Design and Implementation of the Larch/C++ Type System

Matthew W. Markland

TR #98-05
June 1998

Keywords: Specification languages; Larch; LSL; Larch/C++; type systems.

1997 CR Categories:
D.1.5 [*Programming Techniques*] Object-oriented Programming; D.2.1 [*Software Engineering*] Requirements/Specifications — languages, Larch, tools; D.3.3 [*Programming Languages*] Language Constructs and Features — modules, packages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, pre- and post-conditions, specification techniques, LSL; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — type structure.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Contents

List of Figures

Design and Implementation of the Larch/C++ Type System

Matthew W. Markland

Original work: July 14, 1997
Final revision: June 3, 1998

Abstract

This paper describes the design of a type system for the Larch/C++ specification language. To motivate the features of the type system, the type systems of both the Larch Shared Language and C++ are described. After this background, an informal description of the Larch/C++ type system is followed by a formal presentation of the type rules. The implementation of an infrastructure for the type system is then described.

1 Introduction

The goal of Larch/C++ [8] is to give the programmer a formal specification language that is expressive and useful in practice. The work described in this paper focuses on the addition of type checking capabilities to the Larch/C++ Checker. In this context a *type* is a set of values that exhibit uniform behavior under a set of associated operations [13]. *Type checking* defines a process by which a set of formal rules that describe the type system are applied to operations and statements in a given specification to decide whether the operations and statements are type consistent (the terms *sort* and *sort checking* will also be used interchangeably for type and type checking). This functionality allows programmers and specifiers to gain useful information as to whether the design of a program is sensible and type consistent before its actual implementation.

Work has been done in the following areas:

- The creation of a formal description of the Larch/C++ type system.
- The coding of basic functionality to support the implementation of the Larch/C++ type system. This work has many smaller pieces including:
  - Support for User Interaction
    - Improved usability and control over the Larch/C++ Checker via the creation of a number of command line arguments.
  - Support for Larch Shared Language Constructs
    - Modification of the LSL Checker to follow Unix conventions and use less memory.
    - Development of an interface between the existing Larch/C++ Checker and the LSL Checker.
  - Support for the Evolving C++ Language Standard
    - Support for Draft ANSI Standard C++ [3] language constructs.
    - Support for translation of C++ declarations into Larch/C++ sorts.

The paper begins with a background section that introducing formal methods, specification, and the Larch/C++ behavioral interface specification language. Following the general introduction, the type systems of the Larch Shared Language and C++ are described briefly. These descriptions motivate a discussion of the basic functionality of the Larch/C++ type system. This functionality is then formalized in a description of the sort rules for Larch/C++. Finally the details of the implementation of the functionality mentioned above are presented.

2 Background

2.1 Formal Methods

The information in this section is based upon Wing's paper [14].

Formal methods define processes that are used for software development. Built upon a mathematical basis, these processes are designed to reveal ambiguities, incompleteness, and inconsistencies in software as it is developed. A formal method will typically define the specific vocabulary and steps involved in designing a piece of software.

Formal specifications may be a part of a specific formal method. *Formal specification* describes a process by which an abstraction of a problem may be defined and expressed. Usually the abstraction is expressed in some language specifically designed for the purpose. Once clearly expressed, the abstraction may serve as documentation of the problem, a means to communicate the problem clearly, and/or as a contract defining the problem. Specifications and their languages are based upon mathematical properties making them more precise and concise than informal specifications based upon natural languages. The rigorous definition and mathematical basis behind formal specification languages also makes it easier to apply machine analysis and manipulations to them than to specifications in an informal language.

Larch/C++ is a behavioral interface specification language. A *behavioral* specification language is used to define an abstraction for a system based upon the behavior of that system under certain conditions. In other words, it describes a system's behavior as observed from the outside. Larch/C++ describes behavior using a *model-based* approach; a user builds an abstract model of the system which describes its behavior. The abstract model then becomes a way of expressing the real world in a manner that can be controlled and reasoned about. The basic pieces of the model are the interfaces which exist between the pieces of the system. After a user has described the behavior in terms of the interfaces, solutions to the problem may then be designed based upon the formal contract defined by the model.

2.2 Introduction to Larch/C++

Larch/C++ [8] is a model-based, formal specification language tailored for the specification of the behavior of C++ program modules or application program interfaces (API's). Larch/C++ is not designed to specify the behavior of an entire program; instead it allows for the precise, unambiguous documentation of the behavior C++ program modules (functions, classes, etc.). Larch/C++ adds syntax to C++ to allow the specification of complex C++ structures, the inheritance of specifications, and the clear specification of the interface to a class.

Larch/C++ is a two-tiered specification language. Specifications consist of Larch Shared Language (LSL) [6] traits which describe the abstract models, and interface specifications which formalize the behavioral contracts.

This two-tiered approach allows for the clear separation of the definition of the abstract model and its vocabulary from the actual details of the programming language modeled by the interface language. One reason for the separation is so that the abstract models may be written so that they can be reused in other specifications. If the abstract models were written in a language closer to the actual implementation, it would be more difficult to reuse the same model in a specification for an implementation written in a different language.

2.2.1 The Larch Shared Language

The Larch Shared Language [6] allows an user to supply basic semantic information, and a specialized vocabulary for describing abstract values. The basic unit of LSL is the *trait*. Traits contain information on *sorts*, which are like types in a programming language, and *operators* which define various operations upon these sorts. Figure 1 illustrates the LSL portion of a specification for a simple counter [8]. This particular trait illustrates four common parts that are in many traits. The `includes` statement allows for a trait to build upon and reuse previously written traits. All of the information from the traits listed in this statement are available in the following sections of the current trait. Items from the included traits may be renamed syntactically to make the new trait more readable. This provides a shortcut for users, allowing them to reuse previous work. The `introduces` section defines the abstract model's operations. In this case, a Counter may be created via `newCounter` or `inc`, and have its value reported via `value`. The `asserts` section supplies meaning to these operations by logically illustrating how the abstract values are manipulated by the operators. For now, note that a trait defines a model consisting of a set of abstract values and a set of operations upon those values. Examples of possible abstract values for the trait in Figure 1 are `newCounter`, representing a brand new counter, and `inc(newCounter)` representing a counter that has been incremented once. These values are independent of any implementation of Counter.

2.2.2 The Larch/C++ Specification Language

Figure 2 contains the Larch/C++ specification for a class that implements the counter modeled by the previous trait [8]. Larch/C++ specifications consist of a C++ header file which contains additional annotations set off in

```
 @(#)$Id: CounterTrait.lsl,v 1.3 1994/12/09 02:48:06 leavens Exp $
CounterTrait: trait
  includes Natural(Nat), NoContainedObjects(Counter)
  introduces
    newCounter: -> Counter
    inc: Counter -> Counter
    value: Counter -> Nat
    Limit: -> Nat
  asserts
    Counter generated by newCounter, inc
    Counter partitioned by value
    \forall c: Counter
      value(newCounter) == 0;
      value(inc(c)) == value(c) + 1;
      0 < Limit;
```

Figure 1: CounterTrait.lsl

specially marked C++ comments. The use of the comment delimiters, `//@` and `/*@ ... @*/`, allows the annotations to be easily embedded directly into new or existing C++ source code. Larch/C++ annotations contain various keywords. In this example, the first section is a *uses-clause*, which serves a function similar to the `#include` directive in C++; it tells the Larch/C++ Checker the traits that will be used by this specification. Following that, the next annotations define an `invariant` and a `constraint`. The `invariant` describes a condition for the C++ class that must always be true. The `constraint` describes any limits that this class must adhere to.

Most individual function specifications consist of at least three pieces: a requires clause, a modifies clause, and an ensures clause. The *requires clause* states the conditions that must be met before an individual function may be called. If these conditions are not met, there is no guarantee that the function will run correctly. The *modifies* clause lists all objects whose state may be initialized or modified by the execution of this function. Only objects listed in this clause may change state; thus the modifies clause acts as a frame axiom. The *ensures clause* serves to state the expected results of the function provided the conditions specified in the requires clause were met.

A closer look at the `increment` function specification illustrates these sections.

```
virtual void increment();
//@ behavior {
//@   requires value(self^) < Limit;
//@   modifies self;
//@   ensures  self' = inc(self^);
//@ }
```

The first line is the C++ prototype for the function. The `behavior` keyword announces the beginning of the body of the specification. The requires clause states that for this function to be called the value of the counter must be less than the value of `Limit`. If it is not, then the behavior of the function is not specified. If the requires clause is met and function is called, the modifies clause states that the only possible item that can change is the counter itself. The ensures clause then states that, if the previous conditions in the requires clause are met, the value of the counter after the invocation of this function will be the result of incrementing the counter.

3 The Type System

The type system for Larch/C++ is built upon the type systems of LSL and C++. The process of building a type system involves defining how the language is scoped, how individual terms are assigned types, and how the statements and expressions type check. *Scoping* is a description of the visibility of identifiers. The Larch/C++ type system bases its assignment of types and rules for type checking expressions and statements more on the LSL type system, while the scoping rules are based upon those of C++. This leads to an interesting algorithm for looking up variables.

Recall that the terms sort and type, and the corresponding sort checking and type checking, are used interchangeably throughout this paper.

```
// @(#)$Id: Counter.lh,v 1.7 1997/01/12 22:27:38 leavens Exp $
// See J.P. Lejacq's paper in SIGPLAN 26(10), Oct, 1991

//@ uses CounterTrait;

class Counter {
public:
  //@ invariant _value(self\any) <= Limit;
  //@ constraint value(self\pre) <= value(self\post);

  Counter();
  //@ behavior {
  //@   modifies self;
  //@   ensures  self' = newCounter;
  //@ }

  virtual int cnt_value() const;
  //@ behavior {
  //@   ensures result = value(self^);
  //@ }

  virtual void increment();
  //@ behavior {
  //@   requires value(self^) < Limit;
  //@   modifies self;
  //@   ensures  self' = inc(self^);
  //@ }

  virtual void reset();
  //@ behavior {
  //@   modifies self;
  //@   ensures  value(self') = 0;
  //@ }
};
```

Figure 2: Counter.lh

3.1 The LSL Type System

As mentioned earlier in Section 2.2.1 the basic unit of structure in LSL is the trait. Traits are used to define sorts and their associated operations, which exhibit uniform behavior under their associated operations.

Since the Larch/C++ language depends upon LSL to supply abstractions, it was important to learn how the LSL type system works. Unfortunately, little has been written on the LSL type system. Neither the technical report on LSL [7] nor the Larch book [6] offered any details. What was available was the LSL Checker [11]. The LSL Checker is a tool that will perform semantic and syntactic checks on LSL traits. Since the LSL Checker provides type checking of traits, it serves as an operational definition of the LSL type system. Given this definition, information about the type system could be generated by running the Checker on example traits. One option of the LSL Checker that assisted in the development of the traits, and the initial ideas about the system was its `-syms` option. This option will cause the LSL Checker to produce a list of the operators and their signatures contained within the LSL trait. An example of this output, generated by the command line `lsl -syms Example1.lsl` is illustrated in Figure 3 (Note: this is only a portion of the actual output). Please refer to Figure 4 for the trait `Example1.lsl`, which generated this output.

Each line of the `-syms` output consists of a operator name and an associated signature for that operator; these parts are separated by a colon. In the example above, there is an operator named `x` which takes no input and returns an item of sort `->int`. There are also operators such as `f`, which have two different signatures. The ability of an operator to have more than one distinct signature is an example of overloading. *Overloading* occurs when a given name can simultaneously stand for multiple, distinct functions.

After examining the `-syms` output it was clear that LSL supported overloading. However, the extent of LSL's overloading and its overload resolution techniques were still unknown. A series of traits were created that used different sets of functions and the built-in `if-then-else` LSL operator. The `if-then-else` operator was chosen because it was built-in, because it allowed for experimentation with types that might not be numeric, and because it requires the `then` and `else` clauses to have the same unique sort. The LSL Checker was run on the traits to see what kind of errors, if any, it found.

Figure 4 is an example of an LSL trait that successfully sort checks. Working under the assumption that terms could carry sets of sorts, this

```
if __ then __ else __: Bool, Bool, Bool -> Bool
if __ then __ else __: Bool, Int, Int -> Int
if __ then __ else __: Bool, int, int -> int
if __ then __ else __: Bool, float, float -> float
true: -> Bool
false: -> Bool
x: -> int
y: -> Bool
f: int -> float
f: int -> Bool
g: Bool -> float
q: int, Bool -> Bool
q: int, Bool -> float
...
```

Figure 3: Partial output from the `lsl -syms` command

trait sort checks in the following way. The term `q(a,b)` has the set of sorts {`Bool`,`float`} associated with it. Similarly the term `f(a)` has the set {`float`,`Bool`}, and the term `g(b)` has the set {`float`}. Since the `if-then-else` function requires the `then` and `else` clauses to have the same sort, the resulting sort of this operator must be `float`. This illustrates an important point. In this case, the system chooses the version of operator `f` that it will use in the type checking based upon what it needs in the context of the `if-then-else` operator. In this case, by choosing the intersection of the two sets, the LSL Checker chooses to use `f:int -> float` because it works in the context where `g(b)` only has one sort. This is an example of context-dependent overload resolution. *Context-dependent overloading* means that the context in which the function or variable appears is used to help uniquely identify its sort [13]. Finally, since both sides of the `==` operator have a sort in common, and it is known from the `-syms` output that it requires equivalent sorts for its arguments, the expression sort checks.

Figure 5 illustrates a situation where the trait will not sort check. This trait was designed with the purpose of confirming the assumption made in the above example: that all terms may have sets of sorts associated with

```
Example1:trait

   includes Integer

   introduces
        x: -> int
        y: -> Bool
        f: int->float
        f: int->Bool
        g: Bool->float
        q: int,Bool->Bool
        q: int,Bool->float

   asserts
        \forall a:int,b:Bool
        q(a,b) == if true then f(a) else g(b);
```

Figure 4: Example1.lsl

```
Example2:trait

   includes Integer
   introduces
        x: -> int
        y: -> Bool
        f: int->int
        f: int->Bool
        f: int->E
        g: Bool->float
        q: int,Bool->Bool
        q: int,Bool->float

   asserts
     \forall a:int,b:Bool
        q(a,b) == if true then f(a) else g(b);
```

Figure 5: Example2.lsl

them.

```
./Example2.lsl:16: (near col 18): 'if __ then __ else __' not
declared with matching domain sorts
        Possible sorts for arg 1: Bool
        Possible sorts for arg 2: int, Bool, E
        Possible sorts for arg 3: float
Abort: error in checking LSL traits
```

The output from the LSL Checker for this trait describes the possible sorts available for the arguments to the `if-then-else` operation. In this case note that arg 2, which is `f(a)`, has three possible sorts. The sort checking error also shows that there must be an intersection between the sets of the arguments for the operator to sort check. In this case there is an empty intersection between `f(a)`'s set {`int`, `Bool`, `E`} and `g(b)`'s set {`float`}. This leads to the error condition.

To follow up on the previous examples, a trait was created to see if similar errors could be generated with other operators. This trait is shown in Figure 6.

```
./Example3.lsl:18: (near col 8): 'q' sorts of terms in equation
do not match
        Possible sorts for left side: Bool, float
        Possible sorts for right side: int, E
Abort: error in checking LSL traits
```

This error shows that the sort associated with the `if-then-else` operator is the set consisting of {`int`, `E`}. So, the LSL Checker will try to find a sort for `q(a,b)` that would fit the constraint that operator `==` needs to have the same sort for each argument. In this case, since `q(a,b)` has the set {`Bool`,`float`} associated with it, there is no possible solution.

