9-1994

Subtyping, Modular Specification, and Modular Verification for Applicative Object-Oriented Programs

Gary T. Leavens
Iowa State University

William E. Weihl
Iowa State University

Subtyping, Modular Specification, and Modular Verification for Applicative Object-Oriented Programs

Gary T. Leavens and William E. Weihl

TR #92-28d
September 1992, revised September, October 1993,
and January, September 1994

Keywords: verification, specification, supertype abstraction, subtype, message passing, polymorphism, type checking, modularity, soundness, object-oriented, abstract data type.

1992 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages; D.2.4 [*Software Engineering*] Program Verification — Correctness proofs; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types, procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*] Specifying and verifying and reasoning about programs — logics of programs, pre- and post-conditions, specification techniques; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — algebraic approaches to semantics, denotational semantics.

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

Subtyping, Modular Specification, and Modular Verification for Applicative Object-Oriented Programs

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

William E. Weihl
Laboratory for Computer Science, Massachusetts Institute of Technology
545 Technology Square, Cambridge, Mass. 02139 USA
weihl@lcs.mit.edu

September 6, 1994

Abstract

We present a formal specification language and a formal verification logic for a simple object-oriented programming language. The language is applicative and statically typed, and supports subtyping and message-passing. The verification logic relies on a behavioral notion of subtyping that captures the intuition that a subtype behaves like its supertypes. We give a formal definition for legal subtype relations, based on the specified behavior of objects, and show that this definition is sufficient to ensure the soundness of the verification logic. The verification logic reflects the way programmers reason informally about object-oriented programs, in that it allows them to use static type information, which avoids the need to consider all possible run-time subtypes. We also show that the logic does not require reverification of unchanged code when legal subtypes are added to a program.

*The work of both authors was supported in part by the National Science Foundation under Grant CCR-8716884, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988. While a graduate student at MIT, Leavens was also supported in part by a GenRad/AEA Faculty Development Fellowship, and at ISU he has been partially supported by the ISU Achievement Foundation and by the National Science Foundation under Grant CCR-9108654.

1 Introduction

In object-oriented programming, message passing allows the manipulation of objects without knowledge of their exact run-time types. This causes difficulties in verification, because many different operations may be invoked by a single message at different times, and these operations may have different specifications. The technique of *supertype abstraction* [40] [38] [35] overcomes this problem by reasoning using static (what we call *nominal*) type information, and restricting the run-time types of the objects denoted by expressions to be subtypes of their nominal types. That is, supertype abstraction means using supertypes to stand for all their subtypes.

Supertype abstraction has the advantage of *modularity*: one does not have to respecify or reverify unchanged program parts when new subtypes of existing types are added to a program [38] [39]. Our results can thus provide formal support for the common informal practice of object-oriented programming, as it shows conditions under which programmers do not have to rethink unchanged code, and those which might be dangerous. The danger comes when a new type is not a *legal subtype* of some existing type. We give a formal definition of legal subtyping that guarantees that verification using supertype abstraction is sound. To make such a guarantee, the notion of a legal subtype relation has to be stronger than the implementation inheritance (or subclass) relation [58] [31] [55]. The notion of legal subtyping must even be stronger than the syntactic guarantee that the new type will not cause type checking (or "message not understood") errors (see, for example, [8]). It must instead be a behavioral notion, based on the specification of an abstract data type [1] [42] [40] [30] [15] [2] [44] [45]. (See Section 8 for a discussion of related work.)

As an example of the distinction between legal subtyping and subclassing, consider two types `IntSet` and `Interval`, where `Interval` is a type of closed intervals of integers, and `IntSet` is a type of integer sets. (Both types of objects are *immutable*—they have no time-varying state; see Section 2 for their specifications). Since `Interval` is a subtype of `IntSet`, a program can use `Interval` objects as if they were `IntSet` objects. However, one might choose to represent `Interval` objects with two integers (the end-points), instead of inheriting the implementation from a class that implements the type `IntSet`. Thus the type `Interval` may be a legal subtype of `IntSet` without their implementations being related. Similarly, one can implement a subclass of the class `IntSet` in such a way that some of the operations go into infinite loops. The objects of such a class would not behave like objects of type `IntSet`, despite the inheritance of data structures and some code; hence such a class would be a subclass that did not implement a legal subtype.

1.1 Contribution

The main technical contributions of this paper are a formal definition of legal subtype relations and a sound verification logic for object-oriented programs with message passing and subtyping.[1] The programming language to be verified is the Little Object-Oriented Applicative Language, called LOAL (described in Section 5). In more detail the important contributions are as follows.

- A formal interface specification language, called Larch/LOAL (in Section 2), and its model-theoretic semantics (in Section 3).

[1] This verification logic is the first to formally treat code that uses both subtyping and message passing [40].

fun inBoth(s1,s2: IntSet) **returns**(i:Int)
    requires ¬(isEmpty(s1 ∩ s2))
    ensures (i ∈ s1) ∧ (i ∈ s2)

Figure 1: Specification of the function *inBoth*.

- A model-theoretic definition of legal subtype relations (in Section 4), which is based on the semantics of type specifications. Since it is based on the semantics of type specifications, it takes the behavior of objects into account, but is not specific to Larch/LOAL.
- A Hoare logic for LOAL that uses supertype abstraction, and most importantly, a proof of its soundness (both in Section 6). We also formally prove some aspects of Larch/LOAL's modularity.

The key to the soundness of the Hoare logic is the semantic restrictions placed on legal subtype relations. Our definition of legal subtyping can handle *incomplete specifications*— specifications that may have observably distinguishable implementations. For example, `IntSet` with its `choose` operation is incompletely specified when the specification does not say what element of a set `choose` should return. Such specifications are important because they leave design decisions open for both implementors and subtypes. Our definition of legal subtype relations also provides additional intuition beyond the informal motto that each object of a subtype should act like some object of its supertypes [55], for certain kinds of incomplete specifications. For incomplete specifications which do not have a "best" implementation, the informal motto allows surprising behavior— several objects of the subtype could collectively act differently than what one would expect from the supertype's specification.

Less importantly, we present a model theory for specifications that generalizes the work of [53] and [6]. Our simulation relations are tailored to handle incomplete specifications and are preserved by assertions in Larch/LOAL and by LOAL expressions and programs. Simulation relations are the main technical tool used to define legal subtype relations.

Note, however, that we only deal with first-order[2], immutable, abstract data types. Subtyping for mutable types (types whose objects have time-varying state) is still a subject of research [17] [44] [45]. By ruling out mutation, attention is focused on two other features that make reasoning difficult in object-oriented languages: message passing and subtyping.

1.2 An Example of the Reasoning Problem

This section motivates the need for supertype abstraction and modularity. (See also [38] and [35] for more background.)

Consider the specification of Figure 1. In that figure, the meaning of the operators used in the pre-condition (following **requires**) and the post-condition (following **ensures**) is expressed using trait functions from the specification of `IntSet`, for example ∈, ∩, and "isEmpty". What does "i ∈ s1" mean if "s1" is an `Interval`?

Suppose that, before adding the type `Interval` to a program, one has verified that the implementation of *inBoth* in Figure 2 is correct (when it is passed arguments of type

[2]A first-order abstract data type is one without type parameters.

```
fun inBoth (s1,s2:IntSet) =
    testFor(choose(s1), s1, s2);

fun testFor (i:Int, s1,s2:IntSet) =
    if elem(s2, i)
    then i
    else testFor(choose(remove(s1,i)), remove(s1,i), s2)
    fi
```

Figure 2: Implementation of *inBoth*.

fun inBoth[T1,T2: IntSetLikeType] (s1:T1, s2: T2) **returns**(i:Int)
    requires ¬(isEmpty(s1 ∩ s2))
    ensures (i ∈ s1) ∧ (i ∈ s2)

Figure 3: Traditional specification of *inBoth*.

`IntSet`). Does one have to go back and reverify the implementation of *inBoth* when it becomes possible to pass it arguments of type `Interval`?

The standard technique used to specify a polymorphic module is to specify the behavior of the operations that the polymorphic module needs to do its work [25, Page 21] [64, Section 4.2.3] [20, Page 537]. The specification of such operations is often collected into the specification of a "type parameter". For example, roughly following Goguen, one might specify the function *inBoth* as in Figure 3. The conditions that a type would have to satisfy to be an `IntSetLikeType` would be stated elsewhere, but would certainly include a specification that such a type must have operations `choose`, `remove`, and `elem` with appropriate signatures and semantics.

The problem with the specification of Figure 3 is that to check that an instantiation is correct during design or verification, the actual type parameter must be statically shown to satisfy the formal's specification. In a language with message passing, such as Smalltalk-80 [22] or LOAL, the actual type parameter cannot, in general, be uniquely determined during design or verification. One might try an exhaustive case analysis by doing the verification for each possible type parameter. However, this case analysis would have to be extended when new subtypes are added to the program. Therefore this approach is not modular and must be generalized to deal with message passing.

Conventional techniques for program verification suffer from similar problems, because they assume that each expression of type `T` denotes an object of type `T`. Thus they assume that the properties of objects of type `T` can be used to reason about expressions of type `T`. However, to exploit subtype polymorphism in a typed language, one must allow expressions of type `T` to denote objects of subtypes of `T`.

