Ecstatic: An object-oriented programming language with an axiomatic semantics

K. Rustan M. Leino
16 December 1996
Digital Equipment Corporation Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
rustan@pa.dec.com

Abstract. This paper describes a small object-oriented programming language and its axiomatic semantics. The language includes common object-oriented features like methods and subtyping. Objects are implicitly references, and the semantics handles the aliasing that arises. The paper formalizes in first-order logic what it means for a method implementation to meet its specification.

0 Introduction

In the cornucopia of object-oriented programming languages, one is hard-pressed to find a language with an axiomatic semantics. Examples of axiomatic semantics for imperative languages are Hoare logic [5] and Dijkstra's weakest-precondition calculus [4], both of which have achieved considerable success in the last decades. Reasoning using an axiomatic semantics is done at a higher level than with an operational semantics. For example, reasoning about a method invocation is done by reasoning about the method's specification rather than by examining its implementations.

This paper defines a small object-oriented language called Ecstatic. The axiomatic semantics of Ecstatic is based on Dijkstra's weakest liberal preconditions. The language includes common object-oriented features like methods and subtyping. Objects are references to data fields and methods, and the semantics handles the aliasing among these references.

Some efforts have been directed at providing an axiomatic semantics for an object-oriented language. In his thesis, Leavens gave verification rules for a small language with objects [8]. However, those rules apply only to programs without aliasing. Naumann has given a predicate-transformer semantics for a programming language that includes record extension and procedure type variables [12]. However, that semantics also

This paper appears in the Proceedings of the 4th International Workshop on Foundations of Object-Oriented Languages, January 1997.
Republished here by permission.

assumes the absence of aliasing. My thesis shows how to translate features of object-oriented languages into Dijkstra's guarded commands, for which weakest preconditions are defined [10]. Abadi and I have written a Hoare-like logic for object-oriented programs [1]. The logic handles methods, subtyping, and aliasing, but excludes recursive data types. The Ecstatic language allows recursive data structures.

The present work distinguishes itself from object-oriented specification languages like Larch/C++ [7] and those surveyed by Lano and Haughton [6] in that the semantics of Ecstatic is explicit about the connection between method specifications and implementations: the semantics precisely prescribes a *verification condition*, an untyped first-order predicate that formally expresses "the implementation meets its specification".

The Ecstatic language grew from work on the Modula-3 Extended Static Checker (ESC), a static analysis tool for detecting errors like **nil** -dereferencing, array index out-of-bounds errors, deadlocks, and race conditions [0, 3]. ESC translates Modula-3 annotated with specifications into verification conditions that are then passed to a mechanical theorem prover called Simplify [0]. Ecstatic's prescription of verification conditions is similar to that used by ESC. In fact, I have written a checker for Ecstatic that parses and type-checks Ecstatic programs, generates the verification conditions, and passes them to Simplify. This checker was implemented directly from Ecstatic's definition in (a recent version of) this paper.

One can imagine mapping (subsets of) languages like Modula-3 and Java to Ecstatic. Such a mapping combined with the axiomatic semantics of Ecstatic would then provide an axiomatic semantics for the language at hand. For the purpose of being the target of such mappings, the Ecstatic commands could be more primitive. As it stands, Ecstatic is not parsimonious with features: it includes methods with in- and out-parameters, **modifies** clauses, and a rich syntax for expressions, any of which could have been left as a "simple extension". The reason for including these features is to gather in one place the details describing them, and to make Ecstatic a language in which programming is fun. Not included are, most notably, arrays, iteration (but recursion is allowed), abstraction, and modules. Such are left for future inclusion.

The organization of this paper is as follows. Section 1 on declarations describes the major building blocks of the language; Section 2 on commands and expressions describes the smaller building blocks of the language. These sections provide a language definition roughly at the level of the Modula-3 definition [13]: it defines syntax, type-checking, and an operational description that includes checked run-time errors. Section 3 gives the axiomatic semantics of the language, and prescribes how verification conditions are generated. Lastly, Section 4 shows some examples, Section 5 discusses some possible extensions to the language, and Section 6 offers some concluding remarks.

1 Declarations

A program is an (unordered) set of declarations. The language consists of four kinds of declarations. These are used to declare types, data fields, methods, and method implementations. The first three of these introduce identifiers that name the types, data fields, and methods.

Before giving the details, let me give a short (artificial) example program to convey the flavor of the language.

$$
\begin{array}{l}
\mathbf{type}\ T \\
\mathbf{field}\ x\colon T \to \mathbf{int} \\
\mathbf{field}\ y\colon T \to \mathbf{nat} \\
\mathbf{method}\ m(t\colon T,\ z\colon \mathbf{int}) \\
\quad \mathbf{requires}\ 0 \leq y[t] + z \\
\quad \mathbf{modifies}\ x[t] \\
\quad \mathbf{ensures}\ x[t] \geq x_0[t] + y[t] + z \\
\mathbf{impl}\ m(t\colon T,\ z\colon \mathbf{int})\ \mathbf{is} \\
\quad x[t] := x[t] + 2 \cdot y[t] + z
\end{array}
\tag{0}
$$

This program declares an object type T with two fields, x and y. It also declares a method m that takes the self parameter t and another parameter z. The method is given a specification: it requires as a precondition that the sum of the y field of object t and in-parameter z be natural, and it ensures that the post-state value of data field x at object t is at least the sum of its pre-state value, the value of the y field at t, and the given parameter z, and that nothing else is modified. The example program also gives an implementation of m. The implementation consists of one assignment statement.

Now for the details of the four declarations.

1.0 Types

A *type* is a name that denotes a set of values. The types of a program are partially ordered by a *subtype* relation. If type U is a subtype of type T, then every member of (the set of values denoted by) U is a member of T. The subtype relation is reflexive and transitive. A subtype relationship is called *proper* if the two types are distinct. Two types are *compatible* if one is a subtype of the other.

The dual of the subtype relation is the *supertype* relation. A type T is said to be a *direct* supertype of a type U if T is a proper supertype of U and there is no type “in between” T and U, that is, no proper subtype of T is a proper supertype of U.

There are three *simple types*: $\mathbf{bool}$, $\mathbf{nat}$, and $\mathbf{int}$. These represent the booleans $\mathbf{false}$ and $\mathbf{true}$, the natural numbers, and the integers, respectively. Type $\mathbf{nat}$ is a subtype of $\mathbf{int}$.

In addition to the values of simple types, there are *objects*. An object is a value that is either **nil** or a reference to a set of data fields and methods. Equality of objects is reference equality. An *object type* determines the names and types of a subset of the set of fields and methods of its members. Ecstatic features one built-in object type, $\mathbf{obj}$, which is a supertype of all object types. The declaration

$\mathbf{type}\ T <: U$,

where T is a name and U is an object type, introduces the name T as an object type whose direct supertype is U. Hence, each declared object type has a unique direct supertype; in the jargon, Ecstatic features *single inheritance*. If U is $\mathbf{obj}$, "$<:\ U$" may be omitted from the declaration.

No two types may have the same name. Furthermore, from each declared object type, it must be possible to follow the sequence of direct supertypes and arrive at type $\mathbf{obj}$. That is, a program must not contain declarations that produce cycles in the subtype relation. For example, the declarations

$\mathbf{type}\ T <: U$
$\mathbf{type}\ U <: T$

are not both allowed in the same program.

Every expression has a statically determined type. This type contains every value to which the expression can evaluate. For the purpose of describing how static typing is done, I introduce a special type **null**, which is treated as a subtype of every object type (but is not considered to be an object type). The idea is that **null** contains the value **nil**, and that the type of the expression **nil** is **null** (see Section 2.1 on expressions). The type **null** cannot be mentioned in a program; it is used only in the language definition for the purpose just described.

Every non-**nil** object has an *allocated type*, which manifests itself at run-time when the object is allocated and is never changed thereafter. Ecstatic's type rules ensure that if an expression has (static) type T and evaluates at run-time to a non-**nil** object t, then the allocated type of t is a subtype of T.

1.1 Data fields

A *data field* is a map from objects to values. The declaration

field $x\colon T \to U$

introduces the name x as a data field that maps from object type T to type U. We say that type T *possesses* field x, that T is the *index type* of x, and that U is the *range type* of x.

For example, the following program snippet declares a data structure for linked lists of integers.

$$\begin{array}{l} \mathbf{type}\ \mathit{Node} \\ \mathbf{field}\ \mathit{val}\colon \mathit{Node} \rightarrow \mathbf{int} \\ \mathbf{field}\ \mathit{next}\colon \mathit{Node} \rightarrow \mathit{Node} \end{array}$$

The expression $x[t]$ denotes the x field of an object t. Stated differently, since fields are maps, the expression $x[t]$ denotes the value of x at t. The expression $x[t]$ is called a *select expression*, and t is called its *index*. If T is a subtype of U, and U possesses a field x, then, since every member of T is a member of U, objects of type T can be used to index x.

A program can have several fields of the same name. Allowing different fields to have the same name is provided merely as a syntactic convenience; it does not imply any kind of relation between different fields with the same name. However, no two fields may have the same name and index type; that is, no type may possess two fields with the same name. For example, the two declarations

$$\begin{array}{l} \mathbf{field}\ x\colon T \rightarrow \ldots \\ \mathbf{field}\ x\colon T \rightarrow \ldots \end{array}$$

are not both allowed in one program, because then T would possess two fields with the name x, whereas the two declarations

$$\begin{array}{l} \mathbf{field}\ y\colon T \rightarrow \ldots \\ \mathbf{field}\ y\colon U \rightarrow \ldots \end{array}$$

are allowed, because the two fields have different index types. As part of the syntactic convenience is a rule for *resolving* field names occurring in a program, that is, mapping field names to field declarations: for an expression t whose (static) type is an object type T, the name x in $x[t]$ is resolved to the field whose name is x and that is possessed by type U, where U is the closest supertype of T that possesses a field x. The expression does not type-check if no such field exists. This rule is similar to the rule for resolving names of local variables in a program, or resolving names of bound variables in a mathematical formula.

As an artificial example, consider the declarations

$$\begin{array}{l} \textbf{type } U \\ \textbf{field } x\colon U \rightarrow \ldots \\ \textbf{type } T <: U \\ \textbf{field } x\colon T \rightarrow \ldots \end{array}$$

and an expression t of type T. According to the rule for resolving field names, the x in expression $x[t]$ refers to the x field possessed by T. To refer to the other x field of t, one needs to write an expression $x[t']$ where expression t' evaluates to the same value as t but has a different statically determined type. One way to achieve this is to use a **narrow** expression in the following way:

$$x[\textbf{narrow}(t, U)] \qquad .$$

Here, x resolves to the field x possessed by type U. The **narrow** expression is explained in Section 2.1.

Several fields can be introduced at the same time. The declaration

$$\textbf{field } x_0, \ldots, x_{n-1}\colon T \rightarrow U$$

is shorthand for

$$\begin{array}{l} \textbf{field } x_0\colon T \rightarrow U \\ \vdots \\ \textbf{field } x_{n-1}\colon T \rightarrow U \qquad . \end{array}$$

1.2 Methods

A *method* is a procedure that can be invoked on an object. The language features no procedures other than methods. To describe methods, I start with the following definition: a *binding* is a pair $x\colon T$, where x is a name (called a *variable*) and T is a type. The *formal in- and out-parameters* of a method are given as lists of bindings. The list of in-parameters must be nonempty. The first in-parameter is called the *self* parameter, and its type must be an object type.

Let $\mathit{formal\text{-}outs}$ and $\mathit{formal\text{-}ins}$ be lists of bindings, such that $x\colon T$ (where T is an object type) is the first binding of $\mathit{formal\text{-}ins}$, and let spec be a *specification* (defined below). Then, the declaration

$$\textbf{method } \mathit{formal\text{-}outs} := m(\mathit{formal\text{-}ins}) \quad \mathit{spec}$$

introduces the name m as a method for type T. We say that type T *possesses* method m. The method has out-parameters $formal\text{-}outs$, in-parameters $formal\text{-}ins$, and specification $spec$. The names in $formal\text{-}outs$ and $formal\text{-}ins$ must be distinct. If the list $formal\text{-}outs$ is empty, the "$:=$" is omitted from the declaration.

