0;
      modifies this.∗, obs.∗;
   {  unpack this;
      st = y;
      pack this;
      if (obs ≠ null)
         obs.Notify(this);
   }

   int Get()
      requires inv = valid;
   {  return 1000/st; }
}
```

```
class Observer {
   int cache;

   void Notify(Subject s)
      requires inv = valid;
      requires s ≠ null ∧ s.inv = valid;
      modifies cache;
   {  unpack this;
      cache = s.Get();
      pack this;
   }
}
```

Figure 9. The *Subject* and *Observer* classes from Fig. 8, but here using *inv*, **unpack**, and **pack**. The methodology also forced us to change the implementation of *Update*, because there is no way to insert unpack and pack operations in Fig. 8 to live up to the three requirements of: (0) *st* can be updated only if the subject is mutable, (1) the precondition of *Notify* requires that the subject be valid, and (2) the *Subject* invariant must hold at the time of a pack operation. One remaining verification problem, which we address in Section 3.2, is how to make sure the observer *obs* is valid when calling *Notify*.

```
$ModAllowed[\![Spec, o, f]\!] =$
   for each “$\textbf{modifies}\ W$;” in $Spec$ do
      for each “$desig\ suffix$” in $W$ do
         case $suffix$ of
            $(.*):\quad (o = \textbf{old}(Tr[\![desig]\!]) \wedge f \neq inv) \vee$
            $\ldots$
```

Using these defaults and shorthands, we can simplify the *Update* method of the *Subject* class as follows:

```
void $Update$(int $y$)
  requires $y \neq 0$;
  modifies this.$*$, $obs.*$;
{ expose (this) { $st =\ y$; }
  if ($obs \neq$ null)
    $obs.Notify$(this);
}
```

Translation We change the translation to generate proof obligations that guarantee Program Invariant 0.

We start by changing the translation of how objects and references come into being. For objects, we add an assumption in the constructor, saying that the new object starts in a mutable state (*cf.* page 25):

```
$Tr[\![\mathbb{C}\ (Args)\ Spec\ Body]\!] =$
    procedure $\mathbb{C}°\mathbb{C}$ ...
    implementation $\mathbb{C}°\mathbb{C}$ ($Tr^*[\![\mathbb{C}\ this, Args]\!]$) returns ( )
      { assume $wellFormed(\mathcal{H})$ ;
        assume $this \neq$ null $\wedge\ \mathcal{H}[this, inv] = mutable$ ;
        assume $TypeConstraint^*[\![\mathbb{C}\ this, Args]\!]$ ;
        for each field "$F\ f$;" defined in $\mathbb{C}$ do
          $\mathcal{H}[this, \mathbb{C}°f] := Zero[\![F]\!]$ ;
        $Tr[\![Body]\!]$
      }
```

Remember that our defaults and shorthands add a **pack this**; operation at the end of *Body* (during normalization).

For arrays, we make newly allocated arrays appear in the valid state (note that array types themselves do not have any object invariants) (*cf.* page 28):

```
$Tr[\![x =\ $new$\ T[E];]\!] =$
    { var $o$: ref ;
      ...
      $\mathcal{H}[o, inv] := valid$ ; $\mathcal{H}[o, alloc] :=$ true ;
      $x := o$
    }
```

We add an extra precondition to field and array update (*cf.* page 27). For brevity, here and in the rest of this paper, we assume that expressions like o, a, i, and e in the following translations are local variables; in general, we would first apply normalization and then use $Df[\![\cdot]\!]$ and $Tr[\![\cdot]\!]$, as in Section 2.3.3.

```
$Tr[\![o.f =\ e;]\!] =$
    assert $o \neq$ null $\wedge\ \mathcal{H}[o, inv] = mutable$ ;
    $\mathcal{H}[o, f] := e$
```

```
$Tr[\![a[i] = e;]\!] =$
    $\textbf{assert } a \neq \textbf{null} \wedge \mathcal{H}[a, inv] = mutable$ ;
    $\textbf{assert } 0 \leqslant i \wedge i < length(a)$ ;
    $\textbf{assert } type(e) <: elemType(type(a))$ ;
    $\mathcal{H}[a, elems] := Store(\mathcal{H}[a, elems], i, e)$
```

Finally, we define the unpack and pack operations as follows, where o has static type T.

```
$Tr[\![\textbf{unpack } o;]\!] =$
    $\textbf{assert } o \neq \textbf{null} \wedge \mathcal{H}[o, inv] = valid$ ;
    $\mathcal{H}[o, inv] := mutable$

$Tr[\![\textbf{pack } o;]\!] =$
    $\textbf{assert } o \neq \textbf{null} \wedge \mathcal{H}[o, inv] = mutable$ ;
    $\textbf{assert } Df[\![Inv(o)]\!] \wedge Tr[\![Inv(o)]\!]$ ;
    $\mathcal{H}[o, inv] := valid$
```

3.2. Inter-object Invariants

An object invariant can span several objects. Suppose object o refers to object c in its invariant; then changing c might invalidate the invariant of o. There are several strategies for dealing with this situation [61,3,47,7,67,38,60]. In this section, we deal with the common situation where accesses to c are controlled by o. We say that c is part of the *representation* of o and that o is the *owner* of c. We do not assume that c knows its owner, and thus we handle the important case where c is an instance of a class defined in a library. These object invariants are called *ownership-based invariants*, because they use the ownership structure of the heap in the definition of which invariants are admissible.

Suppose we want to design a priority queue of tasks, implemented via a sorted singly-linked list of nodes. Figure 10 shows a possible implementation. For a proper working of the priority queue, we design the list so that it is strictly increasing, that is, we need the following invariant for class $Node$:

$$\textbf{invariant } next \neq \textbf{null} \Rightarrow prio < next.prio; \tag{2}$$

But how can we deal with the fact that the modification of one node's priority might break the invariant in the previous node?

We establish a *hierarchical ownership* relationship on objects. We use ownership to control that, outside an object's invariant, the fields of the object can be mentioned only in the invariants of its transitive owners. The methodology also enforces that when an object is mutable, so are its transitive owners. Consequently, when the fields of an object are changed, it can only violate the invariants of owners, but those owners are mutable, which means the owners are in a state when the invariants are allowed to be violated.

For our list example, we let each node own its successor. To follow the methodology, we must then arrange to expose all predecessors before modify a node.

We encode this ownership regime as follows:

- We extend the domain of the inv field to $\{mutable, valid, committed\}$. We say an object is *committed* if its invariant is known to hold and its owner is not mutable.

```
class PriorityQueue {
  Node hd;

  void Insert(object t, int p)
    requires t ≠ null;
    modifies hd;
  { expose (this) {
      if (hd ≠ null)
        hd = hd.Inject(t, p);
      else
        hd =
          new Node(t, p, null);
    }
  }

  void DeleteMin()
    modifies hd;
  { expose (this) {
      if (hd ≠ null)
        hd = hd.next;
    }
  }

  object Min() {
    if (hd ≠ null)
      return hd.task;
    else
      return null;
  }
}
```

```
class Node {
  object task; int prio;
  Node next;

  Node(object t, int p, Node n) {
    task = t;
    prio = p;
    next = n;
  }

  Node Inject(object t, int p)
    modifies next;
  {
    if (p < prio)
      return new Node(t, p, this);
    expose (this) {
      if (next = null)
        next = new Node(t, p, null);
      else
        next = next.Inject(t, p);
    }
    return this;
  }
}
```

Figure 10. An implementation of a priority queue. Formal parameter t denotes a task and p denotes a priority. This version of the implementation points out three verification problems. First, the object invariant that nodes of the priority queue are sorted needs to mention more than one object (namely, $\mathbf{this}.next.prio$), and that is not allowed by the admissibility condition in Section 3.1. Second, how do we know that the receiver objects of the two calls to *Inject* are valid? Third, method *Insert* might change hd and the $next$ field of an unbounded number of *Node* objects, but all of these modifications could not possibly be listed explicitly in the modifies clause of *Insert*.

- We introduce a field modifier **rep**, which specifies that a field refers to a representation object.

  $$FieldModifier \quad ::= \quad \mathbf{rep}$$

  The **rep** modifier is allowed on fields having reference types.

We can now formalize the meaning of **rep** fields, which establishes an additional property:

Program Invariant 1 *For any* **rep** *field f declared in a class C,*

```
class Subject {
  rep Observer obs;
  int st;
  invariant st ≠ 0;

  Subject(Observer o)
    modifies o.inv;
  { st = 1; obs = o; }

  int Get()
  { return 1000/st; }

void Update(int y)
  requires y ≠ 0;
  modifies this.∗, obs.∗;
{ Observer tmp = obs;
  expose (this) { st = y; obs = null; }
  if (tmp ≠ null)
    tmp.Notify(this);
  expose (this) { obs = tmp; }
}
```

Figure 11. The *Subject* class, updated from Fig. 9. Here, the observer is captured by the subject's constructor to claim ownership of it. The *Update* method temporarily disentangles that ownership relation in order to call *Notify* with two valid objects.

$$(\forall\, o \bullet\ o.inv \neq mutable \ \Rightarrow\ o.f = \mathbf{null} \vee o.f.inv \neq mutable\ )$$

where the quantification ranges over non-null, allocated objects of type C, is a program invariant.

Admissible invariants for our ownership regime are now restricted as follows: An object o may depend only on the fields of o and the fields of objects transitively owned by o. We check this restriction syntactically, allowing an invariant to mention a field-select expression $\mathbf{this}.a.b.\cdots.f.x$ only if $a, b, \ldots, f$ are declared to be **rep** fields. The object invariant (2) of our priority queue example in Fig. 10 has this form, and is thus admissible.

Using our refined methodology, we can solve two of the verification problems in Fig. 10. We declare hd and $next$ as **rep** fields, which make invariant declaration (2) admissible. Program Invariant 1 and the new definition of unpack let us call *Inject*, because they establish that the receiver is valid.

Using our refined methodology, we can also solve the last verification problem with the subject-observer example in Fig. 9. The verifiable code is shown in Fig. 11.

The problem in Fig. 9 was that, on entry to *Update*, we know nothing at all about the validity of *obs*, but we need to know that it is valid when we invoke its *Notify* method. We do know that the subject is valid on entry to *Update*. To entangle the validity of the observer with the validity of the subject, we make the former a representation object of the latter, see the **rep** keyword in Fig. 11.

We now need to admit to "capturing" the valid observer parameter o in the *Subject* constructor. We do that by listing $o.inv$ in the modifies clause of that constructor.

Finally, when the observer is owned by the subject, it is not possible to pass both of them as valid objects to *Notify*. Thus, we temporarily disentangle them, as shown in Fig. 11. While this does make the program verify, it is clumsy. A better solution would be to make the subject and observer *peers*, which means they have the same owner. This solution is explained in detail elsewhere [61,47,54].

Translation We change the translation of unpack and pack operations to reflect changes in the committed-status of objects (*cf.* page 38). For a local variable o of static type T:

```
$Tr[\![\textbf{unpack}\ o;]\!] =$
    $\textbf{assert}\ o \neq \textbf{null} \land \mathcal{H}[o, inv] = valid$ ;
    $\mathcal{H}[o, inv] := mutable$ ;
    for each field “$\textbf{rep}\ F\ f$;” defined in $T$ do
      {  $\textbf{assume}\ \mathcal{H}[o,f] \neq \textbf{null}$ ; $\mathcal{H}[\mathcal{H}[o,f], inv] := valid$
      []  $\textbf{assume}\ \mathcal{H}[o,f] = \textbf{null}$
      }

$Tr[\![\textbf{pack}\ o;]\!] =$
    $\textbf{assert}\ o \neq \textbf{null} \land \mathcal{H}[o, inv] = mutable$ ;
    $\textbf{assert}\ Df[\![Inv_T[\![o]\!]]\!] \land Tr[\![Inv_T[\![o]\!]]\!]$ ;
    for each field “$\textbf{rep}\ F\ f$;” defined in $T$ do
      $\textbf{assert}\ \mathcal{H}[o,f] = \textbf{null} \lor \mathcal{H}[\mathcal{H}[o,f], inv] = valid$ ;
    for each field “$\textbf{rep}\ F\ f$;” defined in $T$ do
      {  $\textbf{assume}\ \mathcal{H}[o,f] \neq \textbf{null}$;  $\mathcal{H}[\mathcal{H}[o,f], inv] := committed$
      []  $\textbf{assume}\ \mathcal{H}[o,f] = \textbf{null}$
      } ;
    $\mathcal{H}[o, inv] := valid$
```

Method Framing Revisited With the meaning of modifies clauses defined in Section 2.3.2, methods *Insert* and *Inject* in Fig. 10 do not verify. That definition insisted on that every non-new object with a modified field be mentioned explicitly in the modifies clause. But that's absurd. We need some form of abstraction in our modifies clauses. Representation objects establish a natural abstraction boundary that we can use.

With ownership, only owners are allowed to have invariants that depend on representation objects. Since representation objects are implementation details, code should not depend on the exact values of committed objects. Therefore, we now state a more relaxed meaning of modifies clauses: committed objects are allowed to be changed without explicitly being mentioned in modifies clauses.

We change the computing of the postcondition for a Spec$^\flat$ modifies clause (*cf.* page 26) by adding another disjunct:

```
$TrMod[\![Spec]\!] =$
    $\textbf{modifies}\ \mathcal{H}$ ;
    $\textbf{ensures}\ (\forall\, o\!:\!\textbf{ref},\ f\!:\!\textbf{name} \bullet$
      $o \neq \textbf{null} \land \textbf{old}(\mathcal{H})[o, alloc] \Rightarrow$
        $ModAllowed[\![Spec, o, f]\!] \lor$
        $\mathcal{H}[x, inv] = committed \lor$
        $\mathcal{H}[o,f] = \textbf{old}(\mathcal{H})[o,f]\ )$
```

With the relaxed meaning, the *Insert* method in Fig. 10 does not need to mention any *Node* object in its modifies clause. Likewise, method *Inject* does not need to explicitly mention the modifications of its successor nodes. Since **this** is valid on entry, methods must still announce modifications of fields of **this**, which is why we wrote the modifies clauses of *Insert* and *Inject* in Fig. 10 the way we did. This solves the third verification problem in Fig. 10.

3.3. Inheritance and Invariants

Inheritance and virtually dispatched calls are key features of object-oriented programming languages. To discuss their verification problems in more detail, let us introduce the concept of a *class frame*. We say: Each subclass defines one class frame, consisting of its instance variables. Applied to our $Cell$ example in Fig. 3, we see that a $Cell$ has two frames: the $\mathbf{object}$ frame, and the $Cell$ frame. The latter contains $Cell$'s only instance field, x. A $BackupCell$ also has a third frame, containing $BackupCell$'s only instance field, b. Single inheritance thus results in a sequence of frames.

Let us now look at the problems involved in virtual calls. First, we see that virtual calls lead to classical callback scenarios. For instance, let c be a $BackupCell$; then a call $c.IncBy(3)$ first enters its $Cell$ frame, which when evaluating $\mathbf{this}.Get()$ reenters its $Cell$ frame; next, it evaluates $Set(\ldots)$, which enters the $BackupCell$ frame, which through a base call reenters the $Cell$ frame. How can we maintain the invariants in $Cell$ and/or $BackupCell$ under such dynamic control flow?

For verification of invariants in the context of inheritance, we let each class frame declare its own invariant. The invariants from different frames of an object are enforced separately. An invariant declared in a class T is admissible if every field-select expression has the form $\mathbf{this}.a.b.\cdots.f.x$ where, as before, $a, b, \ldots, f$ are declared to be $\mathbf{rep}$ fields, and the first field (a or x) is declared in T or a superclass thereof.

Here is an example:

```
class Cell {                  class BackupCell : Cell {
  int x;                        int b;
  invariant 0 ⩽ x;              invariant b ⩽ x;
  ...                           ...
```

The direct superclass frame of $BackupCell$, namely $Cell$, can be viewed as a rep "object", or rep frame, of $BackupCell$ objects. This matches the rep model nicely, since there is only one conceptual pointer from the subclass frame to its immediate superclass frame. Note that, just as for rep objects, the invariant of the "owner" $BackupCell$ is allowed to mention fields declared in rep frames (*i.e.*, superclasses).

As in the rep model, we require that all calls (or, more precisely, all $\mathbf{expose}$ operations) on the rep frame go via calls to its owner, *i.e.*, its subclass frame. Consequently, most methods need to be virtual; you override them in each subclass explicitly, and you always use base calls to transfer control into the superclass, if needed. Embedding a base call in $\mathbf{expose}$ blocks causes an object's frames to be exposed in a stack-like fashion.

We could introduce an inv field for each frame, but since virtual calls and expose statements are used in a highly styled fashion, we can use fewer ghost variables by letting the inv field of an object refer to the most derived frame of the object that is valid. That is, $o.inv <: T$ means that o is valid for frame T and all its superclass frame. The properties $o.inv = valid$ and $o.inv = mutable$ that we introduced in Section 3.1 are now represented as $o.inv = type(o)$ and $o.inv = \mathbf{object}$, respectively, assuming that $\mathbf{object}$ has no invariant.

An object can become committed only when all its class frames are valid. To encode the committed state of an object, we introduce a fictitious type named $Committed$, which is modeled as a subtype of all types in the program. Thus, $o.inv = Committed$ means the the object o is committed, previously written as $o.inv = committed$.

With our new encoding of inv, the following restates the previous Program Invariants 0 and 1 in the context of subclasses:

Program Invariant 2 *For any class C with an admissible invariant,*

$$(\forall\, o \bullet\ type(o) <: C \wedge o.inv <: C \Rightarrow Inv_C[\![o]\!]\ )$$

and for any $\mathbf{rep}$ *field f declared in C,*

$$(\forall\, o \bullet\ o.inv <: C \Rightarrow o.f = \mathbf{null} \vee o.f.inv = Committed\ )$$

where the quantifications range over non-null, allocated objects, are program invariants.

Translation The following definitions of unpack and pack take the new representation into account. Let T be the static type of o, and let S be the immediate superclass of T or, if T is an array type, let S be $\mathbf{object}$; then (*cf.* page 41):

```
$Tr[\![\mathbf{unpack}\ o;]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, inv] = T$ ;
    $\mathcal{H}[o, inv] := S$ ;
    for each field "$\mathbf{rep}\ F\ f$;" defined in $T$ do
      {  $\mathbf{assume}\ \mathcal{H}[o,f] \neq \mathbf{null}$ ; $\mathcal{H}[\mathcal{H}[o,f], inv] := type(\mathcal{H}[o,f])$
      []  $\mathbf{assume}\ \mathcal{H}[o,f] = \mathbf{null}$
      }

$Tr[\![\mathbf{pack}\ o;]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, inv] = S$ ;
    $\mathbf{assert}\ Df[\![Inv_T[\![o]\!]]\!] \wedge Tr[\![Inv_T[\![o]\!]]\!]$ ;
    for each field "$\mathbf{rep}\ F\ f$;" defined in $T$ do
      $\mathbf{assert}\ \mathcal{H}[o,f] = \mathbf{null} \vee \mathcal{H}[\mathcal{H}[o,f], inv] = type(\mathcal{H}[o,f])$ ;
    for each field "$\mathbf{rep}\ F\ f$;" defined in $T$ do
      {  $\mathbf{assume}\ \mathcal{H}[o,f] \neq \mathbf{null}$;  $\mathcal{H}[\mathcal{H}[o,f], inv] := Committed$
      []  $\mathbf{assume}\ \mathcal{H}[o,f] = \mathbf{null}$
      } ;
    $\mathcal{H}[o, inv] := T$
```

Let f be a field declared in a class C; then field update is redefined as follows (*cf.* page 37):