1.3 Overview of the Method

To solve the specification and verification problem discussed above, we use the following method [38].

- One specifies the data types to be used along with their subtype relationships.
- One specifies the operations of data types and functions using supertype abstraction. That is, one assumes that each argument has a specified type, which is that formal's nominal type. However, such a specification means that arguments whose types are subtypes of the corresponding formal's nominal type are allowed.
- The semantics of the type specifications must be checked to ensure that the specified subtype relation is *legal*.
- One verifies that a program meets its specification by reasoning about expressions as if they denoted objects of their nominal types, despite message passing.

In LOAL, each expression's static type is its nominal type. Any static type would do as the nominal type for an expression in a language without a type system, provided that it had the property that the nominal type is an upper bound (in the subtype ordering) on the types of objects the expression can denote.

The use of supertype abstraction in specifications brings up the question of how to reason about a call to a function with arguments whose types are subtypes of the formals' nominal types. The Hoare logic for LOAL requires an assertion describing actual arguments, written in terms of these subtypes, to be translated into the terms of the formals' nominal types. This is because in the logic one reasons at the level of the nominal types of expressions; that is, one uses supertype abstraction. But one would also like to give a direct meaning to such a specification, as a consistency check that this style of reasoning makes sense, and as a way to exploit more exact type information when it is available. One can give such a direct meaning if one can interpret assertions, written in terms tailored to the supertypes, for any subtype. Terms in Larch/LOAL are composed of identifiers, "=", and various *trait functions*, such as $\in$ for the values of `IntSet`. The trait functions come from the traits of the Larch Shared Language [26]. We require that each trait function defined on arguments of a supertype be overloaded for each possible subtype. In this way we can give a meaning to function and type specifications that is independent of any assumption of legal subtyping.

Because "=" is treated differently than trait functions, to avoid anomalies, assertions cannot use "=" freely. That is, Larch/LOAL prohibits specifications from using "=" between terms of user-defined types (but allows "=" to be used between terms of *visible types*: `Int` and `Bool`). Such terms are called *subtype-constraining*. We show that if one can prove a subtype-constraining assertion using the theory of a supertype, then the assertion is also valid for subtype objects, provided the conditions on legal subtyping are met.

IntSetTrait: **trait**
    includes Set(Int **for** E),
    introduces __eqSet__: C,C → Bool
        isEmpty: C → Bool
    asserts ∀ s_1, s_2: C
        (s_1 eqSet s_2) == (s_1 = s_2)
        isEmpty(s_1) == (s_1 = {})

Figure 4: The trait IntSetTrait.

2 Polymorphic Type Specifications

The interface specification language Larch/LOAL is adapted from Wing's interface specification language for CLU [64] [43, Chapter 10] [24] [63] and Chen's Larch/Generic interface specification language [11]. However, unlike Larch/CLU, Larch/LOAL specifications deal only with immutable types.

An interface specification describes both the behavior of abstract types and how they can be used in a program [33] [26]. In Larch/LOAL, the interface describes how a LOAL program can use the types. The program sees a polymorphic *method*, which is implemented by the *operations* of all the abstract types with the same name and number of arguments.

2.1 Traits

Larch/LOAL specifications describe behavior in terms of the abstract values of objects [29] [43] [35] [26]. In Larch/LOAL, the abstract values of objects are specified by a *trait* written in the Larch Shared Language [28] [27] [26]. (The Larch Shared language is used by Larch/LOAL, but is a distinct language. Both are distinct from LOAL itself. A LOAL program uses the abstract types specified in Larch/LOAL, and the assertions in a Larch/LOAL specification are stated using the trait functions described in a trait.) For example, the abstract values of the type `IntSet` (finite sets of integers) are described by the trait IntSetTrait found in Figure 4. The trait functions described in a trait cannot be used by programs, but are only available in specifications; conversely, abstract type operations and methods cannot be used in specifications.

2.2 Meaning of Traits

A trait is similar to a first-order equational algebraic specification. The trait IntSetTrait defines the usual notation for sets. It does this by importing the trait Set found in the Larch Shared Language Handbook [26, Appendix A], renaming the sort "E" in Set to "Int" in IntSetTrait. (A *sort* is a name for a kind of abstract value, which may be defined by a trait.) In IntSetTrait the names and signatures of additional trait functions are described after the keyword **introduces**. For example, IntSetTrait introduces the trait function "eqSet", which can be invoked with an infix-syntax (the characters "_" show the positions of arguments). The **asserts** section presents equational specifications of the trait functions. The first assertion in IntSetTrait says that "eqSet" means the same thing as "=" when one compares terms of sort "C" (i.e., sets).

The traits of a specification can have hidden sorts; that is, sorts that are used for convenience in specification, but which are not related to any type. Such sorts are not types, and hence cannot be used in programs.

2.2.1 Traits and Modularity of Specifications

In Larch/LOAL, operation and function specifications are written as if each argument and result had an abstract value of the specified type. However, to allow use of subtypes, actual arguments and results are allowed to have types that are subtypes of the specified types. An example is the interface specification of *inBoth*, found in Figure 1. It specifies that the arguments may be instances of any subtype of `IntSet`, but its post-condition is written using the trait functions of IntSetTrait, which describe the abstract values of the type `IntSet`. The advantage of this approach is that it is modular — new subtypes added to the program do not affect the specification, since subtypes are not mentioned explicitly.

The problem is to give meaning to such specifications when actual arguments do not have the specified types (for example, when an argument to *inBoth* has the type `Interval`). Our approach is to require that the trait functions defined on abstract values of the supertype must also be defined on abstract values of the subtype. For example, the trait IntSetTrait describes trait functions such as "insert," "delete," "size," etc., all of which must be applicable to the abstract values of a subtype such as `Interval`. Binary operations, such as $\cup$, must be defined for each combination of argument types.

One way to define all the needed trait functions for abstract values of `Interval` objects is to use a coercion function. In Figure 5, the trait IntervalTrait does this by using the coercion function "toSet" which maps intervals constructed with "[,]" to sets. The necessary definitions of the trait functions "insert," "delete," "size," etc., are found in the trait, IntervalSubTrait, where these trait functions are defined on arguments of sort `C` (i.e., `Interval`) by first coercing the abstract values using "toSet". The trait IntervalSubTrait is separated from IntervalTrait so that one may more easily see how it relates to the shorthand version of IntervalTrait in Figure 6, which would expand into Figure 5. The trait IntervalTrait also defines some new trait functions. Its **implies** section states a consequence that the specifier wants to highlight. The trait function "eqSet" is defined for combinations of `IntSet` and `Interval` arguments so that the arguments are viewed as sets.

The way that the trait IntervalSubTrait defines the trait functions that in IntSetTrait take `IntSet` arguments is standard. With some additional syntax, such traits could be defined automatically, as in Figure 6, which would expand into Figure 5. The **subtrait of** line says that the IntervalSubTrait in Figure 5 is to be created and included. Another way to avoid the work of defining the trait functions that apply to the supertype for the subtype would be to use a language for specifying abstract values that had an explicit notion of subsorts; for example, one might use order-sorted algebra [21].

2.3 Type Specifications

Examples of Larch/LOAL type specifications are given in Figures 7 and 10.

To follow Smalltalk and other object-oriented languages we specify types in pairs; for example, in Figure 7 we specify both a type for instances (i.e., `IntSet`) and a type for the class object (i.e., `IntSetClass`). These two types together make what is usually thought of as a type in a language like CLU or Ada, since only the class object can be sent messages to create instances from scratch.

IntervalTrait: **trait**
includes IntervalSubTrait
introduces [_,_]: Int, Int → C
leastElement, greatestElement: C → Int
asserts ∀ x, y: Int
[x,y] == **if** $x \leq y$ **then** [x,y] **else** [x,x] **fi**
leastElement([x,y]) == x
greatestElement([x,y]) == **if** $x \leq y$ **then** y **else** x **fi**
toSet([x,y]) == **if** $y \leq x$ **then** {x} **else** insert(toSet([x,$y \Leftrightarrow 1$]), y) **fi**
implies ∀ x, y: Int
isEmpty([x,y]) == false

IntervalSubTrait: **trait**
includes IntSetTrait(IntSet **for** C)
introduces toSet: C → IntSet
insert, delete: C, Int → IntSet
size: C → Int
__∈__: C,Int → Bool
isEmpty: C → Bool
∪, ∩: C,C → IntSet
∪, ∩: C,IntSet → IntSet
∪, ∩: IntSet,C → IntSet
__eqSet__: C,C → Bool
__eqSet__: C,IntSet → Bool
__eqSet__: IntSet,C → Bool
asserts ∀ c, c_1: C, s: IntSet, i: Int
insert(c, i) == insert(toSet(c),i)
delete(c, i) == delete(toSet(c,i))
size(c) == size(toSet(c))
($i \in c$) == ($i \in$ toSet(c))
isEmpty(c) == isEmpty(toSet(c))
($s \cup c$) == ($s \cup$ toSet(c))
($c \cup s$) == ($s \cup$ toSet(c))
($c \cup c_1$) == (toSet(c) ∪ toSet(c_1))
($s \cap c$) == ($s \cap$ toSet(c))
($c \cap s$) == ($s \cap$ toSet(c))
($c \cap c_1$) == (toSet(c) ∩ toSet(c_1))
(s eqSet c) == (s eqSet toSet(c))
(c eqSet s) == (s eqSet toSet(c))
(c eqSet c_1) == (toSet(c) eqSet toSet(c_1))

Figure 5: The traits IntervalTrait and IntervalSubTrait.