The previous examples have shown that for an operator like `==` to sort check, there needs to be a non-empty intersection between the sets of sorts for its arguments. What if that set has a cardinality larger than one? Figure 7 is an example where the intersection between the sets of sorts for the arguments to `==` does not have an intersection of cardinality one. This generates the following output from the LSL Checker.

```
Example3:trait
   includes Integer

   introduces
        x: -> int
        y: -> Bool
        f: int->int
        f: int->Bool
        f: int->E
        g: Bool->float
        g: Bool->int
        g: Bool->E
        q: int,Bool->Bool
        q: int,Bool->float

   asserts
     \forall a:int,b:Bool
        q(a,b) == if true then f(a) else g(b);
```

Figure 6: Example3.lsl

```
Example4:trait
   introduces
        x: int->int
        x: int->float
        y: int->int
        y: int->float
        p: int,int->int
        p: int,int->float

   asserts
     \forall a,b:int
        p(a,b) == if (a=b) then x(a) else y(a)
```

Figure 7: Example4.lsl

```
./Example4.lsl:12: (near col 8): ‘p’ more than one possible
sort for terms in equation
Possible sorts: int, float
Abort: error in checking LSL traits
```

Since the intersection is not of cardinality one, the expression does not sort check.

Another property of the LSL type system is that any variables declared must have completely unique names. For example the trait in Figure 8 causes the following error to be issued by the LSL Checker:

```
./foo.lsl:15: (near col 9): ‘x’ variable duplicates constant
of same sort
Abort: error in checking LSL traits
```

Notice how the `x:int` declaration within the `\forall` expression interferes with the `x:   ->int` declaration in the `introduces` section. This error occurs because of constraints placed upon the output the LSL Checker generates for the Larch Prover.

The results of these experiments with the LSL Checker can be summarized as follows:

```
foo:trait

includes Integer

introduces
  x: -> int
  y: -> Bool
  f: int->float
  f: int->Bool
  g: Bool->float
  q: int,Bool->Bool
  q: int,Bool->float

asserts
 \forall x:int,b:Bool
    q(x,b) == if true then f(x) else g(b);
```

Figure 8: A LSL Trait with an error in the declaration of variables

```
asserts
  \forall x:E, s1:Set, s2:Set

   s1 \subset s2 == \A x (x \in s1 => x \in s2) /\ s1 \neq s2
```

Figure 9: An example of quantifier scope.

- LSL terms have non-empty sets of sorts associated with them.
- The elements in a LSL term's set of sorts may be dependent on the context in which the term is used.
- When checking is complete, every expression or equation should have a single sort. If this does not occur, there is a type error.
- Declared variables should have unique names.

When the system sort checks, there will be sets of sorts associated with the various terms. As the sort checking progresses, these sets will be narrowed by the contextual information until every expression has an assigned, singular sort, or has sort checked.

The other major issue in type systems is scoping. Scoping has few complications in LSL. There are two scopes, the global scope and quantifier scope. All names go into the global scope, with the appropriate overloading, unless they are declared in a quantified expression.

An example of the creation of quantifier scope is shown in Figure 9. This example shows a portion of a LSL trait which describes Sets. Notice that the definition of the `\subset` function contains a quantified expression of the form `\A x ( ... )`. The scope of `x` is the text between the parentheses.

3.2 The C++ Type System

This section gives a brief overview of the C++ type system. Although not a complete description, it should serve as an introduction for people unfamiliar with it. The section is based upon the C++ Annotated Reference Manual

[4] which should be referred to for a complete description of the C++ type system.

The C++ type system is based upon the C type system with a few additions. There are two broad categories of types, fundamental types and derived types. *Fundamental types* are the basic building blocks of the type system. They consist of the types that are built-in to the implementation. Examples are `int`, `float`, and `char`. *Derived types* are types built from the fundamental types or from other derived types. Examples of these include arrays, functions, and classes. Type names can be created via the `typedef` construct, which assigns a new name to a previously existing type.

C++ also offers a variety of polymorphic forms. These include templates, static overloading, dynamic overloading, and subtype polymorphism. Although these forms are important to the C++ type system, with the exception of templates, they are not an important part of the Larch/C++ type system. Thus it is beyond the scope of this paper to describe them in more detail.

Templates in C++ let users create generic classes that can be instantiated to support a specific type. A template allows the programmer to pass types as parameters to a class. This structure is an example of parametric polymorphism. *Parametric polymorphism* is when a set of operations require a type parameter that defines their behavior.

Compared to LSL, C++ has a complex scope system. The innermost scope level is local scope. *Local* scope refers to the declarations within a given block. Items declared with local scope are visible within the block they are declared in. *Function* scope refers to use of labels within functions. *Class* scope contains the names of all members, both functions and variables, that are contained within a class definition. Finally, *file* scope refers to any declarations that occur outside of all blocks and classes. Declarations at file scope are visible within the given translation unit (usually the source file).

In general, name lookup in C++ begins within the local scope and moves outward to file scope. The process may be modified by using the scope resolution (`::`) operator to state explicitly where to look for the name. Names may also be hidden or overridden within given scopes. The key idea is that even with the above features, C++ requires that any use of a name be unambiguous within a given scope.

3.3 The Larch/C++ Type System

3.3.1 Overview

The Larch/C++ language has its own unique type system. Though this system has many things in common with both the LSL and C++ type systems, Larch/C++ is its own language. Looking at the structure of a Larch/C++ specification will help to explain the system and its unique properties. Below is a specification for a simple C++ function `increment` which increments a global variable `x`. It is based upon the Counter trait shown in Figure 1.

```
int i;
void increment();
//@ behavior {
//@   requires value(i^) < Limit;
//@   modifies i;
//@   ensures  i' = inc(i^);
//@ }
```

Recall that this specification could be broken down into a C++ portion and the actual Larch/C++ specification. In this case the line

```
int i;
```

is the C++ declaration for the variable `i`. The Larch/C++ type system must be able to take this C++ declaration and convert it into a binding of i to a sort. This sort can then be used later when `i` is mentioned.

The items set off by the `behavior` keyword are behavioral annotations. It is within this section that the Larch/C++ system must do its type checking. Within this section, it is not legal to call an actual C++ function. For example, it would be illegal to write the following:

```
ensures self' = increment();
```

Thus any terms within the specification that look like functions do not refer to C++ functions, but rather to LSL operators defined either by user traits or by the inherent traits for the system. This lack of C++ function calls in the specifications makes the type system simpler. It does not have to understand the C++ concepts of static and dynamic overload resolution. It does need to understand LSL operators, though. Because of this, the Larch/C++ type system acts like the LSL type system in that it supports operator overloading by creating sets of sorts for operators, and it will attempt to determine the sorts for expressions via contextual clues.

The basic notion of the Larch/C++ type system is that there is a correspondence between C++ declarations and the LSL sorts. Larch/C++ creates this correspondence by having a set of basic sorts which correspond to the fundamental types in C++. It also adds auxiliary sorts that allow for the discussion of objects. The Larch/C++ Checker automatically has these basic sorts available and uses them to convert C++ declarations into equivalent sorts. Users may also define abstract values and operations in their own traits. The user would place the trait in a `uses` clause before using any theory from it. The `uses` clause causes the Larch/C++ Checker to generate information about the sorts and operations from the trait and make it available to the rest of the specification.

The `uses` clause also supports the C++ template facility. Since there is no equivalent to parametric polymorphism in LSL, the `uses` clause, when combined with renaming, is used to "instantiate" a trait with the correct sort. An example of this is the `SimpleSet` specification from the Larch/C++ Manual, Section 8.2 [8]. Figure 10 is an example of this for a simple set class implemented via templates. There is one template argument, a class `Elem`, which will be the elements of the set. The specifier needs to create an abstract model that can be used with this trait. The specifier creates the following `uses` clause for this purpose.

```
//@ uses SimpleSetTrait(Elem for E, Set<Elem> for C);
```

This `uses` clause has the effect of specializing the trait so that the basic sorts are now `Elem` and `Set[Elem]` (note that the translation of the `<>` notation to the [] notation is done by the Larch/C++ Checker automatically). This means that an LSL operation, such as `empty`, which used to have a result sort of `C` now has a result sort of `Set[Elem]`.

The other major complication in the Larch/C++ type system is the scoping of identifiers and operators. Recall that the scoping system in LSL is simpler than that of C++. Larch/C++ will build upon both of these type systems to create its own.

Larch/C++ expands the basic C++ scoping system by adding three new scope units. *Function-specification* scope refers to the area inside of a specific function specification. It contains the declarations for the function parameters, the function's `result`, the keyword `self`, and any other declarations

```
// @(#)$Id: SimpleSet.lh,v 1.19 1997/06/03 20:29:52 leavens Exp $

template <class Elem>
  //@ expects contained_objects(Elem);
  //@ where Elem is {
  //@   bool operator ==(Elem x, Elem y);
  //@   behavior {
  //@       ensures returns /\ result = (x = y);
  //@   }
  //@ };
class Set {
public:
  //@ uses SimpleSetTrait(Elem for E, Set<Elem> for C);

  Set() throw();
  //@ behavior {
  //@   constructs self;
  //@   ensures liberally self' = empty;
  //@ }

  void insert(Elem e) throw();
  //@ behavior {
  //@   modifies self;
  //@   ensures liberally self' = self^ \U {e};
  //@ }

  bool is_in(Elem e) const throw();
  //@ behavior {
  //@   ensures result = (e \in self);
  //@ }
};
```

Figure 10: SimpleSet.lh

found there. It is a specialization of the local scope from C++. *Spec-case* scope refers to the fact that in a specification that has multiple cases, each case has its own local scope. This is because each spec-case is essentially a miniature function-specification. Finally, as in LSL, there is a *quantifier* scope. All information in Larch/C++ is scoped. This means that the location of a declaration or a `uses` clause may affect the visibility of certain identifiers.

The goal when looking up an identifier in a Larch/C++ specification is to find a single, unambiguous type that is bound to a given identifier. Larch/C++'s mix of both LSL and C++ declarations makes for more complex insertion and lookup functions for identifiers. One reason for the complexity is that information from C++ declarations and information about LSL operations are kept in separate worlds within a scope. This is done for two reasons:

- it allows for name conflict resolution
- it allows the system to simulate the overloading system present in LSL.

These two ideas are described in more detail below.

The Larch/C++ Checker attempts to insert all identifiers into the symbol table as they would be in C++. This leads to LSL operations being scoped as in C++. The insertion process also does name conflict resolution. Name conflict resolution occurs when, in a given scope, a name for a variable or operation tries to reside in both the C++ and LSL worlds. This is not allowed. If this occurs, the offending LSL trait operation is discarded from the symbol table, giving preference to the C++ declaration. Figures 11 and 12 illustrate a typical situation in Larch/C++ where name conflict resolution is needed. Note that the trait defines two operators: `x`, a zero-argument operator that generates an integer value, and `iden`, which takes an argument and returns it. These operators are then put to use in the specification of the function `foo`. In the specification shown in Figure 12, there is no problem with conflicting names. The trait operations defined in the trait are inserted into an outer scope. The argument `int x` is placed within the function-specification scope. Thus, the two versions of `x` are in different scopes. Since the type system tries to find the most local match, it will choose the formal parameter.

Figure 13 shows a specification very similar to the previous one, except for a name conflict that is harder to resolve. In this case the two possibilities

```
Problem:trait
   includes Integer(int for Int)

   introduces
        x:->int
        iden:int->int

   asserts
     \forall i:int
        iden(i) == i;
```

Figure 11: ProblemTrait.lsl

for `x`, the formal parameter and the LSL operator, both exist within the function-specification scope. A lookup algorithm that tries to find the most local identifier has a problem; there are two good choices. One way of solving this problem is to prioritize the worlds. The system would look in the local C++ world first, then in the world containing the local trait operations. If a conflict exists the Checker would always use the C++ name. Another possibility is to reverse the action, so that the the trait operation is always chosen by the Checker. The best choice, and the one implemented in the Larch/C++ Checker, is to flag this as a type error. Once the user has been informed, the system discards the trait operation that has the name conflict. Once informed, the user may either ignore the error, in which case the local identifier would in this example resolve to the formal parameter, or the trait may be modified; renaming the trait operation that causes the conflict. This would then allow for both the trait operation and the local C++ declaration to coexist. This solution is both flexible and has a default behavior that makes sense.

The discussion of name conflict resolution leads to another important question: how are identifiers looked up in Larch/C++? As mentioned earlier, the goal is to find the most local match, while still offering the feel of "global" scope to the trait operations. Recall from Section 3.1 that LSL operators are overloaded. This means that one name may refer to multiple trait operators. Recall also that the names of these operators all reside in a single scope. The

```
//@ uses Problem;
int foo(int x);
//@ behavior {
//@ ensures result = iden(x);
//@}
```

Figure 12: Function `foo`: version 1

```
int foo(int x);
//@ behavior {
//@ uses Problem;
//@ ensures result = iden(x);
//@}
```

Figure 13: Function `foo`: version 2

given operator name should be able to return any of the possible overloads. In Larch/C++, the information on the LSL operations is scoped. Thus a given name may only see overloads that exist within its scope. This does not correctly match the LSL model, where all the trait operations exist in a single global scope. The Larch/C++ name lookup algorithm needs to simulate the LSL global scope or it may miss some valid operations when sort checking.

A first solution for name lookup might simply search up the symbol table, finding all possible C++ and LSL operation identifiers that match. This brute force solution leads to frequent name conflict problems. Identifiers that should not even be visible may be chosen as solutions, or may cause ambiguities. Other possible solutions limit the search by matching the most local C++ identifier and matching all possible trait operations throughout the symbol table. While this solution seems feasible it has the same flaw: it may return an identifier that leads to an ambiguity.

The lookup algorithm chosen attempts to avoid finding ambiguous identifiers by limiting the search in the trait operation side of the table. The algorithm is illustrated in Figure 14.

This algorithm always locates the most local matching C++ declaration, which it should do to model the C++ system, and always stops searching the trait operations side at a point where name conflicts could begin. The essence is that a C++ declaration "shadows" or hides all possible trait operations of the same name in enclosing scopes. Thus, the lookup of trait operations should stop when that shadowing declaration is seen to avoid ambiguity. There will not be any ambiguity at the point of the shadow declaration because of the algorithm for name conflict resolution.

3.3.2 Examples

Here are some examples of how scoping and name lookup work in Larch/C++. In these examples, the enclosing boxes represent the scope boundaries for the various identifiers. Figure 15 shows the simple traits that will be used in these examples. From our previous LSL examples, recall that the LSL Checker allows constant operators such as `x: -> int` to be used where a variable of sort `int` is called for. Due to the fact that these may be substituted for each other, the sort `int` will be used as shorthand for either `int` or `->int` in the following examples.

Figure 16 shows a small example of a specification. The goal is to build the sets of possible sorts for the variables `x` and `y` and the operation `+`. The

```
Begin within the local scope.
Look in the C++ world for an identifier with the correct name.

If a C++ identifier with the correct name is found,
then return it and the search stops.

If a C++ identifier is not found, search the trait operation
world. If any identifier with the correct name is found, begin a
search in all enclosing scopes.
This search is terminated by one of two conditions:

1. A C++ identifier with the correct name is seen.
   Return the set of trait operations that has been generated.

2. The top of the symbol table is reached.
   Return the set of matching trait operations.

If no matching identifier is found in the local scope,
recurse on the next enclosing scope if it is valid.

At the top of the table, return an error.
```

Figure 14: The Larch/C++ name lookup algorithm

```
XTrait:trait
    introduces
        x: -> int
        y: -> Bool

YTrait:trait
     introduces
        y: -> int

fooTrait:trait
     introduces
        foo: int -> int
        y: -> float
```

Figure 15: Example traits for scope and variable lookup examples

process begins within the function-specifier scope represented by the innermost box in the figure. The system attempts to find either a C++ declaration or a trait operation which is usable for `x`. As there is no declaration for `x` there, it finds the formal parameter declared by `int x` in the enclosing scope. At this point, the search stops and the set of sorts for `x` is {`->int`}.

The system continues by building the set of sorts for `y`. Again it starts in the function-specifier scope. It does not find a match in the two inner scopes. It moves to the next enclosing scope, a class scope. Once in this scope, the search begins within the C++ world. Again, no C++ declaration is found. The search continues in the trait operation world. Here the system finds two possibilities for `y`, either `->int` or `->float`, which are provided by the traits `fooTrait` and `YTrait` mentioned in the `uses` clause. Since neither stopping condition holds, the search is continued into the enclosing scope. Here the system finds a C++ declaration `int y`. Because of this, the system stops searching and returns the set of sorts associated with the trait operations that have been seen. This means that the identifier `y` has the set of sorts {`->int,->float`}.