```
$Tr[\![o.f = e;]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \neg(\mathcal{H}[o, inv] <: C)$ ;
    $\mathcal{H}[o,f] := e$
```

and array update is defined as follows:

```
$Tr[\![a[i] = e;]\!] =$
    $\mathbf{assert}\ a \neq \mathbf{null} \wedge \mathcal{H}[a, inv] = object$ ;
    $\mathbf{assert}\ 0 \leqslant i \wedge i < length(a)$ ;
    $\mathbf{assert}\ type(e) <: elemType(type(a))$ ;
    $\mathcal{H}[a, elems] := Store(\mathcal{H}[a, elems], i, e)$
```

So that it can be used when proving programs, we add Program Invariant 2 as axioms.

$$\begin{array}{l}\textbf{axiom}\ (\forall\, h\colon [\textbf{ref}, \textbf{name}]\textbf{any},\ o\colon \textbf{ref}\ \bullet \\ \quad \mathit{wellFormed}(h) \wedge o \neq \textbf{null} \wedge h[o, \mathit{alloc}] \Rightarrow \\ \qquad \mathit{type}(o) <: C \wedge h[o, \mathit{inv}] <: C \Rightarrow \mathit{Df}[\![\mathit{Inv}_C[\![o]\!]]\!] \wedge \mathit{Tr}[\![\mathit{Inv}_C[\![o]\!]]\!]\ )\ ;\end{array}$$

$$\begin{array}{l}\textbf{axiom}\ (\forall\, h\colon [\textbf{ref}, \textbf{name}]\textbf{any},\ o\colon \textbf{ref}\ \bullet \\ \quad \mathit{wellFormed}(h) \wedge o \neq \textbf{null} \wedge h[o, \mathit{alloc}] \Rightarrow \\ \qquad h[o, \mathit{inv}] <: C \Rightarrow h[o, f] = \textbf{null} \vee h[h[o, f], \mathit{inv}] = \mathit{Committed}\ )\ ;\end{array}$$

for every class and applicable field. In these axioms, $\mathit{Inv}_C[\![o]\!]$ is expanded to the expression declared to be the invariant of class C, with o replacing occurrences of **this**.

Finally, we add an axiom that says that *Committed* is a subtype of all types.

$$\textbf{axiom}\ (\forall\, T\colon \textbf{name}\ \bullet\ \mathit{Committed} <: T\ )\ ;$$

Method Preconditions Revisited Exposing an object frame by frame introduces another problem: for every class T, the definition or override of a virtual method m in class T is going to unpack the object, and therefore it needs the precondition $\mathit{inv} = T$. For example, to verify the example in Fig. 3 using our methodology, the $\mathit{Cell}°\mathit{Set}$ and $\mathit{BackupCell}°\mathit{Set}$ method implementations must expose the object for the *Cell* and *BackupCell* frames, respectively, before modifying the fields x and b. To meet with the preconditions of such **expose** statements, $\mathit{Cell}°\mathit{Set}$ would need a precondition of $\mathit{inv} = \mathit{Cell}$ and $\mathit{BackupCell}°\mathit{Set}$ would need a precondition of $\mathit{inv} = \mathit{BackupCell}$, but a virtual method and its overrides cannot arbitrarily change the method precondition! What condition would we check at call sites?

For call sites that invoke the a virtual method m by $\mathbf{base}.m$, we can check different preconditions at different call sites, because base calls are statically bound. That is, calling $\mathbf{base}.m$ invokes a particular implementation, so we can arrange to verify, at the call site, the particular precondition required by that implementation. For a dynamically dispatched call to m, we cannot statically decide which overridden method will be executed, yet we need to verify, at the call site, that the precondition required by the invoked override holds. By demanding that every class override all inherited virtual methods, the condition to be verified at a dynamically dispatched call to $o.m$ is $\mathit{inv} = \mathit{type}(o)$.

To support these scenarios where different implementations of a method need different preconditions, we introduce a *polymorphic invariant level*, written as $\mathit{inv} = \star$:

$$\begin{array}{lcl}\mathit{Literal} & ::= & \ldots \\ & | & \star\end{array}$$

where $\star$ can appear only in the specifications of virtual methods, and the type of $\star$ is that of a run-time type. The idea is that the definition of a method m writes $\mathit{inv} = \star$ in its precondition. For an implementation given in a class T, $\mathit{inv} = \star$ means $\mathit{inv} = T$, and for a dynamically dispatched call to $o.m$, it means $\mathit{inv} = \mathit{type}(o)$.

Our *Cell* and *BackupCell* classes can now be specified, implemented, and verified as shown in Fig. 12. Note that *IncBy* is a non-virtual method. With its given specification, its implementation cannot directly perform any update, because it cannot do the necessary **expose**. However, the implementation can still call virtual methods that will expose the object and modify its state.

```
class Cell {
  int x;
  invariant 0 ⩽ x;

  Cell(int i)
    ensures inv == Cell;
  { x = i; }

  virtual int Get()
    requires inv = ⋆;
  { return x; }

  virtual void Set(int i)
    requires inv = ⋆;
  { expose (this) { x = i; } }

  void IncBy(int i)
    requires inv == Type;
  { int t = Get(); Set(t + i); }
}
```

```
class BackupCell : Cell {
  int b;
  invariant b ⩽ x;

  BackupCell(int i)
    ensures inv == BackupCell;
  { base(i); b = i; }

  override int Get()
  { int g;
    expose (this)
     { g = base.Get(); }
    return g;
  }

  override void Set(int i)
  { expose (this)
     { b = x; base.Set(i); }
  }

  virtual int GetBackup()
    requires inv = ⋆;
  { return b; }

  virtual void Rollback()
    requires inv = ⋆;
  { x = b; }
}
```

Figure 12. The *Cell* and *BackupCell* classes from Fig. 3 with polymorphic invariant levels. For brevity, we omit all other contracts.

Defaults and Shorthands To remove the burden that subclasses must override all virtual methods, our normalization will, for any non-overridden inherited method, insert an override whose body exposes the object and calls the base implementation of the method. For example, for a subclass of *Cell* that does not explicitly override methods *Set* and *Get*, normalization will insert:

```
override void Set(int i)
{ expose (this) { base.Set(i); } }
override int Get()
{ int g; expose (this) { g = Get(); } return g; }
```

Since the value of *inv* is now a type, no longer a boolean, we must make some adjustments in our default method specifications (*cf.* the discussion on Defaults and Shorthands in Section 3.1). For a constructor in a class C, we use the default postcondition

```
ensures inv == C;
```

For every virtual method, we use the default precondition

$$\textbf{requires } inv = \star;$$

Third, for every reference-valued method parameter p, including **this** unless the method is a constructor or virtual method, we add the default precondition

$$\textbf{requires } p.inv = p.Type;$$

Translation We adapt the translation to BoogiePL$^\flat$ as follows. For each definition or override of a virtual method with a polymorphic invariant level, we generate two BoogiePL$^\flat$ procedure declarations (*cf.* page 24):

$Tr[\![MethodModifier\ T\ m\ (Args)\ \ Spec\ Body]\!] =$
  $\textbf{procedure}\ \mathbb{C}{\circ}Virtual{\circ}m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \textbf{returns}\ (Tr[\![T\ \_result]\!]);$
    as the previous translation into $\mathbb{C}{\circ}m$, but replacing $\star$ with $type(\textbf{this})$
  $\textbf{procedure}\ \mathbb{C}{\circ}m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \textbf{returns}\ (Tr[\![T\ \_result]\!])\ ;$
    as before, but replacing $\star$ with $\mathbb{C}$
  $\textbf{implementation}\ \mathbb{C}{\circ}m\ (Tr^*[\![\mathbb{C}\ this, Args]\!])\ \textbf{returns}\ (Tr[\![T\ \_result]\!])$
    as before

For a call $o.m(\ldots)$ where o has static type T and m is a virtual method, we use $T{\circ}Virtual{\circ}m$ in the translation. For a call $\textbf{base}.m(\ldots)$ in a class whose superclass is T, we use $T{\circ}m$ in the translation. The implementation is given for $\mathbb{C}{\circ}m$. Our translation does not give any implementation to $\mathbb{C}{\circ}Virtual{\circ}m$; intuitively, this implementation is provided by the runtime system, which performs the dynamic dispatch by a case split over the run-time type of the receiver object (typically implemented by dereferencing the v-table).

3.4. Summary

The verification of programs requires invariants, but, as we have seen, dealing with invariants presents several verification problems. In this section, we have presented a methodology the structures a program and its specifications in such a way that it is possible to perform sound modular verification. The methodology introduces the field modifier **rep**, the ghost field inv, the **expose** statement, and the invariant-level literal $\star$. We can now specify and generate verification conditions for programs with reentrancy, subclassing, dynamic dispatch, and invariants that span several objects and class frames.

4. Multi-threaded Programs

Multi-threaded object-oriented programs are becoming mainstream: servers are already multi-threaded, but soon we will have multi-cores on every desktop, too. So the question arises: Can we adapt the single-threaded verification methodology to verify multi-threaded programs? In particular, can we maintain invariants and also prevent data races and deadlocks?

Section 4.0 introduces a methodology to avoid data races for individual objects. Section 4.1 extends the methodology to guarantee inter-object invariants over rep objects. Section 4.2 concludes by extending this methodology to protect against deadlocks.