IntervalTrait: **trait**
subtrait of IntSetTrait(IntSet **for** C) **by** toSet **subsort** C **supersort** IntSet
introduces [_,_]: Int, Int → C
leastElement, greatestElement: C → Int
asserts ∀ c, c_1: C, s: IntSet, x, y, i: Int
[x,y] == **if** $x \leq y$ **then** [x,y] **else** [x,x] **fi**
leastElement([x,y]) == x
greatestElement([x,y]) == **if** $x \leq y$ **then** y **else** x **fi**
toSet([x,y]) == **if** $y \leq x$ **then** {x} **else** insert(toSet([x,$y \Leftrightarrow 1$]), y) **fi**
implies ∀ x, y: Int
isEmpty([x,y]) == false

Figure 6: A short-hand version of the trait `IntervalTrait`.

In our type specifications each class type, such as `IntSetClass`, is implicitly specified as a one-object type with a nullary operation, such as `IntSet`, and any class operations specified. The one object is the class object. The trait that specifies its abstract value for this example is given in Figure 9. Figure 8 shows what a type specification for `IntSetClass` would look like, but this would not be explicitly written in Larch/LOAL (and does not follow the Larch/LOAL syntax exactly). In Figure 7, the class operation `null` is specified to take the class object and return an empty `IntSet`. It is invoked as `null(IntSet)`, which is sugar for `null(IntSet())`. The expression `IntSet()` denotes the `IntSet` class object (of type `IntSetClass`), and thus `null(IntSet())` means to pass this class object to the method `null`, which returns an empty `IntSet` instance.

The instance operations of `IntSet` are also specified in Figure 7. None of the operations changes the state of an existing `IntSet`; hence instances of `IntSet` are immutable. Note that:

- `choose` can return an arbitrary element of a nonempty `IntSet` (that is, an implementation can be nondeterministic), and
- the "size" in the post-condition of `size` means the trait function "size", as message names cannot be referred to in assertions.

The instance operations of `IntSet`'s subtype `Interval` (see Figure 10) have the same names as those for `IntSet`. However, the class operations are different; instead of `null` there is an operation `create` that takes the `IntervalClass` object and two integer arguments and returns an `Interval` object representing all the integers between the arguments (inclusive). The integer arguments to `create` must be ordered as specified in the pre-condition. The `ins` and `remove` operations may return either objects of type `IntSet` or `Interval`, depending on their arguments. When applied to an `Interval`, the `choose` operation is deterministic, and will always return the least element of the `Interval`.

Having `choose` be more deterministic on `Interval` than on `IntSet` is desirable for several reasons. If one represents an `IntSet` by a linked list of integers, one might want to return the first element of the list as the result of `choose`. Since this has little to do with the abstract values, it will appear non-deterministic to clients. But it would be strange to specify `Interval` so that `choose` was *required* to be so non-deterministic. Indeed, making

IntSet **immutable type**
class ops [null]
instance ops [ins, elem, choose, size, remove]
based on sort C **from** IntSetTrait
op null(c:IntSetClass) **returns**(s:IntSet)
ensures s eqSet {}
op ins(s:IntSet, i:Int) **returns**(r:IntSet)
ensures r eqSet (s ∪ {i})
op elem(s:IntSet, i:Int) **returns**(b:Bool)
ensures b = (i ∈ s)
op choose(s:IntSet) **returns**(i:Int)
requires ¬ isEmpty(s)
ensures i ∈ s
op size(s:IntSet) **returns**(i:Int)
ensures i = size(s)
op remove(s:IntSet, i:Int) **returns**(r:IntSet)
ensures r eqSet delete(s,i)

Figure 7: The type specification `IntSet`.

IntSetClass **immutable type**
meta ops [IntSet] **instance ops** [null]
based on sort IntSetClass **from** IntSetClassTrait
op IntSet() **returns**(c:IntSetClass)
ensures c eq IntSet
op null(c:IntSetClass) **returns**(s:IntSet)
ensures s eqSet {}

Figure 8: The implicit type specification of `IntSetClass`, which is the type of the class object denoted by `IntSet()`.

IntSetClassTrait: **trait**
introduces IntSet: → IntSetClass
eq: IntSetClass, IntSetClass → Bool
asserts ∀ c_1,c_2: IntSetClass
c_1 == c_2
c_1 eq c_2

Figure 9: The implicit trait IntSetClassTrait, which describes the abstract value of the class object denoted by `IntSet()`.

Interval **immutable type**
subtype of IntSet **by** $[l, u]$ **simulates** toSet($[l, u]$)
class ops [create] **instance ops** [ins, elem, choose, size, remove]
based on sort C **from** IntervalTrait
op create(c:IntervalClass, lb,ub:Int) **returns**(i:Interval)
requires lb $\leq$ ub
ensures i eqSet [lb,ub]
op ins(s:Interval, i:Int) **returns**(r:IntSet)
ensures r eqSet (s $\cup$ {i})
op elem(s:Interval, i:Int) **returns**(b:Bool)
ensures b = (i $\in$ s)
op choose(s:Interval) **returns**(i:Int)
ensures i = leastElement(s)
op size(s:Interval) **returns**(i:Int)
ensures i = size(s)
op remove(s:Interval, i:Int) **returns**(r:IntSet)
ensures r eqSet delete(s,i)

Figure 10: The type specification `Interval`.

`choose` deterministic for `Interval` can be thought of as the record of a design decision. Similarly, one can think of non-determinism in a specification as leaving room for later design decisions, so subtypes should be allowed to be more deterministic.

The Larch/LOAL syntax for type specifications is given in Figure 11. The set of types that can be used in a program is specified in a ⟨type spec list⟩. In this paper we refer to different sets of type specifications by names.

Example 2.1 *The set of type specifications consisting of* `IntSet` *and* `Interval` *(Figures 7 and 10), is called* II. ∎

If we had not given a shorter name to II, we would denote it by "`IntSet + Interval`".

Each individual type specification has a ⟨header⟩ followed by specifications for each of the operations provided by the type. The nonterminal ⟨type⟩ represents type symbols, such as `IntSet`. The nonterminal ⟨identifier⟩ represents message names and other identifiers.

For convenience, the following syntactic sugars are defined. A declaration such as "f,s: Int" is syntactic sugar for the declaration list "f: Int, s: Int." An omitted pre-condition is syntactic sugar for "**requires** true".

In the header of a type specification the operations are divided into *class* and *instance* operations; this distinction is important for subtyping, since message passing only exercises an object's instance operations. The header of a type's specification includes two additional clauses: a **based on** clause, and an optional **subtype of** clause. The **based on** clause describes the abstract values of the objects of the type, by naming a sort and a Larch trait that specifies that sort. For example, the abstract values of objects of type `IntSet` are elements of the sort C, which is taken from the trait IntSetTrait.

⟨type spec list⟩ ::= ⟨type spec⟩ | ⟨type spec list⟩ ⟨type spec⟩
⟨type spec⟩ ::= ⟨type⟩ **immutable type** ⟨header⟩ ⟨op spec list⟩
⟨header⟩ ::= ⟨subtype list⟩ ⟨class ops⟩ ⟨instance ops⟩⟨basis⟩

⟨subtype list⟩ ::= ⟨empty⟩ | ⟨subtype clause⟩ ⟨subtype list⟩
⟨empty⟩ ::=
⟨subtype clause⟩ ::= **subtype of** ⟨type⟩ **by** ⟨simulation list⟩
⟨simulation list⟩ ::= ⟨simulation⟩ | ⟨simulation⟩ , ⟨simulation list⟩
⟨simulation⟩ ::= ⟨term⟩ **simulates** ⟨term⟩

⟨class ops⟩ ::= ⟨empty⟩ | **class ops** [⟨ident list⟩]
⟨instance ops⟩ ::= **instance ops** [⟨ident list⟩]
⟨ident list⟩ ::= ⟨identifier⟩ | ⟨ident list⟩ , ⟨identifier⟩

⟨basis⟩ ::= **based on sort** ⟨identifier⟩ **from** ⟨identifier⟩

⟨op spec list⟩ ::= ⟨op spec⟩ | ⟨op spec list⟩ ⟨op spec⟩

Figure 11: Syntax of Type Specifications. The syntax for ⟨op spec⟩ is given below.

2.4 Specifying the Subtype and Simulation Relations

In a type specification, the optional **subtype of** clauses describe the immediate supertype(s) of the specified type. The "subtype of" relationship among all type symbols is called the subtype relation and is written $\leq$. Formally, the relation $\leq$ specified by a set of type specifications is the reflexive, transitive closure of the **subtype of** relationships given in the type specifications. This is the subtype relation used in type-checking a LOAL program against the type specifications.

For each immediate supertype listed, the specification also states how each object of the subtype simulates at least one object of the supertype. For example, the specification of the type `Interval` states that `Interval` is a subtype of `IntSet`, and that an interval with value $[l, u]$ simulates an integer set with value toSet($[l, u]$), where the trait function "toSet" is described in Figure 5. The family of all such simulation relations, $\mathcal{R}$, indexed by the supertype and made reflexive, is called a simulation relation.

