Formal Semantics and Soundness of an Algorithm

Tim Wahls
Iowa State University

Gary T. Leavens
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Theory and Algorithms Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/159

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Formal Semantics and Soundness of an Algorithm for Translating Model-based Specifications to Constraint Programs

Tim Wahls and Gary T. Leavens

TR #00-02
March 2000

Keywords: Executable specifications, behavioral interface specification language, formal specification, prototyping, translation of specifications to constraints, constraint programming, soundness, denotational semantics, model-based specification, formal methods, precondition, postcondition, SPECS-C++.


Copyright © 2000 by Tim Wahls and Gary T. Leavens.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Formal Semantics and Soundness of an Algorithm for Translating Model-based Specifications to Constraint Programs

Tim Wahls*
Department of Mathematical and Computer Sciences
W256 Olmsted, Penn State Harrisburg,
777 W. Harrisburg Pike, Middletown, PA 17057 USA
wahls@psu.edu

Gary T. Leavens†
Department of Computer Science, 229 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavers@cs.iastate.edu

March 10, 2000

Abstract

This paper presents an algorithm for executing formal specifications, and a proof of the soundness of that algorithm. The algorithm executes specifications written in the model-based specification language SPECS-C++ by transforming such specifications to constraint programs. The generated programs use constraint satisfaction techniques to execute specifications written at a high level of abstraction. Denotational semantics techniques are used for both explaining the algorithm and for proving its soundness.

1 Introduction

This paper presents a proof of the soundness of an algorithm [WLB97] for translating SPECS-C++ [WBL94] specifications to programs in the constraint programming language AKL [JH94]. The proof is accomplished by giving a denotational semantics [Sch86] for SPECS-C++, and then proving the soundness of the valuation functions. The goals of this work are twofold: first, to present the translation algorithm in a relatively language independent manner, and second, to give confidence to those using specification development and testing tools based on our algorithm that the answers returned are correct with respect to the specification.

The semantics given is designed for explaining and proving the soundness of the translation algorithm, and so is considerably different from a semantics designed only to present SPECS-C++. As SPECS-C++ uses first order predicate calculus assertions to specify the behavior of C++ [Str91] functions, a translation algorithm for SPECS-C++ is largely an algorithm for translating assertions. Soundness for such an algorithm means that the program generated by the algorithm will have the same truth value as the assertion it was generated from on the same inputs. However, the translation algorithm is not complete, so the program generated can fail to terminate even when the corresponding assertion has a well defined truth value.

Our basic approach in the semantics can be divided into two parts. First, we process a SPECS-C++ specification and translate it into a powerset of constraints, i.e., into a set of sets of constraints. The most important part of this translation is handling assertions (SPECS-C++ pre-/post-conditions). We use a powerset of constraints because SPECS-C++ specifications can be nondeterministic (and also underdetermined), and so each simple set of constraints represents a different way of satisfying the assertion. Second, we take each set of constraints and and iteratively simplify it, which corresponds to a fixpoint construction in the

*Wahls’s work was supported in part by a fellowship from IBM Rochester.
†Leavens’s work was supported in part by the National Science Foundation under grant CCR-9803843.
semantics. This simplification can expose further nondeterminism, and so simplifying a set of constraints produces a set of results. Each result is a pair consisting of an environment and a final store. Essentially, the fixpoint construction plays the role in the semantics that the AKL interpreter does in the execution algorithm itself. This approach allows us to encapsulate the use of constraints in the semantics — the input to the semantics is a SPECS-C++ specification, and the output is a set of environment and store pairs. Hence, we can reason about the soundness of the algorithm without reference to a particular constraint programming language such as AKL.

An alternative approach to this semantics is for the valuation functions to return constraint logic programs, rather than sets of final states directly. The advantages of this approach are that it would allow us to take advantage of existing formal semantics for such languages [NF89] [JMMS98], and that it would closely model the way our algorithm actually works. However, formal semantics for CLP languages tend to focus on issues of control built into the language (the order that rules are used in, the order that answers are returned in, ... ) that we are not interested in. Additionally, such a semantics would not be a helpful way to explain our work for those unfamiliar with logic and CLP languages. Compounding this problem, existing semantics for such languages tend to be either operational semantics or continuation passing style denotational semantics, which adds another layer of complexity as neither of these approaches lend themselves to directly returning a set of final states. Hence, it is more difficult to see the relationship between the input specification and the set of final states. Finally, we would still need to prove the soundness of the program returned by our semantics with respect to the original specification, and it is not clear to the authors how well existing semantics for CLP languages support such a proof. Hence, we chose not to use this approach.

In showing soundness for our algorithm, we concentrate on the execution of a single call to a SPECS-C++ function, and show that every result environment and store pair from this semantics satisfies the specification of that function. In particular, we show that if the pre-condition of the specification is satisfied by the pre-state store, then any store constructed as specified by this semantics satisfies the post-condition of the specification when used as the post-state store.

We emphasize that this semantics for SPECS-C++ is unusual. Our intent is to present the translation algorithm and provide a mechanism for proving its soundness. A semantics designed strictly for presenting SPECS-C++ would not use constraints, and so would be considerably simpler.

As an example of applying this semantics, consider the following specification for a simple SPECS-C++ function.

```c
void foo(int& x);
/* pre: true
 modifies: x
 post: x' \in \{1, 2, 3\} \land x' != 2 */
```

This specification is nondeterministic, and so applying the first step of the semantics would yield three sets of constraints. These sets could be represented intuitively as follows (we use a more formal notation in presenting the semantics):

\[
\{x' = 1, x' != 2\}
\]

\[
\{x' = 2, x' != 2\}
\]

\[
\{x' = 3, x' != 2\}
\]

As each of these sets is simplified (in the second step of the semantics), the second set would be found to be inconsistent, and so would not contribute to the result. The remaining two sets would each yield one result, where the environments would be unchanged from the original in both cases, and the final store would bind the location of \(x'\) to 1 in the first case, and to 3 in the second. Both of these results are sound, as the specification of \(\text{foo}\) is satisfied if \(x'\) is 1, and also if \(x'\) is 3.

In the following sections, we give an abstract syntax for executable SPECS-C++ specifications, give the semantic algebras used, and define the valuation functions for the semantics. Finally, we show the soundness of (part of) the semantics by showing that if our semantics produces an answer for a call to a SPECS-C++ function, then that answer satisfies the specification of the function.
Abstract syntax domains:

<table>
<thead>
<tr>
<th>$C$ ∈ Class-specification</th>
<th>$F$ ∈ Abstract-function</th>
<th>$S$ ∈ Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$ ∈ Type-declaration</td>
<td>$M$ ∈ Member-function</td>
<td>$I$ ∈ Identifier</td>
</tr>
<tr>
<td>$P$ ∈ Parameter</td>
<td>$Y$ ∈ Result-type</td>
<td>$B$ ∈ Boolean-expr</td>
</tr>
<tr>
<td>$A$ ∈ Data-member</td>
<td>$E$ ∈ Expression</td>
<td>$V$ ∈ Literal</td>
</tr>
<tr>
<td>$T$ ∈ Type-expression</td>
<td>$L$ ∈ Expression-list</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Syntax domains for the abstract syntax.

2 Abstract Syntax

The following is the abstract syntax of an executable SPECS-C++ class specification. We refer to this syntax as abstract syntax for consistency with standard terminology in denotational semantics — the syntax presented is very close to the actual (concrete) syntax of SPECS-C++. As template class specifications and the specifications of the protected and private interfaces of a class are not yet executable, they are not included here.

Figure 1 gives the syntax domains for the abstract syntax. The domain Identifier represents variable, parameter, type and function names. The domain Literal represents the primitive values of SPECS-C++ (integer, float, double, character and string). Since these domains are well known they are not further defined in the syntax. The remaining domains will all be defined syntactically. We need very little of the semantics of types in the rest of the semantics, and so valuation functions are not given for domains Type-declaration, Data-member, Type-expression and Result-type.

Figure 2 gives BNF rules defining the syntax domains. Note that domain $F$ defines a list of abstract functions, which are like the specification functions of VDM [Jon90] [A+93]. $M$ defines a list of pre-post style operation specifications with C++ interfaces. Domain $B$ includes the standard first order boolean operations (including quantifiers \forall and \exists), and domain $E$ gives all operations on expressions.

3 Semantic Algebras

3.1 Values

The built-in types of SPECS-C++ are defined as follows. We use $Object$ in the sense of location or l-value, and $Instance$ for instances of classes. We model instances as a location (containing a tuple composed of the values of the data members of the class) and an identifier containing the name of the class that the instance belongs to. In our semantics, we only need to know the class of an instance when we have the location of that instance. Type $Value$ is a disjoint union of all of the built-in types.

```plaintext
Char = Integer
Float = Real
Double = Real
String = List(Storable-value)
Set = List(Storable-value)
Sequence = List(Storable-value)
Tuple = Identifier → Storable-value
Object = Location
Instance = Location × Identifier

Value = Char + Integer + Float + Double + Boolean + String + Set + Sequence + Tuple + Object +
```
C ::= class I {  
   /* model */  
   domains  
      D  
   data members  
      A  
   constraints  
      B  
   abstract functions  
      F  
   */  
   public:  
      M  
   ];  
   S  

D ::= D1;D2 | T I | ε  
T ::= char | int | float | double | bool | string | T& | set of T | sequence of T | tuple(P)  
A ::= A1;A2 | T I | ε  
F ::= F1;F2 | define I(P) as T such that B | ε  
P ::= P1;P2 | T I | ε  
M ::= M1;M2 | ε |  
   Y I(P);  
   /* pre: B1  
      modifies: L  
      post: B2*/  
   Y ::= void | T | ε  
L ::= E | E, L | ε  
B ::= true | false | !B | B ∧ B | B ∨ B | B ⇒ B  
   \forall T I [D \in E ∧ B ⇒ B] | \exists T I [B]  
   E1 = E2 | E1 != E2 | E1 < E2 | E1 ≤ E2 | E1 > E2 | E1 ≥ E2  
   E1 \in E2 | E1 \subset E2  
E ::= I | V | first(E) | header(E) | last(E) | trailer(E) | length(E) | domain(E) | range(E) |  
   E1 + E2 | E1 - E2 | E1 * E2 | E1 / E2 | E1 % E2 | E1 \union E2 | E1 \intersection E2 | E1 | E2 |  
   {L} | <L> | ⟨L⟩ | |E| | E.1 | E.1 | E.1 | E.1 | E.1 | E[E] | I(L) | result | self | {T I : I \in E ∧ B} | B  
S ::= S1;S2 | T I | I1.I2(L) | I = E | I1 = I2.I3(L)
Instance

Storable values include values of type \textit{Value} as above, and also variables. We use variable here in the sense of logic programming, so that values can be unified. Variables are modeled simply as integers — they will always be injected into another domain, so there is no danger of confusing variables with numeric values.