```
class Counter : Runnable {
  int dangerous;
  Counter() {
    dangerous = 0;
  }
  override void Run() {
    int tmp = dangerous;
    dangerous = tmp + 1;
  }
}
```

```
class Program {
  void Main() {
    Counter ct = new Counter();
    Thread t = new Thread(ct);
    Thread u = new Thread(ct);
    t.Start(); u.Start();
  }
}
```

Figure 13. A simple program to illustrate the possible effects of race conditions. Method Run is invoked twice in this program (via the $\mathit{Thread}{\circ}\mathit{Start}$ method), but the final value of $\mathit{dangerous}$ may end up as either 1 or 2, depending on how the runtime system's thread scheduler happens to interleave the thread executions.

4.0. Data Race Prevention

A *data race* occurs in a multi-threaded program when one thread writes a field or array element, another thread reads or writes the same field or array element, and neither thread performs a synchronization operation that would give it exclusive access to the data. Data races almost always indicate a programming error and such errors are extremely difficult to find and debug due to the nondeterministic interleaving of the thread executions.

Figure 13 shows this problem using a straightforward program. An instance of the $\mathit{Counter}$ class is shared by two threads. Looking at the Run method of the $\mathit{Counter}$ object, which is invoked by the $\mathit{Thread}{\circ}\mathit{Start}$ method, each thread appears to increment the variable by 1. However, in some interleavings of the thread executions, the combined effect is not to increment the variable by 2. In particular, both threads might read the variable when its value is 0, in which case each of the two threads will set the variable to 1.

Like in C$^\sharp$ and Java, every object in Spec$^\flat$ also acts as a lock. These locks can be used to ensure mutual exclusion among threads by using a lock statement:

```
Stmt ::= ...
       |  lock ( Expr ) Block
```

where the expression must be of a reference type. The execution of **lock** (o) $\{S\}$ acquires the lock o, executes S, and then releases o. The acquire operation first waits until a time when no thread holds o, so that the acquisition of o will maintain the program invariant that each lock is held by at most one thread at a time.

The locking mechanism prevents multiple threads from holding o at the same time, but it does not prevent threads from accessing o's fields. We introduce a methodology where a thread t can access a field of an object o only if o is *thread local*—that is, the thread that created the object has not made the object available to other threads—or t holds the lock o. We call the set of objects that a thread can access its *access set*. By making sure that access sets are disjoint, we prevent data races.

The life cycle of each object can now be described as follows.

- A new object is initially thread local (that is, *unshared*), and is included in the access set of the creating thread.

- An unshared object can be made accessible to other threads by sharing it. The sharing operation removes the object from the thread's access set.
- A shared object can be exclusively acquired by locking it. When (and if) the acquisition succeeds, the object is added to the access set of the thread.
- When a locked object is released, it is removed from the access set of the thread and once again becomes available for acquisition.

Language Constructs We introduced the lock statement above. Here, we introduce ghost variables and other constructs needed to write and specify multi-threaded programs.

- We introduce a new keyword that denotes the thread object of the current thread:

$$\begin{array}{lcl} Atom & ::= & \ldots \\ & | & \mathbf{tid} \end{array}$$

  The type of **tid** is the predefined class $Thread$, and it evaluates to a different value for each thread.
- For each object and array, we introduce a boolean ghost field $shared$, which indicates if the object or array is shared or thread local. Note that $shared$ is monotonic in the sense that once an object becomes shared, it remains shared forever.
- For each object and array, we introduce a ghost field $mythread$, which refers to the thread with access to the object or **null** if the object is free. Thus, the access set of a thread t is the set of objects whose $mythread$ field is t. We use this encoding of access sets because it provides us with an appropriate and existing mechanism to handle modifications of access sets (see the discussion on Method Framing Revisited in Section 3.2). The only operation allowed on $mythread$ is to compare it with **tid**, and $mythread$ is not admissible in object invariants.
- We introduce a statement for sharing thread-local objects:

$$\begin{array}{rcl} Stmt & ::= & \ldots \\ & | & ShareStmt \\ ShareStmt & ::= & \mathbf{share}\ Expr\ ; \end{array}$$

  where the expression must be of a reference type.

Finally, in Fig. 14, we give the specifications of the predefined classes $Thread$ and $Runnable$. The precondition and modifies clause of the $Thread$ constructor say that a $Runnable$ object cannot be used with more than one thread. Similarly, the precondition of $Start$ and the inclusion of $\mathbf{this}.mythread$ in the modifies clause of $Start$ prevent a thread from being started more than once.

Example Let us consider a variation of the Fig. 13 introductory counter example, where each thread runs a session object and several session objects share the counter, see Fig. 15.

The main thread creates a new counter, makes it available for sharing, and creates two session objects, a and b. At that point, the sessions objects are thread local to the main thread.

Next, the two threads t and u are created. They take a and b as arguments, both of which are still thread local to the main thread. According to its specification, the thread constructor may remove the $Runnable$ object from the caller's access set; hence, the main thread cannot access a and b after constructing the threads.

```
class Thread {
  Thread(Runnable r)
    requires r ≠ null && r.mythread = tid && ¬r.shared;
    modifies r.mythread, r.inv;
    ensures mythread = tid && ¬shared;
  void Start()
    requires mythread = tid;
    modifies this.∗, mythread;
  . . .
}

class Runnable {
  virtual void Run()
    requires mythread = tid;
    modifies this.∗;
}
```

Figure 14. The predefined classes *Thread* and *Runnable*.

```
/* main thread */
Counter ct = new Counter();
share ct;
Session a = new Session(ct, 0);
Session b = new Session(ct, 1);
Thread t = new Thread(a);
Thread u = new Thread(b);
t.Start();
u.Start();
```

```
class Counter {
  int n;
  Counter()
    ensures mythread = tid &&
              ¬shared;
  { n = 0; }
  virtual void Inc()
    requires mythread = tid;
    modifies this.∗;
  { expose (this) { n = n + 1; } }
}
class Session : Runnable {
  Session(Counter ct, int id)
    requires ct ≠ null &&
              ct.shared;
    ensures mythread = tid &&
              ¬shared;
  . . .
}
```

Figure 15. An example where multiple threads run session objects that use a shared counter object.

Next, threads t and u are started, which implicitly calls the *Run* method on a and b, respectively. Each session object's *Run* method is executed with **tid** set to the executing thread; here, t and u, respectively.

Defaults and Shorthands Like the specifications we saw earlier for single-threaded program, specifications of multi-threaded programs are written in a stylized fashion. To simplify the program text, we introduce the following defaults (applied during normalization). For every constructor, we use the default postcondition

$$\textbf{ensures}\ \mathit{mythread} = \textbf{tid}\ \&\&\ \neg\mathit{shared};$$

and for every non-constructor method, we use the default precondition

$$\textbf{requires}\ \mathit{mythread} = \textbf{tid};$$

These defaults have the additional advantage that they always hold in single-threaded programs. Thus, any part of a program that would verify under the single-threaded methodology will also verify under the multi-threaded methodology.

Most methods do not modify shared or $\mathit{mythread}$, so, as we have already assumed in examples, we change the definition of $o.*$ in modifies clauses to exclude these fields (*cf.* page 36):

```
$\mathit{ModAllowed}[\![\mathit{Spec}, o, f]\!] =$
    for each "$\textbf{modifies}\ W;$" in $\mathit{Spec}$ do
      for each "$\mathit{desig\ suffix}$" in $W$ do
        case $\mathit{suffix}$ of
          $(.*)$ :    $(o = \textbf{old}(\mathit{Tr}[\![\mathit{desig}]\!]) \wedge f \neq \mathit{inv} \wedge$
                      $f \neq \mathit{shared} \wedge f \neq \mathit{mythread}) \vee$
          ...
```

If needed, a modifies clause can list these fields explicitly.

The defaults let us omit several of the specifications we showed explicitly in Fig. 15.

Translation Since we don't intend to verify the implementation of $\mathit{Thread}^\circ\mathit{Start}$, we never need to keep track of more than one value for $\textbf{tid}$. Therefore, we simply encode $\textbf{tid}$ as a global constant with an unknown value:

$$\textbf{const}\ \mathit{tid}\colon \textbf{ref}\ ;$$
$$\textbf{axiom}\ \mathit{tid} \neq \textbf{null} \wedge \mathit{type}(\mathit{tid}) = \mathit{Thread}\ ;$$

The translation of the expression $\textbf{tid}$ is:

$$\begin{aligned} \mathit{Df}[\![\textbf{tid}]\!] &= \\ \mathit{Tr}[\![\textbf{tid}]\!] &= \mathit{tid} \end{aligned}$$

In the translation of constructors, we add the assumption

$$\textbf{assume}\ \neg\mathcal{H}[\mathit{this}, \mathit{shared}] \wedge \mathcal{H}[\mathit{this}, \mathit{mythread}] = \mathit{tid}\ ;$$

on entry to the implementation declaration of $\mathbb{C}^\circ\mathbb{C}$ (*cf.* page 37).

We record the monotonicity of shared as part of the definition of heap successors (*cf.* page 22):