There is a relation $\mathcal{R}_{\mathtt{T}}$ for each type `T`. The relation $\mathcal{R}_{\mathtt{T}}$ says how the abstract values of objects of each type $\mathtt{S} \leq \mathtt{T}$ are to be viewed as objects of type `T`. For example, for each interval value $[l, u]$, the relationship $[l, u]\ \mathcal{R}_{\mathtt{IntSet}}$ toSet($[l, u]$) holds, as specified in `Interval`'s **subtype of** clause. By convention, the following additional relationships are implicit in such specifications. For each type `T`, the relation $\mathcal{R}_{\mathtt{T}}$ includes both the identity relation on the abstract values of objects of type `T` and all relations $\mathcal{R}_{\mathtt{S}}$ such that $\mathtt{S} \leq \mathtt{T}$; for example, the integer set $\{1\}$ is related by $\mathcal{R}_{\mathtt{IntSet}}$ to itself and $\mathcal{R}_{\mathtt{Interval}}$ relates the interval $[1, 2]$ to itself. Furthermore, the relationships compose transitively in the following sense: if $\mathtt{S} \leq \mathtt{T}$ and $a\ \mathcal{R}_{\mathtt{S}}\ b\ \mathcal{R}_{\mathtt{T}}\ c$, then $a\ \mathcal{R}_{\mathtt{T}}\ c$.

The family $\mathcal{R}$ is used to verify that $\leq$ has the necessary semantic properties to be a legal subtype relation, but does not affect the meaning of the specification. The relation

Interval **immutable type**
subtype of IntSet **by** $[l, u]$ **simulates** toSet($[l, u]$)
class ops [create]
instance ops [ins, elem, choose, size, remove]
based on sort C **from** IntervalTrait

op create(c:IntervalClass, lb,ub:Int) **returns**(i:Interval)
requires lb $\leq$ ub
ensures i eqSet [lb,ub]

op choose(s:Interval) **returns**(i:Int)
ensures i = leastElement(s)

Figure 12: The type specification `Interval`, inheriting operation specifications from `IntSet`.

$\leq$ can also be viewed as summarizing information about $\mathcal{R}$. That is, if `S` $\leq$ `T`, then legal subtyping requires that for every object of type `S`, its abstract value is related by $\mathcal{R}_{\mathtt{T}}$ to the abstract value of some object of type `T`, and that $\mathcal{R}$ has the semantic properties described in Section 3.3. Informally these properties require that the relationships be preserved by assertions and programs.

Inheritance of specifications by a subtype specification is a useful extension to a practical specification language. With operations specified by inheritance, one can specify a subtype by specifying only the subtype's class operations and those instance operations that are added by the subtype or that need to be further constrained. This is accomplished in Larch/LOAL by governing the behavior of each method by the most specific applicable operation specification that is consistent with the argument types (Definition 3.2). For example, the specifications of the `Interval` operations `ins`, `elem` and `size`, and `remove` are the same as their specification for `IntSet` arguments and can thus be omitted from the specification of `Interval`, as in Figure 12. Then, for example, when `ins` is passed an `Interval` argument, the specification from `IntSet` would apply. This is similar to copying the operation specification from the supertype and changing the argument types to the subtype.

2.5 Operation Specifications

The syntax of operation specifications is given in Figure 13. The terms used in the pre- and post-conditions of a type specification may use the trait functions from the traits named in the type specification's ⟨basis⟩, and the traits on which any type mentioned is based.

The terms used in the pre- and post-conditions of a type specification may use the trait functions from the traits named in the type specification's ⟨basis⟩, and the traits on which any type mentioned is based (such as `Int` and `Bool`). Arguments having a subtype of the specified type of a formal argument, are not mentioned explicitly, but are premitted whenever the corresponding supertype is mentioned. For example, if an operation were specified to take an `IntSet` as an argument, then the specification permits passing an `Interval`. The idea behind legal subtyping is to ensure that passing such subtypes is sensible.

⟨op spec⟩ ::= **op** ⟨nominal signature⟩ **requires** ⟨pre-condition⟩ **ensures** ⟨post-condition⟩

⟨nominal signature⟩ ::= ⟨identifier⟩ (⟨decl list⟩) **returns** (⟨decl⟩)
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩
⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩

⟨pre-condition⟩ ::= ⟨assertion⟩
⟨post-condition⟩ ::= ⟨assertion⟩
⟨assertion⟩ ::= ⟨term⟩

⟨term⟩ ::= **if** ⟨term⟩ **then** ⟨term⟩ **else** ⟨term⟩ | ⟨secondary⟩
⟨secondary⟩ ::= ⟨prefix operator⟩ ⟨secondary⟩ | ⟨secondary⟩ ⟨postfix operator⟩
    | ⟨secondary⟩ ⟨infix operator⟩ ⟨secondary⟩ | ⟨secondary⟩ = ⟨secondary⟩
    | ⟨mixfix operator start⟩ ⟨secondary⟩ ⟨mixfix continued⟩ | ⟨primary⟩
⟨mixfix continued⟩ ::= ⟨mixfix operator end⟩
    | ⟨mixfix operator continued⟩ ⟨secondary⟩ ⟨mixfix continued⟩
⟨primary⟩ ::= ⟨constant⟩ | ⟨identifier⟩ | (⟨term⟩)
    | ⟨trait function⟩ ( ) | ⟨trait function⟩ (⟨term list⟩)
⟨trait function⟩ ::= ⟨identifier⟩
⟨term list⟩ ::= ⟨term⟩ | ⟨term list⟩ , ⟨term⟩

Figure 13: Syntax of operation specifications and the concrete syntax of terms.

⟨term⟩ ::= ⟨identifier⟩ | ⟨trait function⟩ (⟨term list⟩) | ⟨trait function⟩ ( ) | ⟨term⟩ = ⟨term⟩

Figure 14: Abstract syntax of terms.

For convenience, the following syntactic sugars are defined. A declaration such as "f,s: Int" is syntactic sugar for the declaration list "f: Int, s: Int." An omitted pre-condition is syntactic sugar for "**requires** true".

An ⟨assertion⟩ is a boolean-valued term. A term is boolean-valued if its sort is boolean, as described in Section 3.2.1. The concrete syntax of terms is simplified from the Larch Shared Language [27]. Essentially, terms are as in the predicate calculus with equality, over the language of the traits.

In the concrete syntax, a term can be a ⟨constant⟩ ("27", "empty"), an identifier that names a formal argument or result of an operation ("x"), invocations of trait functions ("f(x, 27)"), invocations of prefix, infix, and postfix operators ("¬ isEmpty(s),", "lb ≤ ub", "p.first"), and invocations of mixfix operators ("**if** b **then** e1 **else** e2 **fi**"). Trait function symbols (such as ∪) are declared to be infix operators (etc.) in the **introduces** section of a trait. We consider the usual boolean connectives (∧, etc.) to be infix operators.

The concrete syntax of terms is unnecessarily complex for the semantic studies to follow. In these studies we use the abstract syntax for terms given in Figure 14. In this abstract syntax, **if then else** is considered a trait function; that is "**if** b **then** e_1 **else** e_2" is considered syntactic sugar for "**if__then__else__**(b,e_1,e_2)". Similarly, all prefix, infix, postfix, and mixfix forms are considered syntactic sugars. Furthermore, we consider all constants to be nullary trait functions; that is, a trait function such as "f" used in a ⟨term⟩ without arguments is syntactic sugar for "f()." This eliminates the special case for constants in proofs by induction on the structure of terms.

In a Larch/LOAL specification, the equals sign (=) can only be used between terms of visible sorts. The *visible sorts* are `Bool` and `Int`, both of which are built-in to LOAL. The need for this restriction on "=" is described in Section 3.2.2.

3 A Model Theory for Type Specifications

In this section algebraic models are defined and used to give a formal semantics to sets of type specifications. We also define simulation relations between algebraic models. Simulation relations are crucial to the definition of subtype relations.

3.1 Algebraic Models

Algebraic models are what a set of abstract data type specifications specifies. They are mathematical abstractions of the objects and procedures of the code that one would write to implement such type specifications. For brevity, we call algebraic models algebras.

Our algebras are an extension of the usual algebraic structures found in the study of equational logic or algebraic specifications [18]. As such, an algebra includes a carrier set and a set of *trait functions*; to these are added a set of *methods*. The trait functions are used to generate the carrier sets as specified in the used traits [26]; they are also used in the evaluation of the assertions used in specifications [64, Chapter 2]. The methods are used by LOAL programs for computation. The trait functions cannot be invoked by programs, and the methods cannot be used in specifications.

To model nondeterministic procedures, the methods are set-valued functions; that is, a method returns a set of the possible results [51] [52]. The special value $\perp$ is used to model procedure calls that do not halt or that encounter run-time errors. So a procedure that might either return 3 or never halt on some argument q would be modeled by a method that has $\{3, \perp\}$ as its set of possible results.

As in object-oriented programming languages, methods may be polymorphic. Message passing is thus modeled by simply invoking a method.

Although the trait functions in LSL are not polymorphic, our semantics of type specifications uses them as if they were. Thus we give a careful explanation of the dynamic overloading resolution used to give the illusion that trait functions are polymorphic.