A program can have several methods with the same name, but a single type may not possess two methods with the same name. Allowing different methods to have the same name is provided merely as a syntactic convenience; it does not imply any kind of relation between different methods with the same name.

A *specification* describes the conditions under which the method may be invoked and the method's effect on the state space when invoked. It is given in three parts: the *precondition* describes those initial states from which a caller is allowed to invoke the method, the *modifies list* specifies which fields of which objects that the method is allowed to modify, and the *postcondition* relates, for terminating method invocations, the pre- and post-states of the invocation (there is no guarantee that the method invocation will actually terminate).

Syntactically, a specification is given as a sequence of **requires**, **modifies**, and **ensures** clauses. The modifies list is the union of the lists given by the **modifies** clauses, and the pre- and postconditions are the conjunctions of the predicates given in the **requires** and **ensures** clauses, respectively.

The forms of **requires** and **ensures** clauses are

$$\begin{array}{l}\textbf{requires}\ pre\\ \textbf{ensures}\ post\end{array} \quad ,$$

where pre and $post$ are *specification predicates*. The free identifiers occurring in pre must be either fields or formal in-parameters. The free identifiers occurring in $post$ must be either fields, formal in- or out-parameters, or *initial-value fields*. An *initial-value field* is a field subscripted by 0. It refers to the value of the field at the time the method is invoked.

A **modifies** clause has the form

$$\textbf{modifies}\ w \quad ,$$

where w is a list of designator expressions. A *designator expression* is a select expression $x[E]$, where x is a field and E is an expression whose type is not **null** and whose free identifiers are fields or formal parameters. The name x is resolved to a field using the type of E as described earlier. Data fields occurring in E refer to their pre-state values.

There is a restriction on the use of initial-value fields in postconditions. If x_0 is an initial-value field occurring in the postcondition, then $x[E]$, for some E, must be present in the modifies list. This restriction does not hamper specification expressiveness, because, roughly, if no $x[E]$ occurred in the modifies list, x would not be allowed to be modified at any object, so x would have the same value in the post-state as it does in the pre-state, and hence there would be no reason to mention x_0 instead of x in the postcondition.

1.3 Method implementations

A method m for a type T may be given one implementation per subtype of T. A method implementation is given by the declaration

$$\mathbf{impl}\ \mathit{formal\text{-}outs} := m(\mathit{formal\text{-}ins})\ \mathbf{is}\ S \qquad ,$$

where $\mathit{formal\text{-}outs}$ and $\mathit{formal\text{-}ins}$ are lists of bindings, m is a name, and S is a *command* (also known as a *statement*). The list $\mathit{formal\text{-}ins}$ must be nonempty, and the type of the first binding in $\mathit{formal\text{-}ins}$, say U, must be an object type. We say this implementation is *given at* type U. The name m is resolved to the method whose name is m and that is possessed by type T, where T is the closest supertype of U that possesses a method m. The method implementation declaration is allowed only if such a method exists.

Furthermore, the number of bindings in $\mathit{formal\text{-}outs}$ and $\mathit{formal\text{-}ins}$, and their types except the type of the first binding in $\mathit{formal\text{-}ins}$, must be the same as those of the formal out- and in-parameters of method m's declaration. That is, the method implementation is allowed to use different names for the formal parameters than those used by the method declaration. The names in $\mathit{formal\text{-}outs}$ and $\mathit{formal\text{-}ins}$ must be distinct. If the list $\mathit{formal\text{-}outs}$ is empty, the " $:=$ " is omitted from the implementation declaration.

For each name v in $\mathit{formal\text{-}outs}$ and $\mathit{formal\text{-}ins}$, free occurrences of variable v in S are resolved to this formal parameter. Command S must not contain any free variables other than these.

1.4 Example: Mapping operations over a linked list

Although commands have not yet been defined, this is a good time to give a program example. This example shows a linked list and its map method, which can map an operation over the values stored in the linked list.

A linked list is built up by objects of type $Node$, declared by

$$\begin{array}{l}\textbf{type } Node \\ \textbf{field } val\colon Node \to \textbf{int} \\ \textbf{field } next\colon Node \to Node \quad .\end{array} \tag{1}$$

It is often useful to map some operation over a linked list. As a simple example, I declare a type Op with a method $apply$ and a data field r. The method takes an integer as a parameter and is allowed to operate on the field r.

$$\begin{array}{l}\textbf{type } Op \\ \textbf{field } r\colon Op \to \textbf{int} \\ \textbf{method } apply(op\colon Op,\ n\colon \textbf{int}) \\ \quad \textbf{modifies } r[op]\end{array}$$

An Op object can be passed to the map method for type $Node$, which invokes the $apply$ method of the Op object for each integer contained in the linked list. The declaration and implementation of the map method are given as follows.

$$\begin{array}{l}\textbf{method } map(node\colon Node,\ op\colon Op) \\ \quad \textbf{requires } op \neq \textbf{nil} \\ \quad \textbf{modifies } r[op] \\ \textbf{impl } map(node\colon Node,\ op\colon Op) \textbf{ is} \\ \quad apply(op, val[node])\ ; \\ \quad \textbf{if } next[node] \neq \textbf{nil then } map(next[node], op) \textbf{ fi}\end{array}$$

I can now show some examples of Op subtypes, each with its own implementation of the $apply$ method.

$$\begin{array}{l}\textbf{type } SumOp <: Op \\ \textbf{impl } apply(sum\colon SumOp,\ n\colon \textbf{int}) \textbf{ is} \\ \quad r[sum] := r[sum] + n\end{array}$$

$$\begin{array}{l}\textbf{type } CountOp <: Op \\ \textbf{impl } apply(cnt\colon CountOp,\ n\colon \textbf{int}) \textbf{ is} \\ \quad r[cnt] := r[cnt] + 1\end{array}$$

$$\begin{array}{l}\textbf{type } PickOp <: Op \\ \textbf{impl } apply(p\colon PickOp,\ n\colon \textbf{int}) \textbf{ is} \\ \quad r[p] := n\end{array}$$

With these declarations, the program snippet

$$\begin{array}{l} op := \mathbf{new}(SumOp) \;; \\ r[op] := 0 \;; \\ map(list, op) \end{array}$$

has the effect of setting $r[op]$ to the sum of the integers stored in $list$ (this assumes that $list$ is non- **nil**). By replacing $SumOp$ by $CountOp$, the program snippet would instead set $r[op]$ to the length of $list$. And by instead using type $PickOp$, the program snippet picks the last element from the list and stores it in $r[op]$.

2 Commands and expressions

This section presents the commands and expressions that can be part of a program. It describes the parsing and type-checking of these, and gives an informal operational description of the execution of commands and evaluation of expressions. A formal semantics is given in Section 3.

2.0 Commands

This subsection introduces the various commands that are part of the language by informally stating the type-checking rules and operational meaning of each command.

The language consists of the following commands.

$v := E$	simple assignment
$v := \mathbf{new}(T)$	allocation
$x[E] := E'$	update
$\mathbf{var}\ bindings\ \mathbf{in}\ S\ \mathbf{end}$	block statement
$S \;;\; S'$	composition
$\mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S'\ \mathbf{fi}$	conditional
$var\text{-}list := m(expr\text{-}list)$	method invocation

In addition, the language provides some convenient shorthands, $\mathbf{skip}$, $\mathbf{wrong}$, and $\mathbf{assert}\ P$, that are defined in terms of these commands.

2.0.0 Simple assignment

The simple assignment statement has the form

$$v := E \qquad ,$$

where E is an expression and v is a *mutable variable*, that is, a local variable (explained below) or a formal out-parameter.

The type of E must be a subtype of the type of v. The statement evaluates E (which may result in a run-time error if E is partial, see Section 2.1), and then assigns the result to v.

2.0.1 Allocation

The allocation statement creates a new object. It is written

$$v := \mathbf{new}(T) \qquad ,$$

where v is a mutable variable and T is an object type. T must be a subtype of the type of v. This statement assigns to v a non- **nil** object that is "new", that is, that has never been the result of any allocation statement previously encountered in the program execution. Thus, the new object is not in use by the executing program by the time the allocation statement is executed. We say that the value was previously not allocated.

The allocated type of the new object is T. The new object v also has the following property, for every data field x whose index type is a supertype of T and whose range type is some type U: the initial value of $x[v]$ is **nil** if U is an object type, and an arbitrary value of type U otherwise.

2.0.2 Update

Data fields are updated with the update command, which has the form

$$x[E] := E' \qquad ,$$

where x is a field, and E and E' are expressions. The type of E must be an object type. The name x is resolved to a field using the type of E as described earlier for select expressions.

The type of E' must be a subtype of the range type of x. The command first evaluates E and E'. It is a run-time error if E produces **nil**; otherwise, the command sets the x field of object E to E'.

2.0.3 Block

The block statement is written

$$\mathbf{var}\ \mathit{bindings}\ \mathbf{in}\ S\ \mathbf{end} \qquad ,$$

where $bindings$ is a nonempty list of bindings, the names in which are all distinct, and S is a command. For each binding $v{:}\ T$ in $bindings$, the block statement introduces v as a *local variable* of type T for use in S. Free occurrences of variable v in S are resolved to this local variable. The initial value of v is $\mathbf{nil}$ if T is an object type, and an arbitrary value of type T otherwise.

2.0.4 Composition

Commands can be sequentially composed using the associative operator $;$. The command

$$S\ ;\ S'$$

executes command S upon whose termination it executes command S'.

2.0.5 Conditional

The conditional statement is written

$$\mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S'\ \mathbf{fi} \qquad ,$$

where E is a boolean expression and S and S' are commands. If E evaluates to $\mathbf{true}$, the conditional statement executes S; otherwise it executes S'.

If S' is $\mathbf{skip}$, then "$\mathbf{else}\ S'$" may be omitted.

2.0.6 Method invocation

The final command, method invocation, is written

$$\mathit{var\text{-}list} := m(\mathit{expr\text{-}list}) \qquad ,$$

where $\mathit{var\text{-}list}$ is a list of distinct, mutable variables, m identifies a method, and $\mathit{expr\text{-}list}$ is a list of expressions. The elements of lists $\mathit{var\text{-}list}$ and $\mathit{expr\text{-}list}$ are called the *actual out- and in-parameters*, respectively, of the invocation. If $\mathit{var\text{-}list}$ is empty, the "$:=$" is omitted.

The (static) type of the first expression in $\mathit{expr\text{-}list}$, call it U, must be an object type. The name m is resolved to the method whose name is m and that is possessed by a type T, where T is the closest supertype of U that possesses a method m. The command does not type-check if no such method exists.

If method m is declared by

$$\mathbf{method}\ \mathit{formal\text{-}outs} := m(\mathit{formal\text{-}ins}) \quad \mathit{spec} \qquad ,$$

then $\mathit{expr\text{-}list}$ must *match* $\mathit{formal\text{-}ins}$ and $\mathit{formal\text{-}outs}$ must match $\mathit{var\text{-}list}$. A list of expressions or bindings e *matches* a list of bindings or variables w just when e and w have the same lengths, and the type of every element in e is a subtype of the type of the corresponding element in w.

A method invocation is executed as follows. First, a new set of formal parameters for the method invocation is created. Then, the actual in-parameters are evaluated, their values are bound to the formal in-parameters, and initial values are bound to the formal out-parameters. It is a run-time error if the first expression in $\mathit{expr\text{-}list}$ evaluates to $\mathbf{nil}$. The initial value of a formal out-parameter of type U is $\mathbf{nil}$ if U is an object type, and an arbitrary value of type U otherwise. Finally, the *appropriate implementation* of m (explained below) is executed, upon whose termination the values of the formal out-parameters are assigned to the actual out-parameters.