The search for the trait operation `+` occurs in a similar fashion. In this case, the search goes to the top level of the symbol table where the builtin

```
//@ uses XTrait;
int y;
class foo {

//@ uses fooTrait,YTrait;
int bar(int x);


//@ behavior {
//@ ensures result = x + y;
//@ }

} ;
```

Figure 16: Scope and variable lookup: example 1

sorts and their operations are inserted. The system will have a set of possible signatures for `+` that come both from the built-in traits and from whatever definitions of `+` have been seen due to `uses` clauses. The system will try to find a signature that can be used with the set of types for `x` and `y`. As seen above, the system knows that the set of sorts for `x` is {`int` and it knows that knows that the set of sorts for `y` is {`->int, ->float`}. So since there are no redefinitions of `+`, in this case the system will end up searching the built-in operations which are included at the highest level of the symbol table structure. One of the signatures it finds is `int,int ->int` (there would be others). In this case, the system would chose this signature for `+` because the arguments to `+`, `x` and `y`, have the sort `int` in their sets.

Finally, note that this example would have generated a warning from the Larch/C++ system. Within the outermost scope, there exists a `uses` clause for `XTrait`, which has a definition of `y`, and a C++ declaration of `y`. As the Larch/C++ Checker processes this file, it would first place the definition of `y` from the `uses` clause into the LSL portion of the symbol table. Then when it saw the C++ declaration for `y`, it would use the rules for name conflict resolution described earlier. In other words, it would discard the LSL operation from the symbol table and issue a warning to the user. In this case, the system would still be able to type check this specification. However, in the general case, type checking results when identifiers are discarded due to the name conflict resolution system may be incomplete and possibly inaccurate.

Figure 17 shows another small example of a specification. This specification differs from the previous example in that there is no declaration of the variable `y` in the outer scope. As before, the set of sorts for `x` is {`int`}.

To build the set of sorts for `y`, the system starts in the function-specifier scope. This time, again, it does not find a match. It moves to the next enclosing scope, a class scope. Once in this scope, the search begins within the C++ world. Again, no C++ declaration is found. The search continues in the trait operation world. Here the system finds two possibilities for `y,` either `->int` or `->float`, both of which are provided by the traits in the `uses` clause. Since the stopping conditions were not met, the search is continued into the enclosing scope. At the file scope, there is no C++ declaration for `y`. So the system finds an trait operation imported from `XTrait` that adds another possibility to the set, { `->Bool`}. The system continues to the next enclosing scope, realizes that it has reached the top of the symbol table, and returns the set of sorts {`->int,->float,->Bool`} for `y`. The search for the trait operation `+` is the same as in the previous example.

```
//@ uses XTrait;

class foo {

//@ uses fooTrait,YTrait;
int bar(int x);


//@ behavior {
//@ ensures result = x + y;
//@ }


} ;
```

Figure 17: Scope and variable lookup: example 2

```
//@ uses XTrait;

class foo {

int bar(int x);

//@ behavior {
//@ ensures result = x + y;
//@ }

} ;
```

Figure 18: Scope and variable lookup: example 3

Figure 18 shows another small example of a specification. This specification differs from the previous two in that it contains a declaration that will cause a type error. Again, the basic goal is to build the sets of possible sorts for the variables `x` and `y` and the operation `+`. The process to build the set for `x` is the same as the previous two examples.

The system continues by building the set of sorts for `y`. As in the previous examples, it begins at the function-specifier scope. Like the first example, a match is not found in the two inner scopes. The system moves to the next enclosing scope, a class scope. Once in this scope, the search begins within the C++ world. Again, no C++ declaration is found. The search continues in the trait operation world. Again, no match is found. The system continues searching upward until it sees the `uses` clause at the global scope. Here it finds a definition for `y` of `->Bool`. Since the system found trait operations that match, the search is continued into the enclosing scope. However, since this match for `y` is found in the outermost scope, the system stops searching and returns the the set of sorts {`Bool`}. This set will lead to a type error unless `+` has the sort `int,Bool -> int`, which is unlikely.

3.3.3 Summary

The Larch/C++ type system has the following qualities:

- Information for symbols is kept in two "worlds." One portion contains C++ declarations, the other contains LSL trait operations.
- A scoping system similar to that of C++ which is applied to all C++ and trait operation names.
- A name conflict resolution algorithm which handles the case where there exists a C++ declaration and a trait operation with the same name in a given scope.
- A name lookup algorithm that attempts to always give the most local C++ declaration that is a match, and gives the illusion that the trait operations are all stored within a single scope, while avoiding the possibility of having an ambiguous solution (i.e. one where there is both a trait operation and a C++ identifier that may be used).

Shorthand	Concrete Syntax Rule
DS	declaration-sequence
EC	ensures-clause
ES	expects-sequence
FI	fun-interface
FR	frame
FSB	fun-spec-body
MC	modifies-clause
P	primary
RC	requires-clause
S	sort-name
SC-B	sc-bracketed
SCS	spec-case-sequence
SC	spec-case
SEC	secondary
SLS	string-literal-sequence
SRL	storage-reference-list
SR	storage-reference
TC	trashes-clause
US	uses-sequence

Figure 19: Abbreviations for Concrete Syntax Rules

4 Formal Sort Rules for Larch/C++

The following sort rules represent the type system for Larch/C++ in a more formal manner. We use the name *sort* here because the system is closely related to the LSL system. The rules presented in this section are based upon the concrete syntax for Function Specifications in Larch/C++, contained in Appendix A of the Larch/C++ Reference Manual [8]. Some of the concrete rules have had their names abbreviated in the formal rules to allow for easier presentation. An abstract syntax grammar has also been created to allow for easier presentation of the rules themselves. Please refer to Figures 20 and Figure 19 for these shorthands.

T ::= if T_1 then T_2 else T_3
| T_1 EQ T_2
| T_1 OP_l T_2
| Q^+ (T)
| T satisfies FI FSB
| T(T^*)

OP_{lsl} ::= $OP_{lsl}{}^+$ SEC
| SEC
| SEC_1 OP_{lsl} SEC_2
| SEC OP_{lsl} OP_{lsl}

Q ::= QS (ID:sort)$^+$

QS ::= \A | \forall | \E | \exists

EQ ::= = | == | \eq | = | != | \neq

OP_l ::= \and | \or |\implies | /\ | \/ | =>

$IDSEQ$ ::= (ID :S be T) $^+$

SCS ::= SC | SCS also SC

ID ::= class_name | built_in_type_name | identifier

$\diamond$ ::= ^ | \pre | ' | \post | \any

SLS ::= string_literal*

Figure 20: Partial Abstract Syntax for Type Rules

4.1 Notation

The following notational conventions are used within these rules. A *type environment* is a finite function which maps identifiers to corresponding sorts. A type environment has the form $\{(\texttt{id:S})\ldots\}$ where $(\texttt{id:S})$ is a pair relating the identifier `id` to the sort `S`. In Larch/C++ a given type environment has two disjoint pieces, the set of C++ relations and the set of LSL operator relations. These sets, denoted by C and L respectively, are kept disjoint for a given scope via the name conflict resolution algorithm mentioned earlier.

The complex type environment for a given scope is denoted by E, and any subscripted or primed variant, such as E_1 or E'. The statement $\texttt{id:}\tau \in E$ represents the fact that within the complex environment E, `id` has a set of sorts τ. Other greek letters, such as α and β, will also be used to represent sets of sorts. Symbolically the two parts of E will be denoted by a pair (C, L), where $C \cap L = \emptyset$.

Type environments can be thought of as set-valued functions. Their domain is given by the following.

$$dom(\{(i : \tau)\}) \quad = \quad i \tag{1}$$

A type environment, such as L, can be "applied" to an identifier i to yield a set of types. If the identifier is not in its domain, then the empty set is returned.

$$L(i) = \begin{cases} \tau, & \text{if } (i : \tau) \in L \\ \{\}, & \text{otherwise} \end{cases} \tag{2}$$

The sub-environments C and L may be combined via the following operations within a given scope.

$$(L_1 \cup L_2) = \{(i : L_1(i) \cup L_2(i)) \mid i \in dom(L_1) \vee i \in dom(L_2)\} \tag{3}$$

$$(C_1 \uplus C_2) = \begin{cases} C_1 \cup C_2, & \text{if } dom(C_1) \cap dom(C_2) = \emptyset \\ \text{undefined}, & \text{otherwise} \end{cases} \tag{4}$$

The above rules try to state what we know about the C++ and LSL scoping systems. The rule for L states that if you combine two L environments, that is equivalent to applying union to the two sets with the special case that if an identifier is in both L_1 and L_2, its set of sorts in L_3 should contain the complete set of sorts. The rule for C environments states that there cannot be a C++ variable name with two distinct types in a single scope.

There are two ways of combining complex type environments. The process of merging two given complex environments to create a single scope is done via disjoint union.*disjoint union* (represented by $\uplus$), is defined as follows:

$$
\begin{aligned}
&(C_1, L_1) \uplus (C_2, L_2) \\
&\quad = \begin{cases} (C_1 \uplus C_2, L_1 \cup L_2), & \text{if } dom(C_1 \uplus C_2) \cap dom(L_1 \cup L_2) = \emptyset \\ & \quad\text{and } dom(C_1) \cap dom(C_2) = \emptyset \\ \text{undefined}, & \text{otherwise} \end{cases}
\end{aligned} \tag{5}
$$

Notice that this $\uplus$ operation may generate errors in two cases:

- If a name is in both C_1 and C_2, there will be an error.
- If a name is in $C_1 \uplus C_2$ and it is in $L_1 \uplus L_2$ also, the system will discard the name from $L_1 \cup L_2$ and issue a warning. This is the name conflict resolution algorithm in action.

A second way to merge two complex type environments together is called shadow union. *Shadow union* ,represented by the $\,\underline{\uplus}\,$ symbol [12], is used to describe the complex type environment created by combining two environments from different scopes. In essence, it embodies how names are hidden due to the scoping system by the name lookup algorithm. The expression $E_1 \underline{\uplus} E_2$ means that a new type environment is created where the following holds:

$$
\begin{aligned}
&(C_1, L_1) \underline{\uplus} (C_2, L_2) \\
&\quad = \begin{array}{l} (C_2 \cup \{(i:\tau) \mid (i:\tau) \in C_1, i \notin dom(C_2), i \notin dom(L_2))\}, \\ \{(i:\tau) \mid (i:\tau) \in (L_1 \cup L_2), i \notin dom(C_2)\}) \end{array}
\end{aligned} \tag{6}
$$

The idea is that, as you enter a new scope, any identifiers declared in that new scope shadow the previous declarations in the type system. In the above formula, E_1 is the existing type environment and E_2 is the type environment of the new scope. If a pair with identifier `i` is in E_1 and not in E_2, it may remain. Otherwise, the pair from E_2 shadows the pair in E_1.

Figure 21 illustrates the general format for the formal sort rules. The structure of each rule is as follows. The bracketed item on the left is the name of the rule. The middle section consists of an optional top portion and a bottom portion separated by dividing line. The top portion is the

$$\texttt{[EXAMPLE]} \quad \frac{E \vdash \texttt{foo} : \beta, \quad E \vdash \texttt{x} : \alpha}{E \vdash \texttt{foo(x)} : \tau} \quad \begin{array}{l} \text{if } \tau = \{n_1 \ldots n_k\}, \\ \alpha = \{m_1 \ldots m_k\} \\ \beta = \{m_1 \rightarrow n_1 \ldots m_k \rightarrow n_k\} \\ k > 0 \end{array}$$

Figure 21: An example type rule

hypothesis, the bottom is the *conclusion*, and the horizontal bar means logical implication. This means that a given rule should be interpreted as follows: if the hypothesis is true, the conclusion should also be true. Within these rules, the $\vdash$ operator also represents implication. In this case, an expression such as $E \vdash x$ means that if E is assumed, then x can be proved. Another way of thinking about the $\vdash$ operator is that the left side represents the attributes inherited from parents in the syntax tree. The set of sorts to the right of the colon (`:`) in the rule represent the synthesized attributes created by checking the syntactic form between the $\vdash$ and the `:`. A term that sort checks correctly, but for which the sort is unimportant has the colon and sort replaced by a $\surd$. Possible sets of types that a given expression may have are represented by τ, α, and β. Function types are represented via the standard $\rightarrow$ notation. To the right of the rule, the *side conditions* state other necessary conditions for the rule to be applied.

For the example in Figure 21, the name of the rule is `[EXAMPLE]`. The rule itself states that given the environment E, if it can be shown that `foo` has set of types β and that `x` has set of sorts α, and the side annotations hold, then it can be stated that given E the expression `foo(x)` has the set of types τ. The side condition states that for the implication to be true, it must be demonstrated that α contains a set of input sorts, and that β contains the correct functions to map the m sorts to their related n sorts. If this holds, then τ should be the set of k sorts $\{n_1 \ldots n_k\}$.

4.2 The Formal Sort Rules

An attempt has been made to break the formal rules into sets of rules that have similar structures or related conclusions. For the most part, the rules are allowed to describe themselves.

$$\texttt{[FUN-SPEC-BODY]} \quad \frac{\begin{array}{c} E \vdash ES \Rightarrow E_1, \\ E \vdash US \Rightarrow E_2, \\ E \vdash DS \Rightarrow E_3, \\ E \uplus (E_1 \uplus E_2 \uplus E_3) \vdash SCS \surd \end{array}}{E \vdash \texttt{behavior}\ \{\ ES\ US\ DS\ SCS\ \} \surd} \quad \begin{array}{c} \text{if } (E_1 \cap E_2) = \emptyset, \\ (E_1 \cap E_3) = \emptyset, \\ (E_2 \cap E_3) = \emptyset \end{array}$$

$$\texttt{[SCS]} \quad \frac{E \vdash SC \surd,\ E \vdash SCS \surd}{E \vdash SC\ \texttt{also}\ SCS \surd}$$

$$\texttt{[SC]} \quad \frac{\begin{array}{c} E \vdash LC \Rightarrow E', \\ E \uplus E' \vdash RFE \surd, \\ E \uplus E' \vdash EX \surd, \\ E \uplus E' \vdash CS \surd \end{array}}{E \vdash LC\ RFE\ EX\ CS \surd}$$

$$\texttt{[RFE]} \quad \frac{E \vdash RC \surd, E \vdash FR \surd, E \vdash EC \surd}{E \vdash RC\ FR\ EC\ \ \surd}$$

Figure 22: Top level rules

4.2.1 Top Level Rules

Figure 22 illustrates the rules that describe the top level of the type checking system. Of these, the rule for [FUN-SPEC-BODY] is probably the most interesting. Notice that the rule creates new type environments from the expects sequence, the uses sequence, and the declaration sequence. These individual environments are then shadow unioned to the existing environment. Remember that the shadow union will override any existing information about a variable in E with the information contained in $(E_1 \uplus E_2 \uplus E_3)$. This modification of the environment corresponds to the creation of the function-specification scope in which the actual specification will be sort checked (see section 3.3 for a description of the scoping system). The function-specification-body will sort check if the specification represented by the SCS sort checks within the newly created environment.