In an algebra there is no separate representation for abstract values and objects. That is, the objects of a type are identified with their abstract values. This is adequate for immutable types, which are the only ones we consider.

3.1.1 Signatures

The algebras that satisfy a set of type specifications will all have the same syntactic interface to a LOAL program; this syntactic interface is called a signature.

Definition 3.1 (signature) *A* signature

$$\Sigma = (SORTS, TYPES, V, \leq, TFUNS, MN, ResSort)$$

consists of:

- *Sets of sort, type, and visible type symbols, such that* $SORTS \supseteq TYPES \supseteq V$ *and* V *is nonempty.*
- *A binary relation,* $\leq$*, which is a preorder on SORTS, such that for all visible types* $\mathtt{T} \in V$*, if* $\mathtt{S} \leq \mathtt{T}$ *then* $\mathtt{S} = \mathtt{T}$*.*

- *Disjoint sets, TFUNS of trait function symbols, and MN of message names. Their union, the set of all operation symbols, is denoted OPS:*

$$OPS \stackrel{\text{def}}{=} TFUNS \cup MN. \tag{1}$$

- *A partial function ResSort: $OPS, SORTS^* \rightarrow SORTS$, which returns an upper bound on the result sort of a trait function or message name applied to a tuple of arguments with the given sorts. ResSort must be monotone in the following sense: for all $\mathtt{g} \in OPS$, and for all tuples of sorts $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$, if $ResSort(\mathtt{g}, \vec{\mathtt{T}})$ is defined, then so is $ResSort(\mathtt{g}, \vec{\mathtt{S}})$, and furthermore $ResSort(\mathtt{g}, \vec{\mathtt{S}}) \leq ResSort(\mathtt{g}, \vec{\mathtt{T}})$ [53, Page 217].*

The restriction on $\leq$ prohibits subtypes of a visible type. This restriction is reasonable, because only visible types can appear as the output of programs, so an object of some other type cannot behave quite like an object of a visible type. It is not clear whether this restriction is absolutely necessary; we view it as a simplifying assumption.

3.1.2 Derivation of a Signature from a Set of Type Specifications

In this section we discuss how a set of type specifications determines a signature.

We use the set of type specifications II (which includes `IntSet` and `Interval`, see Example 2.1) as our example. We write $SIG(II)$ for the signature determined by II, and subscript each part of $SIG(II)$ by II.

For all signatures, the set of visible types, V, is as follows:

$$V_{\text{II}} = V \stackrel{\text{def}}{=} \{\mathtt{Bool}, \mathtt{Int}\}. \tag{2}$$

In general, the set of type symbols, $TYPES$, determined by a type specification consists of the visible types, the type symbols named at the beginning of each ⟨type spec⟩, and a *class type* for each of the types already mentioned, formed by adding "`Class`" as a suffix to each of the other type symbols. For example, the set of type symbols determined by II, $TYPES_{\text{II}}$, includes `IntSet`, `Interval`, `IntSetClass`, and `IntervalClass`, in addition to the visible types and their associated class types.

In general, the set of sorts, $SORTS$, and the set of trait function symbols, $TFUNS$, are determined by the traits referenced in the ⟨basis⟩ clauses of a specification and the traits included (recursively) in those traits, plus a trait of the following form for each class type `TClass`:

TClass: **trait**
  introduces T: → TClass
    eq: TClass, TClass → Bool
  asserts ∀ c_1,c_2: TClass
    c_1 == c_2
    c_1 eq c_2

(The first axiom says that all `TClass` values are equal, and the second says that all are "eq". The trait function "eq" is specified so that one may write subtype-constraining assertions about the `TClass` value.) The set $SORTS$ consists of all the sorts named in those traits, except that each sort name that follows the keywords **based on sort** in the specification of a type named `T` is renamed to `T` in the trait named following **from**. For example, the set of sorts determined by II, $SORTS_{\text{II}}$ is $TYPES_{\text{II}}$, as there are no auxiliary sorts. The set

TFUNS consists of all the trait function symbols mentioned in those traits. For example, the set $TFUNS_{\mathrm{II}}$ includes "IntSet"[3], "{}", "insert", "size", $\_\_ \in \_\_$, and "toSet", among others.

In general, the subtype relation, $\leq$, is the reflexive, transitive closure of the relationships mentioned in the ⟨subtype clause⟩s of each type specification. For example, II states that `Interval` is a subtype of `IntSet`. Hence `Interval` $\leq_{\mathrm{II}}$ `IntSet`. By taking the reflexive, transitive closure, the relationship `IntSet` $\leq_{\mathrm{II}}$ `IntSet` holds, as does `Bool` $\leq_{\mathrm{II}}$ `Bool`, and so on.

The set of message names *MN* of a set of type specifications consists of the symbols following **op** in ⟨op spec⟩s, all type symbols that are not class types (recall that these are nullary operations that return the class object for the type), and message names for the visible types. For example, the MN_{II} includes `null`, `create`, `ins`, `elem`, `IntSet`, `Interval`, `Bool`, `Int`, and message names for the visible types such as `true`, `false`, `not`, and `add`. (See [34, Appendix B] for details on the visible types.)

The requirement on signatures that the *ResSort* mapping is monotone in $\leq$ does not affect the construction of *ResSort*. However, if this requirement is not met, then the set of type specifications is invalid, as it will not determine a proper signature. Thus it is left to the designer to specify the trait functions and methods of subtypes so that signature restrictions are met. (Some automation of this task would be needed in practice.)

In general, the result sort mapping *ResSort* for trait function symbols is determined as follows. Each trait function symbol is introduced in a trait along with a signature (e.g., $\vec{\mathtt{S}} \rightarrow \mathtt{T}$). The mapping *ResSort* is simply another representation for this information. So, if the trait function symbol f is introduced with signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$, then *ResSort* maps the symbol f and tuple of sorts $\vec{\mathtt{S}}$ to `T`. For example, the following shows how $ResSort_{\mathrm{II}}$ works for the trait function symbols "IntSet", "{}", "insert", and the infix "$\in$".

$$
\begin{array}{lcl}
ResSort_{\mathrm{II}}(\mathrm{IntSet}, \langle\rangle) & = & \mathtt{IntSetClass} \\
ResSort_{\mathrm{II}}(\{\}, \langle\rangle) & = & \mathtt{IntSet} \\
ResSort_{\mathrm{II}}(\mathrm{insert}, \langle \mathtt{IntSet}, \mathtt{Int} \rangle) & = & \mathtt{IntSet} \\
ResSort_{\mathrm{II}}(\mathrm{insert}, \langle \mathtt{Interval}, \mathtt{Int} \rangle) & = & \mathtt{IntSet} \\
ResSort_{\mathrm{II}}(\_\_ \in \_\_, \langle \mathtt{Int}, \mathtt{IntSet} \rangle) & = & \mathtt{Bool} \\
ResSort_{\mathrm{II}}(\_\_ \in \_\_, \langle \mathtt{Int}, \mathtt{Interval} \rangle) & = & \mathtt{Bool}
\end{array}
$$

That is, "IntSet" denotes the abstract value of the class object, which has sort `IntSetClass`. Similarly, "{}" denotes an (empty) `IntSet`. The trait function "insert" can take an `IntSet` and an `Int` and return an `IntSet`, as well as taking an `Interval` and an `Int` and returning an `IntSet`. The infix trait function "$\in$" takes an `Int` and either an `IntSet` or an `Interval` and returns a `Bool`.

To define the *ResSort* map for message names we need some terminology to describe the types associated with an operation specification. A message name may be present in many different type specifications. This allows programmers to use message passing and subtype polymorphism. However, unlike the trait functions, the message names are not specified with all possible combinations of argument types. Each ⟨op spec⟩ for a given message name presents a different *nominal signature*, which is a pair consisting of a tuple of type symbols of the formal arguments and a type symbol for the result, written $\mathtt{S}_1, \ldots, \mathtt{S}_n \rightarrow \mathtt{T}$ or $\vec{\mathtt{S}} \rightarrow \mathtt{T}$ or $\rightarrow \mathtt{T}$ if there are no arguments. The nominal signature for a given ⟨op spec⟩ is formed

[3]Recall that each type name is also the name of a nullary trait function that returns the abstract value of the class object for the type

by placing an arrow between the list of the types in the arguments part of the operation signature and the type in the return part. For example, the nominal signature of `choose` in the type specification `IntSet` is $\texttt{IntSet} \rightarrow \texttt{Int}$, while the nominal signature of `choose` in the type specification `Interval` is $\texttt{Interval} \rightarrow \texttt{Int}$. Each type symbol, such as `IntSet`, is also implicitly specified as an operation (see Figure 8), and hence is also a nullary message name; its nominal signature is such that there are no arguments and the result type is the corresponding class type (which is the type of the class object for that type). For example, the nominal signature of the `IntSet` operation is: $\rightarrow \texttt{IntSetClass}$.

In general, the result sort map, *ResSort*, for message names is determined from the nominal signature of each $\langle$op spec$\rangle$, and the subtype relation $\leq$. This determination uses most specific applicable operation specification, as defined below. (In the definition, the subtype ordering $\leq$ is extended to tuples of types pointwise; that is, the formula $\vec{\texttt{S}} \leq \vec{\texttt{T}}$ means that for each i, $\texttt{S}_i \leq \texttt{T}_i$.)