```
$\textbf{axiom}\ (\forall\ \_old\colon [\textbf{ref}, \textbf{name}]\textbf{any},\ \_new\colon [\textbf{ref}, \textbf{name}]\textbf{any}\ \bullet$
  $\mathit{successor}(\_old, \_new) \Rightarrow$
    $\mathit{wellFormed}(\_new) \wedge$
    $(\forall\ r\colon \textbf{ref}\ \bullet\ \_old[r, \mathit{alloc}] \Rightarrow \_new[r, \mathit{alloc}]\ ) \wedge$
    $(\forall\ r\colon \textbf{ref}\ \bullet\ \_old[r, \mathit{shared}] \Rightarrow \_new[r, \mathit{shared}]\ ))$
```

We change the definedness of field-access and array-access expressions. For f a field different from $mythread$, we have (*cf.* page 31):

$$\begin{aligned} Df[\![E.mythread]\!] &= Df[\![E]\!] \land Tr[\![E]\!] \neq \textbf{null} \\ Df[\![E.f]\!] &= Df[\![E]\!] \land Tr[\![E]\!] \neq \textbf{null} \land \mathcal{H}[Tr[\![E]\!], mythread] = tid \\ Df[\![E[F]]\!] &= Df[\![E]\!] \land Tr[\![E]\!] \neq \textbf{null} \land \mathcal{H}[Tr[\![E]\!], mythread] = tid \land \\ &\quad\; Df[\![F]\!] \land 0 \leqslant Tr[\![F]\!] \land Tr[\![F]\!] < length(Tr[\![E]\!]) \end{aligned}$$

Note that the reading of the $mythread$ field may constitute a race condition, but since the only operation we allow on $mythread$ is comparing it with $\mathbf{tid}$, any such race condition is benign because $o.mythread = \mathbf{tid}$ is a stable condition—only a thread itself changes $mythread$ to or from the value $\mathbf{tid}$. This argument also applies to the translations below.

Finally, we change the translation of various statements. For field and array element updates (*cf.* page 43):

$$\begin{aligned} & Tr[\![o.f = \ e;]\!] = \\ & \quad \textbf{assert } o \neq \textbf{null} \land \mathcal{H}[o, mythread] = tid \land \neg(\mathcal{H}[o, inv] <: C) \ ; \\ & \quad \mathcal{H}[o, f] := e \end{aligned}$$

$$\begin{aligned} & Tr[\![a[i] = \ e;]\!] = \\ & \quad \textbf{assert } a \neq \textbf{null} \land \mathcal{H}[a, mythread] = tid \land \mathcal{H}[a, inv] = object \ ; \\ & \quad \textbf{assert } 0 \leqslant i \land i < length(a) \ ; \\ & \quad \textbf{assert } type(e) <: elemType(type(a)) \ ; \\ & \quad \mathcal{H}[a, elems] := Store(\mathcal{H}[a, elems], i, e) \end{aligned}$$

For the unpack and pack operations (*cf.* page 43):

$$\begin{aligned} & Tr[\![\textbf{unpack } o;]\!] = \\ & \quad \textbf{assert } o \neq \textbf{null} \land \mathcal{H}[o, mythread] = tid \land \mathcal{H}[o, inv] = \ldots \ ; \\ & \quad \ldots \end{aligned}$$

$$\begin{aligned} & Tr[\![\textbf{pack } o;]\!] = \\ & \quad \textbf{assert } o \neq \textbf{null} \land \mathcal{H}[o, mythread] = tid \land \mathcal{H}[o, inv] = \ldots \ ; \\ & \quad \ldots \end{aligned}$$

We define the $\mathbf{share}$ statement as follows:

$$\begin{aligned} & Tr[\![\textbf{share } o;]\!] = \\ & \quad \textbf{assert } o \neq \textbf{null} \land \mathcal{H}[o, mythread] = tid \ ; \\ & \quad \textbf{assert } \neg\mathcal{H}[o, shared] \ ; \\ & \quad \mathcal{H}[o, shared] := \textbf{true} \ ; \\ & \quad \mathcal{H}[o, mythread] := \textbf{null} \end{aligned}$$

The lock statement is more involved:

```
$Tr[\![\textbf{lock}\ (o)\ \{S\}]\!] =$
    $\textbf{assert}\ o \neq \textbf{null} \wedge \mathcal{H}[o, \mathit{shared}]$ ;
    $\textbf{assume}\ \mathcal{H}[o, \mathit{mythread}] \neq \mathit{tid}$ ;
    $\{\ \textbf{var}\ \mathit{oldHeap}\colon [\textbf{ref}, \textbf{name}]\textbf{any}$ ;
       $\mathit{oldHeap} := \mathcal{H}$ ;
       $\textbf{havoc}\ \mathcal{H}$ ; $\textbf{assume}\ \mathit{successor}(\mathit{oldHeap}, \mathcal{H})$ ;
       $\textbf{assume}\ (\forall\, x\colon \textbf{ref},\ f\colon \textbf{name} \bullet\ x \neq o\ \Rightarrow\ \mathit{oldHeap}[x, f] = \mathcal{H}[x, f]\ )$ ;
       $\textbf{assume}\ \mathcal{H}[o, \mathit{mythread}] = \textbf{null}$
    $\}$ ;
    $\mathcal{H}[o, \mathit{mythread}] := \mathit{tid}$ ;
    $Tr[\![\{S\}]\!]$ ;
    $\mathcal{H}[o, \mathit{mythread}] := \textbf{null}$
```

Note that the precondition of the lock statement reads the field $o.\mathit{shared}$, which may constitute a race condition. However, any such race condition is benign, since if the precondition $o.\mathit{shared}$ holds, then it is also stable (due to the monotonicity of shared).

The lock statement is thread non-reentrant, which means that a thread will deadlock if it attempts to lock a lock that it already holds. We deal with deadlocks in Section 4.2; here, we simply assume $o.\mathit{mythread}$ to be different from **tid** on entry to the lock statement, since this simplifies the bookkeeping we do around the translation of the body of the lock statement.

The purpose of the **havoc** construction in the translation of the lock statement is to simulate the possible interleavings of other threads, and in particular to simulate their possible effects on the fields of o. The following example illustrates the effect of this **havoc** on the verification. Let o be an object of a class that has an integer field f:

```
$\textbf{int}\ x$;
$\textbf{lock}\ (o)\ \{\ x =\ o.f;\ \}$
$\textbf{lock}\ (o)\ \{\ \textbf{assert}\ x == o.f;\ \}$        /* this assert may fail */
```

This example has no race condition. However, since other threads may acquire o and change $o.f$ between the two lock statements, the assertion may fail. The **havoc** command at the beginning of the translation of the second lock statement causes the verification to “forget” the value of $o.f$ from the first lock statement, which causes the verification of the assert to fail.

The translation says that the executions to be verified are those in which the **havoc** command establishes the assumption $\mathcal{H}[o, \mathit{mythread}] = \textbf{null}$, that is, those where o is not held in the state after the **havoc** command. Intuitively, the command **assume** $\mathcal{H}[o, \mathit{mythread}] = \textbf{null}$ waits until no other thread holds o. The assignments $\mathcal{H}[o, \mathit{mythread}] := \mathit{tid}$ and $\mathcal{H}[o, \mathit{mythread}] := \textbf{null}$ simulate the acquiring and releasing of o’s lock.

4.1. Invariants and Ownership Trees

We have now protected against race conditions. By itself, freedom from race conditions does not guarantee that a program behaves more correctly. For example, consider again our introductory counter example in Fig. 13. If we wrap a **lock** (**this**) block around the reading of field *dangerous* and wrap another **lock** (**this**) block around the update of

$dangerous$, then we have avoided race conditions, but we still end up with a program that may fail to increment $dangerous$ by 2. In this section, we consider the locking of whole data structures, and in particular locking that will maintain object invariants.

To protect invariants by locks, we expand our methodology to guarantee the following property:

Program Invariant 3 *In a multi-threaded program,*

$$(\forall\, o \bullet\ o.shared \wedge o.mythread = \textbf{null} \Rightarrow o.inv = type(o)\,)$$

where o quantifies over non-null, allocated objects, is a program invariant.

This property says that when an object is shared but free, then it is valid. In other words, an object invariant can be violated only when the object is in the access set of some thread. To enforce this program invariant, we must ensure that objects are valid when they become free, which affects the precondition of the $\textbf{share}$ operation.

To refine the multi-threaded methodology to ownership trees, we need to consider what to do with committed objects. According to the single-threaded methodology, operating on a committed object o must start with unpacking o's owner, which makes o valid. Thus, it is natural to let the unpack operation add the representation objects to the thread's access set. To avoid race conditions, we must then prevent other threads from gaining access to committed objects. We will do that by disallowing $\textbf{rep}$ fields from referring to shared objects. Under this refined methodology, one single lock statement locks an entire ownership tree, protecting all its invariants. Note that in the refined methodology, an "unshared" object can be accessed by different threads, but only if the object is part of an ownership tree whose root is shared—that is, when we previously said "thread local", we might now want to say "ownership-tree local".

Program Invariant 4 *In a multi-threaded program,*

$$(\forall\, o \bullet\ o.inv = Committed \Rightarrow o.mythread = \textbf{null}\,)$$

where o quantifies over non-null, allocated objects, is a program invariant.

An object o can in a non-$\textbf{rep}$ field, say $o.f$, hold on to a reference to a shared object. To access the data structure behind $o.f$, one would then first need to lock $o.f$. In order to meet with the precondition of the lock statement, it is necessary to know that $o.f$ is shared. But the conjunct $\textbf{this}.f.shared$ is admissible in an object invariant only if f is a $\textbf{rep}$ field, and we have just disallowed $\textbf{rep}$ fields from referring to shared objects. Instead, we introduce another field modifier:

$$\begin{array}{lcl} FieldModifier & ::= & \ldots \\ & | & \textbf{shared} \end{array}$$

Declaring a field f with $\textbf{shared}$ adds the implicit object invariant:

$$\textbf{this}.f = \textbf{null} \;||\; \textbf{this}.f.shared$$

Because $shared$ is monotonic, it is sound to dereference f in this way in an invariant even when f is not a representation object.

The implicit invariants of $\textbf{rep}$ and $\textbf{shared}$ fields guarantee the following property:

Program Invariant 5 *In a multi-threaded program, for any* $\mathbf{rep}$ *field* f *declared in a class* C*,*

$$(\,\forall\, o \bullet\ \mathit{type}(o) <: C \wedge o.\mathit{inv} <: C \Rightarrow o.f = \mathbf{null} \vee \neg o.f.\mathit{shared}\,)$$

and for any $\mathbf{shared}$ *field* f *declared in* C*,*

$$(\,\forall\, o \bullet\ \mathit{type}(o) <: C \wedge o.\mathit{inv} <: C \Rightarrow o.f = \mathbf{null} \vee o.f.\mathit{shared}\,)$$

where the quantifications range over non-null, allocated objects, are program invariants.

Translation We change the translation of unpack and pack to update the $\mathit{mythread}$ field of representation objects. The pack statement also needs to check the implicit invariants that come from $\mathbf{rep}$ and $\mathbf{shared}$ fields. Let T be the static type of o, and let S be the immediate superclass of T or, if T is an array type, let S be $\mathbf{object}$; then (*cf.* page 51):