The *appropriate implementation* of an invocation of a method m is determined at run-time as follows. Let T be the allocated type of first actual in-parameter of the invocation, and let $\mathcal{S}$ denote the set of supertypes of T at which an implementation of m is given. The appropriate implementation of the method invocation is the implementation given at the smallest ("subtype-most") type of the types in $\mathcal{S}$. Note that the types in $\mathcal{S}$ are totally ordered; hence, there is a smallest type in $\mathcal{S}$, *provided* $\mathcal{S}$ is nonempty. Guaranteeing that $\mathcal{S}$ is nonempty is done by the following (possibly rather strict) requirement on programs: if T is an object type one of whose supertypes possesses a method m, then either there is a supertype of T at which an implementation of m is given, or the program contains no statement of the form

$$v := \mathbf{new}(T) \qquad .$$

This is a static constraint that can be checked by a simple inspection of the program text.

2.0.7 Some convenient shorthands

The commands presented above show the features of the language. However, it is often convenient to use some abbreviations. For that purpose, I define some shorthands in terms of the primary language features. The shorthands are defined in terms of the commands above. This means that they get a precise definition, and also that I don't need to later give a formal semantics for the shorthands.

I define the $\mathbf{skip}$, $\mathbf{wrong}$, and assert statements as follows, where P denotes an expression.

$$\begin{array}{l}
\mathbf{skip} \;=\; \mathbf{var}\ v\colon \mathbf{nat}\ \mathbf{in}\ v := v\ \mathbf{end} \\
\mathbf{wrong} \;=\; \mathbf{var}\ v\colon \mathbf{nat}\ \mathbf{in}\ v := \mathbf{narrow}(-\mathit{1}, \mathbf{nat})\ \mathbf{end} \\
\mathbf{assert}\ P = \mathbf{if}\ P\ \mathbf{then\ skip\ else\ wrong\ fi}
\end{array}$$

The command $\mathbf{skip}$ does not alter the state of the program. Executing $\mathbf{wrong}$ always leads to a run-time error ($\mathbf{narrow}$ is defined below). The assert statement skips if P evaluates to $\mathbf{true}$, and goes wrong otherwise.

2.1 Expressions

Evaluating an expression at run-time produces its value if the expression is defined, and results in a run-time error otherwise. The evaluation does not affect the program state, that is, expressions are "side-effect free". (Thus, for example, $\mathbf{new}(T)$ is not an expression.)

The simplest expressions are constants and variables. The constants $\mathbf{false}$ and $\mathbf{true}$ have type $\mathbf{bool}$, and numeric constants formed from sequences of decimal digits have type $\mathbf{nat}$. The object constant $\mathbf{nil}$ has type $\mathbf{null}$. A variable is an expression whose type and value are the type and value of the variable.

The select expression is written $x[E]$, where x is a field name and E is an expression whose type is an object type. The name x is resolved to a data field as described in Section 1.1. If the value of E is not $\mathbf{nil}$, then $x[E]$ is the value of x at object E. The expression is not defined if E evaluates to $\mathbf{nil}$.

The $\mathbf{narrow}$ expression is written $\mathbf{narrow}(E, T)$, where E is an expression whose type is compatible with the type T. The value of $\mathbf{narrow}(E, T)$ is E and its type is T. The expression is not defined if the value of E is not a member of type T, that is, if T is $\mathbf{nat}$ and E produces a negative integer, or if T is an object type and E produces a non-$\mathbf{nil}$ object whose allocated type is not a subtype of T.

Expressions can have familiar operators. Defined on booleans are $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, and $\equiv$, and defined on integers are the binary operators $+$, $-$, $\cdot$, $\mathbf{div}$, and $\mathbf{mod}$, and the unary operator $-$. These operators have the usual semantics. If both operands of $+$, $\cdot$, or $\mathbf{div}$ have type $\mathbf{nat}$, or if the type of the second operand of $\mathbf{mod}$ is $\mathbf{nat}$, then the expression has type $\mathbf{nat}$ instead of $\mathbf{int}$.

If the run-time evaluation of the first operand to $\wedge$, $\vee$, $\Rightarrow$, or $\Leftarrow$ produces a value that determines the value of the expression, the second operand is not evaluated. That is, if the first operand of these expressions produces $\mathbf{false}$, $\mathbf{true}$, $\mathbf{false}$, and $\mathbf{true}$, respectively, the expression produces $\mathbf{false}$, $\mathbf{true}$, $\mathbf{true}$, and $\mathbf{true}$, respectively, without evaluating the second operand. For this reason, these four operators are called *short-circuit* boolean operators.

Integers can be compared using $=$, $\neq$, $<$, $\leq$, $\geq$, and $>$, producing booleans. Booleans can be compared using $=$ and $\neq$, and so can expressions whose types are object types or $\mathbf{null}$ and are compatible. Note that for comparing booleans, the boolean operator $\equiv$ can be used in place of $=$, the advantage being that $\equiv$ has a lower binding power (and a difference being that $\equiv$ is associative whereas $=$ is chaining, as explained shortly).

Pairs of parentheses can be used in the usual manner around an expression to explicitly express binding. The following table shows the default binding powers of all operators, from strongest to weakest.

unary prefix operators: $-$, $\neg$
$\cdot$, **div**, **mod**
$+$, $-$
$=, \neq, <, \leq, \geq, >$
$\wedge$, $\vee$
$\Rightarrow$, $\Leftarrow$
$\equiv$

The binary arithmetic operators associate to the left. The boolean operator $\equiv$ is associative. So are the boolean operators $\wedge$ and $\vee$, but since $\wedge$ and $\vee$ have the same binding power, it would be ambiguous, and thus not allowed, to write an expression like

$$A \wedge B \vee C \qquad .$$

Other operators do not associate, but the groups of operators $=, \leq, <$ and $=, \geq, >$ can be *chained*. This means that they parse as if they were associative, but the parsing is just a shorthand for writing each operator and its surrounding operands as a separate conjunct. For example, the expression

$$w \leq x = y < z$$

is shorthand for

$$w \leq x \ \wedge \ x = y \ \wedge \ y < z \qquad ,$$

whereas

$$x \leq y \geq z$$

does not parse because $\leq$ and $\geq$ are not in the same chaining group, and

$$x = y \neq z$$

does not parse because $\neq$ is not a chaining operator at all. Note that $=$ is chaining whereas $\equiv$ is associative; thus for booleans x , y , and z ,

$$x = y = z$$

is shorthand for

$$x = y \ \wedge \ y = z \qquad ,$$

whereas

$$x \ \equiv \ y \ \equiv \ z$$

is equivalent to

$$(x \ \equiv \ y) \ \equiv \ z$$

and to

$$x \ \equiv \ (y \ \equiv \ z) \qquad .$$

Since **nat** is a subtype of **int**, natural expressions can be used anywhere integer expressions can be. Similarly, if T is a subtype of an object type U, then an expression of type T can be used anywhere an expression of type U can be used, with one exception: an expression of type **null** cannot be used where an expression of an object type is expected, as is the case, for example, in the select expression.

Since they are not evaluated at run-time, pre- and postconditions can mention a richer set of expressions called *specification predicates*. A specification predicate is one of the following.

- An expression whose operator is $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, or $\equiv$, and whose subexpressions are specification predicates.
- A quantified expression $\langle \forall \, bindings \mid R \triangleright P \rangle$ or $\langle \exists \, bindings \mid R \triangleright P \rangle$, where $bindings$ is a nonempty list of bindings, the names of which are all distinct, and R and P are specification predicates. The names in $bindings$ are called *dummy variables*, R is called the *range* of the quantification, and P is called the *term* of the quantification. If R is the constant $\mathtt{true}$, it and the preceding " $\mid$ " may be omitted. The relation between the range and term is described by

  $$\begin{array}{l} \langle \forall \, bindings \mid R \triangleright P \rangle = \langle \forall \, bindings \triangleright R \Rightarrow P \rangle \\ \langle \exists \, bindings \mid R \triangleright P \rangle = \langle \exists \, bindings \triangleright R \wedge P \rangle \qquad . \end{array}$$

  For each name v is $bindings$, free occurrences of variable v in R and P are resolved to this dummy variable.
- A boolean *specification expression*, defined below.

- The expression $\mathbf{fresh}(E)$, where E is a specification expression whose type is an object type. The **fresh** expression, which is allowed only in postconditions, says that E is a non- **nil** object that is allocated in the post-state of a method, but not in its pre-state.

A *specification expression* is an expression that, when it occurs in a postcondition, is allowed to mention initial-value fields in select expressions. Such expressions are subject to the restrictions placed on the use of initial-value fields (as described in Section 1.2).

3 Formal semantics

This section describes the semantics of the programming language formally in terms of first-order logic. In particular, it defines a proof obligation associated with each method implementation. By discharging the proof obligation for a method implementation S, one is assured that no run-time error will occur in any execution of S, *provided* S is invoked only when its precondition holds and the method invocations occurring in S have the effects prescribed by their respective specifications. (One is not assured that the method will terminate, however.) By discharging the proof obligation for every method implementation of a program, the program is guaranteed never to result in a run-time error.

Throughout this section, I assume that the program text has been preprocessed to rename all identifiers to unique names. A simple way to do that in practice is to rename an identifier (like a field or variable) x to something like $x.53.6$ where 53 and 6 are the line and column in the program text at which x is declared. (See Section 4.4 for an example.)

3.0 Verification conditions

The formal meaning of commands is based on Dijkstra's *weakest liberal preconditions* (wlp) [4]. The function wlp maps commands to predicate transformers: for S a command and R a first-order predicate on the post-state of S, $wlp.S.R$ is a first-order predicate describing those initial states from which execution of S incurs no run-time errors and, if the execution terminates, terminates in a state satisfying R. This wlp is like Dijkstra's, except that it forbids run-time errors [10].

The proof obligation for a method implementation is that it *meet its specification*. Operationally, this means that, when started in a state satisfying the precondition, execution of the implementation does not result in a run-time error and, if it terminates, has the effect prescribed by the postcondition and modifies list. Formally, a method implementation and its specification are transformed into a *verification condition*, an untyped first-order logical formula; the proof obligation is to establish the validity of this formula, a task for which a mechanical theorem prover may be used. This section describes in detail how a verification condition is generated from the program text.

In principle, the formula

$$P \;\Rightarrow\; wlp.S.R$$

expresses that a command S meets a specification whose precondition is P and whose postcondition is R [4]. However, this ignores an important part of a method specification, the modifies list, which together with the postcondition prescribe the method's effect. To account for the modifies list, it is rewritten into a so-called *postcondition contribution*, which constrains modifications of fields according to the modifies list. Let Q denote that postcondition contribution (which is described in detail later). Then, our next approximation of the verification condition is

$$P \;\Rightarrow\; wlp.S.(R \wedge Q) \qquad .$$

Some technical details remain to describe the verification condition in full. First, an implementation is allowed to assume that the self parameter is not **nil**, that the in-parameters have values of appropriate types, and that the out-parameters have been properly initialized. Second, the verification condition must give a value to any initial-value field that occurs in R. Third, the verification condition is a formula written in untyped first-order logic. This means that the type information known about fields must be encoded in some form of "type axioms". These and related axioms form what is called the *background predicate*, which is included in the antecedent of the verification condition. Finally, the formal parameters mentioned in the specification must be renamed to the formal-parameter names used by the implementation, the given pre- and postconditions must be transformed from specification predicates to untyped first-order predicates, and program expressions must be transformed into equivalent expressions in the first-order logic.