\textit{Variable} = \textit{Integer}  \\
\textit{Storable-value} = \textit{Value} + \textit{Variable}

As environments can also store the denotation of member and abstract functions, we give the type of that function denotation here. In type \textit{Function}, the \textit{List(Storable-value)} parameter represents the actual arguments to the function and the \textit{Storable-value} parameter represents the function’s result. Functions are treated as “constraint generators” — when called, they generate constraints as appropriate from their bodies. See Figure 5 for details of how SPECS-C++ functions are translated to this type. The use of type \textit{NextVar} is explained below.

\textit{Function} = \textit{List(Storable-value)} \rightarrow \textit{PowerSet(Constraint)} \rightarrow \textit{Store} \rightarrow \textit{Store} \rightarrow \textit{Storable-value} \rightarrow \textit{NextVar}  \\
\hspace{2cm} \rightarrow \textit{PowerSet(\textit{PowerSet}(\textit{Constraint}) \times \textit{Store} \times \textit{NextVar})}

Denotable values are then just functions and the storable values.

\textit{Denotable-value} = \textit{Function} + \textit{Value} + \textit{Variable}

Now, environments can be defined simply as functions from identifiers to denotable values, as usual.

\text{Domain} \textit{Environment} = \text{Identifier} \rightarrow \textit{Denotable-value}_\perp

\text{Operations:}
\hspace{1cm} \textit{updateenv} : \text{Identifier} \rightarrow \textit{Denotable-value} \rightarrow \textit{Environment} \rightarrow \textit{Environment}  \\
\hspace{2cm} \textit{updateenv} = \lambda i. \lambda v. \lambda e. [i \mapsto v]e  \\
\hspace{1cm} \textit{accessenv} : \text{Identifier} \rightarrow \textit{Environment} \rightarrow \textit{Denotable-value}_\perp  \\
\hspace{2cm} \textit{accessenv} = \lambda i. \lambda e. (e \ i)

We also define a class environment to associate a class name with the member functions of that class. This follows [LP98], except that we use the term class environment in place of method dictionary. For simplicity, we map a class name to an environment as defined above. This provides more generality than needed, as only function definitions will be stored in the environment associated with a class name.

\text{Domain} \textit{Class-Environment} = \text{Identifier} \rightarrow \textit{Environment}_\perp

\text{Operations:}
\hspace{1cm} \textit{updateenv} : \text{Identifier} \rightarrow \textit{Environment} \rightarrow \textit{Class-Environment} \rightarrow \textit{Class-Environment}  \\
\hspace{2cm} \textit{updateenv} = \lambda i. \lambda e. \lambda c. [i \mapsto e]ce  \\
\hspace{1cm} \textit{accessenv} : \text{Identifier} \rightarrow \textit{Class-Environment} \rightarrow \textit{Environment}_\perp  \\
\hspace{2cm} \textit{accessenv} = \lambda i. \lambda e. (ce \ i)

The translation algorithm we are modeling is based on logic and constraint programming, and so makes heavy use of logical variables. In particular, we need the ability to generate fresh (unused) logical variables. We do this by threading a parameter of type \textit{NextVar} (an integer counter) through the semantics and using it as an argument to function \textit{newvar} to generate the “next” logical variable as a \textit{Storable-value} value.

\text{Domain} \textit{NextVar} = \textit{Int}

\text{Operations:}
\hspace{1cm} \textit{newnextv} : \rightarrow \textit{NextVar}  \\
\hspace{2cm} \textit{newnextv} = 0  \\
\hspace{1cm} \textit{newvar} : \textit{NextVar} \rightarrow (\textit{Storable-value} \times \textit{NextVar})  \\
\hspace{2cm} \textit{newvar} = \lambda n. (\text{inVariable}(n), n + 1)

The store algebra is adapted from Schmidt [Sch86], p 147. The only nonstandard operation is \textit{makefresh}, which binds a list of locations in the store to fresh variables. This operation is used to construct the post-state store for executing a member function specification, as the locations that are bound to fresh variables
equal = Storable-value × Storable-value
not_equal = Storable-value × Storable-value
less = Storable-value × Storable-value
member = Storable-value × Storable-value
subset = Storable-value × Storable-value
not = Boolean-expr × Environment × Store × Store
forall = Identifier × Storable-value × Boolean-expr × Environment × Store × Store
first = Storable-value × Storable-value
plus = Storable-value × Storable-value × Storable-value
union = Storable-value × Storable-value × Storable-value
post = Storable-value × Storable-value
comp = Identifier × Storable-value × Boolean-expr × Storable-value × Environment × Store × Store
call = Identifier × List(Storable-value) × Storable-value × Store × Store
false = Unit

constraint = equal + not_equal + member + subset + not + forall + first + plus + minus + union + post + pre + comp + call + false + ...

Figure 3: Type Constraint representing the constraints generated and simplified by the execution technique. Only a representative subset of the various types of constraints are presented.

are precisely the locations for which the post-condition defines new values according to the modifies clause in the member function specification.

Domain Store = (Location → Storable-value) × Location
Operations:

newstore : → Store
newstore = (λ.⊥, first-loc)

update : Location → Storable-value → Store → Store
update = λ.λv.λ(map,last). ([l → v]map,last)

access : Location → Store → Storable-value
access = λ.λ(map,last). map!

allocate-locn : Store → Location × Store
allocate-locn = λ(map,last). (last,(map,next-locn(last)))

makefresh : Store → List(Location) → NextVar → (Store × NextVar)
makefresh = λ(map,last).List.λextv. if list = nil then ((map,last),nextv)
else let (nv,nextv') = newvar(nextv) in

makefresh([hd(l) → nv]map,last) tl(l) nextv'

3.2 Constraints

Because constraints contain values (and not pieces of syntax), we cannot use elements of syntax domain Boolean-expr as constraints. Instead, we introduce a domain Constraint that includes constraints for all of the built-in operators of SPECS-C++. For operators that are not relations, the matching constraint is converted to a relation by making the last “parameter” (the last element of the cartesian product) the result of the operation. This is the standard technique for converting arbitrary functions to relations in logic programming. The constraint False is used as a constraint that is always false. This is needed for explicitly indicating a failure in constraint simplification. We also define constraints representing universal quantification (forall) and negation (not). Figure 3 gives the definition of a representative subset of domain Constraint.

The domain Constraint will be used as follows (roughly). First, some part of a SPECS-C++ specification
\[ C \text{ : Class specification } \rightarrow \text{Environment} \rightarrow \text{Class-Environment} \rightarrow \text{PowerSet(Environment } \times \text{ Store)} \]
\[ C \llbracket \text{class I } \{ \\
/\text{* model} \\
\text{domains} \\
\text{D} \\
\text{data members} \\
\text{A} \\
\text{constraints} \\
\text{B} \\
\text{abstract functions} \\
\text{F} \\
/\}
\text{public:} \\
\text{M} \\
\rrbracket \llbracket \lambda e. \lambda c. \\
\text{let } e' = F[F] e \text{ in} \\
(\lambda (e'' , c, s). \text{if } c = \{\} \text{ then } \{ (e'' , s) \} \text{ else } \{\})^+ \\
\text{let } e = \text{updateenv I (M[M] e' } \lambda i. \bot) ce, \{\}, \text{newstore, newnextv} \rrbracket \]

Figure 4: Function \( C \) for evaluating a class specification.

is converted into a powerset of elements of \( \text{Constraint} \), and then the elements of that powerset are simplified into sets of answers (environment and store pairs) using a fixpoint construction.

4 Valuation Functions

Figure 4 presents \( C \), the valuation function for class specifications. The initial environment \( e \) passed in can be the empty environment \((\lambda i. \bot)\) if no “top level” definitions need to be provided. The initial class environment contains the member functions of any classes already processed. The denotations of the abstract functions are added to the denotation of the class by valuation function \( F \). The denotations of the member functions are stored in a new environment by \( M \) (see Figure 3 for both \( F \) and \( M \)), which is then associated with the class name in the class environment. Both environments are used by \( S \) (Figures 6 and 7) to evaluate the “test case” (sequence of declarations, member function calls and assignment statements) for the specification. \( S \) returns a set of three-tuples (final environment, constraint set, final store) representing the final state. A set of states is necessary because specifications are often underdetermined or nondeterministic. If the constraint set in a state is empty, then the associated environment and store form a valid final state (answer) for the specification — they satisfy the specification and the test cases given as input to \( C \). If the constraint set is not empty, the environment and store may or may not satisfy the specification and test cases, and so do not contribute to the final state. A nonempty constraint set results from the incompleteness of the execution technique — the technique was not strong enough to simplify the remaining constraints.

To extend functions to deal with set-valued inputs, we use the notation \( f^+ \) extensively throughout this paper. Given a function \( A : T_1 \rightarrow \text{PowerSet}(T_2) \), \( A^+ : \text{PowerSet}(T_1) \rightarrow \text{PowerSet}(T_2) \) is defined as:

\[ A^+(X) = \bigcup \{ A(x) \mid x \in X \} \]