An example of how the environments are created and combined for the [FUN-SPEC-BODY] rule can be illustrated using the traits in Figure 23 and

```
fooTrait:trait
     introduces
        foo: int -> int
        y: -> float

Restrictions:trait
     introduces
        x:->int
        y:->int
        foo:int->float
        globalInt:->int
```

Figure 23: Traits used in type environment expansion example

```
int foo(int x);
//@ behavior {
//@    uses fooTrait;
//@    expects Restrictions;
//@    extern int globalInt;
//@
//@    modifies globalInt;
//@    ensures globalInt' = x^;
//@ }
```

Figure 24: Example of type environment expansion

the function specification in Figure 24. To begin with, the uses sequence will process `fooTrait` using the LSL Checker as described before to generate sort information for the operations defined in the trait. This leads to the generation of a type environment (C, L) for `fooTrait` which looks like

$$C = \{\}$$
$$L = \{\texttt{foo:}\{\texttt{int->int}\}\texttt{, y:}\{\texttt{->float}\}\}$$

Similarly, the expects sequence will process its list of traits by running them through the LSL Checker and saving their output. In this case a type environment (C_1, L_1) is generated which looks like

$$C_1 = \{\}$$
$$L_1 = \{\texttt{x:}\{\texttt{->int}\}\texttt{,y:}\{\texttt{->float}\}\texttt{,foo:}\{\texttt{int->float}\}\texttt{, globalInt:}\{\texttt{->int}\}\}$$

Finally, the declaration sequence takes the C++ declaration list which it contains and generates its own type environment (C_2, L_2) which looks like

$$C_2 = \{\texttt{globalInt:}\{\texttt{->int}\}\}$$
$$L_2 = \{\}$$

These individual environments are then combined via disjoint union to create a single type environment that will be shadow unioned to the environment that existed before we entered `foo`'s fun-spec-body to create the environment that will be used to sort check the fun-spec-body. This environment, $(C_x, L_x) = (C, L) \uplus (C_1, L_1) \uplus (C_2, L_2)$, would contain the following

$$C_x = \{\texttt{globalInt:}\{\texttt{->int}\}\}$$
$$L_x = \{\texttt{x:}\{\texttt{->int}\}\texttt{,y:}\{\texttt{->float,->int}\}\texttt{,foo:}\{\texttt{int->float,int->int}\}\}$$

Notice that `y` and `foo` now have sets of sorts associated with them and that the trait operation `globalInt` was removed from the LSL portion of the environment due to the C++ declaration that contained the same name.

Similarly, the rule for `[SC]` shows the creation of the spec-case scope via the shadow union of the existing environment with the environment generated by the let clause. Then the spec-case sort checks if its individual pieces sort check within the newly created environment.

[IDSEQ1]	$\dfrac{E \vdash IDSEQ \Rightarrow E', E \vdash T : \tau, \quad E \vdash S \ \texttt{OK}}{E \vdash IDSEQ\ id : S \ \texttt{be} \ T \Rightarrow E'''}$	if $S \in \tau, E''' = E' \uplus \{(id : \{S\})\}$ See text.
[IDSEQ2]	$\dfrac{E \vdash id : S \Rightarrow E', E \vdash S \ \texttt{OK}}{E \vdash id : S \Rightarrow E'}$	if $E' = \{(id : S)\}$
[LC]	$\dfrac{E \vdash IDSEQ => E'}{E \vdash \texttt{let}\ IDSEQ \Rightarrow E'}$	
[ES]	$E \vdash ES \Rightarrow E'$	See text.
[US]	$E \vdash US \Rightarrow E'$	See text.
[DS]	$E \vdash DS \Rightarrow E'$	See text.
[Q]	$\dfrac{E \vdash \tau_1 \ \texttt{OK}, \ldots, E \vdash \tau_n \ \texttt{OK}}{E \vdash QS\ x_1 {:} \tau_1, \ldots, x_n {:} \tau_n \Rightarrow E'}$	$E' = x_1 : \tau_1, \ldots, x_n : \tau_n$, $n > 0$, See text for a description of OK
[Q1]	$\dfrac{E \vdash Q \Rightarrow E', \quad E \uplus E' \vdash T : \tau}{E \vdash Q(T) : \{\texttt{Bool}\}}$	if $Bool \in \tau$

Figure 25: Rules affecting the type environment

4.2.2 Rules Affecting the Type Environment

The rules in Figure 25 serve to show the points at which the existing type environment may be extended. In some cases, such as the rules for quantifiers (`[Q]` and `[Q1]`, the extension occurs at the point when a new scope is entered. Remember from the description of the Larch/C++ scoping system (Section 3.3) that quantifiers introduce a new scope. The rules show that the type environment for this scope will contain the new identifiers declared within the quantifier. The `OK` marker is there to denote that the sort S is allowable. By `allowable` it is meant that declarations in LSL cannot introduce new sorts; they can only refer to previously mentioned sorts. Thus the judgement $E \vdash m$ `OK` means that within the type environment E the sort m must exist. This modifier will be used in later rules also.

The other rules listed here do not create a new scope to contain the new type environment; instead they augment the existing environment. However, they all create a new environment that may shadow previous declarations. Note that the let clause (`[LC]`) shares the requirement that the sort associated with a declared identifier should have existed in the previous type environment.

4.2.3 Predicate Rules

Figure 26 contains the rules for predicates. Note that as mentioned in Section 3.3, a given construct may have a set of types associated with it. Remember also that if an item sort checks in LSL, it should be possible to assign a unique type to the term. Predicates, represented by $\mathcal{P}$, are a special case. They must have the sort `Bool` as an element of their set of sorts.

4.2.4 Term Rules

Figure 27 contains most of the rules for the sort checking Larch/C++ terms. Recall from Section 3.3 that Larch/C++ and LSL sort check terms identically. Thus rules, such as if-then-else (`[IF]`), represent the ideas expressed in Sections 3.1 and 3.3. Notice the side condition that states that the conditional's test term, T_1, must have `Bool` in its set of types, and that the resulting sort consists of the non-empty intersection of the possible sorts for T_2 and T_3. It might seem that τ should have a cardinality of one at this point so that there would be a single sort associated with the operator. However, it is important to remember that the context surrounding the use of the

[RC1] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{requires}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[RC2] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{requires liberally}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[FR] $$\frac{E \vdash MC\ \surd,\ E \vdash TC\ \surd}{E \vdash FR\ \surd}$$

[EC1] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{ensures}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[EC2] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{ensures liberally}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[EX1] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{example}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[EX2] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{example liberally}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[CS1] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{claims}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[CS2] $$\frac{E \vdash \mathcal{P} : \tau}{E \vdash \texttt{claims liberally}\ \mathcal{P}\ \surd} \quad \text{if } Bool \in \tau$$

[informally] $$E \vdash \texttt{informally}\ SLS : \{\texttt{Bool}\}$$

Figure 26: Predicate rules

$$\texttt{[IF]} \quad \frac{E \vdash T_1 : \tau',\ E \vdash T_2 : \alpha,\ E \vdash T_3 : \beta}{E \vdash \texttt{if } T_1 \texttt{ then } T_2 \texttt{ else } T_3 : \tau} \qquad \begin{array}{l} \text{if } \mathit{Bool} \in \tau', \\ \tau = \alpha \cap \beta, \\ \tau \neq \emptyset \end{array}$$

$$\texttt{[LT]} \quad \frac{E \vdash (OP_l)(T_1, T_2) : \tau}{E \vdash T_1\ OP_l\ T_2 : \tau}$$

$$\texttt{[OP}_{lsl}\texttt{]} \quad \frac{E \vdash (OP_{lsl})(SEC) : \tau}{E \vdash OP_{lsl}\ SEC : \tau}$$

$$\texttt{[OP}_{lsl}\texttt{2]} \quad \frac{E \vdash (OP_{lsl})(SEC_1, SEC_2) : \tau}{E \vdash SEC_1\ OP_{lsl}\ SEC_2 : \tau}$$

$$\texttt{[P0]} \quad (C, L) \vdash ID : \tau \qquad \text{if } ID{:}\tau \in C$$

$$\texttt{[P1]} \quad (C, L) \vdash ID : \tau \qquad \text{if } ID{:}\tau \notin C,\ ID{:}\tau \in L$$

$$\texttt{[P2]} \quad \frac{E \vdash T_1 : \tau_1, \ldots, E \vdash T_n : \tau_n,\ E \vdash F : \tau}{E \vdash F(T_1, \ldots, T_n) : \tau'} \qquad \begin{array}{l} \text{if } \tau' = \{m'_1, \ldots, m'_k\}, \\ m_1 \in \tau_1 \times \ldots \times \tau_n,\ m_1 \rightarrow m'_1 \in \tau, \\ \vdots \\ m_k \in \tau_1 \times \ldots \times \tau_n,\ m_k \rightarrow m'_k \in \tau, \\ k > 0 \end{array}$$

$$\texttt{[PRIM1]} \quad \frac{E \vdash (\_\_.ID)(P) : \tau}{E \vdash P.ID : \tau}$$

Figure 27: Term rules

if-then-else operator should serve to narrow τ as the sort checking process continues. Thus, the cardinality need not be one at this point.

Of the rules listed here, the operator application rule ,[P3], probably has the most impact on the system. The behavior embodied in this rule is used within any rule that may act like a function call ([PRIM1], [OP$_{lsl}$], [OP$_{lsl}$2], the sc-bracketed rules,[LT], and others). Since trait functions are overloaded, and the overload resolution involves context, a given operator name can have a set of possible return sorts. Each of these return sorts has a corresponding set of domain sorts that is the cross-product of the sorts of the arguments. For a function application to sort check, it must be shown that given a set of return sorts there must be a function signature that consists of the cross-product of the sets of types of the arguments. For example, given the following signatures for a function foo:

```
foo: int -> float
foo: char -> Bool
foo: float -> int
foo: int -> int
```

and a use of foo in the following specification:

```
int bar(int x);
//@ behavior {
//@   ensures result = foo(x);
//@ }
```

Here `foo` has the set of signatures {`int -> float,char -> Bool,float -> int, int -> int`} associated with it. In this case, since `=` requires that the two arguments have the same sort and the sort of `result` is known to be `int`, the set of possible signatures for `foo` is {`float -> int, int -> int`}. From this set, only one signature has the correct sort for the argument `x`. So in this case, the operator `foo` will have the signature `int -> int`, and the statement

```
 result = foo(x)
```

will sort check.

$$\texttt{[SC-B]} \quad \frac{E \vdash (\texttt{[]})(T_1, \ldots, T_n) : \tau}{E \vdash [T_1, \ldots, T_n] : \tau}$$

$$\texttt{[SC-B2]} \quad \frac{E \vdash (\{\})(T_1, \ldots, T_n) : \tau}{E \vdash \{(T_1, \ldots, T_n)\} : \tau}$$

$$\texttt{[SC-B3]} \quad \frac{E \vdash (\mathtt{\backslash langle\backslash rangle})(T_1, \ldots, T_n) : \tau}{E \vdash \mathtt{\backslash langle}(T_1, \ldots, T_n)\mathtt{\backslash rangle} : \tau}$$

$$\texttt{[SC-B4]} \quad \frac{E \vdash (\mathtt{\backslash<\backslash>})(T_1, \ldots, T_n) : \tau}{E \vdash \mathtt{\backslash<}(T_1, \ldots, T_n)\mathtt{\backslash>} : \tau}$$

Figure 28: sc-bracketed rules

$$\texttt{[STATE1]} \quad \frac{E \vdash P : \tau}{E \vdash P\diamond : \tau'} \qquad \text{if } \tau' = strip(selobjs(\tau))$$

$$\texttt{[STATE2]} \quad \frac{E \vdash P : \tau}{E \vdash P\mathtt{\backslash obj} : \tau}$$

Figure 29: State functions

4.2.5 sc_bracketed Rules

The sc-bracketed rules are used for operators that have signatures similar to the following.

```
[ __ , __ ]: int , int -> Pair[int]
```

These functions may be formed by builtins, such as tuple constructors, or may be defined by the specifier.

4.2.6 State Function Rules

Larch/C++ specifications have the ability to describe the abstract value contained within a given object. To do this, Larch/C++ has a formal model that describes the relationship between objects and their values. This is described in Section 2.8 of the Reference Manual [8]. Most of the time, a C++ declaration will create an object containing an abstract value. The state of the variable "associates to each object, an abstract value." [8, page 21]. State functions allow a specifier to extract the abstract value for a variable for a specific state. For example, the C++ declaration

```
int i;
```

creates an object with the following sort.

```
Obj[int]
```

A state function, when applied to `i`, returns the value for the state, which has sort `int`. Please refer to the Reference Manual [8, Section 6.2.1] for more details.

Figure 29 illustrates the rules for state functions. Two auxiliary functions, *strip* and *selobjs*, defined below, allow for the extraction of abstract values.

$$strip(\texttt{ConstObj[T]}) = \texttt{T}$$

$$strip(\texttt{Obj[T]}) = \texttt{T}$$

$$strip(\{S_1, \ldots, S_n\}) = \{strip(S_1), \ldots, strip(S_n)\}$$

$$\begin{aligned} &selobjs(\{S_1, \ldots, S_n\}) \\ &\quad = \{S_i | 1 \leq i \leq n, S_i \text{ has form } \texttt{Obj[T]} \text{ or } \texttt{ConstObj[T]}\}, \end{aligned}$$

strip takes an object type, and strips off the object portion, leaving the value. For example, *strip*(`Obj[int]`) would return `int`. *selobjs* takes a set of sorts and builds a subset consisting of the object sorts. Used in conjunction, *strip* and *selobjs* create a set of values that the object may have for the given state.

[HOC]	$$\frac{E \vdash FI \Rightarrow E', \quad E \uplus E' \vdash FSB : \surd}{E \vdash OP_{lsl}\ \texttt{satisfies}\ FI\ FSB : \{\texttt{Bool}\}}$$	If E' is as described in the text.
[EQ]	$$\frac{E \vdash OP_{lsl_1} : \alpha,\ E \vdash OP_{lsl_2} : \beta}{E \vdash OP_{lsl_1}\ EQ\ OP_{lsl_2} : \{\texttt{Bool}\}}$$	if $\lvert\alpha \cap \beta\rvert = 1$

Figure 30: Miscellaneous rules

4.2.7 Miscellaneous Rules

In Larch/C++ *higher-order* functions are functions which either take pointers to functions as arguments, or return pointers to functions [8]. The rule `[HOC]` in Figure 30 is the rule for the higher-order-comparison used in the specification of higher-order functions. The function-interface (FI) is used to create a new type environment, E', from the formal parameters of the function interface. Then the fun-spec-body(FSB) is sort checked in accordance with the rule presented earlier. Please see the Reference Manual [8, Section 6.12] for more information on higher-order functions.

4.2.8 Literal Rules

Figures 31 and 32 illustrate the sort rules for some of the literals. They are divided into the Larch/C++ basic sorts and the special C++ literals. These rules show that the basic building blocks have the specific sorts dictated by C++. The rule `[LIT2]` serves as a model for the formulation of sorts for the unsigned types. For a complete list, please see the Reference Manual [8, Chapter 11].