Thus, the verification condition for a method implementation

$$\mathbf{impl}\; \mathit{formal\text{-}outs}' := m(\mathit{formal\text{-}ins}')\;\mathbf{is}\;\; S \qquad ,$$

whose specification is

$$\begin{array}{l}\mathbf{method}\; \mathit{formal\text{-}outs} := m(\mathit{formal\text{-}ins}) \\ \quad \mathbf{requires}\; P\; \mathbf{modifies}\; w\; \mathbf{ensures}\; R\end{array}$$

has the form

$$\begin{array}{ll} \mathit{BackgroundPred} \land \mathit{InitialFields} \land \mathit{Pr}(P') \land Y_0 = Y \land \mathit{alloc}_0 = \mathit{alloc} \land & \\ \mathit{self} \neq \mathbf{nil} \land \mathit{Types}(\mathit{formal\text{-}ins}') \land \mathit{Reset}(\mathit{formal\text{-}outs}') & (2) \\ \Rightarrow \mathit{wlp}.S.(\mathit{Pr}(R') \land Q) \quad , & \end{array}$$

where

- $\mathit{BackgroundPred}$ is the background predicate,
- $\mathit{InitialFields}$ states that the values of all data fields have appropriate types,
- P', w', and R' are P, w, and R, respectively, in which formal parameters from $\mathit{formal\text{-}outs}$ and $\mathit{formal\text{-}ins}$ have been replaced by the names of the corresponding parameters in $\mathit{formal\text{-}outs}'$ and $\mathit{formal\text{-}ins}'$,
- Pr is a function that maps a specification predicate to a first-order predicate,
- Y is the union of the lists $\mathit{Fields}(w')$ and $\mathit{Targets}(S)$, the data fields that appear in the modifies list w' and those that are updated by the command S, respectively,
- Y_0 is Y with every field initial-valued, so $Y_0 = Y$ means the conjunction of equalities $y_0 = y$ for every field y in Y,
- alloc represents the set of allocated objects, and alloc_0 is the initial value of alloc,
- self is the name of the self parameter, that is, the first name given in $\mathit{formal\text{-}ins}'$,
- $\mathit{Types}(\mathit{formal\text{-}ins}')$ says that the in-parameters have values of the appropriate types,
- $\mathit{Reset}(\mathit{formal\text{-}outs}')$ says that the formal out-parameters have appropriate initial values, and
- Q denotes $\mathit{PostCondContrib}(Y, w')$, the postcondition contribution resulting from the modifies list.

Antecedents $\mathit{BackgroundPred}$, $\mathit{InitialFields}$, and $\mathit{alloc}_0 = \mathit{alloc}$ are the same for all verification conditions in a program, whereas the rest of the verification condition is specific to a particular method implementation.

In the rest of this section, I give the details of the components of this formula.

3.1 The background predicate

This subsection describes how the program state and type information are encoded in untyped first-order logic. It prescribes the construction of the background predicate.

I assume the background predicate does not need to mention any axioms for logical operators (like $\neg$, $\wedge$, $\forall$), integer arithmetic ($+$, $-$, $\cdot$, $\mathbf{div}$, $\mathbf{mod}$), and integer inequalities (like $\leq$).

To encode the state of data fields, I use two functions: $[\ ]$ ("select") and $store$. As we have seen before, $x[t]$ denotes the x field at object t. The value of $store(x, t, E)$ responds in the same way to applications of select as does x, except possibly at t where the former yields the value E. The relation between select and $store$ is described formally by the following two axioms, called the *select-of-store axioms*:

$$\begin{array}{l} \langle \forall\, x, t, v \triangleright store(x, t, v)[t] = v \rangle \\ \langle \forall\, x, t, t', v \triangleright t \neq t' \Rightarrow store(x, t', v)[t] = x[t] \rangle \quad . \end{array} \tag{3}$$

For every object type T in the program, the first-order logic contains a term $tc\$T$, called the *typecode* of T. (Here and throughout, I assume "$\$$" to be a character that does not appear in Ecstatic programs.) The typecodes are distinct: for every two distinct object types T and U, the background predicate contains a conjunct

$$tc\$T \neq tc\$U \quad . \tag{4}$$

To model the allocated type of an object, I use a function $typecode$ from objects to typecodes. The subtype relation can then be defined in terms of typecodes. I introduce two functions, $subtype1$ and $subtype$. For every type declaration

$$\mathbf{type}\ T <: U$$

in the program, the background predicate contains the axiom

$$subtype1(tc\$T, tc\$U) \quad . \tag{5}$$

Function $subtype$ is defined as the reflexive, transitive closure of $subtype1$, as expressed by the following axioms:

$$\begin{array}{l} \langle \forall\, tc \triangleright subtype(tc, tc) \rangle \\ \langle \forall\, tc0, tc1 \triangleright subtype1(tc0, tc1) \Rightarrow subtype(tc0, tc1) \rangle \\ \langle \forall\, tc0, tc1, tc2 \triangleright subtype(tc0, tc1) \wedge subtype(tc1, tc2) \\ \qquad\qquad \Rightarrow subtype(tc0, tc2) \rangle \quad . \end{array} \tag{6}$$

Since the direct supertype of a type is uniquely determined —that is, Ecstatic uses single inheritance—, if two object types are not compatible, the only member they have in common is **nil** . This is described by the *incomparable subtype axiom*:

$$\begin{array}{l} \langle \forall\, t0, t1, tc0, tc1, tc \;\triangleright \\ \quad t0 \neq \mathbf{nil} \wedge t1 \neq \mathbf{nil} \wedge \\ \quad subtype(typecode(t0), tc0) \wedge subtype1(tc0, tc) \wedge \\ \quad subtype(typecode(t1), tc1) \wedge subtype1(tc1, tc) \wedge \\ \quad tc0 \neq tc1 \\ \quad \Rightarrow\; t0 \neq t1 \;\rangle \qquad . \end{array} \tag{7}$$

The axiom says that if the types of two non- **nil** objects ($t0$ and $t1$) have a common supertype (whose typecode is) tc , and the objects are of different direct subtypes of tc ($tc0$ and $tc1$), then the two objects are distinct.

For each type T , there is a *type predicate* $is\$T$. The idea is that $is\$T(E)$ holds just when E denotes a value of type T . The background predicate contains a definition for each type predicate: for each object type T , $is\$T$ is defined by

$$\langle \forall\, t \;\triangleright\; is\$T(t) \;\equiv\; t = \mathbf{nil} \vee subtype(typecode(t), tc\$T) \;\rangle \qquad , \tag{8}$$

and the type predicates for simple types are defined by

$$\begin{array}{l} \langle \forall\, v \;\triangleright\; is\$\mathbf{bool}(v) \;\equiv\; v = \mathbf{false} \vee v = \mathbf{true} \;\rangle \\ \langle \forall\, v \;\triangleright\; is\$\mathbf{nat}(v) \;\equiv\; 0 \leq v \;\rangle \\ \langle \forall\, v \;\triangleright\; is\$\mathbf{int}(v) \;\equiv\; true \;\rangle \qquad . \end{array} \tag{9}$$

Note that if T is a subtype of U , then $is\$T(E) \;\Rightarrow\; is\$U(E)$.

The definition of $is\$\mathbf{int}$ may seem puzzling: is every value an integer? The reason $is\$\mathbf{int}(v)$ is defined simply as the predicate $true$ is that the Ecstatic type system is strict enough that integers are never mixed with booleans or objects. For example, a programmer is never under the obligation to show that some particular value is indeed an integer and not, say, an object. (In a similar way, $is\$\mathbf{obj}(t)$ could have been defined as simply $true$.)

The boolean program values $\mathbf{false}$ and $\mathbf{true}$ are not the first-order predicates $false$ and $true$. The reason for this, and the consequences thereof, are as follows. Variables are terms in the first-order logic, and so are expressions. Consequently, boolean variables and the boolean constants $\mathbf{false}$ and $\mathbf{true}$ are terms of the logic. Perhaps more surprisingly, since $x < y$ is also an expression, it too must be a term of the logic, not a predicate. For example, if b is a boolean field, a program may contain a command $b[t] := x < y$, which, as we shall see, gives rise to an expression like $store(b, t, x < y)$

in the verification condition, except that the third argument to $store$ must be a term. To accommodate such expressions in the first-order logic, one needs to introduce special functions for rudimentary boolean operations like $=$, $<$, and $\wedge$, and then transform program expressions into first-order expressions that use these functions. The special functions require the introduction of a dozen axioms into the background predicate. Luckily, in many particular cases, it is possible to transform program expressions into first-order expressions without appeal to these functions, as is encoded by function Pr defined in a later subsection, but the special functions are needed in general. The six special functions are $equal$, $less$, $atmost$, not, and, and or, and the dozen axioms about these are

$$
\begin{array}{l}
\langle \forall\, c, d \triangleright\; equal(c, d) = \mathbf{false} \vee equal(c, d) = \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; less(c, d) = \mathbf{false} \vee less(c, d) = \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; atmost(c, d) = \mathbf{false} \vee atmost(c, d) = \mathbf{true} \,\rangle \\
\langle \forall\, c \triangleright\; not(c) = \mathbf{false} \vee not(c) = \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; and(c, d) = \mathbf{false} \vee and(c, d) = \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; or(c, d) = \mathbf{false} \vee or(c, d) = \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; equal(c, d) = \mathbf{true} \;\equiv\; c = d \,\rangle \\
\langle \forall\, c, d \triangleright\; less(c, d) = \mathbf{true} \;\equiv\; c < d \,\rangle \\
\langle \forall\, c, d \triangleright\; atmost(c, d) = \mathbf{true} \;\equiv\; c \leq d \,\rangle \\
\langle \forall\, c \triangleright\; not(c) = \mathbf{true} \;\equiv\; c \neq \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; and(c, d) = \mathbf{true} \;\equiv\; c = \mathbf{true} \wedge d = \mathbf{true} \,\rangle \\
\langle \forall\, c, d \triangleright\; or(c, d) = \mathbf{true} \;\equiv\; c = \mathbf{true} \vee d = \mathbf{true} \,\rangle \quad .
\end{array}
\tag{10}
$$

In addition, the background predicate contains an axiom that expresses that the two boolean values are distinct:

$$\mathbf{false} \neq \mathbf{true} \quad . \tag{11}$$

An example that uses the special functions is given in Section 4.3.

Two functions needed in the transformation of **narrow** expressions are $natural$ and $narrow$, for which the background predicate contains the two axioms

$$
\begin{array}{l}
\langle \forall\, v \triangleright\; 0 \leq v \;\Rightarrow\; natural(v) = v \,\rangle \\
\langle \forall\, t, tc \triangleright\; t = \mathbf{nil} \vee subtype(typecode(t), tc) \;\Rightarrow\; narrow(t, tc) = t \,\rangle \quad .
\end{array}
\tag{12}
$$

The type information provided by the declared types of data fields is encoded. For each pair of types (T, U) that appears in some data field declaration

$$\mathbf{field}\; x{:}\, T \to U \quad ,$$

the background predicate defines a *field-type predicate* $\mathit{field}\$T\$U$. The background predicate relates each such field-type predicate to select by including the conjunct

$$\langle \forall\, x, t \,\triangleright\, \mathit{field}\$T\$U(x) \land \mathit{is}\$T(t) \land t \neq \mathbf{nil} \;\Rightarrow\; \mathit{is}\$U(x[t]) \,\rangle \quad . \tag{13}$$

This says that if x is a field with index type T and range type U , then select x at a non- **nil** T object results in a value of type U .

To encode which objects are allocated, I make use of an *allocation state* called alloc . The fact that an object t has been allocated in an allocation state a is denoted by the predicate $\mathit{isDecl}(t, a)$. For a field x whose range type is an object type, x is said to be *consistent* with an allocation state a , written $\mathit{isConsistent}(x, a)$, if, for every object t , $x[t]$ is allocated in a if t is. This is encoded as follows: for every field-type predicate $\mathit{field}\$T\$U$ where U denotes an object type, the background predicate contains

$$\begin{array}{l}\langle \forall\, x, a, t \,\triangleright\, \mathit{field}\$T\$U(x) \land \mathit{isConsistent}(x, a) \land \\ \qquad\qquad \mathit{is}\$T(t) \land t \neq \mathbf{nil} \land \mathit{isDecl}(t, a) \\ \qquad\qquad \Rightarrow\; \mathit{isDecl}(x[t], a) \,\rangle \quad .\end{array} \tag{14}$$

Although the background predicate is not explicit about the isDecl of **nil** , note that axiom (14) is accordant with $\mathit{isDecl}(\mathbf{nil}, \mathit{alloc})$, but not with $\neg\mathit{isDecl}(\mathbf{nil}, \mathit{alloc})$.