Figure 5 presents \( F \) and \( M \), the valuation functions for abstract and member functions, respectively. In both cases, the function is compiled to type \( \text{Function} \), and then associated with its name in an environment so that it can be invoked later. Abstract functions are stored in the top level (global) environment, while the member functions of each class are stored in a class environment. Hence, \( M \) takes both the global environment and a class environment argument (that holds the member functions for the current class that have already been compiled), and returns a class environment. Function \( P \) is used to associate the formal
\[ \mathcal{F}: \text{Abstract-function} \rightarrow \text{Environment} \rightarrow \text{Environment} \]
\[ \mathcal{F}[F_1; F_2] = \lambda e. \mathcal{F}[F_2] (\mathcal{F}[F_1] e) \]
\[ \mathcal{F}[\text{define } I(P)] \text{ as } T \text{ such that } B] = \lambda e. \]
\[ \text{let } f = \lambda \epsilon. \lambda s_{\text{pre}}. \lambda s_{\text{post}}. \lambda v. \lambda n_{\text{extv}}. (\lambda(e', n_{\text{extv}'}) \cdot \{(e', s_{\text{post}}, n_{\text{extv}'})\})^+ \]
\[ B[B] \ (P[P] \ l \ e, c, n_{\text{extv}}) \ s_{\text{pre}} \ s_{\text{post}} \ \text{in} \]
\[ \text{updateenv } [I] \ \text{inFunction}(f) \ e \]
\[ \mathcal{F}[e] = \lambda e. e \]

\[ \mathcal{M}: \text{Member-function} \rightarrow \text{Environment} \rightarrow \text{Class-Environment} \rightarrow \text{Class-Environment} \]
\[ \mathcal{M}[M_1; M_2] = \lambda \epsilon. \lambda e. \mathcal{M}[M_2] (\mathcal{M}[M_1] e \ c) \]
\[ \mathcal{M} \parallel Y \ I(P); \]
\[ /* \text{pre: } B_1 \]
\[ \text{modifies: } L \]
\[ \text{post: } B_2 */ ] = \lambda \epsilon. \lambda e. \]
\[ \text{let } f = \lambda \epsilon. \lambda s_{\text{pre}}. \lambda s_{\text{post}}. \lambda n_{\text{extv}}. \]
\[ \text{let } e' = \text{updateenv } [\text{self}] \ h d(l) \ (P[P] \ t d(l) \ e \ v) \ \text{in} \]
\[ (\lambda(e'', s', e'', n_{\text{extv}})). \{(e''', s'', n_{\text{extv}}'')\})^+ \]
\[ ((\text{fix } I^+) \ ((\lambda(lst, c', n_{\text{extv}}) = \text{makefresh}(s_{\text{pre}}) \ lst \ n_{\text{extv}}') \ \text{in} \]
\[ (\text{adjustState } e' \ s_{\text{pre}} \ s_{\text{post}} \ ((\lambda(e', n_{\text{extv}})). \ B[B_2] \ (e', c'', n_{\text{extv}})) \ s_{\text{pre}} \ s_{\text{post}})^+ \]
\[ \text{updateenv } [I] \ \text{inFunction}(f) \ c \ e \]
\[ \mathcal{M}[e] = \lambda \epsilon. \lambda e. \ c \ e \]

\[ \mathcal{P}: \text{Parameter} \rightarrow \text{List(Storable-value)} \rightarrow \text{Environment} \rightarrow \text{Storable-value} \rightarrow \text{Environment} \]
\[ \mathcal{P}[P_1; P_2] = \lambda \epsilon. \lambda v. \mathcal{P}[P_2] \ t d(l) \ (P[P_1] \ cons(h d(l), \ nil) \ e \ v) \ v \]
\[ \mathcal{P}[T I] = \lambda \epsilon. \lambda v. \ \text{updateenv } [I] \ h d(l) \ e \]
\[ \mathcal{P}[e] = \lambda \epsilon. \lambda v. \ \text{updateenv } [\text{result}] \ v \ e \]

Figure 5: Function \( \mathcal{F} \) for storing abstract function definitions in an environment, function \( \mathcal{M} \) for storing member function definitions, and function \( \mathcal{P} \) that sets up the environment to evaluate the function body in.
S: Statement → (Environment × Class-Environment × PowerSet(Constraint) × Store × NextVar) →
               PowerSet(Information × PowerSet(Constraint) × Store × NextVar))
S[S_1; S_2] = \begin{align*}
\lambda(e, ce, c, s, \text{nextv}).
& (\lambda(e', c', s', \text{nextv}'). \ S[S_2]^+ (e', c', s', \text{nextv}'))^+
\end{align*}

(S[S_1] (e, ce, c, s, nextv))

S[T I] = \begin{align*}
& \text{let } (s', l) = \text{allocate-locen } s \text{ in } \\
& \quad \text{if } T \in \text{Identifier} \text{ then } \\
& \quad \quad \{(\text{updateenv } \llbracket I \rrbracket \text{ inValue(inInstance(l, T)) } e, c, s', \text{nextv})\} \\
& \quad \text{else } \{(\text{updateenv } \llbracket I \rrbracket \text{ inValue(inObject(l)) } e, c, s', \text{nextv})\} \\
\end{align*}

S[E I = E] = \lambda(e, ce, c, s, \text{nextv}).
\begin{align*}
\text{cases } \text{accessenv } \llbracket I \rrbracket \ e \ of \\
& \text{ isValue(v) } \rightarrow \text{ cases } v \ of \\
& \quad \text{isObject(l) } \rightarrow \\
& \quad \quad \text{let } (v', \text{nextv}') = \text{newvar(\text{nextv}) } \text{ in } \\
& \quad \quad \quad (\lambda(e', c', s', s'', \text{nextv}''). \ {(e', c', s'', \text{nextv}'')}^+) \\
& \quad \quad \quad \text{ (fix } I^+ \text{) (adjustState } e \text{ s (update } l v' s) \text{ (E}[E] \ v' (e, c, \text{nextv}') s s)) \\
& \quad \text{isInstance(l, ci) } \rightarrow \\
& \quad \quad \text{let } (v', \text{nextv}') = \text{newvar(\text{nextv}) } \text{ in } \\
& \quad \quad \quad (\lambda(e', c', s', s'', \text{nextv}''). \ {(e', c', s'', \text{nextv}'')}^+) \\
& \quad \quad \quad \text{ (fix } I^+ \text{) (adjustState } e \text{ s (update } l v' s) \text{ (E}[E] \ v' (e, c, \text{nextv}') s s)) \\
& \quad \text{else } \{(e, \{\text{inFalse}\}) s, \text{nextv})\} \\
\end{align*}

\text{end}

\text{end}

\begin{figure}
\begin{center}
\begin{align*}
\text{Figure 6: Function } S \text{ for evaluating statements that can call the member functions of the class specification.}
\text{These statements include variable declarations and assignment statements. (Part 1 of 2.)}
\end{align*}
\end{center}
\end{figure}

parameters with the actuals in the environment used for evaluating the function's body. Note that parameter v representing the function's result is associated with result in that environment. Abstract functions can call other abstract functions (and themselves recursively). We allow recursion by including the name of an abstract function in a Call constraint (see Figure 10) and then retrieving the function from the environment when simplifying such a constraint (see Figure 14). This delay ensures that all abstract functions are stored in the environment before any call to an abstract function is evaluated.

For member functions (valuation function M), some additional work is required. The modifies clause specifies what objects can change from the pre-state to the post-state. This list of objects is used by function makefresh (previously described) to create the initial post-state store. Note that the post-state store passed to the function is ignored in this case — this store is only passed in because it is needed for abstract functions. After constraints are generated from both the pre and post-condition, functional I (see Figure 11) is used in a fixpoint construction to simplify the constraints into an environment and result store. The fixpoint construction is needed because this simplification is iterative, and should continue until no more constraints can be simplified. Functional I is used for member functions because they are called from outside of the class — abstract functions are only called from member functions and other abstract functions, and so the constraints generated from such calls will be simplified with the other constraints generated from the body of the member function. Member function specifications cannot be recursive — member functions can only be called from outside of the class.

Figures 6 and 7 present valuation function S, which provides the semantics for syntax domain Statement — the “test cases” (where each test case is a sequence of object declarations, calls to member functions, and assignment statements) used for executing class specifications. For declarations of objects (including instances that are objects), the object name is associated with a newly allocated location in the environment. For member function calls, the class of the object receiving the message is retrieved and then passed to the
\[ S[1, I_2 (L)] = \lambda(e, ce, c, s, nextv). \text{cases accessenv } [I_1] \text{ e of } \\
\quad \begin{cases} 
    \text{isValue}(v) \rightarrow \text{cases } v \text{ of } \\
    \quad \text{isInstance}(l, ci) \rightarrow \\
    \quad \begin{cases} 
        \text{cases accessenv } [I_2] \text{ (accessenv } ci \text{ ce) of } \\
        \quad \text{isFunction}(f) \rightarrow \\
        \quad \begin{cases} 
            \text{let } (v', nextv') = \text{newvar}(nextv) \in \\
            \quad \begin{cases} 
                (\lambda(lst, c', nextv''). (\lambda(c'', s', nextv'''). \{(e, c'', s', nextv''')\})^+ \\
                (f \text{ cons}(v, lst) c' s s' \text{ nextv'}))^+ \\
                \text{L}[L] (e, c, nextv') s s
            \end{cases}
        \end{cases}
    \end{cases}
\end{cases}
\end{cases}
\]

Figure 7: Function \( S \) for evaluating statements that can call the member functions of the class specification. \( S \) uses function \( \mathcal{L} \) (Figure 8) to evaluate actual parameter lists in member function calls. (Part 2 of 2.)
class environment to get the environment containing the (compiled versions of the) member functions for that class. The name of the called function is then used to obtain the member function from that environment. The member function is then called with the appropriate arguments. Function $\mathcal{L}$ (Figure 8) is used to evaluate the actuals into a list of Storable-values and an associated set of constraints to be passed to the denotation of the member function. Note that the object receiving the message (member function call) is packaged and passed as a regular parameter.

For simple assignment statements (assigning an expression result to an object), a new post-state store need not be allocated because such a statement can only change the value stored at one location. However, the fixpoint construction used earlier must be applied here as well to ensure that the set of constraints is simplified if the assignment statement is the last statement in the test case. Assignment statements are the only case requiring constraint simplification where a member function is not invoked, and so we cannot rely on the execution of a member function to apply the fixpoint construction in this case.

Figure 9 presents valuation function $B$, which is used to translate assertions (type Boolean-expr) into constraints. Negated assertions are converted directly to constraints to be evaluated later. Note that the environment and stores are included so the assertion can be evaluated in the proper context. For conjunctions, first one conjunct is translated, and the resulting constraint set is used as input to the translation of the next conjunct. This ensures that constraints generated from both conjuncts are in the same set of constraints. For disjunctions, each conjunct is translated and the resulting sets of constraints are unioned. An implication $P \Rightarrow Q$ is translated as the equivalent $(P \land \neg Q) \lor \neg P$.

Universally quantified assertions can only be executed if the domain that the bound variable ranges over is known and finite. The variable’s domain can be given either by restricting it to be an element of a set or sequence, or by bounding it as a finite subrange of the integers. Here, the domain is evaluated, and included in a $\forallall$ constraint to be evaluated later. Existential quantification simply introduces a new variable.

The remaining boolean assertions are applications of relations, and are translated directly to the matching constraints. The case for relation $=$ is shown. The remaining cases (including cases for relations $\in$ and $\subsetset$) are similar.

Function $\mathcal{E}$ (Figure 10) translates expressions into sets of constraints. Since there is a matching constraint for each built in operator of SPECS-C++, this consists of translating the actual operands into sets of constraints, and then building the appropriate constraint. In the majority of cases, this is simple enough to essentially be a type conversion — we are taking an element of domain Expression and converting it to an element of domain Constraint. However, we do lose some structure in this process. For example, an arithmetic expression like $x \times 2 + y$ is translated (roughly) to: $\{\inTimes(x, 2, v), \inPlus(v, y, v')\}$, where $v$ and $v'$ are fresh variables and $v'$ represents the overall result. This flattening of the structure of the expression implies that only local constraint propagation is possible — no propagation can be done based directly on the larger surrounding context of a term. However, this does accurately model the current version of the translation algorithm.