4.2.9 Storage Rules

In Larch/C++ storage references are used in the `modifies` clause, and in a few other places. The storage reference rules illustrated in Figures 33 and 34 show how these references interact with the sort system. Rule `[SRL4]` is the most complex of the rules. Its side condition states that a storage reference must either have the sort `Set[TypeTaggedObject]` or if it does

$$
\begin{array}{ll}
\texttt{[LIT1]} & E \vdash \mathit{int\_const} : \{\mathrm{int}\} \\
\texttt{[LIT2]} & E \vdash \mathit{unsigned_int\_const} : \{\mathrm{unsignedInt}\} \\
\texttt{[LIT3]} & E \vdash \mathit{float\_const} : \{\mathrm{double}\} \\
\texttt{[LIT4]} & E \vdash \mathit{char\_const} : \{\mathrm{char}\} \\
\texttt{[LIT5]} & E \vdash \texttt{L}\ \ \mathit{char\_const} : \{\mathrm{wchar\_t}\} \\
\texttt{[LIT6]} & E \vdash \mathit{string\_literal} : \{\mathtt{Arr}[\mathrm{Obj}[\mathrm{char}]]\} \\
\texttt{[LIT7]} & E \vdash \texttt{L}\ \ \mathit{string\_literal} : \{\mathrm{Arr}[\mathrm{Obj}[\mathrm{wchar\_t}]]\} \\
\texttt{[LIT8]} & E \vdash \mathit{abstract\_string\_literal} : \{\mathtt{String}[\mathrm{char}]\}
\end{array}
$$

Figure 31: A sampling of literal rules

$$
\begin{array}{lll}
\texttt{[this]} & E \vdash \texttt{this} : \tau & \text{if } E(\texttt{this}) = \tau \\
\texttt{[self]} & E \vdash \texttt{self} : \tau & \text{if } E(\texttt{self}) = \tau \\
\texttt{[result]} & E \vdash \texttt{result} : \tau & \text{if } E(\texttt{result}) = \tau \\
\texttt{[pre]} & E \vdash \texttt{pre} : \{\mathit{State}\} & \\
\texttt{[post]} & E \vdash \texttt{post} : \{\mathit{State}\} & \\
\texttt{[any]} & E \vdash \texttt{any} : \{\mathit{State}\} & \\
\texttt{[sizeof]} & E \vdash \texttt{sizeof(}\mathit{type}\texttt{)} : \{\mathit{int}\} &
\end{array}
$$

Figure 32: More literal rules

$$[\texttt{fresh}] \quad \frac{E \vdash T_1 : \tau_1, \ldots, E \vdash T_n : \tau_n}{E \vdash \texttt{fresh}(T_1,\ldots,T_n) : \{\texttt{Bool}\}} \quad \begin{array}{l} n > 0, \\ \forall k (1 \leq k \leq n) \Rightarrow \tau_k \neq \emptyset \end{array}$$

$$[\texttt{trashed}] \quad \frac{E \vdash SRL \checkmark}{E \vdash \texttt{trashed}(\mathrm{SRL}) : \{\texttt{Bool}\}}$$

$$[\texttt{unchanged}] \quad \frac{E \vdash SRL \checkmark}{E \vdash \texttt{unchanged}(\mathrm{SRL}) : \{\texttt{Bool}\}}$$

$$[\texttt{thrown}] \quad \frac{E \vdash S \;\; \texttt{OK}, E \vdash SRL \checkmark}{E \vdash \texttt{thrown}(S) : \{S\}}$$

$$[\texttt{throws}] \quad \frac{E \vdash S \;\; \texttt{OK}}{E \vdash \texttt{throws}(S) : \{\texttt{Bool}\}}$$

$$[\texttt{returns}] \quad E \vdash \texttt{returns} : \{\texttt{Bool}\}$$

$$[\texttt{modifies}] \quad \frac{E \vdash SRL \checkmark}{E \vdash \texttt{modifies} \;\; SRL : \checkmark}$$

$$[\texttt{constructs}] \quad \frac{E \vdash \texttt{modifies}\; SRL \checkmark}{E \vdash \texttt{constructs} \;\; SRL : \checkmark}$$

Figure 33: Storage reference rules, part one

[SRL] $\dfrac{E \vdash SR_1 \surd, \ldots, E \vdash SR_n \surd}{E \vdash SR_1, \ldots, SR_n \surd}$ if $n \geq 0$

[SRL2] $E \vdash$ `nothing` $\surd$

[SRL3] $E \vdash$ `everything` $\surd$

[SRL4] $\dfrac{E \vdash SR : \tau}{E \vdash SR \surd}$ if `Set[TypeTaggedObject]` $\in \tau$ or ($\tau = \{S\}$ and $S \neq$ `Set[TypeTaggedObject]` and $S \rightarrow$ `Set[TypeTaggedObj]` $\in E($`contained_objects`$)$)

Figure 34: Storage reference rules, part two

not have this sort, there must be a operation `contained_objects` within the type environment E which has the signature `S -> Set[TypeTaggedObject]` associated with it, and the sort of the storage reference must be `S`. The constraints in the side condition are there because of the definition of storage references. For more information on the semantics of storage references, see the Reference Manual [8, Section 6.2.3.3 and following].

5 Implementation of the System

As mentioned in the Introduction, the development process for type checking in the Larch/C++ Checker has occurred in many pieces. This section of the paper begins with a short description of the Checker's source files and the tools used in development. Following that, the modifications to the Checker to support the C++ `namespace` construction are described. A description of the modifications to the Larch/C++ Checker and the LSL Checker to allow for the support of LSL constructs in Larch/C++ continues the section. Then a description of how the system converts C++ declarations to LSL sorts is given. The concluding section discusses the lookup algorithm and preliminary type checking results. Note that the source code presented in the following sections has been stripped of any comments or specifications in order to save space. Please refer to the actual implementation for more details.

5.1 Overview of the System

The task of creating tools for a language as complex as Larch/C++ is difficult. Any implementation of the tools will also be complex. Larch/C++'s implementation can be broken down into the following major pieces:

- The lexer and parser for Larch/C++
- The lexer and parser for LSL operation signatures
- The temporary file support system
- The LSL Checker
- The symbol table system

Note that the LSL Checker is included here because it required modification to work within the system, and because it is an important part of the system as a whole.
Figure 35 shows a pictorial representation of the structure of the system.

The majority of the Larch/C++ Checker's implementation lies within the lexer, the parser, and the support code. The lexer and parser are built using an attribute grammar system called Ox [1]. The Checker does not take full advantage of Ox, but the syntax for the creation and access of attributes for rules in Ox can be clearer than the equivalent expressions in `yacc`. Ox takes a decorated grammar as input and generates output in the form of `.l` and `.y` files to be sent through `lex` and `yacc`.

The support files for the lexer and parser include all of the code to generate the various data structures used within the Checker. The code has been developed using C++. Names of the source files and classes attempt to clearly state what they are used for. The file names can be confusing because they are limited to 8.3 format for compatibility with MS-DOS (Originally, MS-DOS allowed only filenames which had eight characters followed by a three character extension). An example of how the files and classes are named is the case of type specifiers. Type specifiers are a part of the C++ declaration syntax. The support code for type specifiers in the Larch/C++

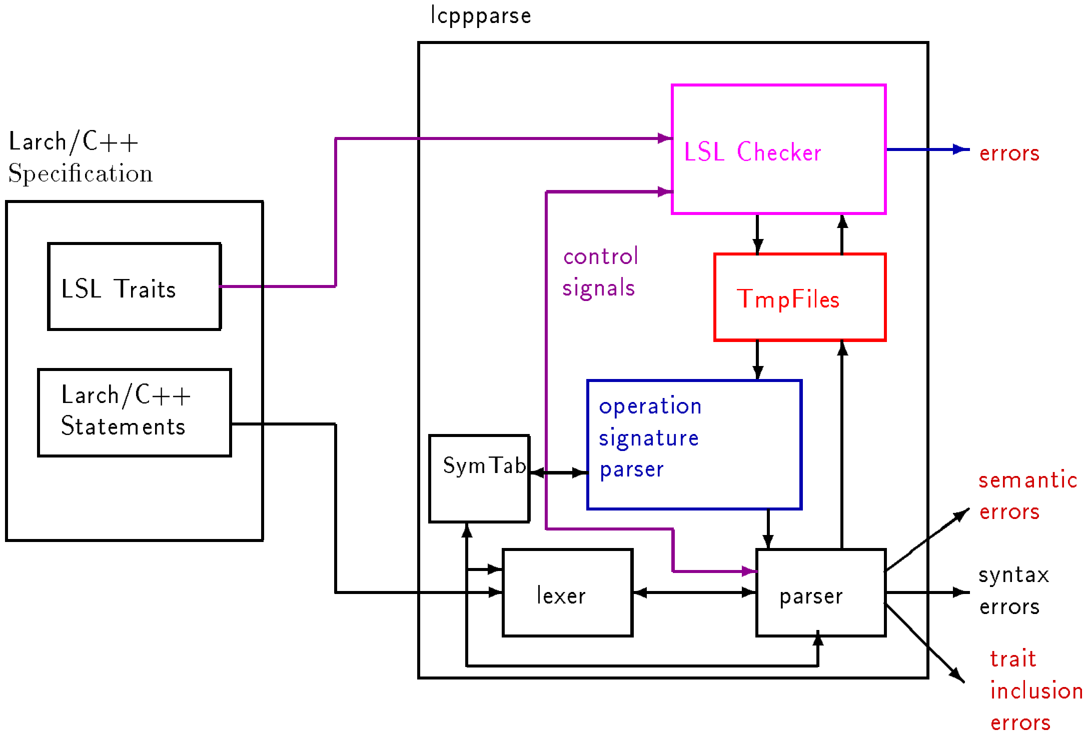

Figure 35: The structure of the Larch/C++ Checker

Checker is contained in files named `TypeScfr.*` and the class implemented is actually named `TypeSpecifier`. It is important to note that some header files are broken into four pieces, the `.pre` file, the `.h` file, the `.pri` file, and the `.bse` file. Not all headers have all of these files, but most have at least the `.h` and `.pri`. The `.pre` file is used to hold any private declarations or `#include` directives for the `.h` file that might occur before a class definition. The `.h` file contains the public portion of the class definition. The `.pri` file contains protected or private members. The `.bse` file is used for private or protected inheritance.

The common code portion of the system contains code that is not specific to the Larch/C++ Checker. This includes implementations of dynamic strings, debugging code, interfaces to the environment, and other functions. These classes and functions are used throughout the whole of the system.

Finally, not mentioned in the above list but still very important to the Larch/C++ Checker is the Boehm-Demers-Weiser Conservative Garbage Collector [2]. All reclamation of allocated storage is handled by the garbage collector. This is important because the code makes heavy use of pointers and dynamic allocation of objects.

5.2 Support for C++ `namespace`

The Draft C++ Standard [3] contains definitions for many new language constructs. One of these is `namespace`. The `namespace` construct was added to C++ to allow for an additional level of scoping for names. Before namespaces, the possibility existed for vendors to supply libraries of code that would have name conflicts. For example, vendor A ships a linked-list library with the operation `Delete`. It was possible that this `Delete` could name conflict with some other piece of code. The only solution was either to not use the library or to wrap the library in another class to isolate the name. Namespaces allow for declarations to be wrapped in another name without the need for additional classes. These wrapped names can then be made available via a `using` declaration.

Figure 36 shows an example of the use of the `namespace` construction.The `namespace` declaration creates a new `namespace` named `foo` which contains the declarations for `i` and `j`. These variables will not be visible at any scope level (see Section 3.2 for a review of the C++ scoping system) unless there is a `using` directive or a `using` declaration involving them. Within the function `inc()` there is an example of a `using` directive. The declaration

```
namespace foo{
  int i;
  int j;
}

void inc(){
  using namespace foo;
}

void dec(){
  using foo::i;
}
```

Figure 36: The C++ `namespace` construction

```
using namespace foo;
```

informs the system that all the declarations contained within `namespace foo` should become visible at the current scope. Similarly the `using` declaration:

```
uses foo::i;
```

informs the system that the declaration for `i` contained in `namespace foo` should be visible at the current scope. Another form of the `namespace` declaration, the *unnamed namespace*, has a similar syntax, except a name is not supplied for the namespace. The semantics of unnamed namespaces are a little different, however. An unnamed `namespace` has the effect of declaring a namespace followed by an immediate `using` directive. Thus, the names become visible immediately.

Before describing the implementation of the support for the `namespace` construction within the Larch/C++ Checker, some background on the symbol table system used within the Checker is needed. Figure 37 illustrates the structure of the system's symbol table at some point. The basic structure of the symbol table is contained within the `class SymTab`. In this illustration, there are two pointers, `globalSymTab` and `currentSymTab`, which point to objects of type `SymTab`. `globalSymTab` always points to the top of the symbol table, while `currentSymTab` always points to the current local scope.

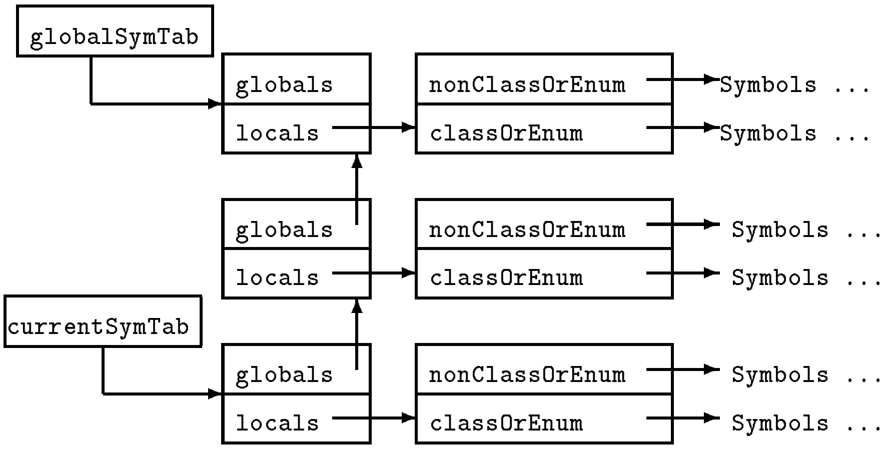

Figure 37: The structure of the symbol table

Each `SymTab` object contains a link to its enclosing scope, `global`, and a link to the information in that table's current scope, `locals`. At the top level of the symbol table, the `globals` pointer is `NULL` to signify the top of the table. The local information in a `SymTab` is contained with a `Locale` object. A `Locale` contains references to two lists of `Symbols`, those that are classes or enumerations, `classOrEnum`, and those which are not, `nonClassOrEnum`. The `Symbol` class, and its derived classes, are used to represent and store the various types of C++ declarations. Please see Figure 38 for the class hierarchy diagram for `Symbol` and its derived classes.

Modifying the Larch/C++ system to support the `namespace` and `using` constructs was not difficult. Within the Larch/C++ checker, there was already support for the parsing of the `namespace` and `using` syntax. `Symbols` of the form `OriginalNamespaceName` were created when a `namespace` declaration was parsed. The `OriginalNamespaceName` object contains a local symbol table with a structure similar to the system's symbol table. The declarations within a namespace are placed within this local table so that they are available whenever the namespace object is referenced. Since this portion of the implementation was already complete, all that needed to be done was to implement the semantics of the using declarations and directives.

At first, the best solution was thought to be that when a `using` directive

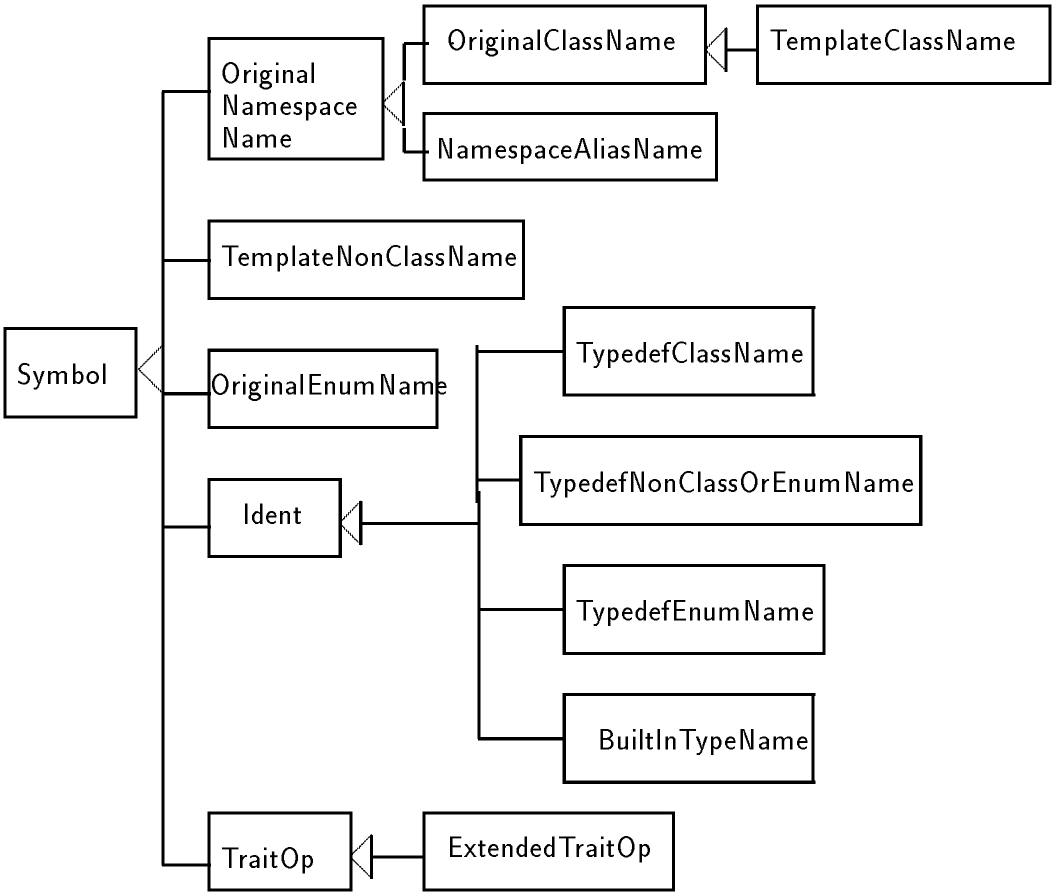

Figure 38: The `Symbol` class hierarchy

or declaration was seen, simply copy the symbol for that declaration out of the namespace's local symbol table into the current symbol table for that scope. However, the question was raised, are the symbols we copy really like the other symbols within the table? The declarations introduced by `using` declarations and directives appear to be or are synonyms for the previous declarations. It was decided that the declarations introduced by the `using` forms should be differentiated from other forms of declarations. This was done because `using` directives and declarations don't really create new declaration information, they simply change its visibility. Thus simply creating new regular declaration objects that were copies of the previous declaration objects would not suffice.