Definition 3.2 (most specific applicable operation specification) *Let SPEC be a set of type specifications. Let* $\vec{\texttt{S}}$ *be a tuple of types.*

An operation specification with nominal signature $\vec{\texttt{T}} \rightarrow \texttt{U}$ *is the* most specific applicable operation specification for $\vec{\texttt{S}}$ *if and only if its tuple of formal arguments types,* $\vec{\texttt{T}}$*, is such that* $\vec{\texttt{S}} \leq \vec{\texttt{T}}$ *and it is the unique least, in the* $\leq$ *ordering, tuple of formal argument types for all operation specifications in SPEC with the same message name and number of arguments.*

For example, the specification of `choose` in the type specification `Interval` is the most applicable operation specification in II for $\langle$`Interval`$\rangle$. The specification of `choose` in the type specification `IntSet` is the most applicable operation specification in II for $\langle$`IntSet`$\rangle$.

For each message name `g`, and each tuple of sorts $\vec{\texttt{S}}$, $\mathit{ResSort}(\texttt{g}, \vec{\texttt{S}})$ is defined to be equal to `T` if and only if $\vec{\texttt{U}} \rightarrow \texttt{T}$ is the nominal signature of the most specific applicable operation specification for $\vec{\texttt{U}}$ whose operation symbol is `g`. Hence the unique operation specification with the most specific argument type requirements that apply to $\vec{\texttt{S}}$ determines the nominal result type. Furthermore, unless such a most specific applicable operation specification exists for a tuple of argument types, *ResSort* is not defined on that tuple of argument types.

For example, the following shows how $\mathit{ResSort}_{\mathrm{II}}$ acts on the message names `IntSet`, `Interval`, `null`, `create`, `ins`, and `choose`.

$$
\begin{array}{lcl}
\mathit{ResSort}_{\mathrm{II}}(\texttt{IntSet}, \langle\rangle) & = & \texttt{IntSetClass} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{Interval}, \langle\rangle) & = & \texttt{IntervalClass} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{null}, \langle\texttt{IntSetClass}\rangle) & = & \texttt{IntSet} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{create}, \langle\texttt{IntervalClass}, \texttt{Int}, \texttt{Int}\rangle) & = & \texttt{Interval} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{ins}, \langle\texttt{IntSet}, \texttt{Int}\rangle) & = & \texttt{IntSet} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{ins}, \langle\texttt{Interval}, \texttt{Int}\rangle) & = & \texttt{IntSet} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{choose}, \langle\texttt{IntSet}\rangle) & = & \texttt{Int} \\
\mathit{ResSort}_{\mathrm{II}}(\texttt{choose}, \langle\texttt{Interval}\rangle) & = & \texttt{Int}
\end{array}
$$

The use of *ResSort* for determining the result sort of a method allows the specification of binary operations where the code executed depends on the types of more than one argument. An example is given in [38].

3.1.3 Subsignatures

The notion of subsignature is used in the study of modular verification, and for technical purposes in this paper. When a new type specification is added to a set of type specifications, the old specification's signature is a subsignature of the new signature.

Definition 3.3 (subsignature) *We say that* Σ' *is a* subsignature *of* Σ *if* $SORTS' \subseteq SORTS$, $TYPES' \subseteq TYPES$, $V' \subseteq V$, $TFUNS' \subseteq TFUNS$, $MN' \subseteq MN$, $\leq'$ *is the restriction of* $\leq$ *to* $SORTS'$, $ResSort(OPS' \times (SORTS')^*) \subseteq SORTS'$, $ResSort'$ *is the restriction of* $ResSort$ *to* $OPS' \times (SORTS')^*$, *and for all sorts* $\mathtt{S}', \mathtt{T}' \in SORTS'$, *if there is a sort* $\mathtt{U}'$ *that is the least upper bound of* $\mathtt{S}'$ *and* $\mathtt{T}'$ *in* $\leq'$, *then for all sorts* $\mathtt{S}, \mathtt{T} \in SORTS$, *whenever* $\mathtt{S} \leq \mathtt{S}'$ *and* $\mathtt{T} \leq \mathtt{T}'$, *then the least upper bound of* $\mathtt{S}$ *and* $\mathtt{T}$ *exists and is a sort* $\mathtt{U} \leq \mathtt{U}'$.

For example, $SIG(\mathtt{IntSet})$ is a subsignature of $SIG(II)$.

The restriction about least upper bounds is necessary because in LOAL, the type of an expression **if** b **then** e_1 **else** e_2 **fi** is the least upper bound of the types of e_1 and e_2. When new types are added to a program, one needs to be sure the least upper bound still exists and is no larger than the original least upper bound. (See Lemma 5.2.)

3.1.4 Algebras

An algebra with signature Σ is called a Σ-algebra. In an algebra A, the interpretation of a message name $\mathtt{g}$ is written $\mathtt{g}^A$, and the interpretation of a trait function "f" with nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$ is written ${\mathtt{f}_{\vec{\mathtt{S}} \rightarrow \mathtt{T}}}^A$.

To allow recursive functions to be defined over algebras, we require that each "carrier set" be a flat domain and that each trait function and method be monotonic and continuous. (See, for example, [56] for definitions of these terms.) Since a method is a set-valued function, we need to define precisely what we mean by "monotonic" and "continuous" for set-valued functions.

We first extend the domain ordering to sets of possible results. Let $\sqsubseteq$ be the domain ordering on a carrier set. For sets of possible results, the ordering $\sqsubseteq_E$ is defined so that $Q \sqsubseteq_E R$ if for each $q \in Q$ there is some $r \in R$ such that $q \sqsubseteq r$ [4, Page 13].

Definition 3.4 (monotonic) *A set-valued function* g *is* monotonic *if and only if for all* $\vec{q}_1$, $\vec{q}_2$, *if* $\vec{q}_1 \sqsubseteq \vec{q}_2$, *then* $g(\vec{q}_1) \sqsubseteq_E g(\vec{q}_2)$.

That is, g is monotonic if whenever $\vec{q}_1 \sqsubseteq \vec{q}_2$ and $r_1 \in g(\vec{q}_1)$, then there is some $r_2 \in g(\vec{q}_2)$ such that $r_1 \sqsubseteq r_2$.

A continuous method will be defined as preserving least upper bounds of sequences. A *sequence in* $\sqsubseteq$ is a nonempty set $Q = \{q_i \mid i \in I\}$ indexed by some well-ordered set [23, Page12] I (whose elements are ordered by $\leq$) with the property that, if $i \leq j$, then $q_i \sqsubseteq q_j$.

Definition 3.5 (continuous) *A monotonic set-valued function* g *is* continuous *if and only if for every sequence in* $\sqsubseteq$, $Q = \{q_i\}$, *whenever* $R = \{r_i\}$ *is a sequence in* $\sqsubseteq$ *indexed by the same set as* Q *such that for all indexes* i, $r_i \in g(\vec{q}_i)$, *then* $\mathrm{lub}(R) \in g(\mathrm{lub}(Q))$.

The reader not familiar with denotational semantics is urged to ignore the parts of the following definition of algebras that refer to monotonicity and continuity. Such a reader can replace "flat domain" by "set" in the following.

Definition 3.6 (Σ-algebra) *Let Σ be a signature.*
A Σ-algebra, *$A = (|A|, TFUNS^A, MN^A)$, consists of:*

- *a* carrier set, *$|A|$, which is a SORTS-indexed family of flat domains: $|A| \stackrel{\text{def}}{=} \{A_{\mathtt{T}} \mid \mathtt{T} \in SORTS\}$, such that for each sort $\mathtt{T}$, $\perp \in A_{\mathtt{T}}$,*
- *a family of* trait functions, *$TFUNS^A$, which is a $(TFUNS \times SORTS^* \times SORTS)$-indexed family of monotonic and continuous functions such that for each $\mathrm{f} \in TFUNS$, for each tuple of sorts $\vec{\mathtt{S}}$, if there is some sort $\mathtt{T}$ such that $ResSort(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}$, then:*
  - *the trait function ${\mathrm{f}_{\vec{\mathtt{S}} \to \mathtt{T}}}^A$ has $\perp$ as its result if and only if at least one of its arguments is $\perp$,*
  - *for each tuple $\vec{q} \in A_{\vec{\mathtt{S}}}$, ${\mathrm{f}_{\vec{\mathtt{S}} \to \mathtt{T}}}^A(\vec{q}) \in A_{\mathtt{T}}$, and*
  - *for all tuples of sorts $\vec{\mathtt{U}}$, for all sorts $\mathtt{W}$, if $ResSort(\mathrm{f}, \vec{\mathtt{U}}) = \mathtt{W}$, then for all tuples $\vec{q} \in (A_{\vec{\mathtt{S}}} \cap A_{\vec{\mathtt{U}}})$, ${\mathrm{f}_{\vec{\mathtt{S}} \to \mathtt{T}}}^A(\vec{q}) = {\mathrm{f}_{\vec{\mathtt{U}} \to \mathtt{W}}}^A(\vec{q})$,*
- *a family of* methods, *MN^A, which is a MN-indexed family of monotonic and continuous set-valued functions such that for each $\mathtt{g} \in MN$, for each tuple of types $\vec{\mathtt{S}}$, and for each tuple $\vec{q} \in A_{\vec{\mathtt{S}}}$, if $ResSort(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{U}$, then:*
  - *$\mathtt{g}^A(\vec{q})$ is a nonempty set, and*
  - *for each $r \in \mathtt{g}^A(\vec{q})$, there is some type $\mathtt{T} \leq \mathtt{U}$ such that $r \in A_{\mathtt{T}}$.*

We now make some general remarks about the definition of algebras, define our notion of dynamic overloading for trait functions, and then give an example algebra.