Many of the cases for $\mathcal{E}$ are similar, and in fact most are omitted from Figure 10. However, the case

\[
\mathcal{L}(\text{Expression-list} \rightarrow (\text{Environment} \times \text{PowerSet(Constraint)} \times \text{NextVar}) \rightarrow \text{Store} \rightarrow \text{Store} \\
\rightarrow \text{PowerSet(List(Storable-value))} \times \text{PowerSet(Constraint)} \times \text{NextVar}) \\
\mathcal{L}[e] = \lambda(e, c, nextr). \lambda_{spre}. \lambda_{spost}. \{(\text{nil}, c, nextrv)\} \\
\mathcal{L}[\mathcal{E}] = \lambda(e, c, nextr). \lambda_{spre}. \lambda_{spost}. \\
\text{let } (v, nextrv) = \text{newvar}(nextr) \text{ in} \\
(\lambda(c', nextrqv). \{(\text{cons}(v, nil), c', nextrqv))\}+) \mathcal{E}[\mathcal{E}] v (e, c, nextrv) \; spre \; spost \\
\mathcal{L}[\mathcal{L}] (e, c, nextrv) \; spre \; spost
\]
\[
B: \text{Boolean-expr} \rightarrow (\text{Environment} \times \text{PowerSet(Constraint)} \times \text{NextVar}) \rightarrow \text{Store} \rightarrow \text{Store} \rightarrow \text{PowerSet(PowerSet(Constraint)} \times \text{NextVar}) \\
B[\text{true}] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \{(c,\text{nextv})\} \\
B[\text{false}] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \{\{\text{infalse()},\text{nextv}\}\} \\
B[!B] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \{(c \cup \{\text{inNot}(B,e,\text{pre},\text{post})\},\text{nextv})\} \\
B[B_1 \land B_2] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \\
(\lambda(c',\text{nextv}''). \{\lambda(c''',\text{nextv}'''). \{(c \cup c' \cup c'''.\text{nextv}'').\}^+ B[B_2] \{e,\},\text{nextv}'') \text{ s}_{\text{pre}} \text{ s}_{\text{post}}^+ \\
B[B_1 \lor B_2] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. B[B_1] (e,c,\text{nextv}) \text{ s}_{\text{pre}} \text{ s}_{\text{post}} \cup B[B_2] (e,c,\text{nextv}) \text{ s}_{\text{pre}} \text{ s}_{\text{post}} \\
B[B_1 \Rightarrow B_2] = B[B_1 \land B_2] \lor \neg B_1] \\
B[\forall I T (\{I \in E \land B_1 \Rightarrow B_2\} \Rightarrow B_1] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \\
\text{let} \ (v,\text{nextv}') = \text{newvar(}\text{nextv}) \ \text{in} \\
\quad \lambda(c',\text{nextv}''). \{(c' \cup \{\text{inforall}(I,v,B_1 \Rightarrow B_2,e,\text{pre},\text{post})\},\text{nextv}'')\}^+ \\
\quad E[E] \ v (e,c,\text{nextv}') \text{ s}_{\text{pre}} \text{ s}_{\text{post}} \\
B[\exists I T [B]] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \\
\text{let} \ (v,\text{nextv}') = \text{newvar(}\text{nextv}) \ \text{in} \\
\quad B[B] \ \text{updateenv} [\{I\}] \ v e c,\text{nextv}') \text{ s}_{\text{pre}} \text{ s}_{\text{post}} \\
B[E_1 = E_2] = \lambda(e,c,\text{nextv}).\lambda_{\text{pre}}.\lambda_{\text{post}}. \\
\text{let} \ (v_1,\text{nextv}') = \text{newvar(}\text{nextv}) \ \text{in} \\
\text{let} \ (v_2,\text{nextv}'') = \text{newvar(}\text{nextv}) \ \text{in} \\
\quad \lambda(c',\text{nextv}'''). \lambda(c''',\text{nextv}'''). \{(c' \cup \{\text{inEqual}(v_1,v_2)\},\text{nextv}'''.\}^+ \\
\quad E[E_2] v_2 (e,c',\text{nextv}''') \text{ s}_{\text{pre}} \text{ s}_{\text{post}}^+ \\
\quad E[E_1] v_1 (e,c,\text{nextv}') \text{ s}_{\text{pre}} \text{ s}_{\text{post}} \\
:\end{align*}

Figure 9: Function $B$ for evaluating boolean assertions into sets of constraints.
\( \mathcal{E} \) {} Expression → Storable-value → (Environment × PowerSet(Constraint) × NextVar) → Store → Store
\( \mathcal{E}[B] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \)

\begin{align*}
\text{cases } v \text{ of} & \\
& \text{isValue}(b) \rightarrow \\
& \text{if } b \text{ then } \mathcal{B}[B] (e, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \text{ else } \mathcal{B}[!]B (e, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \\
& \text{isVariable}(i) \rightarrow \\
& \mathcal{B}[B] (e, c \cup \{\text{inEqual}(v, \text{inValue}(\text{true}))\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \\
& \cup \mathcal{B}[!]B (e, c \cup \{\text{inEqual}(v, \text{inValue}(\text{false}))\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \\
\end{align*}

\( \mathcal{E}[I] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \{ (e \cup \{\text{inEqual}(v, \text{accessenv}[I] e)\}, \text{nextv}) \} \)

\( \mathcal{E}[V] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \{ (e \cup \{\text{inEqual}(v, \text{inValue}(V))\}, \text{nextv}) \} \)

\( \mathcal{E}[\text{first}E] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \)

\( \text{let } (v', \text{nextv}') = \text{newvar}(\text{nextv}) \text{ in } \)

\( (\lambda(c', \text{nextv}''). \{ (c' \cup \{\text{inFirst}(v', v)\}, \text{nextv}'') \})^+ \mathcal{E}[E] v' (e, c, \text{nextv'}) s_{\text{pre}} s_{\text{post}} \)

\( \mathcal{E}[E + E_2] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \)

\( \text{let } (v_1, \text{nextv}') = \text{newvar}(\text{nextv}) \text{ in } \)

\( \text{let } (v_2, \text{nextv}'') = \text{newvar}(\text{nextv}) \text{ in } \)

\( (\lambda(c', \text{nextv}''). \{ (c'' \cup \{\text{inPlus}(v_1, v_2, v)\}, \text{nextv}'') \})^+ \mathcal{E}[E_2] v_2 (e, c', \text{nextv}'') s_{\text{pre}} s_{\text{post}} \)

\( \mathcal{E}[E_1] v_1 (e, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \)

\( \mathcal{E}[E'] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \)

\( \text{let } (v', \text{nextv}') = \text{newvar}(\text{nextv}) \text{ in } \)

\( (\lambda(c', \text{nextv}''). \{ (c' \cup \{\text{inPost}(v', v)\}, \text{nextv}'') \})^+ \mathcal{E}[E] v' (e, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \)

\( \mathcal{E}[\text{result}] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \{ (e \cup \{\text{inEqual}(v, \text{accessenv}[\text{result}] e)\}, \text{nextv}) \} \)

\( \mathcal{E}[\text{self}] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \{ (e \cup \{\text{inEqual}(v, \text{accessenv}[\text{self}] e)\}, \text{nextv}) \} \)

\( \mathcal{E}[\text{if } I : I \text{ in } E \land B] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \)

\( \text{let } (v', \text{nextv}') = \text{newvar}(\text{nextv}) \text{ in } \)

\( (\lambda(c', \text{nextv}''). \{ (c' \cup \{\text{inComp}(I, v, B, v, e, s_{\text{pre}}, s_{\text{post}})\}, \text{nextv}'') \})^+ \mathcal{E}[E] v' (e, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \)

\( \mathcal{E}[I \text{ if } L] = \lambda v. \lambda(e, c, \text{nextv}). \lambda_{\text{pre}}. \lambda_{\text{post}}. \)

\( (\lambda(v, c', \text{nextv}'). \{ (c' \cup \{\text{inCall}(I, I, v, s_{\text{pre}}, s_{\text{post}})\}, \text{nextv}') \})^+ \mathcal{E}[L] v (e, c, \text{nextv}) s_{\text{pre}} s_{\text{post}} \)

: 

\[ \text{end} \]

Figure 10: Representative cases of valuation function \( \mathcal{E} \).
\[
I : \mathcal{P}(\text{Environment} \times \text{Store} \times \text{Store} \times \mathcal{P}(\text{Constraint}) \times \text{Next Var}) \\
\rightarrow \mathcal{P}(\text{Environment} \times \text{Store} \times \text{Store} \times \mathcal{P}(\text{Constraint}) \times \text{Next Var}) \\
I = \lambda(e, sprec, spost, cs, nextv). \begin{cases} 
\text{if } cs = \emptyset & \{ (e, sprec, spost, cs, nextv) \} \\
\text{else} & \bigcup \{ S \ c \ (e, sprec, spost, cs - \{ c \}, nextv) \mid c \in cs \} 
\end{cases}
\]

Figure 11: The functional \( I \) used for simplifying constraints. The least fixed point of \( I^+ \) is the denotation of an assertion. \( I \) uses function \( S \) (defined in Figures 13 and 14) to simplify each constraint.

\[
adjustState : \text{Environment} \rightarrow \text{Store} \rightarrow \text{Store} \rightarrow (\mathcal{P}(\mathcal{P}(\text{Constraint}))) \times \text{Next Var} \\
\rightarrow \mathcal{P}(\text{Environment} \times \text{Store} \times \text{Store} \times \mathcal{P}(\text{Constraint}) \times \text{Next Var}) \\
adjustState = \lambda e. \lambda sprec. \lambda spost. \lambda cs. \{(e, sprec, spost, c, nextv) \mid (c, nextv) \in cs\}
\]

Figure 12: Function \( adjustState \), which is used to adjust the representation of a state to the form needed for applying \( I \).

for elements of syntax domain Boolean-expr (for \( B \in \text{Boolean-expr} \)) is considerably different. This case is for elements of Boolean-expr used in an expression (term) context. Hence, we must evaluate an element of Boolean-expr for its value, rather than just generating constraints from it. We divide this case into two subcases. If we already know what truth value \( B \) must have for the set of constraints to be consistent, we simply generate constraints for \( B \) or \( !B \) as appropriate. If the needed truth value for \( B \) is unknown, then we return the union of two sets of states. For one set, we generate constraints for \( B \) and specify that the value of \( B \) is \text{true}. For the other set, we generate constraints for \( !B \) and specify that the value of \( B \) is \text{false}. Clearly, if the specification determines the value of \( B \), all states in one of these sets must fail and so not contribute any answers to the final result.