Various solutions to this problem were proposed. One solution involved the creation of a new form of `Symbol`, the `Alias` which would have contained a reference back to the original symbol in the namespace symbol object. These `Alias` objects would have been placed into the scope containing the `using` form. While this solution had promise, it also added complexity. The actual declaration information would not have been easily accessible within the `Alias` object because it would have had to have been dereferenced in some manner.

While the `Alias` object idea was rejected, the basic idea behind it, the idea of aliases, formed the cornerstone for the actual solution. The basic structure of the `Symbol` class and its derived classes was modified to support a new data member, the alias field. The `alias` field is a boolean value that stated whether this object was an alias or a simple symbol. Besides the data member, member functions for the observation and modification of this data member were created. The use of C++ and inheritance shined at this point. Only the code for the `Symbol` class had to be modified, with the change affecting the derived classes via inheritance. This solution also avoided the extra complexity of creating a new type of `Symbol` and the difficulty in getting at the declaration information. When a `using` statement was seen, all that needed to be done was create a copy of the symbols within the namespace object, flag these copies as aliases, and insert them into the symbol table.

The code to create the alias copies and do the insertion into the symbol table ended up going through two revisions. The first implementation took a non object-oriented path to the solution. Remember that the `namespace` objects contained a symbol table with the same structure as the global symbol table described earlier. To create copies of the symbols, the system needed to be able to access them easily. The first implementation modified the

Locale class, adding member functions that gave access to the protected data members. Once the system had the lists of Symbols out of the Locale, it processed these Symbols one at a time via a large case statement. The case statement checked to see what kind of Symbol it was, created a copy of the symbol with the alias field set, and passed this copy back to the system to add to the current scope. While this solution worked, it seemed to void some of the advantages of object-oriented design. First, member functions should not pass private or protected data members to their callers. This breaks the property of encapsulation. Second, with the use of virtual functions, there should be no need for large case statements to differentiate between the type of symbols. Thus, the code was rewritten to be more object-oriented.

The final implementation consists of the following. The Symbol class and its derived classes were modified to have member functions CloneSym and CloneSymAsAlias. A Symbol object receiving one of these method calls creates a copy of itself, setting the alias flag as necessary. The functions to allow for access to the data within the Locale objects and the SymTab objects were moved into member functions for those classes. This kept the encapsulation of the protected data members intact. Figure 39 illustrates the CopySymbols functions and Figure 40 shows their use in the Larch/C++ grammar file for the case of a using directive.

The way the functions work for this example is as follows. Looking at Figure 40, notice the comment. The LocalSyms variable contains the local symbol table associated with the namespace. The function CopySymbols is overloaded for SymTab objects. In this case the call

```
LocalSyms->CopySymbols(currentSymTab,true)
```

resolves to the first function in Figure 39. The arguments to this function are the destination symbol table into which the copies will be placed, and a boolean value representing whether the copies should be aliases. This function then calls Locale::CopySymbols passing along the arguments. The Locale::CopySymbols passes the work onto the more complex of the two SymTab::CopySymbols functions to do the actual copying of the symbols and insertion within the symbol table. The insertions are done as follows. The first item is taken off of the list of symbols. The item is copied either as an alias or as a plain Symbol object depending on the value of the asAlias parameter. The copy is then inserted into the symbol table referenced by the default parameter if it is not already there. The next Symbol to copy

```
void SymTab::CopySymbols(SymTab *Table, bool asAlias){
    locals->CopySymbols(Table,asAlias);
}

void Locale::CopySymbols(SymTab *Table, bool asAlias){
  Table->CopySymbols(asAlias,nonClassOrEnum);
  Table->CopySymbols(asAlias,classOrEnum);
}

void SymTab::CopySymbols(bool asAlias, SymNodePtr SymList){

  SymNodePtr currentSym;
  Symbol *copySym;
  currentSym = SymList;

  while (currentSym != NULL) {
    if (asAlias) {
      copySym = Car(currentSym)->CloneSymAsAlias();
    } else {
      copySym = Car(currentSym)->CloneSym();
    }
    if ((this->Defined(Car(currentSym)->GetName()))) {
      Warning(*(currentSym->value->GetName()) +
              DyString(" is multiply defined in this scope\n")+
              DyString("The new definition was ignored and not inserted"));
    } else {
      this->AddSym(copySym);
    }
    SymNodePtr currentSym2 = Cdr(currentSym);
    currentSym = currentSym2;
  }

}
```

Figure 39: The `CopySymbols` functions

```
using_directive
    : USING NAMESPACE complete_namespace_name T_SEMI
        @{ @m using_directive.sym;
           @using_directive.sym@ = @complete_namespace_name.sym@;
           SymTab *LocalSyms = @using_directive.sym@->GetLocals();
           /*
             LocalSyms contains a SymTab that contains the
             symbols that must be inserted.
           */
           LocalSyms->CopySymbols(currentSymTab,true);

        @}
```

Figure 40: One production for `using` directives

is taken off the list, and the process continues until all symbols have been copied.

5.3 Support for LSL Constructs

As its name suggests, Larch/C++ is dependent upon the Larch Shared Language. Specifiers use LSL traits to create the abstract models for their specifications. These models can then be referenced in Larch/C++ in a number of ways: `uses` sequences, `expects` sequences, etc. The Larch/C++ Checker needs to be able to extract information about the operations defined within traits. The system must handle the overloading of trait operations and the renaming of sort parameters within traits. One possible solution was to write a filter for LSL traits that would extract the necessary information. This appeared to be a time sink, however. It was decided to see how other languages in the Larch family handled this interface.

5.3.1 Interface to the LSL Checker

After examining an early implementation for the LCLint tool [10], it was realized that the LSL Checker handles the renaming of traits in `includes` clauses, and will generate the needed information about the trait operations via the `-syms` option. Reusing this functionality would save time and effort. Thus, an interface to the LSL Checker was needed. The interface and its development occurred in a series of steps. First, the information within the Larch/C++ specifications needed to be converted into a form useful to the LSL Checker. Then a system of passing this information to the LSL Checker and capturing the output was needed. Finally the information from the LSL Checker needed to be translated and stored in a form that the Larch/C++ Checker could later use.

The first step, converting Larch/C++ `uses` clauses into useful LSL information, was a non-trivial task. For example, there needed to be a way to convert a `uses` clause like

```
//@ uses SimpleSetTrait(Elem for E, Set<Elem> for C);
```

into an equivalent LSL `includes` line. To do this, the Larch/C++ parser was modified to build strings of trait information. Two attributes were added to the parse tree nodes, a pointer to a `DyString` called `info` and a boolean called `TooComplex`. The `info` attribute points to the string that will eventually contain the trait and renaming information for the LSL `includes` line. The string is built as expected, bottom up, as the parse is conducted. The system handles template types, like the type `Set<int>` above, by translating the `<>` symbols into legal `[]` symbols. The system can do this because Larch/C++ requires that template types have equivalent sorts [8, Page 20].However, C++ allows for complex expressions as template arguments. How should the system handle those? The answer is, it does not. The `TooComplex` attribute is used to keep track of the simplicity of the template arguments. If something that the system cannot handle, such as an expression `1 < 2`, appears as an argument, the `TooComplex` flag is set. This causes the Checker to generate a warning that the `uses` clause was too complex to translate into an LSL `includes` directive.

Once the information in a `uses` clause has been translated, it needs to be passed to the LSL Checker. This requires some sort of interface between the Larch/C++ Checker and the LSL Checker. The interface used owes a lot of its basic design to the interface used by LCLint 1.4a [10]. The LCLint tool generates a temporary trait file that contains an `includes` statement for the trait information. It then uses the Standard C `system` call to run the LSL Checker over that file, redirecting the Checker's output into another temporary file. If `system` returns an error condition, then the LSL Checker is run again over the file to allow the error messages to be passed to the user.

As in the LCLint tool, the Larch/C++ Checker takes the information generated from the `uses` clause (or other expressions involving traits) and creates a temporary trait. This trait consists of an `includes` statement that uses the generated traits. The LSL Checker is then run on this temporary trait to both check its syntax and to generate the `-syms` output. Once this basic interface was sketched out, a few questions surfaced. How are the temporary files generated and managed? What is the best way to call the LSL Checker? Does the LSL Checker need to be run twice?

Temporary file management was the first item to be addressed. It was known that there was a `tmpfile` function in Standard C that would generate a uniquelynamed temporary file. The problems with this method were that the file was always automatically deleted, and that the return value of the call was a `FILE *`. Since the system was implemented in C++, it was felt that the system should use streams as the interface rather than `FILE` pointers. The system should also have the ability to keep its temporary files around both for debugging purposes, and perhaps for caching purposes in the future (see Section 6). It was also necessary to design the implementation of temporary files in a way that would be more easily portable to other operating systems.

The best solution to all of these issues was to design a temporary file class. The class would encapsulate the creation of the file names within its constructor, allowing us the ability to provide our own name or let the system choose one. It would also provide the ability to flag a file as undeleteable, so that it would not be reclaimed by the system. The destructor could check this flag, and delete the file as necessary. Figure 41 shows the header for the class `TmpFile` designed to meet these goals.

`TmpFile`, which is part of the common code for the system, is an abstract class. Derived classes may be specialized to allow for different behaviors. In the Larch/C++ Checker, a derived class `OutTmpFile` generates temporary files for system outputs. The classes carry along their path information, which is based upon a default path stored in a static member variable. The value of this variable defaults to the current directory, but may be reset as needed. The `Deleteable` member function is used to change the deleteable flag on the file. If the deleteable flag is true, the file will be deleted when its destructor is called. The class currently allows for an extension to be added to the temporary file. This was added because LSL requires traits to reside in files with the extension `.lsl`.

Once a system for generating temporary files was ready, it was time to design the method for calling the LSL Checker. All of the efforts sur-

```
#ifndef _TmpFile_h
#define _TmpFile_h 1
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include "DyString.h"
#include "dirname.h"

class TmpFile {

public:
  TmpFile();
  TmpFile(char *extension);
  virtual ~TmpFile();
  virtual bool open();
  virtual bool close();
  virtual char *GetPath() const;
  virtual char *GetFileName() const;
  virtual char *GetCompletePath() const;
  virtual bool IsDeleteable() const;
  virtual void Deleteable(bool flag);
  friend ostream& operator << (ostream& out, const TmpFile& tmp);
  static void SetDefaultPath(char *def_dir);

protected:
  static DyString default_path;
  DyString path;
  DyString filename;
  DyString extension;
  DyString CompletePath;
  bool deleteable;
  fstream mystream;
};
#endif
```

Figure 41: The `TmpFile` header

rounding the call are encapsulated within the `call_lsl` function. Figures 42 and 43 show the source code for this function. The first half of the function (Figure 42) sets up the temporary files that are needed. It creates two `OutTmpFile`s, `tmptrait` to hold the temporary trait information, and `symsout` to hold the output from the LSL Checker. The function checks to make sure that these files can be opened for writing. If not, it exits with an error. Once it has opened the files, it proceeds to create the temporary trait. Traits in LSL are required to have the trait name and the file name be the same. Thus the first thing `call_lsl` does is get the file name from `tmptrait`. It takes the name provided by the `OutTmpFile` and the `includes` line passed as an argument, and creates the temporary trait. For example, given the `uses` clause

```
//@ uses SimpleSetTrait(Elem for E, Set<Elem> for C);
```

`call_lsl` will be called with the `includes` parameter set to

```
SimpleSetTrait(Elem for E, Set[Elem] for C)
```

and will generate a temporary trait similar to the following.

```
eaaa04712:trait
includes SimpleSetTrait(Elem for E,Set[Elem] for C)
```

The second part of `call_lsl` handles the generation of the call to the LSL Checker and any resulting errors from that call; then it passes the results on to the next stage. `call_lsl` uses the environment variable `LSL_EXE_PATH` to locate the LSL Checker in the operating system. The use of an environment variable here was more flexible than either using a compilation macro to set the path, or simply choosing a path. The environment variable makes it easy for users to customize their installation of the tools. A call to the `system` function is built from the path to the LSL Checker and the name of the temporary files. The macro variable `EXECIT` is used to prepend an `exec` to the front of the string on systems that support `exec`. This was done to improve the ability of the system to handle user interrupts. The first version of `call_lsl` did not use `exec` and it had a tendency to not report errors or respond to user signals. For example, if a user hit Ctrl-C in the middle of checking, the system would return the user to the command prompt, but continue to execute. The abnormal behavior was traced to the shell invoked by the `system` call catching the signals that should have been

```
#include <signal.h>
#include <stdlib.h>
#include "OutTmpFile.h"
#include "LSLsup.h"
#include "debug.h"
#include "DGetEnv.h"
#include "execit.h"

extern int lslparse();
extern bool LSL_keep_on_error;
extern bool verbose;

int call_lsl(DyString includes){
  extern bool debug_flag;
  int val,pid,status;
  DyString sys_str;

  OutTmpFile tmptrait("lsl");
  OutTmpFile symsout;
  if(debug_flag){
    tmptrait.Deleteable(false);
    symsout.Deleteable(false);
  }
  if(!tmptrait.open()){
    return(TMPFILE_ERR);
  }
   if(!symsout.open()){
    return(TMPFILE_ERR);
  }
  tmptrait.write(tmptrait.GetFileName());
  tmptrait.write(":trait\n");
  tmptrait.write("includes ");
  tmptrait.write(includes.ToCppString());
  tmptrait.write("\n");
  tmptrait.close();
```

Figure 42: The call_lsl function, part one

passed upward. The addition of the `exec` to the command line passed to the `system` call allows for the messages to return back to `system` as expected.

After the actual `system` call, most of the remainder of `call_lsl` handles error conditions that may be returned. The return value of `call_lsl` is checked within the parser. If the system has been interrupted, this fact is passed up the call chain to the system driver, which halts the system. If the `system` call has executed correctly, the Larch/C++ Checker is ready to move on to parse the resulting `-syms` output.

Once the basic interface was present in `call_lsl`, and before the parser for the `-syms` output was written, the interface to the LSL Checker was tested. At first everything seemed to work fine. However, on very large traits the Larch/C++ checker consistently caused core dumps. After investigation, it was discovered that the LSL Checker had a large memory footprint that seemed to be causing the system to run out of memory. After consulting with the maintainers of the LSL Checker for ideas, the LSL Checker was modified to use the garbage collector also. This reduced its memory footprint to one-half the previous size, and allowed the Larch/C++ Checker to execute as expected.

Another discovery was that the LSL Checker did not follow the UNIX convention of sending output that reported errors to the `stderr` stream. This made clear why the implementors of the LCLint tool made two calls to the LSL Checker. Since the errors were reported on `stdout`, and they captured `stdout` to a file, the second execution was to allow for the errors to appear to the screen. Since modifications had already been made to support the garbage collector, the LSL Checker's output system was modified to support standard UNIX conventions. This allows the Larch/C++ checker to make a single system call and still get user messages output to the screen, and the `-syms` output to a file.

5.3.2 Data Structures for Sorts

Before one can parse and store the output from the LSL Checker, there must be data structures to store it in. The design of the data structures for sorts needed to embody the concept of sorts as they are used in LSL, while also being easy to build. The belief was that the sorts for C++ declarations might need to be built in pieces, bottom-up, as the parse occurred. This influenced the functionality of the design.