```
$Tr[\![\mathbf{unpack}\ o;]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, \mathit{mythread}] = \mathit{tid} \wedge \mathcal{H}[o, \mathit{inv}] = T$ ;
    $\mathcal{H}[o, \mathit{inv}] := S$ ;
    for each field “$\mathbf{rep}\ F\ f$;” defined in $T$ do
      {  $\mathbf{assume}\ \mathcal{H}[o,f] \neq \mathbf{null}$ ;
         $\mathcal{H}[\mathcal{H}[o,f], \mathit{mythread}] := \mathit{tid}$ ;
         $\mathcal{H}[\mathcal{H}[o,f], \mathit{inv}] := \mathit{type}(\mathcal{H}[o,f])$
      []  $\mathbf{assume}\ \mathcal{H}[o,f] = \mathbf{null}$
      }

$Tr[\![\mathbf{pack}\ o;]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, \mathit{mythread}] = \mathit{tid} \wedge \mathcal{H}[o, \mathit{inv}] = S$ ;
    $\mathbf{assert}\ \mathit{Df}[\![\mathit{Inv}_T[\![o]\!]]\!] \wedge \mathit{Tr}[\![\mathit{Inv}_T[\![o]\!]]\!]$ ;
    for each field “$\mathbf{rep}\ F\ f$;” defined in $T$ do
      $\mathbf{assert}\ \mathcal{H}[o,f] = \mathbf{null}\ \vee$
         $(\mathcal{H}[\mathcal{H}[o,f], \mathit{mythread}] = \mathit{tid} \wedge \mathcal{H}[\mathcal{H}[o,f], \mathit{inv}] = \mathit{type}(\mathcal{H}[o,f]) \wedge$
           $\neg\mathcal{H}[\mathcal{H}[o,f], \mathit{shared}])$ ;
    for each field “$\mathbf{shared}\ F\ f$;” defined in $T$ do
      $\mathbf{assert}\ \mathcal{H}[o,f] = \mathbf{null} \vee \mathcal{H}[\mathcal{H}[o,f], \mathit{shared}]$ ;
    for each field “$\mathbf{rep}\ F\ f$;” defined in $T$ do
      {  $\mathbf{assume}\ \mathcal{H}[o,f] \neq \mathbf{null}$;
         $\mathcal{H}[\mathcal{H}[o,f], \mathit{inv}] := \mathit{Committed}$ ;
         $\mathcal{H}[\mathcal{H}[o,f], \mathit{mythread}] := \mathbf{null}$
      []  $\mathbf{assume}\ \mathcal{H}[o,f] = \mathbf{null}$
      } ;
    $\mathcal{H}[o, \mathit{inv}] := T$
```

To maintain Program Invariant 3, we add precondition $o.\mathit{inv} = o.\mathit{Type}$ to the $\mathbf{share}$ statement (*cf.* page 51):

```
$Tr[\![\mathbf{share}\ o;]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, mythread] = tid$ ;
    $\mathbf{assert}\ \neg\mathcal{H}[o, shared]$ ;
    $\mathbf{assert}\ \mathcal{H}[o, inv] = type(o)$ ;
    $\mathcal{H}[o, shared] := \mathbf{true}$ ;
    $\mathcal{H}[o, mythread] := \mathbf{null}$
```

When locking, we also have to forget the knowledge about owned objects (*cf.* page 52):

```
$Tr[\![\mathbf{lock}\ (o)\ \{S\}]\!] =$
    $\mathbf{assert}\ o \neq \mathbf{null} \wedge \mathcal{H}[o, shared]$ ;
    $\mathbf{assume}\ \mathcal{H}[o, mythread] \neq tid$ ;
    $\{\ \mathbf{var}\ oldHeap: [\mathbf{ref}, \mathbf{name}]\mathbf{any}$ ;
      $oldHeap := \mathcal{H}$ ;
      $\mathbf{havoc}\ \mathcal{H}$ ; $\mathbf{assume}\ successor(oldHeap, \mathcal{H})$ ;
      $\mathbf{assume}\ (\forall\, x: \mathbf{ref},\ f: \mathbf{name} \bullet$
        $\mathcal{H}[x, mythread] = tid \Rightarrow oldHeap[x, f] = \mathcal{H}[x, f]\ )$ ;
      $\mathbf{assume}\ \mathcal{H}[o, mythread] = \mathbf{null}$
    $\}$
    $\mathcal{H}[o, mythread] := tid$ ;
    $Tr[\![\{S\}]\!]$ ;
    $\mathcal{H}[o, mythread] := \mathbf{null}$
```

Note how we deal with forgetting the knowledge about owned objects. Unlike the previous translation of the lock statement, where we only forgot the fields of the object being locked, we now erase knowledge about the entire program state, except for the state of the objects that are accessible by the current thread. This reflects the recent possible effects of other threads on the ownership tree rooted at o. It also erases knowledge of committed objects whose transitive owners *are* held by the current thread. This encoding is convenient, because it lets us write the **havoc** construction without defining exactly which committed objects are reachable from the thread's accessible objects—something that presents a difficulty for automatic theorem provers anyway—and we argue that this erasing is okay, because a program should rely only on the invariants of, not the exact field values of, committed objects (this is analogous to how we encoded the postcondition contribution of modifies clauses, see the discussion on Method Framing Revisited in Section 3.2).

Finally, the translation also encodes Program Invariants 3 and 5 as axioms (but not Program Invariant 4, because we don't need it in verification).

```
$\mathbf{axiom}\ (\forall\, h: [\mathbf{ref}, \mathbf{name}]\mathbf{any},\ o: \mathbf{ref} \bullet$
  $wellFormed(h) \wedge o \neq \mathbf{null} \wedge \mathcal{H}[o, alloc] \Rightarrow$
    $\mathcal{H}[o, shared] \wedge \mathcal{H}[o, mythread] = \mathbf{null} \Rightarrow \mathcal{H}[o, inv] = type(o)\ )$ ;
```

For every class C:

```
class Session : Runnable {
  shared Counter ct;
  invariant ct ≠ null;
  int id;

  Session(Counter ct, int id)
    requires ct ≠ null;
  { this.ct = ct; this.id = id; }

  override void Run()
  { while (true) {
      lock (ct) { ct.Inc(); }
    }
  }
}
```

Figure 16. The full *Session* class for the counter example in Fig. 15. Field *ct* is declared with **shared**, since the session object needs to keep track of the fact that it is okay to lock it.