The functional \( I \) (Figure 11) is used to simplify a set of constraints into a store. \( I \) is always used in a fixed point construction, as constraint simplification proceeds until no more simplification is possible — which occurs when an attempt at simplification does not change the set of constraints. \( I \) makes heavy use of function \( S \) which attempts to simplify a single constraint. Several of the more interesting cases of function \( S \) are presented in Figures 13 and 14. For each kind of constraint, \( S \) determines if enough information is available to make a simplification step. If not, then \( S \) leaves the constraint set unchanged (by returning the constraint to the set). Otherwise, \( S \) performs a simplification, in which case the constraint used is not returned to the constraint set. However, simplifying a constraint may cause one or more new constraints to be added, and could also cause values in the current post-state store to become more defined. This models the local propagation used in the implementation. We use \text{unify} to mean the standard unification algorithm, except that this version returns a set of constraints rather than a substitution, using the obvious equivalence between a substitution and a set of equality (\text{Equal}) constraints. We use the notation \([v'/v]X\) to denote replacing all occurrences of \( v \) by \( v' \) in \( X \), where \( v \) must be a variable, \( v' \) is a \text{Variable} or a \text{Value}, and \( X \) is an environment, store or constraint set. Note that elements of \text{Variable} can occur within each of these structures.

5 Proof of Soundness

In this section, we concentrate on proving the soundness of valuation functions \( B \) (Figure 9) and \( E \) (Figure 10), and on proving the soundness of \((\text{fix } I^+\) (Figure 11). These are the parts of the semantics that deal directly with assertions, and so we can show soundness by showing that any environment and store produced by the semantics satisfies the assertion that was input.

The valuation functions that precede \( B \) (\( C, M, F \) and \( S \) in Figures 4 through 7) describe translating
$S : \text{Constraint} \rightarrow \text{Environment} \times \text{Store} \times \text{Store} \times \text{PowerSet(Constraint)} \times \text{NextVar}$

$S = \lambda(c, e, s_{prev}, s_{post}, cs, nextv). \text{cases } c \text{ of}$

$\text{cases } ds \text{ of}$

isValue(d) \rightarrow cases d of

iset(vl) \rightarrow (if vl = nil then \{(e, s_{prev}, s_{post}, cs, nextv)\})

else let (v, nextv') = newvar(nextv) in

let e'' = updateenv [I] v e' in

let (v', nextv'') = newvar(nextv') in

($\lambda(cs', nextv'')$. \{(e, s_{prev}, s_{post},

\in Member(v, ds), \text{inMinus}(ds, v, v'),

inFormul(I, v', B, e', s'_{prev}, s'_{post})\},

nextv''))

\{B[B] (e'', cs, nextv''), s'_{prev}, s'_{post}\}

else \{(e, s_{prev}, s_{post}, \text{inFalse}(), nextv)\}

end

end

$\text{isNot}(B, e', s'_{prev}, s'_{post}) \rightarrow \text{let res = (fix } I^+)(\text{adjustState } e \text{ s_{prev} s_{post} \{B[B] (e', cs, nextv) s'_{prev} s'_{post}\}})$

\text{in if } \forall(e', s'_{prev}, s'_{post}, cs', nextv') \in \text{res . } cs' = \{} \text{ then } \{(e, s_{prev}, s_{post}, \text{inFalse}(), nextv)\}

else if \text{inFalse}() \in \text{cs} \text{ then } \{(e, s_{prev}, s_{post}, cs, nextv')\}

else \{(e, s_{prev}, s_{post}, cs \cup \{\text{inNot}(B, e', s'_{prev}, s'_{post})\}, nextv)\}

end

$\text{isEqual}(v_1, v_2) \rightarrow \text{cases } v_1 \text{ of}$

isVariable(i) \rightarrow \{(e, s_{prev}, s_{post}, cs, nextv)\}

| isValue(n) \rightarrow \text{cases } v_2 \text{ of}$

isVariable(i) \rightarrow \{(e, s_{prev}, s_{post}, cs, nextv)\}

| isValue(n) \rightarrow \text{let } cs', nextv' = \text{unify}(n, v_2, cs, nextv) \text{ in}$

\{(e, s_{prev}, s_{post}, cs', nextv')\}

end

$\text{isFirst}(v_1, v_2) \rightarrow \text{cases } v_1 \text{ of}$

isVariable(i) \rightarrow \text{cases } v_2 \text{ of}$

isVariable(i) \rightarrow \{(e, s_{prev}, s_{post}, cs \cup \{\text{inFirst}(v_1, v_2)\}, nextv)\}

| isValue(eI) \rightarrow \text{let } (nv, nextv') = \text{newvar}(nextv) \text{ in}$

\{(e, s_{prev}, s_{post}, cs \cup \{\text{inConcat}(\text{inValue(inSequence(cons(v_2, nil)), nv, v_1)})\}, nextv')\}

end

$\text{isValue}(seq) \rightarrow \text{cases } seq \text{ of}$

isSequence(l) \rightarrow \{(e, s_{prev}, s_{post}, cs \cup \{\text{isEqual}(v_2, \text{id}(l))\}, nextv)\}

else \{(e, s_{prev}, s_{post}, \text{inFalse}(), nextv)\}

end

end

Figure 13: Representative cases of function S, which defines the effect of simplifying a single constraint. (Part 1 of 2.)
\[ \text{isPlus}(v_1, v_2, v_3) \rightarrow \]
\[ \text{cases } v_1 \text{ of} \]
\[ \text{isVariable}(i_1) \rightarrow \text{cases } v_2 \text{ of} \]
\[ \text{isVariable}(i_2) \rightarrow \{ (e, s_{pre}, s_{post}, cs \cup \{ \text{isPlus}(v_1, v_2, v_3) \}, \text{nextv}) \} \]
\[ \text{isValue}(n_2) \rightarrow \text{cases } v_2 \text{ of} \]
\[ \text{isVariable}(i_3) \rightarrow \{ (e, s_{pre}, s_{post}, cs \cup \{ \text{isPlus}(v_1, v_2, v_3) \}, \text{nextv}) \} \]
\[ \text{isValue}(n_3) \rightarrow \{ (e, s_{pre}, s_{post}, cs \cup \{ \text{isEqual}(v_1, \text{inValue}(n_3 - n_2)) \}, \text{nextv}) \} \]
\end{verbatim}

Figure 14: More representative cases of function \textit{S}, which defines the effect of simplifying a single constraint. (Part 2 of 2.) Function \textit{filter} is defined in Figure 15.

16
filter : Identifier × List(Storable-value) × Boolean-expr × Environment × PowerSet(Constraint) × NextVar
→ PowerSet(List(Storable-value) × PowerSet(Constraint) × NextVar)

filter = λ(I, vl, B, e, cs, nextv).
if vl = null then \{null, cs, nextv\}
else let e' = updateenv[I] hd(vl) e in
(λ(vl', cs', nextv'). (λ(cs'', nextv''). \{cons(hd(vl), vl'), cs'', nextv''\}ι)
\[ E[B] \text{ inValue(inBoolean(true))} \] (e', cs', nextv')
∪ (λ(cs'', nextv''). \{cons(hd(vl), vl'), cs'', nextv''\}ι)
\[ E[B] \text{ inValue(inBoolean(false))} \] (e', cs', nextv'))ι

filter(I, tl(vl), B, e, cs, nextv)

Figure 15: Function filter is used to select only the elements of a list that satisfy some boolean condition.

larger parts of SPECS-C++ specifications. To show soundness for these functions, we would need to show that they are correct with respect to some independent formal semantics for SPECS-C++. While useful, this part of the proof is not as interesting as showing soundness for assertions, and so we omit it here.

We state the soundness result for assertions as a theorem below. The notation B[e, s_pre, s_post] means the truth value of B under the given environment and stores.

**Theorem 5.1 Soundness Theorem:**

∀B ∈ Boolean-expr. ∀(cs, nextv') ∈ B[e, \{\}, nextv] s_pre s_post.
∀(e', s'_pre, s'_post, cs', nextv'') ∈ (fix I+) \{cons(hd(vl), vl'), cs', nextv''\}ι.

That is, if the constraints derived from B are simplified into the environment e', pair of stores s'_pre and s'_post, and set of constraints cs', and all constraints can be simplified (cs' is empty), then the assertion B is true under e, s_pre and s_post. We ignore s'_pre and e' because simple inspection of the semantics shows that the pre-state store is never changed, and solving constraints never changes the environment. The environment and pre-state store are part of the result simply to make the type of I work out for the fixed point construction.

In the proof, we work with the following (stronger) formulation of this theorem to make the proof easier. The predicate consistent determines whether or not a set of constraints is consistent (all constraints in the set can be satisfied simultaneously) under a given environment and pair of stores. Thus, this version of the theorem states that consistency of the set of constraints returned by the algorithm (under the environment and stores returned by the algorithm) is equivalent to the truth value of the original assertion under the same environment and stores. If the set of constraints is empty, then consistent is defined to be true. Hence, this second version of the theorem implies the original.

**Lemma 5.2 Soundness Theorem (stronger version):**

∀B ∈ Boolean-expr. ∀(cs, nextv') ∈ B[e, \{\}, nextv] s_pre s_post.
∀(e', s'_pre, s'_post, cs', nextv'') ∈ (fix I+) \{cons(hd(vl), vl'), cs', nextv''\}ι.
consistent(cs', e, s'_pre, s'_post) ⇔ B[e, s_pre, s_post]

We define consistent more carefully as follows.

**Definition 5.3 Definition of consistent**

A set of constraints is consistent if and only if there exists at least one labeling (assignment of values) of the logical variables of the constraint set that satisfies every constraint in that set.
We use function eval to evaluate one constraint in the given environment and stores, assuming that all logical variables have been labeled. Hence, a set of constraints is consistent if there is some labeling for which eval is true for each constraint in the set. For most constraints, eval has the same truth value as the result of applying the SPECS-C++ operator associated with the constraint. We provide the definition for three of the more interesting cases (including constraints representing universal quantification and negation) below. Using $v \in ds$ is a slight abuse of notation, since $ds$ is of type Set and not a mathematical set. Note that if a set of constraints contains any inFalse() constraint, the set is not consistent.