The allocation state is changed by the allocation and method invocation commands. As an allocation state changes, the new allocation state is said to *succeed* the previous one. An object allocated in one allocation state remains allocated in successive allocation states. This is encoded by a predicate $\mathit{succeeds}$ and the axiom

$$\langle \forall\, t, a, b \,\triangleright\, \mathit{isDecl}(t, a) \land \mathit{succeeds}(b, a) \;\Rightarrow\; \mathit{isDecl}(t, b) \,\rangle \quad . \tag{15}$$

Finally, succession of allocation states preserves consistency, as expressed by the axiom

$$\begin{array}{l}\langle \forall\, x, a, b \,\triangleright\, \mathit{isConsistent}(x, a) \land \mathit{succeeds}(b, a) \\ \qquad\qquad \Rightarrow\; \mathit{isConsistent}(x, b) \,\rangle \quad .\end{array} \tag{16}$$

The background predicate is formed by conjoining the axioms (3), (6), (7), (9), (10), (11), (12), (15), and (16), and the axiom schemas (4), (5), (8), (13), and (14) applied to the types mentioned in the program.

3.2 Proper types and initial values

This subsection defines three functions on lists of bindings, Vars , Types , and Reset , one function on lists of designator expressions, Fields , and two functions on lists of data fields, $\mathit{FieldTypes}$ and $\mathit{FieldReset}$. It also defines the conjunct $\mathit{InitialFields}$ that is used in the verification condition (2).

Function Vars projects to the variable components of the bindings, Types is used in encoding the type and accessibility of formal in-parameters and dummy variables, and Reset is used in encoding the initial values of local variables and formal out-parameters. Function Fields is similar to Vars, but is used with modifies lists, and functions $\mathit{FieldTypes}$ and $\mathit{FieldReset}$ are similar to Types and Reset, but are used with data fields.

For B a list of bindings, $\mathit{Vars}(B)$ is the list of variables v occurring in some binding $v\colon T$ in B.

For B a list of bindings, $\mathit{Types}(B)$ and $\mathit{Reset}(B)$ are first-order predicates. For each binding $v\colon T$ in B where T is not an object type, each of these predicates contains a conjunct $\mathit{is}\$T(v)$. For those bindings where T is an object type, $\mathit{Types}(B)$ contains a conjunct $\mathit{is}\$T(v) \wedge \mathit{isDecl}(v, \mathit{alloc})$, and $\mathit{Reset}(B)$ contains a conjunct $v = \mathbf{nil}$.

For example, if B is the list of bindings $u\colon \mathbf{int}$, $v\colon T$, where T is some object type, then

$$\begin{array}{lcl} \mathit{Vars}(B) & = & u, v \\ \mathit{Types}(B) & = & \mathit{is}\$\mathbf{int}(u) \wedge \mathit{is}\$T(v) \wedge \mathit{isDecl}(v, \mathit{alloc}) \\ \mathit{Reset}(B) & = & \mathit{is}\$\mathbf{int}(u) \wedge v = \mathbf{nil} \end{array} .$$

For w a list of designator expressions, $\mathit{Fields}(w)$ is the list of data fields x occurring in some designator expression $x[E]$ in w. For example, if w is the list $x[E], y[E'], x[E'']$, then $\mathit{Fields}(w)$ is the list x, y.

For W a list of data fields, $\mathit{FieldTypes}(W)$ is a first-order predicate. For each data field x in W, if x is declared in the program by

$$\mathbf{field}\ x\colon T \to U \quad ,$$

then $\mathit{FieldTypes}(W)$ contains the conjunct $\mathit{field}\$T\$U(x)$. If U is an object type, $\mathit{FieldTypes}(W)$ also contains the conjunct $\mathit{isConsistent}(x, \mathit{alloc})$.

For W a list of data fields and v a variable, $\mathit{FieldReset}(W, v)$, too, is a first-order predicate. A field x in W contributes a conjunct to $\mathit{FieldReset}(W, v)$ only if the range type of x is an object type, in which case $\mathit{FieldReset}(W, v)$ contains the conjunct $x[v] = \mathbf{nil}$.

For example, suppose that, in the context of the program snippet (1) on page 8, W is the list $\mathit{val}, \mathit{next}$. Then, $\mathit{FieldTypes}(W)$ is

$$\mathit{field}\$\mathit{Node}\$\mathbf{int}(\mathit{val}) \wedge \mathit{field}\$\mathit{Node}\$\mathit{Node}(\mathit{next}) \wedge \mathit{isConsistent}(\mathit{next}, \mathit{alloc})$$

and $\mathit{FieldReset}(W, v)$ is

$$next[v] = \mathbf{nil} \qquad .$$

In the verification condition (2), $\mathit{InitialFields}$ is the predicate $\mathit{FieldTypes}(W)$, where W is the list of all data fields declared in the program.

3.3 Transforming expressions and specification predicates

This subsection defines two functions, Pr and Tr, which transform expressions into first-order logic.

Both Pr and Tr are defined inductively on the structure of their argument. Function Pr produces a first-order predicate, whereas Tr produces a first-order term. Only Pr is ever applied to a specification predicate.

I first describe Pr.

- Pr distributes over the operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, and $\equiv$.
- $Pr(\langle \mathbf{Q}\, \mathit{bindings} \mid R \triangleright P \rangle) = \langle \mathbf{Q}\, \mathit{vars} \mid \mathit{types} \wedge Pr(R) \triangleright Pr(P) \rangle$, where $\mathit{vars} = \mathit{Vars}(\mathit{bindings})$ and $\mathit{types} = \mathit{Types}(\mathit{bindings})$.
- $Pr(\mathbf{fresh}(E)) = (\neg \mathit{isDecl}(e, \mathit{alloc}_0) \wedge \mathit{isDecl}(e, \mathit{alloc}) \wedge e \neq \mathbf{nil})$, where $e = Tr(E)$.
- The following rules reduce the number of times the six special functions introduced in Section 3.1 have to be used.

$$\begin{array}{rcl}
Pr(E0 = E1) & = & (Tr(E0) = Tr(E1)) \\
Pr(E0 \neq E1) & = & (Tr(E0) \neq Tr(E1)) \\
Pr(E0 < E1) & = & (Tr(E0) < Tr(E1)) \\
Pr(E0 \leq E1) & = & (Tr(E0) \leq Tr(E1)) \\
Pr(E0 \geq E1) & = & (Tr(E0) \geq Tr(E1)) \\
Pr(E0 > E1) & = & (Tr(E0) > Tr(E1))
\end{array}$$

- For any other expression E, which necessarily is a boolean specification expression, $Pr(E) = (Tr(E) = \mathbf{true})$.

Function Tr is defined as follows.

- For any constant c (*i.e.*, $\mathbf{false}$, $\mathbf{true}$, $\mathbf{nil}$, or a numeric constant), $Tr(c) = c$.

- For any variable v , $Tr(v) = v$.
- Tr distributes over the arithmetic operators ($+$, $-$, $\cdot$, $\mathbf{div}$, and $\mathbf{mod}$).
- For any select expression (possibly with an initial-valued field), $Tr(x[E]) = x[Tr(E)]$.
- If the type of E is a subtype of T , then $Tr(\mathbf{narrow}(E, T)) = Tr(E)$. If T is $\mathbf{nat}$ and the type of E is $\mathbf{int}$, then $Tr(\mathbf{narrow}(E, T)) = natural(Tr(E))$. Otherwise, $Tr(\mathbf{narrow}(E, T)) = narrow(Tr(E), tc\$T)$.
- The comparison operators are transformed using the special functions.

$$
\begin{array}{lcl}
Tr(E0 = E1) & = & equal(Tr(E0), Tr(E1)) \\
Tr(E0 \neq E1) & = & not(equal(Tr(E0), Tr(E1))) \\
Tr(E0 < E1) & = & less(Tr(E0), Tr(E1)) \\
Tr(E0 \leq E1) & = & atmost(Tr(E0), Tr(E1)) \\
Tr(E0 \geq E1) & = & atmost(Tr(E1), Tr(E0)) \\
Tr(E0 > E1) & = & less(Tr(E1), Tr(E0))
\end{array}
$$

- Boolean operators are also transformed using the special functions.

$$
\begin{array}{lcl}
Tr(\neg E) & = & not(Tr(E)) \\
Tr(E0 \wedge E1) & = & and(Tr(E0), Tr(E1)) \\
Tr(E0 \vee E1) & = & or(Tr(E0), Tr(E1)) \\
Tr(E0 \Rightarrow E1) & = & or(not(Tr(E0)), Tr(E1)) \\
Tr(E0 \Leftarrow E1) & = & or(Tr(E0), not(Tr(E1))) \\
Tr(E0 \equiv E1) & = & equal(Tr(E0), Tr(E1))
\end{array}
$$

Section 4.3 gives an example involving Pr and Tr .

3.4 Definedness of expressions

This subsection defines a function *Defined* from expressions to first-order predicates. The predicate $Defined(E)$ holds in those states in which the evaluation of E does not result in a run-time error. *Defined* is used only with expressions that are evaluated at run-time.

Function $\mathit{Defined}$ is defined inductively on the structure of expressions. It yields true for constants and variables. For other expressions, except as noted below, $\mathit{Defined}$ is the conjunction of $\mathit{Defined}$ on each subexpression.

$$\begin{array}{lcl}
\mathit{Defined}(x[E]) & = & \mathit{Defined}(E) \wedge \mathit{Tr}(E) \neq \mathbf{nil} \\
\mathit{Defined}(E \,\mathbf{div}\, E') & = & \mathit{Defined}(E) \wedge \mathit{Defined}(E') \wedge \mathit{Tr}(E') \neq \mathit{0} \\
\mathit{Defined}(E \,\mathbf{mod}\, E') & = & \mathit{Defined}(E) \wedge \mathit{Defined}(E') \wedge \mathit{Tr}(E') \neq \mathit{0} \\
\mathit{Defined}(E \wedge E') & = & \mathit{Defined}(E) \wedge (\neg \mathit{Tr}(E) \vee \mathit{Defined}(E')) \\
\mathit{Defined}(E \vee E') & = & \mathit{Defined}(E) \wedge (\mathit{Tr}(E) \vee \mathit{Defined}(E')) \\
\mathit{Defined}(E \Rightarrow E') & = & \mathit{Defined}(E) \wedge (\neg \mathit{Tr}(E) \vee \mathit{Defined}(E')) \\
\mathit{Defined}(E \Leftarrow E') & = & \mathit{Defined}(E) \wedge (\mathit{Tr}(E) \vee \mathit{Defined}(E'))
\end{array}$$

For $\mathbf{narrow}$ expressions,

$$\mathit{Defined}(\mathbf{narrow}(E, T)) = \mathit{Defined}(E)$$

if the static type of E is a subtype of T, and otherwise

$$\mathit{Defined}(\mathbf{narrow}(E, T)) = \mathit{Defined}(E) \wedge \mathit{is\$T}(\mathit{Tr}(E)) \qquad .$$

The axiomatic semantics of Ecstatic gives the first-order logic some freedom in its approach to dealing with undefined expressions. One approach is to treat an undefined expression as an uninterpreted function of its subexpressions, but other approaches are also possible.

3.5 Weakest liberal preconditions

This subsection defines the semantics of each command by giving its weakest liberal precondition, wlp. Throughout this subsection, I use R as any arbitrary first-order predicate.

The wlp of the simple assignment statement $v := E$ is defined by

$$\mathit{wlp}.(v := E).R = \mathit{Defined}(E) \wedge R[v := e] \qquad ,$$

where $e = \mathit{Tr}(E)$. This says that to guarantee that the command $v := E$ ends in a state satisfying R (if the command terminates at all) without incurring any run-time error, the command must be started in a state in which E is defined and R with free occurrences of variable v replaced by e holds.