Because trait functions only return $\perp$ when one of their arguments is $\perp$ and vice versa, we can think of $\perp$ as added to the set of abstract values defined in the traits. An element of a carrier set that is not $\perp$ is a *proper* element.

To allow more intuitive discussions, we use the following phrases in a stylized way. The phrase "q has sort $\mathtt{T}$" means that $q \in A_{\mathtt{T}}$; furthermore, if $\mathtt{T}$ is also a type, the phrases "q has type $\mathtt{T}$" and "q is an instance of type $\mathtt{T}$" also mean $q \in A_{\mathtt{T}}$.

To be unambiguous, we need to describe more of our vector notation for tuples. Tuples can be zero-length. The tuple $\vec{q}$ has type $\vec{\mathtt{S}}$, written $\vec{q} \in A_{\vec{\mathtt{S}}}$, if each q_i has type $\mathtt{S}_i$. The notation $A_{\vec{\mathtt{S}}}$ stands for $A_{\mathtt{S}_1} \times \cdots \times A_{\mathtt{S}_n}$, and $A_{\langle\rangle} \stackrel{\text{def}}{=} \{\langle\rangle\}$.

The last restriction on trait functions in the definition of an algebra ensures that we can think of each trait function symbol, such as "insert", as interpreted by a polymorphic function. In an algebra, the trait functions are not polymorphic. However, for our semantics of specifications, we need to do dynamic overloading resolution for trait functions. For example, given a term such as "insert(`s`, `i`)", we need to know what it means, even though its meaning depends on the sort of the value of `s`. There is no problem in determining which trait function to call if the carrier sets of each sort, such as `Interval` and `IntSet` are disjoint (ignoring $\perp$). However, we also permit models where the carrier sets are not disjoint. Such models have been used by Cardelli and others to discuss subtyping [8] [6] [5] [21]. In such a model, when one is given a tuple of arguments to a trait function $\vec{q}$, there may be no unique tuple of sorts $\vec{\mathtt{S}}$ such that $\vec{q} \in A_{\vec{\mathtt{S}}}$. So the last requirement on trait functions says that the algebra must be such that if $\vec{q}$ has more than one tuple of sorts, then the algebra must be arranged so that invoking the trait function associated with any of the tuples of sorts that $\vec{q}$ has gives the same result. So in particular, if $\mathtt{U} \leq \mathtt{S}$, $A_{\mathtt{U}} \subseteq A_{\mathtt{S}}$, $ResSort(\mathrm{f}, \langle \mathtt{S} \rangle) = \mathtt{T}$,

ResSort$(\mathrm{f}, \langle \mathtt{U} \rangle) = \mathtt{W}$, and $q \in A_{\mathtt{U}}$, then we can think of $\mathrm{f}^A(q)$ as ${\mathrm{f}_{\mathtt{S}\to\mathtt{T}}}^A(q) = {\mathrm{f}_{\mathtt{U}\to\mathtt{W}}}^A(q)$. In this case it must be that the range of ${\mathrm{f}_{\mathtt{U}\to\mathtt{W}}}^A$ is a subset of $A_{\mathtt{W}} \cap A_{\mathtt{T}}$, and so the result has both sort `W` and sort `T`. Therefore, by this restriction on trait functions, we can write $\mathrm{f}^A(\vec{q})$ without ambiguity.

Definition 3.7 (dynamic overloading of trait functions) *Let $\vec{\mathtt{S}}$ be a tuple of sorts. If $\vec{q} \in A_{\vec{\mathtt{S}}}$ and* *ResSort*$(\mathrm{f}, \vec{\mathtt{S}})$ *is defined and equals* `T`*, then* $\mathrm{f}^A(\vec{q}) \stackrel{\text{def}}{=} {\mathrm{f}_{\vec{\mathtt{S}}\to\mathtt{T}}}^A(\vec{q})$.

The return type of a method is more loosely constrained than the return sort of a trait function. That is, a method can return an element of some other carrier set than the one predicted by *ResSort*.

Another difference from the trait functions is that the methods of algebras can be non-strict. An operation is *strict* if whenever one of its arguments is $\perp$, then the only possible result is $\perp$. Non-strict methods are useful for modeling types with lazy evaluation, such as streams.

As an example, we describe some parts of a $SIG(II)$-algebra, A^{II}. The carrier sets of the visible types are fixed by convention, for example, ${A^{\mathrm{II}}}_{\mathtt{Bool}} = \{\perp, \mathit{true}, \mathit{false}\}$ and ${A^{\mathrm{II}}}_{\mathtt{Int}} = \{\perp, 0, 1, \Leftrightarrow 1, 2, \Leftrightarrow 2, \ldots\}$. The carrier sets of the non-visible types are given in Figure 15. These carrier sets are disjoint. The trait functions are largely determined by the traits, so only a few examples are presented in Figure 15. On the other hand, we present all of the methods associated with the non-visible types in Figure 15; we use trait functions in some of the definitions of the methods in A^{II}. We omit cases where the arguments are $\perp$, but for these cases the only possible result is $\perp$.

The `choose` operation of A^{II} is defined on nonempty arguments of type `IntSet` to have the entire set argument as its set of possible results, but is deterministic for arguments of type `Interval`. Also, if `choose` is applied to an empty `IntSet`, then the possible results are the entire carrier set of `Int`, including $\perp$. The method $\mathtt{ins}^{A^{\mathrm{II}}}$ has an interval as its only possible result whenever it can. In contrast, $\mathtt{remove}^{A^{\mathrm{II}}}$ only has sets as possible results.

3.1.5 Visible Types are the Same in All Algebras

Sets of type specifications written in Larch/LOAL all have the same set of visible (i.e., built-in) types. Thus we will assume from now on that the visible types are the same in all algebras. To state this assumption precisely requires the notion of the reduct of an algebra [18, Section 6.8] [34]. Briefly, the reduct $A_{(\Sigma')}$ has as its carrier sets the carrier sets of the sorts in A that appear in Σ', and as its trait functions and methods those named in Σ'.

For Larch/LOAL, there is a fixed signature ΣB and a fixed ΣB-algebra, B, that defines the visible types [34, Appendix B]. We assume that all signatures have ΣB as a subsignature and all algebras have B as their ΣB-reduct.

3.2 Formal Semantics of Type Specifications

In this section we formalize the semantics of sets of type specifications. We first formalize the static semantic constraints: sort-checking and the notion of subtype-constraining assertions. Then we describe the usual model-theoretic concepts of evaluation of assertions and validity.

Carrier Sets for the Non-visible Types

$$
\begin{array}{lcl}
A^{\mathrm{II}}{}_{\texttt{IntSet}} & \stackrel{\text{def}}{=} & \{\bot\} \cup \text{“finite sets of proper elements of } A^{\mathrm{II}}{}_{\texttt{Int}}\text{”} \\
A^{\mathrm{II}}{}_{\texttt{IntSetClass}} & \stackrel{\text{def}}{=} & \{\bot, \mathit{IntSet}\} \\
A^{\mathrm{II}}{}_{\texttt{Interval}} & \stackrel{\text{def}}{=} & \{\bot\} \cup \{[x,y] \mid x,y \in A^{\mathrm{II}}{}_{\texttt{Int}}, x \neq \bot, y \neq \bot, x \leq y\} \\
A^{\mathrm{II}}{}_{\texttt{IntervalClass}} & \stackrel{\text{def}}{=} & \{\bot, \mathit{Interval}\}
\end{array}
$$

A sampling of Trait Functions

$$
\begin{array}{lcl}
\mathrm{IntSet}_{\rightarrow\texttt{IntSetClass}}{}^{A^{\mathrm{II}}}() & \stackrel{\text{def}}{=} & \mathit{IntSet} \\
\mathrm{insert}_{\texttt{IntSet,Int}\rightarrow\texttt{IntSet}}{}^{A^{\mathrm{II}}}(\{i_1,\ldots,i_n\}, i) & \stackrel{\text{def}}{=} & \{i_1,\ldots,i_n\} \cup \{i\}, \text{ for } n \geq 0 \\
\mathrm{insert}_{\texttt{Interval,Int}\rightarrow\texttt{IntSet}}{}^{A^{\mathrm{II}}}([x,y], i) & \stackrel{\text{def}}{=} & \mathrm{toSet}^{A^{\mathrm{II}}}([x,y]) \cup \{i\} \\
[\_\_,\_\_]_{\texttt{Int,Int}\rightarrow\texttt{Interval}}{}^{A^{\mathrm{II}}}(x,y) & \stackrel{\text{def}}{=} & \begin{cases} [x,y] & \text{if } x \leq y \\ [x,x] & \text{if } x > y \end{cases} \\
\mathrm{toSet}_{\texttt{Interval}\rightarrow\texttt{IntSet}}{}^{A^{\mathrm{II}}}([x,y]) & \stackrel{\text{def}}{=} & \begin{cases} \{x, x+1, \ldots, y\} & \text{if } x < y \\ \{x\} & \text{otherwise} \end{cases}
\end{array}
$$