**Definition 5.4 Definition of eval**

eval : Constraint $\times$ Environment $\times$ Store $\times$ Store $\rightarrow$ Boolean
eval(c, e, s'$_{pre}$, s'$_{post}$) = cases c of
  isForall(I, ds, B, e', s'$_{pre}$, s'$_{post}$) $\rightarrow$
    $\forall v \in ds$. B[updateenv (I) v e', s'$_{pre}$, s'$_{post}$]
  | isNot(B, e', s'$_{pre}$, s'$_{post}$) $\rightarrow$
    if B[e', s'$_{pre}$, s'$_{post}$] then false else true
  | isFalse() $\rightarrow$ false
  ...

To show the Soundness Theorem, we use the following lemmas. First, we must show that the valuation function $B$ is sound.

**Lemma 5.5 Soundness of $B$:**

$\forall B \in Boolean$-expr. $\forall (cs, nextTv) \in B[e, \{\}, nextTv] s_{pre} s_{post}$.

$\forall (e', s'_{pre}, s'_{post}, cs', nextTv') \in (fix I^+) \{(e, s_{pre}, s_{post}, cs, nextTv)\}.

consistent(cs, e, s_{pre}, s_{post}) \Leftrightarrow B[e, s_{pre}, s_{post}]$

That is, that under a valid final post-state store $s'_{post}$, the set of constraints returned by $B$ is consistent if and only if the assertion $B$ is true.

Then we show that the construction $(fix I^+)$ used for simplifying constraints is sound:

**Lemma 5.6 Soundness of $(fix I^+)$:**

$\forall B \in Boolean$-expr. $\forall (cs, nextTv) \in B[e, \{\}, nextTv] s_{pre} s_{post}$.

$\forall (e', s'_{pre}', s'_{post}', cs', nextTv') \in (fix I^+) \{(e, s_{pre}, s_{post}, cs, nextTv)\}.

consistent(cs, e, s_{pre}, s_{post}) \Leftrightarrow consistent(cs', e, s_{pre}', s_{post}')$

Together, Lemmas 5.5 and 5.6 imply the Soundness Theorem (Lemma 5.2). Note that we are not proving completeness (even though we are using $\Rightarrow$), because we do not show that $fix I^+$ always simplifies a set of constraints to the empty set. In fact, $fix I^+$ does not always simplify a set of constraints, even when such simplification is possible.

What remains is to prove Lemmas 5.5 and 5.6.

We prove Lemma 5.5 by induction on the structure of Boolean-expr and the definition of $B$ given in Figure 9. The base cases are true, false and the applications of SPECS-C++ relations. For true, $B[true] (e, \{\}, nextTv) s_{pre} s_{post} = \{(\{\}, nextTv)\}$. The only element of the constraint set returned is $\{\}$, and so:

consistent(cs, e, s_{pre}, s_{post})

$\Leftrightarrow (by \ cs = \{\})$

consistent(\{\}, e, s_{pre}, s_{post})

$\Leftrightarrow (by \ the \ definition \ of \ consistent)$

true

$\Leftrightarrow (by \ the \ definition \ of \ SPECS-C++)$
true[e, s_{pre}, s'_{post}]

For false, \( B[false] (e,\{\},nextv) s_{pre} s_{post} = \{\{inFalse()\}, nextv\} \). Hence:

consistent\( (cs, e, s_{pre}, s'_{post}) \)
\( \Leftrightarrow \) (by \( cs = \{inFalse()\} \))
consistent\( (\{inFalse()\}, e, s_{pre}, s'_{post}) \)
\( \Leftrightarrow \) (by the definition of consistent)
false
\( \Leftrightarrow \) (by the definition of SPECS-C++)
false\( [e, s_{pre}, s'_{post}] \)

Applications of SPECS-C++ relations are translated directly into the corresponding constraint. Applying consistent to that constraint is equivalent to evaluating the relation application by definition of eval, assuming the soundness of valuation function \( \mathcal{E} \) for syntax domain Expression (see Lemma 5.9).

The inductive step consists of proving the lemma for the boolean connectives and quantifiers.

For negation, we first need to comment on the role of the post-state store that is recorded as part of a Not constraint. Because the simplification of a constraint may be delayed for an arbitrarily length of time after the constraint is generated, the current post-state store at the time the constraint is simplified may be different from the post-state store at the time the constraint was generated. In particular, this will occur when a Not constraint is generated as part of processing a “test case” (an element \( S \) of syntax domain Statement), but is simplified during the processing of a different element of syntax domain Statement. As elements of this domain are statements, a different post-state store results from evaluating each. However, in the case we are discussing here, we are processing one element \( B \) of syntax domain Boolean-expr (which is a component of some element of syntax domain Statement). Hence, the current post-state store and the post-state store recorded as part of a Not constraint (or any other constraint that has a post-state store component) must be the same. We record this fact as a lemma below.

**Lemma 5.7** Property of stores contained within constraints:

When simplifying constraints obtained from a single element \( B \) of Boolean-expr, any post-state store contained within a constraint is the same as the current (global) post-state store. \( \blacksquare \)

Now, we are ready to prove Lemma 5.5 for Not constraints. We have:

\( B[\neg B] (e,\{\},nextv) s_{pre} s_{post} = \{\{inNot(B, e, s_{pre}, s_{post})\}, nextv\} \)

and hence:

consistent\( (\{inNot(B, e, s_{pre}, s_{post})\}, e, s_{pre}, s'_{post}) \)
\( \Leftrightarrow \) (by Lemma 5.7)
consistent\( (\{inNot(B, e, s_{pre}, s'_{post})\}, e, s_{pre}, s'_{post}) \)
\( \Leftrightarrow \) (by the definition of consistent)
eval\( (\{inNot(B, e, s_{pre}, s'_{post})\}, e, s_{pre}, s'_{post}) \)
\( \Leftrightarrow \) (by the definition of eval)
not \( B[e, s_{pre}, s'_{post}] \)
\( \Leftrightarrow \) (by the definition of ! in SPECS-C++)
\( (\neg B)[e, s_{pre}, s'_{post}] \)

For conjunction,

\( B[B_1 \land B_2] (e,\{\},nextv) s_{pre} s_{post} \)
\( = \) (by the definition of \( B \))
\( (\lambda(e', nextv'). (\lambda(e'', nextv''). \{(e' \cup e'', nextv'')\}) + B[B_2] (e,\{\},nextv') s_{pre} s_{post})^+ \)
\( B[B_1] (e,\{\},nextv) s_{pre} s_{post} \)
\( = \) (by the definition of \( f^+ \) for any function \( f \))
\( (\lambda(e', nextv'). \bigcup\{(e' \cup e'', nextv'')\} | (e'', nextv'') \in B[B_2] (e,\{\},nextv') s_{pre} s_{post})^+ \)
\[ \mathcal{B}[B_1] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \]
\[ = \text{(by the definition of } \cup) \]
\[ (\lambda(c', \text{nextv}''). \{ (c' \cup c'', \text{nextv}'') | (c'', \text{nextv}'') \in \mathcal{B}[B_2] (e, \{\}, \text{nextv}') s_{\text{pre}} s_{\text{post}} \})^+ \]
\[ \mathcal{B}[B_1] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \]
\[ = \text{(by the definition of } f^+ \text{ for any function } f) \]
\[ \bigcup \{ (c', c'') | (c'', \text{nextv}'') \in \mathcal{B}[B_2] (e, \{\}, \text{nextv}') s_{\text{pre}} s_{\text{post}} \} \]
\[ \cup \{ (c', \text{nextv}') \in \mathcal{B}[B_1] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \} \]

Hence, any constraint set passed to consistent is of the form \( c' \cup c'' \), where \( c' \) consists of constraints generated from \( B_1 \) and \( c'' \) consists of constraints generated from \( B_2 \), and:

\[ \text{consistent}(c, e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{(by } cs = c' \cup c''\text{)} \]
\[ \text{consistent}(c', c''', e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{(any subset of a consistent set of constraints is consistent)} \]
\[ \text{consistent}(c', e, s_{\text{pre}}, s_{\text{post}}') \land \text{consistent}(c'', e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{(by the inductive hypothesis twice)} \]
\[ B_1[e, s_{\text{pre}}, s_{\text{post}}] \land B_2[e, s_{\text{pre}}, s_{\text{post}}'] \]
\[ \Rightarrow \text{(by the definition of } \text{SPECS-C++)} \]
\[ (B_1 \lor B_2)[e, s_{\text{pre}}, s_{\text{post}}'] \]

For the only if direction of the proof:

\[ (B_1 \lor B_2)[e, s_{\text{pre}}, s_{\text{post}}'] \]
\[ \Rightarrow \text{(by the definition of } \lor \text{ in } \text{SPECS-C++)} \]
\[ B_1[e, s_{\text{pre}}, s_{\text{post}}] \land B_2[e, s_{\text{pre}}, s_{\text{post}}'] \]
\[ \Rightarrow \text{(tautology)} \]
\[ \neg(B_1[e, s_{\text{pre}}, s_{\text{post}}] \land B_2[e, s_{\text{pre}}, s_{\text{post}}']) \]
\[ \Rightarrow \text{(DeMorgan's Law)} \]
\[ \neg(B_1[e, s_{\text{pre}}, s_{\text{post}}'] \lor \neg B_2[e, s_{\text{pre}}, s_{\text{post}}']) \]
\[ \Rightarrow \text{(by the inductive hypothesis twice)} \]
\[ \neg \text{consistent}(c', e, s_{\text{pre}}, s_{\text{post}}') \lor \neg \text{consistent}(c'', e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{(adding constraints to an inconsistent set cannot make it consistent)} \]
\[ \neg \text{consistent}(c', c''', e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{consistent}(c, e, s_{\text{pre}}, s_{\text{post}}') \]

For disjunction:

\[ \mathcal{B}[B_1 \lor B_2] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \]
\[ = \text{(by the definition of } \mathcal{B}) \]
\[ \mathcal{B}[B_1] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \lor \mathcal{B}[B_2] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \]

Hence, for any \( cs \in \mathcal{B}[B_1 \lor B_2] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \), either \( cs \in \mathcal{B}[B_1] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \) or \( cs \in \mathcal{B}[B_2] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \).