Originally, the Larch/C++ Checker had one class, `LCPPSort`, which was

```
  if(verbose){
    cerr << "Checking traits: " << includes.ToCppString() << endl;
  }
  DyString * lsl_exe = DGetEnv("LSL_EXE_PATH", new DyString("lsl"));
  sys_str = DyString(EXECIT) + *lsl_exe + DyString(" -syms ") +
            DyString(tmptrait.GetFileName());
  sys_str = sys_str + DyString(" > ") + DyString(symsout.GetFileName());
  val = system(sys_str.ToCppString());
  if(val < 0){
    return(SYSCALL_ERR);
  }
  if (WIFEXITED(val) && WEXITSTATUS(val) != 0) {
    if (LSL_keep_on_error) {
      tmptrait.Deleteable(false);
    }
    return(LSL_ERR);
  }
  if (WIFSIGNALED(val)) {
    tmptrait.close();
    return(WTERMSIG(val));
  }
  symsout.close();
  tmptrait.close();
  val = parse_lsl(symsout.GetFileName());
  return(val);
}
```

Figure 43: The `call_lsl` function, part two

used to store sort information. It simply stored the `TypeSpecifier` and `Declarator` information for a declaration. The design of the new sort system began with the `LCPPSort` class so that it would be easy to plug into the existing tools.

After examining documentation for LSL and a multitude of traits, a basic idea of what was needed to represent sorts was built. There are three types of sorts within LSL, atomic sorts, parameterized sorts, and arrow sorts. *Atomic* sorts are the basic sort building blocks. An example would be the sort `int`. Atomic sorts take no arguments, and cannot be broken down into smaller pieces.

Parameterized sorts are built by combining atomic sorts, or other parameterized sorts, into a more complex whole. An example would be the sort `Set[int]`, which consists of the two atomic sorts, `Set` and `int`. Even though most observed parameterized sorts only had one parameter, the decision was made to support the general case of an indeterminant number of parameters for a given sort.

Arrow sorts represent the signatures for LSL operators. An example would be `int , int -> int`. Arrow sorts consist of two pieces, an argument list which contains a list of atomic and parameterized sorts, and a result sort which is either a atomic or parameterized sort. Note that LSL does not support the concept of higher order operations; that is, arrow sorts cannot be used as either arguments or results. This fact simplifies the concrete implementation of these sorts.

Originally, the class hierarchy in Figure 44 was proposed. Within this hierarchy, `LCPPSort` was viewed as an abstract class, allowing for basic behavior to be shared, and so that the implementation could use `LCPPSort` pointers to hold any type of sort. The class structure represents the feeling that atomic sorts are basic and parameterized sorts build upon them. The arrow sorts exist in their own tree, but contain both atomic and parameterized sorts. While this design captured the behavior of the various sorts well, it was uncertain whether it would be practical in practice. As mentioned earlier, it was believed that the sorts for a C++ declaration might have to be built bottom-up. For example, the sort for a declaration `int i;` would first build an atomic sort `int`, later converting it to the parameterized sort `Obj[int]`. The question was raised whether a design for the classes could be found where these conversions would not have to happen. It was also noted that the list of parameters inside of a parameterized sort and the list of arguments within an arrow sort could be a mix of atomic and parameter-

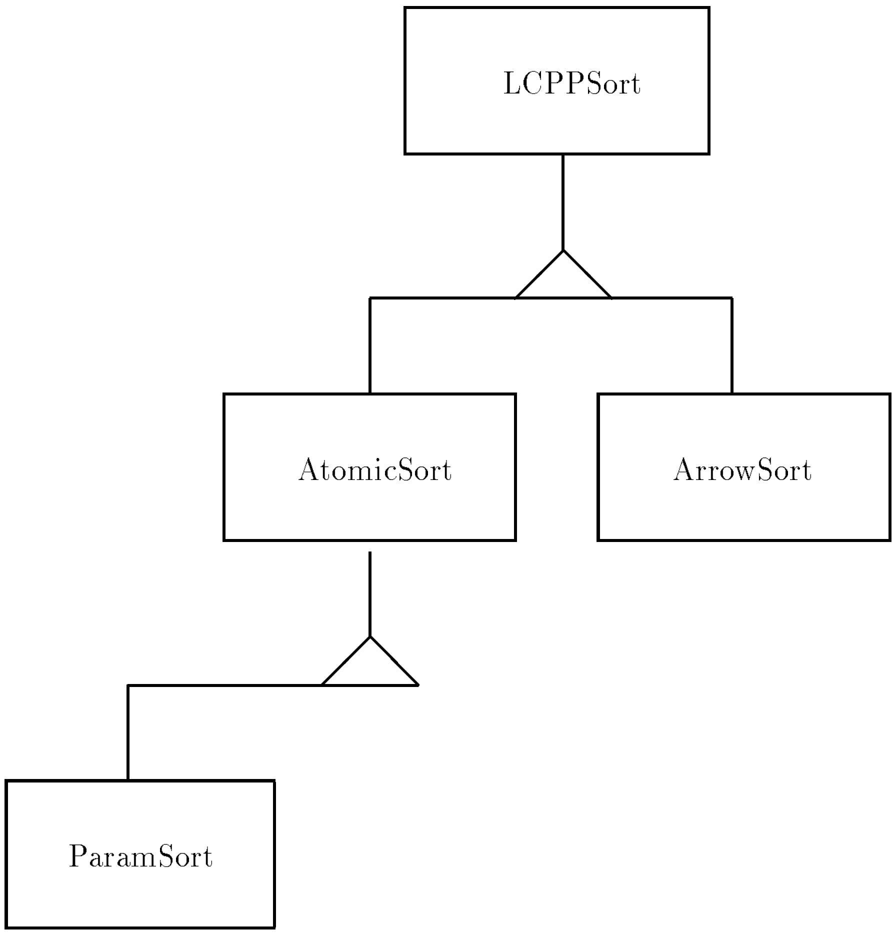

Figure 44: Class hierarchy for `LCPPSort`: first design

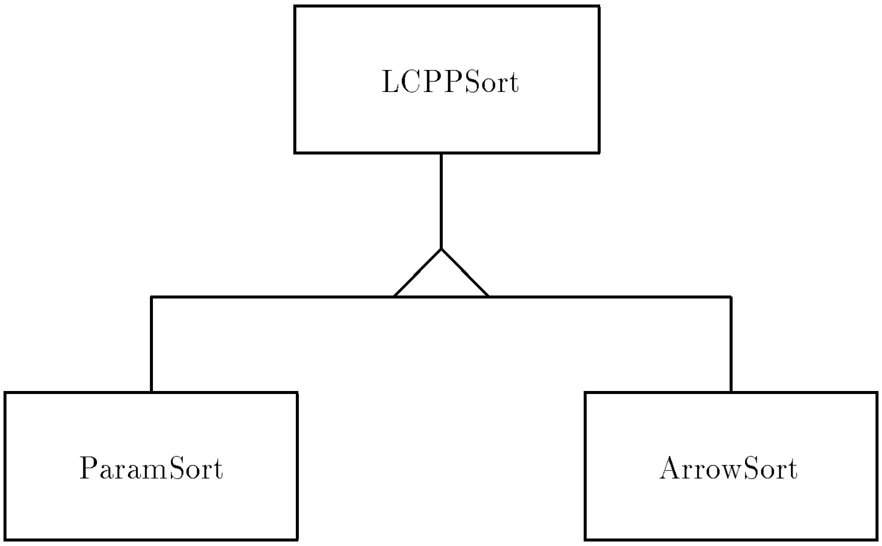

Figure 45: Class hierarchy for `LCPPSort`: final design

ized sorts. This could lead to later headaches when trying to decide if two parameterized sorts were equal. The lack of homogeneity would make the task more complex.

The realization that atomic sorts were equivalent to parameterized sorts with zero parameters simplified the class hierarchy for sorts to that shown in Figure 45. This hierarchy still supports the three forms of sorts, while not needing to be able to convert between atomic and parameterized sorts. With the new hierarchy a declaration `int i;` will first build a parameterized sort `int` which has no parameters. Later it would create the final sort `Obj[int]` by adding the `int` as a parameter to the `Obj` sort. The change in design also made lists of parameters in both parameterized and arrow sorts homogeneous.

The final piece of the puzzle for representing sorts was how to support the overloading of LSL operators. Remember from earlier discussion (Section 3.1) that LSL allows for a given operator name to have many different signatures associated with it. The choice of which signature to use is then made based upon the context of the use of the operator. The data structure needed to handle basic insertion, and insertion of duplicates easily. It would be convenient if it could handle retrieval based on different pieces of the arrow sort structure. The design chosen was an iterated set of arrow sorts. This data structure had the advantage of easily handling insertion, especially insertion of duplicates. Although the data structure itself does not handle retrieval of a given arrow sort by the parts of arrow sorts, i.e. by either parameter list or result sort, helper functions could provide that functionality.

Figure 46 shows the class `ArrowSet` used to implement the set of arrow sorts. The set is actually built on top of a singly linked list provided by the macros in the `SINGLYLL.h` file. The functionality is as expected for sets with the member functions modeling the usual mathematical behavior for sets. The member functions `Union`, `Intersection`, and `Diff` generate their results within the default parameter, rather than generating a completely new set. For example, given sets `S1` and `S2`, the operation `S1.Union(S2)` will change `S1` to hold `S1` $\cup$ `S2`. This behavior, while documented in the specification for the class, can be somewhat confusing.

The major addition to the implementation of set shown here is the ability to iterate through the items in the set. Most sets allow you to look to see if an item is in a set. When dealing with sets of arrow sorts, however, the user might only know the result sort he is looking for. This means that there needs to be a way to examine each member in the set, allowing for comparisons with the individual parts of an arrow sort. Thus the iterator, and its controlling functions, were added. The `Reset_Iterate` function clears any information about previous iterations and makes the set ready to iterate. The `Iterate` function returns the next member of the set in the iteration, or NULL if there are no elements remaining. The `Save_Iterate` and `Store_Iterate` member functions were added to help support the ability to print out the sets. The `operator <<` function was written using the `Iterate` function to output one member at a time. However, what if a user wanted to print out the set in the middle of an iteration without losing the iteration information. `Save_Iterate` saves the current iteration point, which can be later restored via `Restore_Iterate`. The use of all of the iteration functions is illustrated in the code for `operator <<` shown in Figure 47.

5.3.3 The Operator Signature Parser

Once the LSL Checker has been used to generate the `-syms` output, there needs to be a way of getting that information into the Larch/C++ Checker's symbol table. A tool was needed that would convert the `-syms` output into

```
#include "SINGLYLL.h"
#include "ArrowSrt.h"
typedef ArrowSort *ArrowSortPtr;
DECLARE_SINGLYLL(ArrowSortPtr,ArrowSortNode)
typedef ArrowSortNode *ArrowSortNodePtr;
class ArrowSet{
public:
  ArrowSet();
  ArrowSet(const ArrowSortPtr item);
  ArrowSet(const ArrowSet& set);
  bool In(const ArrowSortPtr item) const;
  int Cardinality() const;
  bool IsEmpty() const;
  bool Subset(const ArrowSet& set) const;
  bool Insert(const ArrowSortPtr item);
  bool Remove(const ArrowSortPtr item);
  void Union(const ArrowSet& set1);
  void Intersection(const ArrowSet& set1);
  void Diff(const ArrowSet& set1);
  bool operator == (const ArrowSet& set1) const;
  bool operator != (const ArrowSet& set1) const;
  void Save_Iterate();
  void Restore_Iterate();
  void Reset_Iterate();
  ArrowSortPtr Iterate();
protected:

  ArrowSortNodePtr theSet;
  ArrowSortNodePtr iteratePoint;
  ArrowSortNodePtr old_iteratePoint;
};
ostream& operator << (ostream& out, ArrowSet item);
```

Figure 46: The `ArrowSet` class

```
ostream& operator << (ostream& out, ArrowSet item){

  ArrowSortPtr val;
  int iter;

  iter = 0;
  out << "{";
  item.Save_Iterate();
  item.Reset_Iterate();
  val = item.Iterate();
  while(val != NULL){
    if (iter == 0){
      val->Print(out);
      iter++;
    }
    else {
      cout << ",";
      val->Print(out);
    }
    val = item.Iterate();
  }
  out << "}";
  item.Restore_Iterate();
}
```

Figure 47: The `operator <<` function for `class ArrowSet`

operator names and associated arrow sorts. The development of this portion of the system again looked to the previous work of LCLint for an idea of how to proceed. The LCLint tool used a hand-crafted lexer, combined with a `yacc`-based parser to generate data structures from the `-syms` data. A close examination revealed that the hand-crafted lexer was too complex to be easily converted for use within the Larch/C++ Checker. This led to the development of a `flex`-based lexer for this input. Starting from the token definitions in LCLint's signature file and the information in the LSL technical report [7], a set of regular expressions was developed to represent the tokens. The major difficulty here was that some characters had multiple meanings depending upon context. For example, the signature

```
[ __ , __ ] : int , int -> Tuple[int]
```

represents a function named `[]` which takes two `int` arguments and constructs a `Tuple[int]`. At first glance this does not seem difficult to break into tokens. However, the square brackets that form the name of the function are actually the tokens `OPENSYM` and `CLOSESYM`, while the square brackets in the sort `Tuple[int]` are the tokens `LBRACKET` and `RBRACKET`. How should the system be designed to return the correct tokens? The solution was to create a lexer with start states to allow for the differentiation of the uses for square brackets. *Start states* allow `flex` to use different matching rules based upon what has been seen in the input. In this case, the switching of states in the lexer is based upon having seen either a colon or a new-line character. The lexer knows that when it has seen a colon, it is within the signature for the operation. Thus the lexer always returns `LBRACKET` and `RBRACKET` in this case. When the lexer reaches a new-line character, it knows that it has reached the end of a complete signature. At this point it switches to a start state where square brackets are `OPENSYM`s and `CLOSESYM`s. Please see [9] for more details on the use of start states.

The initial testing of the operator signature parser revealed a minor problem. The system was consistently returning parse errors for input that looked correct. The problem turned out to be that the LSL Checker automatically broke long lines by inserting a new-line character and continuing the output on the next line. This was done to allow for users to easily read long lines that appeared on their screens. However, the insertion of the new-line character was causing the lexer for the signatures to switch states prematurely, leading to the parse errors. There were a number of possible solutions. One possibility was to change the lexer to use a different set of start states. While this was probably possible, the lexer as it existed was relatively simple to understand. If the lexer was to be modified, it would probably become more complex and difficult to understand. The other solutions were all based upon modifying the LSL Checker to change its output format. Any change to the output, however, had to preserve the output format used by the Larch Prover, and should continue to make the messages readable to the user. Since all output in the LSL Checker passed through the same output functions, some sort of conditional check was needed to turn on and off the output of the offending new-line. An additional boolean variable, `PrintingSyms` was added to the system to flag when the `-syms` output was being printed. If the flag is true, then no new-lines are inserted into the output. This allows the parser to work as expected.

The final piece of the operator signature parser is the interface to the Larch/C++ symbol table. This involved the creation of a new type of `Symbol` to hold the operator's information and a modification to the `Locale` portion of the symbol table. New classes `TraitOp` and `ExtendedTraitOp` were derived from the `Symbol` base class. Basically `TraitOp` objects look like all other `Symbols`, especially `Idents`, except for the fact that their sort information is an `ArrowSet` not a simple `LCPPSort`. `ExtendedTraitOp` is derived from `TraitOp` and has an extra name field that is used in reporting errors. For example, the signature `if __ then __ else __: Bool, Bool, Bool -> Bool` creates an `ExtendedTraitOp`. The name used to index the function in the symbol table is `ifthenelse`, while the name reported for errors is `if __ then __ else __`. Other signatures, such as `f: int -> float`, generate `TraitOps` because the name reported for errors is the same as the name used to index the operation in the symbol table.

The parser creates a local symbol table into which it places `TraitOp` and `ExtendedTraitOp` symbols as it parses the signatures. It is at this point where the system does the name conflict resolution described in Section 3.3. If the parser-generated symbol has a name that already exists in the `currentSymTab`, the trait operation is discarded, and a warning is reported to the user. Similarly, if the system is inserting a C++ identifier into the symbol table and discovers an existing trait operation with the same name, it discards the trait operation and warns the user.