```
axiom (∀ h: [ref, name]any, o: ref •
  wellFormed(h) ∧ o ≠ null ∧ H[o, alloc] ⇒
    type(o) <: C ∧ H[o, inv] <: C ⇒
      for each field “rep F f;” defined in C do
        H[o, f] = null ∨ ¬H[H[o, f], shared]
      for each field “shared F f;” defined in C do
        H[o, f] = null ∨ H[H[o, f], shared]
  ) ;
```

Example Let us continue the example from Fig. 15 by showing the whole *Session* class, see Fig. 16.

It is instructive to take a closer look at the verification of the $Session{}^\circ Run$ method:

0. On entry, the typing assumption in the translation of method implementations tells us $type(this) <: Session$. Also, the default precondition tells us $this.inv = Session$, which by the reflexivity of $<:$ yields $this.inv <: Session$.
1. The heap, $\mathcal{H}$, is a syntactic target of the loop, since the loop calls a method (*Md*, page 30). What is known about the heap on an arbitrary iteration thus comes from *LoopMod* (page 30), which applies the method's modifies clause to the loop. The modifies clause of *Run*, declared in class *Runnable* in Fig. 14, is **this**.$*$, which stands for the fields of **this** except *inv*, *shared*, and *mythread*. Thus, every iteration of the loop starts with $\mathcal{H}[this, inv]$ and $\mathcal{H}[this, mythread]$ having the same values as when the loop was first reached.
2. By steps 0 and 1, we conclude that

$$type(this) <: Session \wedge this.inv <: Session$$

holds on entry to each loop iteration.
3. By step 2 and Program Invariant 2, we conclude that *this* satisfies the *Session* invariant on entry to each loop iteration, namely $\mathcal{H}[this, ct] \neq$ **null**. And by

step 2 and Program Invariant 5, we conclude that the **shared** field ct is shared: $\mathcal{H}[\mathcal{H}[this, ct], shared]$. These properties about $\mathcal{H}[this, ct]$ are what we need to discharge the precondition of the lock statement.

4. By the encoding of the lock statement, and in particular by the conditions assumed after the **havoc** command, we have that the object being locked, $\mathcal{H}[this, ct]$, is free. Since $\mathcal{H}[this, ct]$ is checked to be shared before the **havoc** , the definition of $successor$ tells us that it remains shared after the **havoc** . By Program Invariant 3, we thus have that $\mathcal{H}[this, ct]$ is valid, which is the precondition we need to establish for the call to Inc.
5. We deduce that $\mathcal{H}[this, ct]$ is unchanged by the call to Inc, which among other things means it is still non-null, as follows:
   - The encoding of the modifies clause of Inc lets us conclude that the call does not change any field of $this$, provided $this$ is not the target of the call ($\mathcal{H}[this, ct]$) and provided $this$ is not committed at the time of the call.
   - We deduce the dis-equality $this \neq \mathcal{H}[this, ct]$ from the assumption about $\mathcal{H}[\mathcal{H}[this, ct], mythread]$ at the beginning of the encoding of the lock statement.
   - According to proof step 2, $this$ is not committed on entry to the loop. Moreover, lock acquisition does not change fields of objects in the thread's access set, and proof step 1 tells us that $\mathcal{H}[this, mythread] = tid$ holds on entry to the loop.)
6. By the definition of $successor$, which upon return from a call we get to assume relates the old and new heap of the call, we have that $\mathcal{H}[this, ct]$ remains shared after the call to Inc.
7. By steps 5 and 6, we are able to discharge the proof obligation associated with the pack operation at the end of the **expose** block.

4.2. Deadlock Prevention

A *deadlock* occurs when there is a nonempty set of threads, each of which waits for a lock held by another thread in the set. Deadlocks are programming errors.

The prototypical example for a deadlock is the dining philosophers problem [17], where n philosophers (the threads) sit at a round table, spending their time eating and thinking. There are n forks available (the shared objects), placed between adjacent philosophers at the table. Eating requires the use of two forks. A philosopher can only pick up one fork at a time (philosopher locks a fork). In this setting, there is a possibility of a deadlock, for example if every philosopher holds a left fork and waits for a right fork.

Deadlocks can be avoided if all shared objects are partially ordered and each thread acquires shared objects in ascending order.

We let a program construct a partial order on shared objects. We make this order available in Spec$^\flat$ programs by introducing an irreflexive operator:

$$\begin{array}{lll} \otimes & ::= & \ldots \\ & \mid & \sqsubset \end{array}$$

where the operands of $\sqsubset$ must be the keyword **lockbound** (explain shortly) or be of a reference type. Operator $\sqsubset$ has the same binding power as $<$.

We also change the share statement so that one can specify the position of a newly shared object in this order:

$$\mathit{ShareStmt} \quad ::= \quad \mathbf{share}\ \mathit{Expr}^* \sqsubset \mathit{Expr} \sqsubset \mathit{Expr}^*\ ;$$

where all expressions must have a reference type. In the statement $\mathbf{share}\ LL \sqsubset o \sqsubset UU;$, it is the expression o that is being shared. It is checked to evaluate to a non-null value. The objects specified by LL are lower bounds and the objects specified by UU are upper bounds. For every pair of objects l and u in LL and UU, respectively, if both l and u are non-null, then the share statement requires $l \sqsubset u$, which ensures that there is a place for o between LL and UU.

Finally, we introduce a keyword **lockbound** that indicates an upper bound on all the locks acquired by the current thread.

$$\begin{array}{lll} \mathit{Atom} & ::= & \ldots \\ & | & \mathbf{lockbound} \end{array}$$

To avoid deadlocks, a precondition of the $\mathbf{lock}\ (o)$ statement is that o lies above **lockbound**. Statement $\mathbf{lock}\ (o)\ \{S\}$ then sets **lockbound** to o before executing S, and restores **lockbound** to the old value of **lockbound** after executing S. **lockbound** can be used only as an argument to $\sqsubset$.

Example Dijkstra proposed a solution to avoid deadlocks of dining philosophers by ordering all the forks and requiring the philosophers to pick up their respective forks in that order [17]. We show that solution for $n = 3$ in Fig. 17. The forks are named x, y, and z and the philosophers are named a, b, and c. Philosopher a will pick up fork x before fork y, philosopher b will pick up fork y before fork z, and philosopher c will pick up fork x before fork z. Since all philosophers adhere to the same global fork order, thus creating an asymmetry around the table, we avoid deadlocks.

Due to timing issues, this solution might still suffer from starvation. To avoid that problem, one can for example introduce queues of eating requests, that guarantee equal access to a fork by adjacent philosophers. We do not discuss this solution any further.

Prelude We extend the prelude by encoding **lockbound** as a global variable:

$$\mathbf{var}\ \mathit{lockbound}\colon \mathbf{ref}\ ;$$

We introduce an strict partial order called *LockOrder* (*i.e.*, the relation *LockOrder* is irreflexive and transitive).

$$\begin{array}{l} \mathbf{function}\ \mathit{LockOrder}(\mathbf{ref}, \mathbf{ref})\ \mathbf{returns}\ (\mathbf{bool})\ ; \\ \mathbf{axiom}\ (\forall\, o\colon \mathbf{ref} \bullet\ \neg \mathit{LockOrder}(o, o)\ )\ ; \\ \mathbf{axiom}\ (\forall\, o\colon \mathbf{ref},\ p\colon \mathbf{ref},\ q\colon \mathbf{ref} \bullet \\ \quad \mathit{LockOrder}(o, p) \land \mathit{LockOrder}(p, q) \Rightarrow \mathit{LockOrder}(p, q)\ )\ ; \end{array}$$

Just like we predefined the **object** class and added a constructor to the prelude, we predefine the *Runnable* class (see Fig. 14). This lets us give the *Run* method a specification that is not expressible in the Spec$^\flat$ language; in particular, we include a precondition that says that *lockbound* is below all references in the Spec$^\flat$ program:

```
class Program {
  void Main() {
    Fork x = new Fork(); share ⊏ x ⊏;
    Fork y = new Fork(); share x ⊏ y ⊏;
    Fork z = new Fork(); share y ⊏ z ⊏;
    Philosopher a = new Philosopher(x, y);
    Philosopher b = new Philosopher(y, z);
    Philosopher c = new Philosopher(x, z);
    Thread A = new Thread(a);
    Thread B = new Thread(b);
    Thread C = new Thread(c);
    A.Start(); B.Start(); C.Start();
  }
}

class Fork { }

class Philosopher {
  shared Fork left; shared Fork right;
  invariant left ≠ null && right ≠ null && left ⊏ right;

  Philosopher(Fork left, Fork right)
    requires left ≠ null && left.shared;
    requires right ≠ null && right.shared;
    requires left ⊏ right;
  { this.left = left; this.right = right; }

  override void Run()
  { while (true) {
      lock (right) { lock (left) {
        /* use the forks to eat ... */
    } } }
  }
}
```

Figure 17. A program that runs 3 dining philosophers.

```
procedure Runnable∘Run(this: ref) returns ( ) ;
  requires (∀ o: ref •
    o ≠ null ∧ type(o) <: object ⇒ LockOrder(lockbound, o) ) ;
  . . .
```

We omit here the constant *Runnable*, its associated type axioms, and the translation of the constructor.

Translation The translation of the expression **lockbound** is:

$$Df[\![\mathbf{lockbound}]\!] \; = $$
$$Tr[\![\mathbf{lockbound}]\!] \; = \; lockbound$$

Since the lock statement is a structured block statement, the value of **lockbound** on exit from a method is the same as it was on entry. However, **lockbound** can have different values during the execution of a method. We therefore put **lockbound** into the modifies and ensures clause of every procedure in our translation (*cf.* page 41):

```
$TrMod[\![Spec]\!] =$
    $\textbf{modifies}\ \mathcal{H}, lockbound\ ;$
    $\textbf{ensures}\ lockbound = \textbf{old}(lockbound)\ ;$
    $\textbf{ensures}\ (\forall\, o\colon \textbf{ref},\ f\colon \textbf{name} \bullet\ \ldots\ \mathcal{H}[o,f] = \textbf{old}(\mathcal{H})[o,f]\ )$
```

We replace the translation of the previous **share** statement (*cf.* page 55) with the following translation of the new **share** statement:

```
$Tr[\![\textbf{share}\ LL \sqsubset o \sqsubset UU;]\!] =$
    $\textbf{assert}\ o \neq \textbf{null} \land \mathcal{H}[o, mythread] = tid\ ;$
    $\textbf{assert}\ \neg\mathcal{H}[o, shared]\ ;$
    $\textbf{assert}\ \mathcal{H}[o, inv] = type(o)\ ;$
    for each expression “$l$” in $LL$ do
      for each expression “$u$” in $UU$ do
        $\textbf{assert}\ l = \textbf{null} \lor u = \textbf{null} \lor LockOrder(l, u)\ ;$
    for each expression “$l$” in $LL$ do
      $\textbf{assume}\ l = \textbf{null} \lor LockOrder(l, o)\ ;$
    for each expression “$u$” in $UU$ do
      $\textbf{assume}\ u = \textbf{null} \lor LockOrder(o, u)\ ;$
    $\mathcal{H}[o, shared] := \textbf{true}\ ;$
    $\mathcal{H}[o, mythread] := \textbf{null}$
```

Finally, we change the translation of the lock statement to check for possible deadlock violations and to update *lockbound* (*cf.* page 55):