Case 1 (\( cs \in \mathcal{B}[B_1] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \)):

\[ \text{consistent}(c, c', e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{(by the inductive hypothesis)} \]
\[ B_1[e, s_{\text{pre}}, s_{\text{post}}'] \]

Case 2 (\( cs \in \mathcal{B}[B_2] (e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}}' \)):

\[ \text{consistent}(c, c', e, s_{\text{pre}}, s_{\text{post}}') \]
\[ \Rightarrow \text{(by the inductive hypothesis)} \]

20
\[ B_2[e, s_{\text{pre}}, s'_{\text{post}}] \]

Hence:

\[ \text{consistent}(cs, e, s_{\text{pre}}, s'_{\text{post}}) \]
\[ \iff \text{(by the above and proof by cases)} \]
\[ B_1[e, s_{\text{pre}}, s'_{\text{post}}] \lor B_2[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \iff \text{(by the definition of SPECS-C++)} \]
\[ (B_1 \lor B_2)[e, s_{\text{pre}}, s'_{\text{post}}] \]

For implication:

\[ B[B_1 \Rightarrow B_2][e, \{\}, \text{nextv}] s_{\text{pre}} s_{\text{post}} \]
\[ = \text{(by the definition of } B) \]
\[ B[(B_1 \lor B_2) \lor !B_1][e, \{\}, \text{nextv}] s_{\text{pre}} s_{\text{post}} \]

Hence, for any \( cs \in B[B_1 \Rightarrow B_2][e, \{\}, \text{nextv}] s_{\text{pre}} s_{\text{post}} \):

\[ \text{consistent}(cs, e, s_{\text{pre}}, s'_{\text{post}}) \]
\[ \iff \text{(by previous cases of Lemma 5.5 for } \lor, \lor \text{ and } !) \]
\[ (B_1 \lor B_2) \lor !B_1[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \iff \text{(tautology } (P \Rightarrow Q) \iff ((P \land Q) \lor \neg P)) \]
\[ (B_1 \Rightarrow B_2[e, s_{\text{pre}}, s'_{\text{post}}] \]

For universal quantification:

\[ B[orall I \{ (I \land \text{in E} \lor B_1) \Rightarrow B_2 \}][e, \{\}, \text{nextv}] s_{\text{pre}} s_{\text{post}} \]
\[ = \text{(by the definition of } B) \]
\[ \text{let } (v, \text{nextv'} = \text{newvar(} \text{nextv}) \text{ in} \]
\[ (\lambda (v', \text{nextv'}). \{ (v' \cup \text{in} \text{Forall}(I, v, B_1 \Rightarrow B_2, e, s_{\text{pre}}, s_{\text{post}})), \text{nextv'}) \})^+ \]
\[ E[E][v, e, \{\}, \text{nextv}] s_{\text{pre}} s_{\text{post}} \]
\[ = \text{(by the definition of } f^+) \text{ for any } f \]
\[ \text{let } (v, \text{nextv'} = \text{newvar(} \text{nextv}) \text{ in} \]
\[ \{ (v' \cup \text{in} \text{Forall}(I, v, B_1 \Rightarrow B_2, e, s_{\text{pre}}, s_{\text{post}})), \text{nextv'}) \}| (v', \text{nextv'}) \in E[E][v, e, \{\}, \text{nextv}] s_{\text{pre}} s_{\text{post}} \}

Hence (using \( E[e, s_{\text{pre}}, s'_{\text{post}}] \) to mean the value of \( E \) under the given environment and stores):

\[ \text{consistent}(v' \cup \{ \text{in} \text{Forall}(I, v, B_1 \Rightarrow B_2, e, s_{\text{pre}}, s_{\text{post}})\}, e, s_{\text{pre}}, s'_{\text{post}}) \]
\[ \iff \text{(by Lemma 5.7)} \]
\[ \text{consistent}(v' \cup \{ \text{in} \text{Forall}(I, v, B_1 \Rightarrow B_2, e, s_{\text{pre}}, s'_{\text{post}})\}, e, s_{\text{pre}}, s'_{\text{post}}) \]
\[ \iff \text{(by Lemma 5.9 — the soundness of } E) \]
\[ \text{consistent}(\{ \text{in} \text{Forall}(I, ds, B_1 \Rightarrow B_2, e, s_{\text{pre}}, s'_{\text{post}})\}, e, s_{\text{pre}}, s'_{\text{post}}), \text{with } ds = E[e, s_{\text{pre}}, s'_{\text{post}}], \text{where } E \text{ is the domain of the bound variable } I \text{ from the original assertion (above)} \]
\[ \iff \text{(by the definition of consistent)} \]
\[ \text{eval(} \text{in} \text{Forall}(I, ds, B_1 \Rightarrow B_2, e, s_{\text{pre}}, s'_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) \]
\[ \iff \text{(by the definition of eval)} \]
\[ \land_{v' \in ds} (B_1 \Rightarrow B_2)[\text{updatev}[I][v', e, s_{\text{pre}}, s'_{\text{post}}]], \text{where } v' \text{ is a fresh variable} \]
\[ \iff \text{(by the definition of } \land) \]
\[ v' \in ds \Rightarrow ((B_1 \Rightarrow B_2)[\text{updatev}[I][v', e, s_{\text{pre}}, s'_{\text{post}}]) \]
\[ \iff \text{(by the definition of } \text{updatev}) \]
\[ (I \land \text{in ds} \Rightarrow (B_1 \Rightarrow B_2))[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \iff (ds = E[e, s_{\text{pre}}, s'_{\text{post}}]) \]
\[ (I \land \text{in E} \Rightarrow (B_1 \Rightarrow B_2))[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \iff \text{(tautology: } (P \Rightarrow (Q \Rightarrow R)) \iff ((P \land Q) \Rightarrow R)) \]
\[ (I \land \text{in E} \land B_1) \Rightarrow B_2[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \iff \text{(generalization)} \]
\[ (\forall I ((I \land \text{in E} \land B_1) \Rightarrow B_2))[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ (\forall T \in \text{Specs}(\mathbb{C} \cup \mathbb{S})) \]
\[ (I \text{ in } E \land \text{B}_1 \Rightarrow \text{B}_2)[e, s_{\text{pre}}, s'_{\text{post}}] \]

For existential quantification:

\[ B[\exists \text{t I } [B]](e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}} \]
\[ = (\text{by the definition of } B) \]
\[ \text{let } (v, \text{nextv'}) = \text{newvar}(\text{nextv}) \text{ in } \]
\[ B[B](\text{updateenv}[I] v e, \{\}, \text{nextv'}) s_{\text{pre}} s_{\text{post}} \]

Hence, for any \((c, \text{nextv}^g) \in B[\exists \text{t I } [B]](e, \{\}, \text{nextv}) s_{\text{pre}} s_{\text{post}}:\]

consistent\((c, e, s_{\text{pre}}, s'_{\text{post}})\)
\[ \Leftrightarrow (\text{by the inductive hypothesis}) \]
\[ B[\text{updateenv}[I] v e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \Leftrightarrow (\text{by the definition of } \text{updateenv} \text{ and tautology: } P(v) \Leftrightarrow \exists x P(x)) \]
\[ \exists I \text{B}[e, s_{\text{pre}}, s'_{\text{post}}] \]
\[ \Leftrightarrow (\text{by the definition of Specs-}\mathbb{C} \cup \mathbb{S}) \]
\[ \exists \text{t I } [B](e, s_{\text{pre}}, s'_{\text{post}}) \]

This completes the proof of Lemma 5.5. \[ \blacksquare \]

To prove Lemma 5.6, we note that \(I\) simply applies function \(S\) repeatedly (in the fixed point construction) to simplify a set of constraints (see Figure 11). Hence, we prove Lemma 5.6 by induction on the number of applications of \(S\).

For 0 applications of \(S\), we have 0 iterations of the fixed point construction, which yields the identity function, and:

consistent\((c, e, s_{\text{pre}}, s'_{\text{post}}, \text{nextv})\) \(\Leftrightarrow\) consistent\((c, e, s_{\text{pre}}, s'_{\text{post}}, \text{nextv})\)

is a tautology.

For the inductive step, we need to look at the sequence of applications of \(S\) that produces the final constraint set \(c'\) and final store \(s'_{\text{post}}\). We state this inductive step as a lemma.

**Lemma 5.8 Soundness of \(S\):**

At some arbitrary point in a sequence of applications of \(S\), let \(cs''\) be the current constraint set and let \(c\) be the constraint chosen for simplification. Furthermore, let \(cs'''\) be one of the constraint sets returned by \(S\), and let \(cs''''\) be the input constraint set for the next application of \(S\) in the sequence. That is:

\((c'', s''_{\text{pre}}, s''_{\text{post}}, cs''''', \text{nextv''''}) \in S \text{c } (e, s_{\text{pre}}, s_{\text{post}}, cs''' - \{c\}, \text{nextv''})\)

then:

consistent\((cs''', e, s_{\text{pre}}, s'_{\text{post}})\) \(\Leftrightarrow\) consistent\((c, e, s_{\text{pre}}, s'_{\text{post}})\)

Lemma 5.8 captures precisely how \(S\) is used by \(I\). We prove Lemma 5.8 by showing that it is true for each kind of constraint, i.e. for each possible value of \(c\).

For \(\forall\) constraints (representing universal quantification), we have from the definition of \(S\) (Figure 13), with some renaming of variables for clarity:

\[ S \text{inForall}(I, ds, B', e', s_{\text{pre}}, s_{\text{post}})(e, s_{\text{pre}}, s''_{\text{post}}, cs'' - \{\text{inForall}(I, ds, B', e', s_{\text{pre}}, s_{\text{post}})\}, \text{nextv}) \]
\[ = \text{(by Lemma 5.7)} \]
\[ S \text{inForall}(I, ds, B', e', s''_{\text{post}})(e, s_{\text{pre}}, s''_{\text{post}}, cs'' - \{\text{inForall}(I, ds, B', e', s''_{\text{post}})\}, \text{nextv}) \]
\[ = \text{(by definition)} \]

**cases ds of**
\[ \text{isValue(d)} \rightarrow \text{cases d of} \]
isSet(vl) → (if vl = nil then \{e, spre, s′_{post}, cs''_{, nextv}\}
else let (v, nextv') = newvar(nextv) in
let e'' = updateenv [[l] v e'’ in
let (v', nextv'') = newvar(nextv) in
(\lambda(cs’, nextv''). \{e, spre, s′_{post},
  cs’ \cup \{inMember(v, ds), inMinus(ds, v, v’),
inForall(I, v’, B’, e’, spre, s′_{post})\},
  nextv''\})^+
B[B’’ (e'', cs'', nextv'') spre s′_{post}]
else \{e, spre, s′_{post}, \{inFalse()\}, nextv\}
end)

end

isVariable(i) → \{\{e, spre, s′_{post}, cs'' \cup \{inForall(I, ds, B’, e’, spre, s′_{post})\}, nextv\}\}

If ds is a variable, we have that cs''_{= cs'' where cs''_{ is the constraint set component of the value
returned from S, and so:

consistent(cs'', e, spre, s′_{post}) ⇔ consistent(cs''_{, e, spre, s′_{post}}).