The other major modifications to the symbol table involved adding an additional field to the `Locale` class, with its associated member functions, and the modification of the `SymTab` class to allow for copying of the new fields. Figure 37 shows the old structure of the symbol table. `Locales` contained two namespaces, `nonClassOrEnum` and `classOrEnum`. This form was modified to add a third field, `TraitOps`, to hold lists of the trait operations. This was needed to support the split of the symbol table into separate C++ and LSL operator "worlds", as described in Section 3.3. The addition of this field required the creation of supporting functions that operated upon this new field. The new functions are simply copies of the existing `Locale` functions which operate upon this new field. All operations on the new field were kept separate to make sure that the existing behavior of the Larch/C++ Checker did not change. Of these new functions, the only one examined in detail here is the new `CopyTraitOps` series of functions within `Locale` and `SymTab`. These functions are basically clones of the `CopySymbols` functions described in Section 5.2 modified to work with the new `TraitOps` field in the `Locale` class. These functions are used to copy the local symbol table created by the LSL operator signature parser into the Larch/C++ Checker's main symbol table. Please see the implementation for further details on all of the new functions.

The operator signature parser is used for more than support for `uses` clauses within specifications. It is also used to load all of the built-in traits, such as the traits for the basic C++ types, in the initialization of the Larch/C++ Checker. This information is stored at the top level of the symbol table, outside of any C++ scope so that it will not be removed by the name conflict resolution system.

5.3.4 Conversion of C++ declarations to LSL sorts

The final major piece of support required for sort checking within the Larch/C++ Checker is the ability to convert C++ declarations into their equivalent sorts. Recall from the discussion of the rules for state functions, Section 4.2.6, that C++ declarations consist of an abstract value wrapped within an object. This interaction between the sorts of the abstract values and the sorts of the enclosing objects can make it difficult to automatically generate sorts. Examples of declarations and their equivalent sorts taken from the Reference Manual [8, Chapter 5 and Section 6.1.8.1] are illustrated in Figures 48 and 49. Notice that the same form of declaration may have a different sort depending upon whether it is a declaration of a variable or a declaration of a formal parameter. Notice also the complexities surrounding the sorts for `struct`, `union`, and `class` types. These special cases make the code for the translation more complex.

```
Declaration              Sort of x (used as global)
-------------            --------------------------
const int &const x;      ConstObj[int]
const int *const x;      ConstObj[Ptr[ConstObj[int]]]
const int x[3][4];       Arr[Arr[ConstObj[int]]]
int *x[10];              Arr[Obj[Ptr[Obj[int]]]]
struct IntList {
  int val;
  IntList *next;
  };
IntList x;               ConstObj[IntList]
const IntList x;         ConstObj[Const[IntList]]
int (*x)(int i);         Obj[Ptr[ConstObj[cpp_function]]]
int (* const x)(int i); ConstObj[Ptr[ConstObj[cpp_function]]]
int (*x[10])(int i);     Arr[Obj[Ptr[ConstObj[cpp_function]]]]
```

Figure 48: Sorts for global C++ declarations

At first, the thought was that sort information would have to be gathered and passed as attributes throughout the parse tree. However, this method was discarded once the realization occurred that the parser was already gathering the information necessary for building sorts within the `TypeSpecifier` and `Declarator` objects. In Larch/C++, as in C++, type specifiers contain a type name and an associated qualifier (either `const` or `volatile`). `int`, `unsigned int`, `enum color`, and `const int`, are all examples of valid type specifiers [8]. Declarators contain the names of the items being declared along with a list of qualifiers that provide additional information. Figure 50 lists the qualifiers as presented in the Larch/C++ Reference Manual [8].

After it was realized that the type specifier and declarator information was enough to build sorts, the question became, at what point during a parse is this information complete and available? It turns out that the key was within the function `Declare`. `Declare` is used to add declarations to the current symbol table. It was clear at this point that all of the information required would be available. In examining the `Declare` function, it was noted that at certain points it made calls to the `LCPPSort` constructor. It was decided

```
Declaration      VarId   Its Sort (when used as a formal parameter)
-------------    -----   -----------------------------------------
int i            i       int
int & ir         ir      Obj[int]
int * ip         ip      Ptr[Obj[int]]
int ai[]         ai      Ptr[Obj[int]]
struct IPair {
  int fst, snd;
  };
Ipair sip;       sip     Val[IPair]
union FI {
  float f;
  int i;
  };
FI fi;           fi      Val[FI]
```

Figure 49: Sorts for formal parameter declarations

```
  Operator     Meaning
  ---------    -----------------
   *            Pointer
   ::*          Pointer to Member
   &            Reference
   []           Array
   ()           Function
```

Figure 50: Declarator qualifiers

that this would be the best place to generate sorts.

Once the location was known, a function needed to be developed to convert the type specifier and declarator information into the correct sorts. A preliminary version of this function for global declarations, called `BuildSort`, which does not handle classes, unions, or structs, is pictured in Figures 51 and 52. The portion of the function in Figure 51 handles the translation of the type specifier information into an equivalent sort. It first checks to see if a `const` specifier is associated with the declaration. If one is, it creates a parameterized sort `ConstObj`; otherwise it creates the sort `Obj`. Next it examines the `Symbol` associated with this type specifier. The Symbol carries the name of the type or `typedef`, for which this specifier was created. If you examine the class hierarchy for `Symbol` in Figure 38, you will see that one form of identifier is `BuiltInTypeName`. Many declarations carry `Symbol`s of this sort whose name fields are the built in types. If the `Symbol` is not a `typedef`, the function takes the name from the `Symbol`, creates a parameterized sort from the name, and then uses that name as a parameter to the sort created from the `const` specifier. If the `Symbol` associated with this type specifier is a `typedef`, it looks inside that symbol to find the correct sort to use as a parameter to the previously created sort. For example, this algorithm works as follows on a declaration of the form `int i;`.

- It creates a sort `Obj` because there is no `const` associated with the declaration.

- Since the symbol carried along with the `TypeSpecifier` is a `BuiltInTypeName`, the system creates another sort `int`.

- The system uses the second sort as a parameter to the first, creating the correct sort `Obj[int]`.

It is important to note that this portion of the system is dependent upon the system creating the correct type names for `BuiltInTypeName` `Symbol`s. When preliminary testing began, it was discovered that type specifiers such as unsigned int did not create the correct names. Functionality was added to the function `TypeScfr::Combine` to generate these names correctly.

The second half of `BuildSort` handles any associated declaration qualifiers. These are handled by building up a sort in pieces. The individual pieces are not semantically complete sorts, but when it finishes, it will have generated a complete sort. `BuildSort` starts out by checking to see if a pointer, `*`

```
ParamSort *BuildSort(const TypeScfr *typescfr,
                    DQNode *dqlist, bool IsMember){

  ParamSort *typescfrsort, *result;
  ParamSort *tmp1, *tmp2, *tmp3;
  DQNode *currlist, *revlist;

  if(typescfr == NULL){
    result = NULL;
  } else {
    if(typescfr->isConst){
      tmp1 = new ParamSort(DyString("ConstObj"),NULL);
    }
    else {
      tmp1 = new ParamSort(DyString("Obj"),NULL);
    }

    if((typescfr->sym->GetKeywordKind())
           == TypedefNonClassOrEnumNameTag){
      tmp1 = (ParamSort *)typescfr->sym->GetLCPPSort();
    } else {
      tmp2 = new ParamSort(*(typescfr->sym->GetName()),NULL);
      tmp1->AddParam(tmp2);
    }

    typescfrsort = tmp1;
    result = typescfrsort;
  }
```

Figure 51: Function `BuildSort`, part one

```
  currlist = dqlist;
  while(currlist != NULL) {
    DeclQual *tmp = Car(currlist);
    if(tmp->IsPointer()){
      if(tmp->IsConst()){
        tmp1 = new ParamSort(DyString("ConstObj"),NULL);
      }
      else {
        tmp1 = new ParamSort(DyString("Obj"),NULL);
      }
      tmp2 = new ParamSort(DyString("Ptr"),NULL);
      tmp2->AddParam(result);
      tmp1->AddParam(tmp2);
      result = tmp1;
    }
    else if (tmp->IsArray()){
      tmp1 = new ParamSort(DyString("Arr"),NULL);
      tmp1->AddParam(result);
      result = tmp1;
    }
    else if (tmp->IsFunction()){
      tmp1 = new ParamSort(DyString("ConstObj"),NULL);
      if (IsMember){
        tmp2 = new ParamSort(DyString("cpp_member_function"),NULL);
      } else {
        tmp2 = new ParamSort(DyString("cpp_function"),NULL);
      }
      tmp1->AddParam(tmp2);
      result = tmp1;
    }
    else {
    }
    currlist = Cdr(currlist);
  }
  return(result);
}
```

Figure 52: Function BuildSort, part two

has been seen. Declaration qualifiers carry information about whether they have been modified with a `const` keyword also. If the declaration qualifier is a pointer, the system generates one of the following incomplete sorts, either `ConstObj[Ptr]` or `Obj[Ptr]` from the pointer qualifier depending upon the `const`ness of the pointer. Then the sort created from the type specifier is added to the incomplete sort as a parameter, yielding a complete sort. As an example, look what happens to the declaration `int * const i;`:

- The type specifier builds a sort `Obj[int]` as described above.
- The first declaration qualifier is a pointer. The system checks to see if it is modified by a `const` keyword. Since it is, it generates the incomplete sort `ConstObj[Ptr]`.
- The system combines the sort from the type specifier and the declaration qualifier to get the final sort `ConstObj[Ptr[Obj[int]]]`.

The other declaration qualifiers work in a similar way, building the correct sorts for themselves based upon the definitions in the Reference Manual [8]. As the algorithm cycles through the list of qualifiers, it builds the sort from the inside out.

Once the basic declaration qualifier code was built, another abnormality was discovered in testing. Member functions which were referenced outside of the actual class declaration were not given the sort `cpp_member_function` as they should have been. The class `Point` in Figure 53 is a specification that illustrates this problem.

This specification attempts to redeclare the functions outside of the class declaration. When this happened, the system had no way of knowing that the functions were actually member functions and that their sorts were redeclared as `cpp_function`. To solve this problem, the `Declare` function was modified to check the `Symbol` it was declaring to see if it already has a sort. If it does, `Declare` keeps that type and does not create a new one. The code for this is illustrated in Figure 54

The code that performs the check

```
sym->GetLCPPSort() == NULL
```

is making sure that the `Symbol`'s sort pointer has not been assigned to previously. If this is the case, the system builds a sort for the `Symbol sym`.

```
class Point {
public:
  uses informally "pairs of [x,y] values";
  Point();
  int x_val() const;
  int y_val() const;
  void set_x(int xv);
  void set_y(int yv);
};
extern Point::Point();
behavior {
  constructs self;
  ensures self\post = [0,0];
}
extern int Point::x_val() const;
behavior {
  ensures result = self\any.x;
}
extern int Point::y_val() const;
behavior {
  ensures result = self\any.y;
}
extern void Point::set_x(int xv);
behavior {
  modifies self;
  ensures self\post.x = xv /\ (self\pre).y = (self\post).y;
}
extern void Point::set_y(int yv);
behavior {
  modifies self;
  ensures self\post.y = yv /\ (self\pre).x = (self\post).x;
}
```

Figure 53: The Point.lcc specification

```
if (sym->GetKeywordKind() == OriginalClassNameTag
    || sym->GetKeywordKind() == TemplateClassNameTag) {
  // it's a constructor
  Symbol *ctorSym = new Ident(sym->GetName(), DD_Declared);
  ctorSym->SetLCPPSort(BuildSort(
              currentTypeScfr, dcltor->dcl_quals,IsMember));
  sym->AddCtor(ctorSym);
} else {
  if(sym->GetLCPPSort() == NULL){
    sym->SetLCPPSort(BuildSort(
             currentTypeScfr, dcltor->dcl_quals,IsMember));
  }
  illFormed = AddCheckingDeclarable(currentSymTab, sym) || illFormed;
}
```

Figure 54: Code from Declare checking for previous sort

As mentioned earlier, formal parameters may have sorts that differ from similar global declarations. A function BuildArgSort which works in a similar manner as BuildSort handles the different formulations.

5.3.5 Additional Work

To complete the development of the sort checking system for the Larch/C++ Checker, additional work needs to be done. The lookup algorithm described in Section 3.3 needs to be implemented and tested. Also, the support functions for dealing with the ArrowSet objects need to be defined and implemented. Finally, the main grammar file needs to be modified to actually perform the sort checking. It is expected that work will continue on this portion of the project in the future.

6 Future Work

In the future, steps need to be taken to improve the performance of the Larch/C++ Checker. One idea, raised when the memory problems were found in the LSL Checker, is the use of a cache to hold traits that have been processed previously. This "symscache" could then be checked to see if a trait had already had the `-syms` output created. It it had been created and is up-to-date, the Larch/C++ checker could simply parse that file rather than calling the LSL Checker to regenerate the information. Problems that would need to be addressed include how to keep track of which traits are in the cache, how to keep track of how the traits have been renamed, and the policy for regenerating the information. Another major issue has to do with renaming. When items are renamed in a uses clause or other trait use, the output generated by the LSL Checker reflects that renaming. Somehow, the cache will need to be able to look at a file, tell how it was renamed, and see if that renaming is equivalent to the current use.

Another area for future work is an examination of the structure of the Larch/C++ source code. Perhaps the application of design patterns could create a clearer, less cluttered design. One place where this could be applied is in the creation of an class based on the iterator pattern [5] to handle the iteration of the sets of arrow sorts.

Another place in the code that could stand a redesign is the `TmpFile` classes. The current system could be improved by rewriting the base `TmpFile` class so that it inherits from the class `fstream`. This implementation would require fewer classes, because the functionality of the `OutTmpFile` class could be moved into `TmpFile`. It also would allow the use of the C++ `putto` (`<<`) and `getfrom` (`>>`) operators on `TmpFile`s without the complex overloading that is required in the present implementation.

7 Conclusions

The goal of this project was to add the ability to sort check specifications to the Larch/C++ Checker. The current status of the project shows the the original goal is obtainable. This project has contributed the basic support needed within the Larch/C++ Checker, and the formalization of the Larch/C++ type system, that will allow for a sort checker to be added. The contributions are as follows:

- Creation of a formal and systematic statement describing the Larch/C++ type system.

- Development of an interface between the existing Larch/C++ Checker and the LSL Checker.

- Development of an automated system for translating C++ declarations into equivalent LSL sorts.

- Support for the Evolving C++ Language Standard

This work, and the work to develop the sort checker itself, continue as this is written. It is my hope that the work I have done will make the future work of others easier.

8 Acknowledgments

This work was partially funded under NSF grant CCR-9503168.

Many people assisted in the above work. Dr. Gary Leavens gave me the opportunity to work on this project and supported me every step of the way. My family was always there for me when I needed them. Marybeth Gurski offered support and insight every step of the way. Professor John Hagge filled in at the last minute on my committee and offered outstanding criticism on this paper. Dr. Kelvin Nilsen offered moral support throughout my graduate career. The following people created the tools used in my project: Kurt Bischoff, Hans Boehm, et al., previous members of the Larch/C++ project, and others who previously have worked with the Larch Shared Language and its associated languages. Finally, to any and all others who have been forgotten, thanks one and all.

References

[1] Kurt Bischoff. *Ox: Users Reference Manual.* IA, 1993.

[2] Hans J Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, Sept 1988.

[3] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, April 1995.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., Reading, Mass., 1990.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, MA, 1995.

[6] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[7] John V. Guttag, James J. Horning, and Andrés Modet. Report on the Larch Shared Language: Version 2.3. Technical Report 58, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, April 1990. Order from src-report@src.dec.com.

[8] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.6. ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or http://www.cs.iastate.edu/~leavens/larchc++.html, June 1997.

[9] John R. Levine, Tony Mason, and Doug Brown. *lex and yacc.* O'Reilly and Associates, Inc., CA, 1992.

[10] MIT/LCS. LCLint checker, version 1.4a. http://larch-www.lcs.mit.edu:8001/larch/lclint/, June 1997.

[11] MIT/LCS. LSL checker, version 3.1beta3. http://larch-www.lcs.mit.edu:8001/larch/lsl.html, June 1997.

[12] David A. Schmidt. *The Structure of Typed Programming Languages.* MIT Press, MA, 1994.

[13] David A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall, New York, N.Y., 1990.

[14] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.