For the allocation statement, we have

$$\begin{array}{l} wlp.(v := \mathbf{new}(T)).R \ = \\ \quad \langle \forall\, v, alloc' \mid \; typecode(v) = tc\$T \wedge v \neq \mathbf{nil} \wedge FieldReset(W, v) \wedge \\ \qquad\qquad\qquad\quad \neg isDecl(v, alloc) \wedge succeeds(alloc', alloc) \wedge \\ \qquad\qquad\qquad\quad \langle \forall\, t \triangleright isDecl(t, alloc') \equiv isDecl(t, alloc) \vee t = v \,\rangle \triangleright \\ \qquad\qquad R[alloc := alloc'] \,\rangle \end{array} ,$$

where W is the list of all data fields whose index type is a supertype of T. Viewed operationally, this says that **new** is free to "pick" any non-**nil**, unallocated object whose allocated type is T and whose fields have proper initial values, and such an object is always assumed to exist. (Note that a compiler writer may choose to implement **new** by allocating a sufficient amount of memory to hold the new object and then setting the fields to proper initial values. Such an implementation is allowed by the semantics given here, since the difference is not observable to programs.)

The update command is defined by

$$\begin{array}{l} wlp.(x[E] := E').R \ = \\ \qquad Defined(E) \wedge Defined(E') \wedge e \neq \mathbf{nil} \wedge R[x := store(x, e, e')] \end{array} ,$$

where $e = Tr(E)$ and $e' = Tr(E')$.

The block statement is defined by

$$wlp.(\mathbf{var}\ bindings\ \mathbf{in}\ S\ \mathbf{end}).R \ = \ \langle \forall\, vars \mid \; reset \triangleright wlp.S.R \,\rangle \qquad ,$$

where $vars = Vars(bindings)$ and $reset = Reset(bindings)$.

Composition is defined as follows.

$$wlp.(S\ ;\ S').R \ = \ wlp.S.(wlp.S'.R)$$

The conditional statement is defined by

$$\begin{array}{l} wlp.(\mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S'\ \mathbf{fi}).R \ = \\ \qquad Defined(E) \wedge (g \Rightarrow wlp.S.R) \wedge (\neg g \Rightarrow wlp.S'.R) \end{array} \qquad ,$$

where $g = Pr(E)$.

Finally, an invocation of a method m is defined in terms of the specification of m. I will show the wlp for the invocation of a method that takes two in- and two out-parameters. Generalizations to other numbers of parameters is straightforward. If a method m is declared by

$$\begin{array}{l} \mathbf{method}\ u0\colon U0,\ u1\colon U1 := m(t0\colon T0,\ t1\colon T1) \\ \quad \mathbf{requires}\ Pre \\ \quad \mathbf{modifies}\ w \\ \quad \mathbf{ensures}\ Post \end{array} \qquad ,$$

then we have

$$
\begin{array}{l}
wlp.(v0, v1 := m(E0, E1)).R = \\
\quad Defined(E0) \wedge Defined(E1) \wedge \\
\quad \langle \forall\, t0, t1 \mid t0 = Tr(E0) \wedge t1 = Tr(E1) \triangleright \\
\qquad t0 \neq \textbf{nil} \wedge pre \wedge \\
\qquad \langle \forall\, u0, u1, W, alloc \mid types \wedge fieldtypes \wedge succeeds(alloc, alloc_0) \wedge \\
\qquad\qquad\qquad\qquad\qquad post \wedge Q \triangleright \\
\qquad\qquad\qquad R[v0, v1 := u0, u1] \,\rangle [W_0, alloc_0 := W, alloc] \\
\quad \rangle \quad ,
\end{array}
$$

where

- $pre = Pr(Pre)$ and $post = Pr(Post)$,
- W is $Fields(w)$, and W_0 is W with every identifier initial-valued,
- $types = Types(u0{:}\ U0,\ u1{:}\ U1)$, and $fieldtypes = FieldTypes(W)$, and
- $Q = PostCondContrib(W, w)$.

Note the conjunct $t0 \neq \textbf{nil}$ which enforces that the actual self parameter isn't **nil** at the time of the method invocation.

I will give an example application of this rule in Section 4.2.

3.6 Postcondition contributions from modifies lists

This subsection defines the predicate $PostCondContrib(Y, w)$ for a list of designator expressions w and a list of data fields Y. Informally, the predicate, which relates a post-state and a pre-state, expresses that every data field y in Y differs from y_0 only at newly allocated objects and objects that are listed as indices of y in w.

Predicate $PostCondContrib(Y, w)$ contains a conjunct $pcc(y, w, Y)$ for each data field y in Y. For a data field y whose index type is T, the macro $pcc(y, w, Y)$ is defined as

$$\langle \forall\, s \mid is\$T(s) \wedge isDecl(s, alloc_0) \triangleright y_0[s] = y[s] \vee IsModPoint(y, w, s, Y) \,\rangle,$$

where s is an identifier not occurring in the program. Macro $IsModPoint(y, w, s, Y)$ yields a disjunct

$$s = e[Y := Y_0]$$

for every designator expression $y[E]$ in w, where $e = Tr(E)$ and Y_0 is Y with every identifier initial-valued.

For example, consider a program that contains that following declarations:

$$\begin{array}{l} \textbf{type } T \\ \textbf{field } \mathit{val}\colon T \to \textbf{int} \\ \textbf{field } \mathit{left}, \mathit{right}, \mathit{parent}\colon T \to T \end{array} \qquad .$$

Let w denote the list

$$\mathit{left}[t],\ \mathit{val}[\mathit{left}[t]],\ \mathit{val}[\mathit{right}[t]] \qquad ,$$

and let Y denote the list val, left, parent. It may help to think of w as being the modifies list of a method and Y as being the data fields that occur on the left-hand side of update commands in an implementation. Predicate $\mathit{PostCondContrib}(Y, w)$ is then the conjunction

$$\mathit{pcc}(\mathit{val}, w, Y) \wedge \mathit{pcc}(\mathit{left}, w, Y) \wedge \mathit{pcc}(\mathit{parent}, w, Y) \qquad ,$$

which expands to

$$\begin{array}{l} \langle \forall s \mid \mathit{is\$T}(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \\ \qquad\qquad \mathit{val}_0[s] = \mathit{val}[s] \vee \mathit{IsModPoint}(\mathit{val}, w, s, Y) \rangle \wedge \\ \langle \forall s \mid \mathit{is\$T}(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \\ \qquad\qquad \mathit{left}_0[s] = \mathit{left}[s] \vee \mathit{IsModPoint}(\mathit{left}, w, s, Y) \rangle \wedge \\ \langle \forall s \mid \mathit{is\$T}(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \\ \qquad\qquad \mathit{parent}_0[s] = \mathit{parent}[s] \vee \mathit{IsModPoint}(\mathit{parent}, w, s, Y) \rangle \end{array} \quad ,$$

which in turn expands to

$$\begin{array}{l} \langle \forall s \mid \mathit{is\$T}(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \\ \qquad\qquad \mathit{val}_0[s] = \mathit{val}[s] \vee s = \mathit{left}_0[t] \vee s = \mathit{right}[t] \rangle \wedge \\ \langle \forall s \mid \mathit{is\$T}(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \mathit{left}_0[s] = \mathit{left}[s] \vee s = t \rangle \wedge \\ \langle \forall s \mid \mathit{is\$T}(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \mathit{parent}_0[s] = \mathit{parent}[s] \rangle \end{array} \qquad .$$

By conjoining this predicate to the postcondition, the method implementation is constrained to modify val, left, and parent only at the objects specified in the modifies list and at any newly allocated object.

3.7 Targets

This subsection defines $\mathit{Targets}$, a function from a command to a list of data fields. Intuitively, a data field x is in $\mathit{Targets}(S)$, and is thus said to be a *target* of S, if x is possibly changed by an execution of S, as determined by a simple inspection of the program text.

Function $\mathit{Targets}$ is defined inductively on the structure of its argument. For a statement composition S, $\mathit{Targets}(S)$ is the union of the targets of the command components of S. If S is an update command $x[E] := E'$, $\mathit{Targets}(S)$ is the singleton list x. If S is an invocation of a method whose modifies list is w, $\mathit{Targets}(S)$ is the list $\mathit{Fields}(w)$. For all other commands S, $\mathit{Targets}(S)$ is the empty list.

The verification condition (2) on page 18 mentions $\mathit{Targets}(S)$ so that updates of the targets of S are constrained to the updates allowed by the modifies list and updates at newly allocated objects. Data fields that are not targets of S cannot possibly be changed by S, and thus they need not be constrained explicitly as a postcondition contribution.

Well, there is one exception to the claim that data fields that aren't targets cannot be modified: S may invoke methods, and method implementations are allowed to modify data fields at newly allocated objects without getting explicit permission from a modifies list. However, an object considered newly allocated with respect to a nested method invocation is also considered newly allocated with respect to S. Hence, S is allowed the modifications of newly allocated objects done by nested method invocations, so such modifications need not be constrained explicitly as postcondition contributions.

4 Examples

This section gives some additional program examples and discusses their verification conditions.

The first example shows the background predicate for a small program. The second example gives a proof that a simple method implementation meets its specification. The third example contrasts the wlp generated for the implementation of a method with the wlp generated for an invocation of the method. The fourth example shows the interplay between boolean program values and first-order predicates. The final example illustrates the renaming of identifiers to unique ones.

4.0 An example background predicate

This subsection shows the background predicate that results from the declarations (1) on page 8.

Regardless of the program at hand, the background predicate contains the axioms

$$(3) \wedge (6) \wedge (7) \wedge (9) \wedge (10) \wedge (11) \wedge (12) \wedge (15) \wedge (16) \quad . \tag{17}$$

In addition, we need to instantiate some axiom schemas according to the types and fields declared in the program.

Program (1) declares one type, $Node$. Like the built-in object type $\mathbf{obj}$, $Node$ gives rise to a typecode. Axiom schema (4) says that the typecodes are distinct. Applied to the program at hand, we get

$$tc\$\mathbf{obj} \neq tc\$Node \quad . \tag{18}$$

According to axiom schema (5), the declaration of type $Node$ also gives rise to a $subtype1$ relation:

$$subtype1(tc\$Node, tc\$\mathbf{obj}) \quad . \tag{19}$$

The type predicates for types $\mathbf{obj}$ and $Node$, as dictated by axiom schema (8), are defined by

$$\begin{array}{l} \langle \forall\, t \triangleright is\$\mathbf{obj}(t) \;\equiv\; t = \mathbf{nil} \vee subtype(typecode(t), tc\$\mathbf{obj}) \rangle \\ \langle \forall\, t \triangleright is\$Node(t) \;\equiv\; t = \mathbf{nil} \vee subtype(typecode(t), tc\$Node) \rangle \quad . \end{array} \tag{20}$$

Program (1) declares two fields, val and $next$. The types of these give rise to field-type predicate, about which the following axioms are generated, by axiom schema (13):

$$\begin{array}{l} \langle \forall\, x, t \triangleright field\$Node\$\mathbf{int}(x) \wedge is\$Node(t) \wedge t \neq \mathbf{nil} \Rightarrow is\$\mathbf{int}(x[t]) \rangle \\ \langle \forall\, x, t \triangleright field\$Node\$Node(x) \wedge is\$Node(t) \wedge t \neq \mathbf{nil} \Rightarrow is\$Node(x[t]) \rangle \,. \end{array} \tag{21}$$

Finally, since the range type of the field $next$ is an object type, axiom schema (14) stipulates that the background predicate contain the axiom

$$\begin{array}{l} \langle \forall\, x, a, t \triangleright field\$Node\$Node(x) \wedge isConsistent(x, a) \wedge \\ \qquad\qquad is\$Node(t) \wedge t \neq \mathbf{nil} \wedge isDecl(t, a) \\ \qquad\qquad \Rightarrow isDecl(x[t], a) \rangle \quad . \end{array} \tag{22}$$

In summary, the background predicate for program (1) is the conjunction of (17), (18), (19), (20), (21), and (22). The next example shows how parts of the background predicate are used in the proof of a method implementation.

4.1 The proof of a simple program

In this example, I prove that the implementation of method m in program (0) on page 2 meets its specification.