If ds is a value but not a set, the constraint set component of the value returned from S (always referred to hereafter as cs''_{ is \{inFalse()\}. Hence, consistent(cs''_{, e, spre, s′_{post}}) is false. We assume that specifications
have been type checked before being presented to the semantics, and so this case cannot occur.

If ds is the empty set, we have the following:

cs'' = cs''_{ = \{isForall(I, ds, B’, e’, spre, s′_{post})\}

eval(isForall(I, ds, B’, e’, spre, s′_{post}), e, spre, s′_{post}) = true

Hence:

consistent(cs''_{, e, spre, s′_{post}})
⇒ (by eval(isForall(I, ds, B’, e’, spre, s′_{post}), e, spre, s′_{post}) = true and the definition of consistent)
consistent(cs''_{, e, spre, s′_{post}})

For ds a nonempty set, we have:

cs''_{ = (cs''_{ = \{isForall(I, ds, B’, e’, spre, s′_{post})\}} \cup
  \{isForall(I, v’, B’, e’, spre, s′_{post}), inMember(v, ds), inMinus(ds, v, v’)\}

Hence:

consistent(cs''_{, e, spre, s′_{post}})
⇒ (from the equality given above)
consistent((cs''_{ = \{isForall(I, ds, B’, e’, spre, s′_{post})\}}) \cup
  \{isForall(I, v’, B’, e’, spre, s′_{post}), inMember(v, ds), inMinus(ds, v, v’)\}, e, spre, s′_{post})
⇒ (tautology ((\forall x. x \in S \Rightarrow P(x)) \land P(v)) ⇔ (\forall x. x \in S \cup \{v\} \Rightarrow P(x)) and the definition of consistent)
consistent(cs''_{, e, spre, s′_{post}})

For Not constraints (representing negation), we have from the definition of S (Figure 13), with some
renaming of variables for consistency:

S inNot(B’, e’, spre, s′_{post}) (e, spre, s′_{post}, cs''_{ = \{inNot(B’, e’, spre, s′_{post})\}, nextv}\)
= (by Lemma 5.7)

S inNot(B’, e’, spre, s′_{post}) (e, spre, s′_{post}, cs''_{ = \{inNot(B’, e’, spre, s′_{post})\}, nextv} =
let res = (fix I') (adjustState e’, spre, s′_{post} (B[B’] (e’, cs''_{ = \{inNot(B’, e’, spre, s′_{post})\}, nextv) spre s′_{post}))
in if \forall (e'', spre, s′_{post}, cs''_{, nextv'}) \in res . cs''_ = \{\} then \{e’, spre, s′_{post}, \{inFalse()\}, nextv\}
else if inFalse() \in cs''_ then \{e’, spre, s′_{post}, cs''_{, nextv'}

\{inNot(B’, e’, spre, s′_{post})\}, nextv’\}

23
\[
\text{\texttt{else}} \{ (e, s_{\text{pre}}, s'_{\text{post}}, cs \cup \{\text{inNot}(B', e', s_{\text{pre}}, s'_{\text{post}})\}, \text{nextv}) \}
\]

We inductively assume that the Soundness Theorem holds for the nested use of fix \(I^+\) above. That is:

\[
\forall (e^m, s'_{\text{pre}}, e, s_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . \text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}}) \iff B'[e, s_{\text{pre}}, s'_{\text{post}}]
\]

If \(\forall (e^m, s'_{\text{pre}}, e, s_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . cs^m = \emptyset\), then we have that \(cs^m = \{\text{inFalse()}\}\). To show this case, we first show that the \text{Not} constraint involved evaluates to false. We can then show that neither \(cs^m\) nor \(cs'^m\) is consistent, and so are equivalent.

First:

\[
\text{eval(isNot}(B', e', s_{\text{pre}}, s_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by Lemma 5.7})
\]

\[
\text{eval(isNot}(B', e', s_{\text{pre}}, s'_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by the definition of eval})
\]

\[
\text{not } B'[e', s_{\text{pre}}, s'_{\text{post}}] \\
\iff (\text{by inductive use of the Soundness Theorem})
\]

\[
\text{not } \forall (e^m, s'_{\text{pre}}, e, s_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . \text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by } \forall (e^m, s'_{\text{pre}}, e, s_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . cs^m = \emptyset \text{ and the definition of consistent})
\]

\[
\text{not true} \\
\iff \text{false}
\]

Now:

\[
\text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by } cs = \{\text{isNot}(B', e', s_{\text{pre}}, s_{\text{post}})\} \cup (cs' = \{\text{isNot}(B', e', s_{\text{pre}}, s_{\text{post}})\}))
\]

\[
\text{consistent}(\text{isNot}(B', e', s_{\text{pre}}, s_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by the above and the definition of consistent})
\]

\[
\text{false} \\
\iff \text{consistent}(\text{inFalse()}) \\
\iff (\text{by } cs^m = \{\text{inFalse()}\})
\]

\[
\text{consist}\text{ent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by the definition of consistent})
\]

\[
\text{eval(isNot}(B', e', s_{\text{pre}}, s_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by Lemma 5.7})
\]

\[
\text{eval(isNot}(B', e', s_{\text{pre}}, s'_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by the definition of eval})
\]

\[
\text{not } B'[e', s_{\text{pre}}, s'_{\text{post}}] \\
\iff (\text{by inductive use of the Soundness Theorem})
\]

\[
\text{not } \forall (e^m, s'_{\text{pre}}, e, s_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . \text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by } \forall (e^m, s'_{\text{pre}}, e, s_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . \text{inFalse()} \in cs^m \text{ and the definition of consistent})
\]

\[
\text{not false} \\
\iff \text{true}
\]

Hence:

\[
\text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}}) \\
\iff (\text{by } cs = cs' = \{\text{isNot}(B', e', s_{\text{pre}}, s_{\text{post}})\} \text{ and eval(isNot}(B', e', s_{\text{pre}}, s_{\text{post}}), e, s_{\text{pre}}, s'_{\text{post}}) = \text{true})
\]

\[
\text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}})
\]

Finally, if neither \(\forall (e^m, s'_{\text{pre}}, s'_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . cs^m = \emptyset\) nor \(\forall (e^m, s'_{\text{pre}}, s'_{\text{post}}, cs^m, \text{nextv}) \in \text{res} . \text{inFalse()} \in cs^m\), is true, then we have that \(cs^m = cs'^m\), and so:

\[
\text{consistent}(cs^m, e, s_{\text{pre}}, s'_{\text{post}})
\]
\\leftrightarrow (by \(cs''' = cs''\))
\[\text{consistent}(cs'', e, s_{\text{pre}}, s'_{\text{post}})\]

For \(False\) constraints (used to indicate failure), we have from the definition of \(S\) (Figure 13), with some renaming of variables for consistency:
\[S \text{in}False((e, s_{\text{pre}}, s_{\text{post}}, cs'') = \{e, s_{\text{pre}}, s_{\text{post}}, \{\text{in}False\}, nextv\})\]

Since \(\text{in}False() \in cs''\) and \(cs''' = \{\text{in}False\}\), we have:
\[
\text{consistent}(cs''', e, s_{\text{pre}}, s'_{\text{post}}) \\
\leftrightarrow (by \(cs''' = \{\text{in}False\}\) and the definition of consistent)
false
\leftrightarrow (by \(\text{in}False() \in cs''\) and the definition of consistent)
\text{consistent}(cs'', e, s_{\text{pre}}, s'_{\text{post}})
\]

The proof of Lemma 5.8 for the remaining types of constraints is omitted. ■

To complete the proof of Lemma 5.2, we need to show the soundness of \(\mathcal{E}\).

**Lemma 5.9 Soundness of \(\mathcal{E}\):**
\[
\forall E \in \text{Expression}. \forall (cs, nextv') \in \mathcal{E}[E] \upsilon(e, \{\}, nextv) s_{\text{pre}} s_{\text{post}}.
\forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', nextv'') \in (\text{fix } \Gamma^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, nextv')\}.
\text{consistent}(cs', e, s_{\text{pre}}, s'_{\text{post}}) \leftrightarrow \upsilon = \mathcal{E}[e, s_{\text{pre}}, s'_{\text{post}}]
\]

Note that the argument \(\upsilon\) to \(\mathcal{E}\) represents the value of the expression. Here, we mean \(\upsilon\) as a \textit{Value}, not a \textit{Variable}.

To prove the soundness of \(\mathcal{E}\), we use two lemmas. These lemmas play the same role in this proof as Lemmas 5.5 and 5.6 in the proof of the Soundness Theorem.

**Lemma 5.10** \(\forall E \in \text{Expression}. \forall (cs, nextv') \in \mathcal{E}[E] \upsilon(e, \{\}, nextv) s_{\text{pre}} s_{\text{post}}.
\forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', nextv'') \in (\text{fix } \Gamma^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, nextv')\}.
\text{consistent}(cs', e, s_{\text{pre}}, s'_{\text{post}}) \leftrightarrow \upsilon = \mathcal{E}[e, s_{\text{pre}}, s'_{\text{post}}]■

**Lemma 5.11** \(\forall E \in \text{Expression}. \forall (cs, nextv') \in \mathcal{E}[E] \upsilon(e, \{\}, nextv) s_{\text{pre}} s'_{\text{post}}.
\forall (e', s'_{\text{pre}}, s'_{\text{post}}, cs', nextv'') \in (\text{fix } \Gamma^+) \{(e, s_{\text{pre}}, s_{\text{post}}, cs, nextv')\}.
\text{consistent}(cs, e, s_{\text{pre}}, s'_{\text{post}}) \leftrightarrow \text{consistent}(cs', e, s_{\text{pre}}, s'_{\text{post}})■

Together, these two lemmas imply Lemma 5.9. The proof of Lemma 5.10 (omitted) is accomplished by induction on the structure of syntax domain Expression, and so is very similar to the proof of Lemma 5.5. The proof of Lemma 5.11 (omitted) is accomplished by induction on the number of of applications of \(S\), and so is similar to the proof of Lemma 5.6. ■

### 6 Conclusion

We have presented an algorithm for executing a formal specification by translation to constraint programs, and we have shown the soundness of that algorithm. For software engineers, this result has two important benefits. First, a specifier can use our algorithm for executing and testing specifications with confidence that all results returned will be correct. Second, by presenting our algorithm in a way that is relatively independent of programming language (or even paradigm), we have provided a correctness criterion for the execution of SPECS-C++ or other model-based specifications.

Our work should also be of interest to those working on semantics of logic and constraint logic programming languages. While our semantics is similar in scope to the semantics for these kinds of programming languages, our approach is considerably different. Our approach directly returns a set of final states rather than using continuations or transition axioms, and abstracts away from the control built into a particular language. It thus provides a more denotational, as opposed to operational, semantics for such languages.
References