```
$Tr[\![\textbf{lock}\ (o)\ \{S\}]\!] =$
    $\textbf{assert}\ o \neq \textbf{null} \land \mathcal{H}[o, shared] \land LockOrder(lockbound, o)\ ;$
    $\{\ \textbf{var}\ oldHeap\colon [\textbf{ref}, \textbf{name}]\textbf{any}\ ;$
      $oldHeap := \mathcal{H}\ ;$
      $\textbf{havoc}\ \mathcal{H}\ ;\ \textbf{assume}\ successor(oldHeap, \mathcal{H})\ ;$
      $\textbf{assume}\ (\forall\, x\colon \textbf{ref},\ f\colon \textbf{name} \bullet$
        $\mathcal{H}[x, mythread] = tid\ \Rightarrow\ oldHeap[x,f] = \mathcal{H}[x,f]\ )\ ;$
      $\textbf{assume}\ \mathcal{H}[o, mythread] = \textbf{null}$
    $\}$
    $\{\ \textbf{var}\ oldLockbound\colon \textbf{ref}\ ;$
      $oldLockbound := lockbound\ ;$
      $lockbound := o\ ;\ \mathcal{H}[o, mythread] := tid\ ;$
      $Tr[\![\{S\}]\!]\ ;$
      $\mathcal{H}[o, mythread] := \textbf{null}\ ;\ lockbound := oldLockbound$
    $\}$
```

Note, since we now deal with deadlocks, we have removed the assumption

$$\textbf{assume}\ \mathcal{H}[o, mythread] \neq tid\ ;$$

which previously was part of our translation of the lock statement—the checked condition $\mathbf{lockbound} \sqsubset o$ implies that the thread does not already hold o.

4.3. Summary

In this section, we have extended the single-threaded methodology to multi-threaded code. The basic idea is to limit the interaction between threads, so that reasoning can proceed mostly as for single-threaded code, except at certain synchronization points. We have shown how to maintain objects invariants in a multi-threaded setting. A single lock statement acquires exclusive access to all objects in an ownership tree. A program can decide the degree of sharing in a program by deciding to make fields either **rep** fields or **shared** fields. We prevent deadlocks by allowing a program to incrementally and locally specify a global partial order among the objects in the program. Locking objects in ascending order then prevents deadlocks.

5. History and Acknowledgments

Program-Verifier Architecture Roots of the program-verifier architecture we have described trace back to ESC/Modula-3 [14], a project spearheaded by Greg Nelson. The ESC/Modula-3 checker translated Modula-3 programs into a form of Dijkstra's guarded commands [18,65], from which it generated verification conditions. The ESC/Java checker [23] refined this approach by more clearly defining two forms of an intermediate language [53]. The BoogiePL intermediate language [11] took two more steps by adding a mathematical part to the language, which previously had been passed directly to the theorem prover, and by adding a parser for the language, which for debugging the verifier has been shown to have great value [4].

Filliâtre has also proposed a generation of verification conditions via an intermediate language based on type theory [20]. This has served as the basis for the tool and intermediate verification language Why [21]. Why is being used as the intermediate language for the Java verifier Krakatoa [57] and the C verifier Caduceus [22].

Rather than using VC generation and first-order logic, a program verifier can encode more of the program into the formulas passed to the theorem prover. This approach is followed, for example, by the KeY tool [1] for JavaCard programs, which uses dynamic logic, and the LOOP [36,58] and Jive [62] verifiers for Java, which use extensions of Hoare logic.

Translation of Languages and Language Features We have shown a translation of core object-oriented language features. The use of updatable maps (arrays) to model references goes back to Burstall [8]. Modeling the heap as a 2-dimensional array, as done by Poetzsch-Heffter [68], has the advantage that one can quantify over all field names, as we have done extensively.

Leino's thesis [43] gave a translation of object-oriented source-language features, together with constructs like exceptions, records, and deallocation, into guarded commands, along the lines of what was done in ESC/Modula-3. Ecstatic [44] is a core object-oriented language with a weakest-precondition semantics, and includes axioms that encode types and allocation. The ESC/Java translation of annotated Java into guarded commands and axioms employed a number of encoding tricks aimed at improving the performance of the underlying theorem prover [52]. Boogie uses similar encodings, but in this paper we have avoided such “optimizations” in order to make the presentation more straightforward.

Efficient Formulas We defined the semantics of BoogiePL$^\flat$ commands in terms of classical weakest preconditions. However, such a definition gives rise to a lot of redundancy that can lead a theorem prover to unnecessary case splits, which easily can turn into unbearable performance [24]. ESC/Modula-3 and ESC/Java used techniques for reducing this redundancy, which is important for a practical checker [24,46].

Unlike the structured commands of BoogiePL$^\flat$ that we used in this paper, BoogiePL has unstructured goto commands. Boogie uses a redundancy-reducing technique based on weakest preconditions to define the semantics of these unstructured commands [5].

Quantifiers Another important consideration in the design of a practical automatic verifier is how to give the theorem prover directives of how to instantiate universally quantified expressions. The SMT solver Simplify [12] calls these directives *triggers*, and getting good results from Simplify requires good use of triggers.

Specification, Abstraction, and Methodology The first sound modular verification methodology for a significant subset of a modern object-oriented language was given by Müller in his thesis [61]. The particular methodology we presented for using object invariants in single-threaded [3] and multi-threaded [35,37] programs is based on joint work with our Spec$^\sharp$ colleagues. There are extensions of this methodology to visibility-based invariants [47,7,60], static class invariants [48], iterators [33], pure methods [9,34], model fields [49], and subject-observers structures [54].

We made use of committed objects to get abstraction in modifies clauses. Other approaches have used abstraction dependencies [43,50,61], data groups [45,51], separation logic [67], and dynamic frames [38].

We know the technique of changing the specification for method overrides from the work on Fugue [10], which called such specifications *sliding*. We justified the soundness of dereferencing shared fields in implicit object invariants on the grounds that $shared$ is monotonic, which is an idea further explored for type states [19]. The technique of capturing parameters, which we specify by a modifies clause that mentions inv and/or $shared$, was used in the work on ESC/Modula-3 [13].

6. Conclusion

Program verification, although as old as computer science [25], is still one of its grand challenges [31].

This paper developed a verifying compiler for a multi-threaded object-oriented subset of Spec$^\sharp$ [6], here called Spec$^\flat$. Correctness of Spec$^\flat$ programs is specified by types, method specifications, object invariants, field modifiers, ghost state, and new statements. The developed compiler (defined in Sections 2, 3, and 4) takes as input a Spec$^\flat$ program, and generates, via an intermediate language called BoogiePL$^\flat$ (defined in Section 1), first-order verification conditions, which can be processed by an SMT solver. If its proof attempt succeeds, then the program is correct and can be run. If it fails, advice is sought from the user. Experience shows that this is a viable approach. By now, many thousands of Spec$^\sharp$ lines have been verified, although often only with shallow properties like freedom from raised exceptions.

This paper addresses many challenges of verifying modern multi-threaded object-oriented languages. We have shown how to deal with reentrancy, aliasing, inheritance, representation abstraction, method framing, and multi-threading. We have also shown how to engineer a basic verifier by introducing an intermediate verification language. Much remains to be done: for instance, we have to learn how to verify more object-oriented design patterns [26], different kinds of concurrent code [41], and we have to learn how to verify abstractions [32].

We hope that this paper guides students toward understanding program verification, and we encourage them to build their own verifier. Program verification is a rich and rewarding research field.

References

[0] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[3] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[4] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[5] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05*, pages 82–87. ACM, September 2005.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[7] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Seventh International Conference on Mathematics of Program Construction (MPC 2004)*, Lecture Notes in Computer Science, pages 54–84. Springer-Verlag, July 2004.

[8] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 6:23–50, 1971.

[9] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[10] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004—Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, June 2004.

[11] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[12] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[13] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, DEC Systems Research Center, July 1998.

[14] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[15] L. Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California, Berkeley, Berkeley, CA 94720, 1973.

[16] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[17] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971.

[18] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[19] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *Proceedings of International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, July 2003.

[20] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[21] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[22] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[23] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[24] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[25] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[27] Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1978.

[28] Donald I. Good, Ralph L. London, and W. W. Bledsoe. An interactive program verification system. In *Proceedings of the international conference on reliable software*, pages 482–492. ACM, 1975.

[29] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[31] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[32] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, USA, 2006.

[33] Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: Proof rules and implementation. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2005)*, July 2005.

[34] Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2006)*, July 2006.

[35] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 137–147. IEEE, September 2005.

[36] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, April 2001.

[37] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming

model for concurrent object-oriented programs. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 420–439. Springer, November 2006.

[38] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

[39] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[40] James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, Pittsburg, PA 15213, September 1969.

[41] Doug Lea. *Concurrent programming in Java: design principles and patterns*. The Java series. Addison-Wesley, Reading, MA, USA, 1996.

[42] Gary Todd Leavens. *Verifying Object-Oriented Programs that Use Subtypes*. PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.

[43] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Technical Report Caltech-CS-TR-95-03.

[44] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[45] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[46] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.

[47] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[48] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, July 2005.

[49] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, March 2006.

[50] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[51] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 246–257. ACM, May 2002.

[52] K. Rustan M. Leino and James B. Saxe. Java to guarded commands translation. Design note ESCJ 16c, ESC/Java source distribution, August 1998.

[53] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.

[54] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. Manuscript KRML 166, October 2006.

[55] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[56] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.

[57] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–

2):89–106, January–March 2004.

[58] Tiziana Margaria and Wang Yi, editors. *The LOOP compiler for Java and JML*, volume 2031 of *Lecture Notes in Computer Science*. Springer, April 2001.

[59] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[60] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Invariants for non-hierarchical object structures. In Anamaria Martins Moreira and Leila Ribeiro, editors, *Brazilian Symposium on Formal Methods, SBMF 2006*, pages 233–248. SBC, September 2006.

[61] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

[62] Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter. Programming and interface specification language of JIVE—specification and design rationale. Technical Report 223, Fernuniversität Hagen, 1997.

[63] John Nagle and Scott Johnson. Practical program verification: Automatic program proving for real-time embedded systems. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 48–58, January 1983.

[64] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also available as Technical Report CSL-81-10, Xerox PARC, June 1981.

[65] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[66] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[67] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 247–258. ACM, January 2005.

[68] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.