I start by writing down the antecedent of the verification condition (2):

$$\begin{array}{lll}
\mathit{BackgroundPred} & : & \text{omitted (for an example, see previous subsection)} \\
\mathit{InitialFields} & : & \mathit{field}\$T\$\mathbf{int}(x) \wedge \mathit{field}\$T\$\mathbf{nat}(y) \\
Pr(P') & : & 0 \leq y[t] + z \\
Y_0 = Y & : & x_0 = x \\
\mathit{alloc}_0 = \mathit{alloc} & : & \mathit{alloc}_0 = \mathit{alloc} \\
\mathit{self} \neq \mathbf{nil} & : & t \neq \mathbf{nil} \\
\mathit{Types}(\mathit{formal\text{-}ins}') & : & \mathit{is}\$T(t) \wedge \mathit{isDecl}(t, \mathit{alloc}) \wedge \mathit{is}\$\mathbf{int}(z) \\
\mathit{Reset}(\mathit{formal\text{-}outs}') & : & \mathit{true} \quad .
\end{array}$$

The consequent of the verification condition is

$$\mathit{wlp}.(x[t] := x[t] + 2 \cdot y[t] + z).(Pr(R') \wedge Q) \qquad , \tag{23}$$

where $Pr(R')$ is simply the given postcondition

$$x[t] \geq x_0[t] + y[t] + z$$

and Q is $\mathit{PostCondContrib}(x, x[t])$ which is

$$\langle \forall s \mid \mathit{is}\$T(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright x_0[s] = x[s] \vee s = t \rangle \qquad .$$

Applying the wlp for the update statement, (23) becomes

$$\begin{array}{l}
\mathit{true} \wedge (t \neq \mathbf{nil} \wedge t \neq \mathbf{nil}) \wedge t \neq \mathbf{nil} \wedge \\
\mathit{store}(x, t, x[t] + 2 \cdot y[t] + z)[t] \geq x_0[t] + y[t] + z \wedge \\
\langle \forall s \mid \mathit{is}\$T(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \\
\qquad\qquad x_0[s] = \mathit{store}(x, t, x[t] + 2 \cdot y[t] + z)[s] \vee s = t \rangle \qquad .
\end{array} \tag{24}$$

We are now ready to embark on a calculation to discharge the verification condition. Under the antecedent of the verification condition, we massage the consequent (24) of the verification condition in the following calculation.

$$\begin{array}{ll}
& \mathit{true} \wedge (t \neq \mathbf{nil} \wedge t \neq \mathbf{nil}) \wedge t \neq \mathbf{nil} \wedge \\
& \mathit{store}(x, t, x[t] + 2 \cdot y[t] + z)[t] \geq x_0[t] + y[t] + z \wedge \\
& \langle \forall s \mid \mathit{is}\$T(s) \wedge \mathit{isDecl}(s, \mathit{alloc}_0) \triangleright \\
& \qquad\qquad x_0[s] = \mathit{store}(x, t, x[t] + 2 \cdot y[t] + z)[s] \vee s = t \rangle \\
= & \qquad \{ \text{ from the antecedent, } t \neq \mathbf{nil} \text{ and } x_0 = x \ \}
\end{array}$$

$$
\begin{array}{ll}
& store(x, t, x[t] + 2 \cdot y[t] + z)[t] \geq x[t] + y[t] + z \ \wedge \\
& \langle \forall\, s \mid is\$T(s) \ \wedge\ isDecl(s, alloc_0) \triangleright \\
& \qquad\qquad x[s] = store(x, t, x[t] + 2 \cdot y[t] + z)[s] \ \vee\ s = t \,\rangle \\
= & \quad \{ \text{ logic } \} \\
& store(x, t, x[t] + 2 \cdot y[t] + z)[t] \geq x[t] + y[t] + z \ \wedge \\
& \langle \forall\, s \mid is\$T(s) \ \wedge\ isDecl(s, alloc_0) \triangleright \\
& \qquad\qquad s \neq t \ \Rightarrow\ x[s] = store(x, t, x[t] + 2 \cdot y[t] + z)[s] \,\rangle \\
= & \quad \{ \text{ the two select-of-store axioms (3) } \} \\
& x[t] + 2 \cdot y[t] + z \geq x[t] + y[t] + z \ \wedge \\
& \langle \forall\, s \mid is\$T(s) \ \wedge\ isDecl(s, alloc_0) \triangleright s \neq t \ \Rightarrow\ x[s] = x[s] \,\rangle \\
= & \quad \{ \text{ logic, and arithmetic } \} \\
& y[t] \geq 0 \\
= & \quad \{ \text{ background predicate (9) } \} \\
& is\$\mathbf{nat}(y[t]) \\
\Leftarrow & \quad \{ \text{ background predicate (13) } \} \\
& field\$T\$\mathbf{nat}(y) \ \wedge\ is\$T(t) \ \wedge\ t \neq \mathbf{nil} \\
= & \quad \{ \text{ antecedent } \} \\
& true
\end{array}
$$

And with that calculation, we have discharged the proof obligation.

4.2 Implementation versus invocation

In this example, I contrast the proof obligation for a method implementation with the proof obligation for invoking the method.

In the context of the linked list declarations (1) on page 8, consider the following method declaration and implementation.

$$
\begin{array}{l}
\mathbf{method}\ n\colon Node := prepend(node\colon Node,\ value\colon \mathbf{int}) \\
\quad \mathbf{ensures}\ \mathbf{fresh}(n) \wedge next[n] = node \wedge val[n] = value \\
\mathbf{impl}\ n\colon Node := prepend(node\colon Node,\ value\colon \mathbf{int})\ \mathbf{is} \\
\quad n := \mathbf{new}(Node)\ ; \\
\quad next[n] := node\ ; \\
\quad val[n] := value
\end{array}
$$

This method prepends $value$ to the given list $node$ and returns the new head of the list n. Note that the method has an empty modifies list, yet the implementation modifies the $next$ and val fields of n. This is permitted, because n is allocated inside the method implementation. To see how this rule manifests itself in the formal semantics, let's take a look at the wlp of the method implementation and the wlp of an invocation of the method.

The consequent of a verification condition has the form

$$wlp.S.(Pr(P') \land Q) \qquad .$$

In this example, $Pr(P')$ differs from the given postcondition because of the use of $\mathbf{fresh}$. Predicate $Pr(P')$ is

$$\begin{array}{l} \neg isDecl(n, alloc_0) \land isDecl(n, alloc) \land n \neq \mathbf{nil} \land \\ next[n] = node \land val[n] = value \qquad . \end{array}$$

The postcondition contribution Q, that is $PostCondContrib(Y, w')$, is more interesting. The modifies list w' is empty, and Y, which considers the targets of the method implementation, is the list $next, val$. Expanding the postcondition contribution yields

$$\begin{array}{l} \langle \forall s \mid is\$Node(s) \land isDecl(s, alloc_0) \triangleright next_0[s] = next[s] \rangle \land \\ \langle \forall s \mid is\$Node(s) \land isDecl(s, alloc_0) \triangleright val_0[s] = val[s] \rangle \qquad . \end{array}$$

Applying the appropriate wlp to $Pr(P') \land Q$ gives the formula

$$\begin{array}{l} \langle \forall n, alloc' \mid typecode(n) = tc\$Node \land n \neq \mathbf{nil} \land next[n] = \mathbf{nil} \land \\ \qquad\qquad \neg isDecl(n, alloc) \land succeeds(alloc', alloc) \land \\ \qquad\qquad \langle \forall t \triangleright isDecl(t, alloc') \equiv isDecl(t, alloc) \lor t = n \rangle \triangleright \\ \quad \neg isDecl(n, alloc_0) \land isDecl(n, alloc') \land n \neq \mathbf{nil} \land \\ \quad store(next, n, node)[n] = node \land store(val, n, value)[n] = value \land \\ \quad \langle \forall s \mid is\$Node(s) \land isDecl(s, alloc_0) \triangleright \\ \qquad\qquad next_0[s] = store(next, n, node)[s] \rangle \land \\ \quad \langle \forall s \mid is\$Node(s) \land isDecl(s, alloc_0) \triangleright \\ \qquad\qquad val_0[s] = store(val, n, value)[s] \rangle\rangle \qquad . \end{array}$$

Under the range of this quantification and the antecedent of the verification condition, we embark on a calculation on the term of the quantification:

$$\begin{array}{ll} & \neg isDecl(n, alloc_0) \land isDecl(n, alloc') \land n \neq \mathbf{nil} \land \\ & store(next, n, node)[n] = node \land store(val, n, value)[n] = value \land \\ & \langle \forall s \mid is\$Node(s) \land isDecl(s, alloc_0) \triangleright \\ & \qquad next_0[s] = store(next, n, node)[s] \rangle \land \\ & \langle \forall s \mid is\$Node(s) \land isDecl(s, alloc_0) \triangleright \\ & \qquad val_0[s] = store(val, n, value)[s] \rangle \\ = & \quad \{ \ alloc_0 = alloc, \neg isDecl(n, alloc), isDecl(n, alloc'), n \neq \mathbf{nil} \ \} \end{array}$$

$$
\begin{array}{ll}
& store(next, n, node)[n] = node \;\wedge\; store(val, n, value)[n] = value \;\wedge \\
& \langle \forall\, s \;\mathbf{|}\; is\$Node(s) \;\wedge\; isDecl(s, alloc_0) \;\triangleright \\
& \qquad\qquad next_0[s] = store(next, n, node)[s] \;\rangle \;\wedge \\
& \langle \forall\, s \;\mathbf{|}\; is\$Node(s) \;\wedge\; isDecl(s, alloc_0) \;\triangleright \\
& \qquad\qquad val_0[s] = store(val, n, value)[s] \;\rangle \\
= & \qquad \{ \;\; \text{the first select-of-store axiom (3)} \;\; \} \\
& \langle \forall\, s \;\mathbf{|}\; is\$Node(s) \;\wedge\; isDecl(s, alloc_0) \;\triangleright \\
& \qquad\qquad next_0[s] = store(next, n, node)[s] \;\rangle \;\wedge \\
& \langle \forall\, s \;\mathbf{|}\; is\$Node(s) \;\wedge\; isDecl(s, alloc_0) \;\triangleright \\
& \qquad\qquad val_0[s] = store(val, n, value)[s] \;\rangle \\
\Leftarrow & \qquad \{ \;\; next_0 = next \text{ , } val_0 = val \text{ , the second select-of-store axiom (3)} \;\; \} \\
& \langle \forall\, s \;\mathbf{|}\; is\$Node(s) \;\wedge\; isDecl(s, alloc_0) \;\triangleright\; n \neq s \;\rangle \\
= & \qquad \{ \;\; \text{logic} \;\; \} \\
& \langle \forall\, s \;\mathbf{|}\; is\$Node(s) \;\wedge\; n = s \;\triangleright\; \neg isDecl(s, alloc_0) \;\rangle \\
= & \qquad \{ \;\; alloc_0 = alloc \text{ , and } \neg isDecl(n, alloc) \;\; \} \\
& true \qquad .
\end{array}
$$

So much for the implementation. Now let us consider an invocation of the method:

$$head := prepend(head, 5) \qquad ,$$

and let us construct the wlp of this command with respect to some first-order predicate R . According to the definition of wlp for method invocation, we get

$$
\begin{array}{l}
true \wedge true \wedge \\
\langle \forall\, node, value \;\mathbf{|}\; node = head \wedge value = 5 \;\triangleright \\
\qquad node \neq \mathbf{nil} \wedge true \wedge \\
\qquad \langle \forall\, n, alloc \;\mathbf{|}\; (is\$Node(n) \wedge isDecl(n, alloc)) \wedge true \wedge \\
\qquad\qquad\qquad succeeds(alloc, alloc_0) \wedge \\
\qquad\qquad\qquad \neg isDecl(n, alloc_0) \wedge isDecl(n, alloc) \wedge n \neq \mathbf{nil} \wedge \\
\qquad\qquad\qquad next[n] = node \wedge val[n] = value \wedge \\
\qquad\qquad\qquad true \;\triangleright \\
\qquad\qquad R[head := n] \;\rangle[alloc_0 := alloc] \\
\rangle \qquad .
\end{array}
$$

The formula can be simplified, especially if one renames the dummy variable $alloc$ to, say, $alloc'$ (because then the substitution $[alloc_0 := alloc]$ can be applied directly):

$$\begin{array}{l} head \neq \mathbf{nil} \wedge \\ \langle \forall\, n, alloc' \mid is\$Node(n) \wedge isDecl(n, alloc') \wedge succeeds(alloc', alloc) \wedge \\ \qquad \neg isDecl(n, alloc) \wedge isDecl(n, alloc') \wedge n \neq \mathbf{nil} \wedge \\ \qquad next[n] = head \wedge val[n] = 5 \ \triangleright \\ \quad R[head := n] \ \rangle \end{array} \quad .$$

The thing to notice here is that the postcondition contribution is simply $true$, because the modifies list is empty—the targets of the implementation are not taken into account. Thus, as far as the caller can tell, the method invocation does not change the $next$ and val fields of any object, but instead just allocates an object whose fields happen to have the desired values. The fact that the implementation allocates an object and sets the fields of that object to appropriate values cannot be observed by the caller.

4.3 Boolean values and predicates

Section 3.1 discussed the difference between boolean program values (like $\mathbf{true}$) and first-order predicates (like $true$), and introduced some special functions. This subsection gives an example to show the impact of that discussion on verification conditions.

Consider the command

$$b[t] := x < y \quad ,$$

where b is a boolean-valued field, x and y are integer variables, and t is a variable of an appropriate object type. The wlp of this command with respect to a predicate R is

$$R[b := store(b, t, less(x, y))] \quad .$$

The use of $less$ is necessary because the arguments of functions, like $store$, are expected to be terms of the logic. Therefore, the wlp of the update command makes use of function Tr.

Contrast the command above with the command

$$\mathbf{if}\ x < y\ \mathbf{then}\ S\ \mathbf{else}\ S'\ \mathbf{fi} \quad ,$$

where x and y are the same variables as in the command above, and S and S' are commands. The wlp of this command is

$$(x < y \Rightarrow wlp.S.R) \wedge (\neg(x < y) \Rightarrow wlp.S'.R) \quad .$$

Here, it is possible to use the first-order predicate $x < y$ directly, rather than using $less(x, y) = \mathbf{true}$, and function Pr, which is used by the wlp for **if**, takes advantage of this possibility.

4.4 Preprocessing names

In this final example, I show an artificial program to illustrate the details of dealing with names.

Consider the obscure program

$$
\begin{array}{l}
\textbf{type}\ T \\
\textbf{field}\ x\colon T \to \textbf{bool} \\
\textbf{type}\ U <: T \\
\textbf{field}\ x\colon U \to \textbf{int} \\
\textbf{method}\ x(t\colon T,\ x\colon \textbf{int}) \\
\quad \textbf{modifies}\ x[t] \\
\textbf{impl}\ x(u\colon U,\ x\colon \textbf{int})\ \textbf{is} \\
\quad \textbf{var}\ t\colon T\ \textbf{in} \\
\qquad t := u\ ; \\
\qquad x[t] := x < x[u] \\
\quad \textbf{end} \qquad .
\end{array}
$$

Note that types, fields, methods, and variables live in four different name spaces. The use of a name in a program reveals to which of these four categories the name belongs. For example, field names occur left of the " [" in select expressions, and method names to the left of the first " (" in method invocation commands.

To show how the names in this program are resolved, I rename every identifier x to a unique name of the form $x.m.n$, where m and n are the line and column at which x is declared. The program then becomes:

$$
\begin{array}{l}
\textbf{type}\ T.0.5 \\
\textbf{field}\ x.1.6\colon T.0.5 \to \textbf{bool} \\
\textbf{type}\ U.2.5 <: T.0.5 \\
\textbf{field}\ x.3.6\colon U.2.5 \to \textbf{int} \\
\textbf{method}\ x.4.7(t.4.9\colon T.0.5, x.4.15\colon \textbf{int}) \\
\quad \textbf{modifies}\ x.1.6[t.4.9] \\
\textbf{impl}\ x.4.7(u.6.7\colon U.2.5, x.6.13\colon \textbf{int})\ \textbf{is} \\
\quad \textbf{var}\ t.7.6\colon T.0.5\ \textbf{in} \\
\qquad t.7.6 := u.6.7\ ; \\
\qquad x.1.6[t.7.6] := x.6.13 < x.3.6[u.6.7] \\
\quad \textbf{end} \qquad .
\end{array}
$$

(I omitted such renamings in the previous examples to avoid clutter.)

The consequent of the verification condition for this implementation is

$$\begin{array}{l}
\langle \forall\, t.7.6 \;\mid\; t.7.6 = \mathbf{nil} \;\triangleright \\
\quad (t.7.6 \neq \mathbf{nil} \;\wedge \\
\quad\; \langle \forall\, s \;\mid\; is\$T.0.5(s) \wedge isDecl(s, alloc_0) \;\triangleright \\
\qquad\qquad x.1.6_0[s] = x.1.6[s] \;\vee\; s = u.6.7 \,\rangle \\
\quad)[x.1.6 := store(x.1.6, t.7.6, less(x.6.13, x.3.6[u.6.7]))] \\
\quad\; [t.7.6 := u.6.7] \\
\rangle \quad ,
\end{array}$$

which simplifies to

$$\begin{array}{l}
u.6.7 \neq \mathbf{nil} \;\wedge \\
\langle \forall\, s \;\mid\; is\$T.0.5(s) \wedge isDecl(s, alloc_0) \;\triangleright \\
\qquad x.1.6_0[s] = store(x.1.6, u.6.7, less(x.6.13, x.3.6[u.6.7]))[s] \\
\qquad \vee\; s = u.6.7 \,\rangle \quad .
\end{array}$$

Note that all name resolution is done using static types.

5 Language extensions

This section discusses some possible extensions to the language.

In addition to objects, Ecstatic provides booleans and integers. One can easily extend the language to accommodate other simple types, such as real numbers. To do so, the logical theory underlying the data type may need to be included. A very useful data type is (open) arrays, which can be added to the language as described in KRML 55 [9].

A useful shorthand that can easily be defined is a block statement where variables list their initial values. For example,

$$\mathbf{var}\; x\!: T := E \;\mathbf{in}\; S \;\mathbf{end} \tag{25}$$

would be a shorthand for

$$\mathbf{var}\; x\!: T \;\mathbf{in}\; x := E \,;\, S \;\mathbf{end} \quad .$$

If the type of E is T, then “ : T ” can be omitted from (25).

Ecstatic does not feature iteration. Instead, a program must resort to using recursion. However, an iterative construct can be added to the language. Such an addition would have to include a convenient syntax for expressing loop invariants, and the verification condition generation would have to be changed accordingly.

Ecstatic allows expressions to be partial, but does not allow them to have side effects. Convenient expressions with side effects include invocations of $\mathbf{new}$ and invocations of methods with exactly one out-parameter. Also, allowing a statement that combines the update statement with method invocation is often convenient. These conveniences can be defined as shorthands in terms of the present language. The following example conveys the idea: the statement

$$y[E] := m(\mathbf{new}(T), n(E', E''))$$

would be a shorthand for

$$\begin{array}{l} \mathbf{var}\ t0 \colon T0,\ t1 \colon T1,\ t2 \colon T2,\ t3 \colon T3\ \mathbf{in} \\ \quad t0 := E \ ;\ t1 := \mathbf{new}(T) \ ;\ t2 := n(E', E'') \ ; \\ \quad t3 := m(t1, t2) \ ;\ y[t0] := t3 \\ \mathbf{end} \qquad , \end{array}$$

where $T0$, $T1$, $T2$, and $T3$ are appropriate types. Defining shorthands like these is a clerical task. The only possibly non-trivial part is making sure the second operand of short-circuit boolean operators is evaluated conditionally.

Ecstatic provides methods, but not regular procedures. Syntactically, methods require at least one in-parameter (the self parameter), which must be of an object type; procedures have no such requirement. Operationally, the allocated type of the first actual in-parameter determines which method implementation is called, whereas a procedure has one implementation that is used for every invocation of the procedure. Axiomatically, the only difference between methods and procedures is that a method has an implicit precondition $self \neq \mathbf{nil}$.

Now to the major issue. It is fair to ask: "Isn't it boring that a method can be given only one specification?". Yes, it is. As things stand, the language is mostly like a programming language with procedures—not much is gained from object types and inheritance. For example, in the linked-list example in Section 1.4, I showed two different Op subtypes, each giving its own implementation of method $apply$. Each subtype is allowed to modify the r field in its own way. However, a subtype is not allowed to introduce new fields and modify them, unless the modifies list of $apply$ were changed to permit such modifications.

It would be possible to allow a subtype to weaken the precondition and strengthen the postcondition of a method. However, the big problem is that it seems desirable to let a subtype enlarge the modifies list of a method, so that the subtype can modify the additional fields it introduces. This problem can be solved by *data abstraction* and abstractional *dependencies* [10], which can be layered on top of Ecstatic. Such an endeavor includes allowing the declaration of abstract data fields that are functions of concrete ones, and appropriately rewriting the abstract fields that occur in pre- and postconditions and modifies list. (To get an idea of how this is done in ESC, see the ESC home page [0] or my thesis [10].) Once that is done, the semantics of the language remains unchanged.

6 Summary and conclusion

I have described a simple language for writing object-oriented programs. The language uses four kinds of declarations: types, data fields, methods specifications, and method implementations. The language is different from most other object-oriented programming languages in several ways. For example, types are declared separately from data fields and methods—knowing the complete set of data fields and methods is necessary when laying out the data structures of a compiled program in memory, but is not necessary to describe the language and its semantics. Also, compared to many common notions of subtyping (see, for example, Liskov and Wing [11]), subtyping in Ecstatic is very simple: it is simply an ordering among the names of types.

More importantly, the present language is given a precise axiomatic semantics. The semantics shows how object-oriented features like data fields, subtyping, aliasing, methods, modifies lists in the presence of objects, and **new** allocations are handled.

I have shown some examples that demonstrate some of the basic features of the language. It should be clear that the language is powerful enough to write interesting programs, but because of the lack of data abstraction in the specification language, many such programs cannot be specified and verified using the rules given here. Data abstraction can be layered on top of Ecstatic. However, the present paper contains enough novelties that it would be inappropriate to describe data abstraction here at the same time.

Acknowledgements

I'm grateful to Dave Detlefs, Allan Heydon, Greg Nelson, Jim Saxe, Raymie Stata, and Mark Vandevoorde for valuable comments on different versions of this paper. The name of the language was inspired by a news group posting by Lance Berc [2].

References

[0] Extended Static Checking home page, Digital Equipment Corporation, Systems Research Center. On the Web at `http://www.research.digital.com/SRC/esc/Esc.html`.

[1] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. To appear in the proceedings of TAPSOFT/FASE'97, 1997.

[2] Lance Berc. Re: Angles and angels. Posting to `src.writing` by `berc@pa.dec.com`, 13 April 1995, 23:33:48 -0700, Digital Equipment Corporation Systems Research Center.

[3] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM SIGSOFT, January 1996.

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

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

[6] Kevin Lano and Howard Haughton. *Object-Oriented Specification Case Studies*. Prentice Hall, New York, 1994.

[7] Gary T. Leavens. Larch/C++ reference manual, draft, $Revision: 4.16$, 16 July 1996. On the Web at `http://www.cs.iastate.edu/~leavens/larchc++manual/lcpp_toc.html`, 1996.

[8] 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.

[9] K. Rustan M. Leino. Modeling subtypes with only one object type. KRML 55, Digital Equipment Corporation Systems Research Center, August 1995.

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

[11] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[12] David A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi, San Miniato, Italy, 6–10 June 1994*, pages 467–487. Elsevier, 1994.

[13] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.