Methods for the Non-visible Types

$$
\begin{array}{lcl}
\texttt{IntSet}^{A^{\mathrm{II}}}() & \stackrel{\text{def}}{=} & \{\mathit{IntSet}\} \\
\texttt{Interval}^{A^{\mathrm{II}}}() & \stackrel{\text{def}}{=} & \{\mathit{Interval}\} \\
\texttt{null}^{A^{\mathrm{II}}}(\mathit{IntSet}) & \stackrel{\text{def}}{=} & \{\{\}\} \\
\texttt{create}^{A^{\mathrm{II}}}(\mathit{Interval}, x, y) & \stackrel{\text{def}}{=} & \{[\_,\_]^{A^{\mathrm{II}}}(x,y)\} \\
\texttt{ins}^{A^{\mathrm{II}}}(s,i) & \stackrel{\text{def}}{=} & \begin{cases} \{[x,y]\} & \text{if } s=[x,y],\ x \leq i \leq y \\ \{[i,y]\} & \text{if } s=[x,y],\ i = x \Leftrightarrow 1 \\ \{[x,i]\} & \text{if } s=[x,y],\ i = y+1 \\ \{\mathrm{insert}^{A^{\mathrm{II}}}(s,i)\} & \text{otherwise.} \end{cases} \\
\texttt{elem}^{A^{\mathrm{II}}}(s,i) & \stackrel{\text{def}}{=} & \{\_ \in \_\_^{A^{\mathrm{II}}}(s,i)\} \\
\texttt{choose}^{A^{\mathrm{II}}}(s) & \stackrel{\text{def}}{=} & \begin{cases} \texttt{Int}^{A^{\mathrm{II}}} & \text{if } s \in A^{\mathrm{II}}{}_{\texttt{IntSet}},\ s = \{\} \\ s & \text{if } s \in A^{\mathrm{II}}{}_{\texttt{IntSet}},\ s \neq \{\} \\ \{x\} & \text{if } s \in A^{\mathrm{II}}{}_{\texttt{Interval}},\ s = [x,y] \end{cases} \\
\texttt{size}^{A^{\mathrm{II}}}(s) & \stackrel{\text{def}}{=} & \{\mathrm{size}^{A^{\mathrm{II}}}(s)\} \\
\texttt{remove}^{A^{\mathrm{II}}}(s,i) & \stackrel{\text{def}}{=} & \{\mathrm{delete}^{A^{\mathrm{II}}}(s,i)\}
\end{array}
$$

Figure 15: An algebra A^{II} for the specification II.

$$[\text{ident}] \quad \Sigma; H, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}$$

$$[\text{tfinvoc}] \quad \frac{\Sigma; H \vdash \vec{E} : \vec{\mathtt{S}},\ \Sigma \vdash \mathit{ResSort}(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}}{\Sigma; H \vdash \mathrm{f}(\vec{E}) : \mathtt{T}}$$

$$[=] \quad \frac{\Sigma; H \vdash E_2 : \mathtt{T},\ \Sigma; H \vdash E_3 : \mathtt{T}}{\Sigma; H \vdash E_1 \ = \ E_2 : \mathtt{Bool}}$$

Figure 16: Sort Inference Rules for Larch/LOAL terms.

3.2.1 Sort-Checking of Assertions

Sort checking for terms depends on a signature, Σ, and a sort-environment, H. The signature gives the sorts of constants and trait functions (through *ResSort*). The sort environment gives the nominal sort of each identifier declared in a surrounding ⟨op spec⟩ (as a formal argument or formal result). The nominal sort of a trait function application of the form $\mathrm{f}(\vec{e})$ is T if the nominal sort of $\vec{e}$ is $\vec{\mathtt{S}}$, $\mathit{ResSort}(\mathrm{f}, \vec{\mathtt{S}})$ is defined, and $\mathit{ResSort}(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}$, otherwise the term does not sort-check. The nominal sort of an equation $e_1 = e_2$ is Bool if e_1 and e_2 have the same nominal sort, otherwise the equation does not sort-check.

Formal sort inference rules for sort-checking terms are given in Figure 16. The figure is based on the abstract syntax for terms (see Figure 14), and so does not make a special case for infix, etc. trait functions. A sort environment H can be thought of as a set of sort assumptions, which are the pairs of the mapping. An assumption of the form $\mathtt{x} : \mathtt{T}$ means that the identifier x has nominal sort T. The notation $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ means that each $\mathtt{x}_i$ has nominal sort $\mathtt{S}_i$. The notation $H, \mathtt{x} : \mathtt{T}$ means $H[\mathtt{T}/\mathtt{x}]$; that is, H extended with the assumption $\mathtt{x} : \mathtt{T}$ (where the extension replaces all assumptions about x in H). The notation $\Sigma; H \vdash E : \mathtt{T}$ means that given the signature Σ and the sort environment H one can prove that the expression E has nominal sort T using the inference rules. The notation $\Sigma \vdash \mathit{ResSort}(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}$ means that the *ResSort* mapping of Σ maps "f" and $\vec{\mathtt{S}}$ to the sort T. An inference rule of the form:

$$\frac{h_1, h_2}{c}$$

means that to prove the conclusion c one must first show that both hypotheses h_1 and h_2 hold. Rules written without hypotheses and the horizontal line are axioms.

Using the sort inference rules we can make the following definitions.

Definition 3.8 (Σ-term, Σ-assertion, nominal sort) *Let Σ be a signature. Let SORTS be the sorts of Σ.*

A ⟨term⟩ E is a Σ-term *if and only if there is some sort environment H and some sort $\mathtt{T} \in$ SORTS such that $\Sigma; H \vdash E : \mathtt{T}$.*

A Σ-term is a Σ-assertion *if and only if there is some sort environment H such that $\Sigma; H \vdash E : \mathtt{Bool}$.*

If $\Sigma; H \vdash E : \mathtt{T}$, then T is the nominal sort of the Σ-term E in H.

Usually the environment needed for determining the nominal sort of a term is determined from the surrounding declarations in the specification. We leave off the phrase "in H" when the sort environment is clear from context, as is always the case for assertions.

3.2.2 Subtype-Constraining Assertions

In Larch/LOAL specifications, equality (=) may only be used between terms of visible sort. We call such terms subtype-constraining.

Definition 3.9 (subtype-constraining) *Let Σ be a signature.*

A Σ-term is subtype-constraining *if and only if it uses "=" only between subterms whose nominal sort is a visible sort.*

For example, "$\mathtt{x} = 27$" is subtype-constraining, because `Int` is a visible sort, but "$\mathtt{s} = \{\}$" is not subtype-constraining.

We say "subtype-constraining" because a subtype-constraining term should mean approximately the same thing (see Section 6 for details) when some of its identifiers denote abstract values of subtypes of their nominal types. The following lemma looks at the sort-checking of terms and what happens when some identifiers in a term are replaced by identifiers whose types are subtypes of the types of the identifiers they replace. Such substitutions are used in the verification logic. The lemma states that the nominal sort of the substituted term is a subtype of the original term's nominal sort. For an assertion that is not subtype-constraining, such as "$\mathtt{s} = \{\}$", substituting "`iv`" of sort `Interval` for "`s`" produces "$\mathtt{iv} = \{\}$", which does not sort-check (because the sort of "$\{\}$" is `IntSet`). However, if in $E_1 = E_2$ the terms E_1 and E_2 have a visible sort, then there is no problem, because the only subtype of a visible sort is itself.

In the following, the notation $\{\vec{\mathtt{x}} : \vec{\mathtt{T}}\}$ means the set $\{\mathtt{x}_i : \mathtt{T}_i \mid i \text{ an index of } \vec{\mathtt{x}}\}$. The notation $Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ means the assertion Q with the $\mathtt{v}_i$ simultaneously substituted for free occurrences of the $\mathtt{x}_i$.

Lemma 3.10 *Let Σ be a signature. Let $\leq$ be the subtype relation of Σ. Let X be a set of identifiers containing $\{\vec{\mathtt{x}} : \vec{\mathtt{T}}\}$. Let Y be a set of identifiers containing $\{\vec{\mathtt{v}} : \vec{\mathtt{S}}\}$ such that $(Y \cup \{\vec{\mathtt{x}} : \vec{\mathtt{T}}\}) \supseteq X$. Let Q be a Σ-term with free identifiers from X. Let H be a sort environment containing the assumptions $\vec{\mathtt{x}} : \vec{\mathtt{T}}$ and $\vec{\mathtt{v}} : \vec{\mathtt{S}}$.*

Suppose Q is subtype-constraining and $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$. If $\Sigma; H \vdash Q : \mathtt{U}$, then there is some sort $\mathtt{V} \leq \mathtt{U}$ such that $\Sigma; H \vdash Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}] : \mathtt{V}$.

Proof Sketch: The proof is by induction on the structure of terms[4].

The basis is if that Q is an identifier. If Q is an identifier different from the $\mathtt{v}_i$, Q is the same as $Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$. If $\mathtt{y}$ is $\mathtt{x}_i$ for some i, then by definition of substitution, $Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ is $\mathtt{v}_i$. The result then follows by the hypothesis, that $\mathtt{S}_i \leq \mathtt{T}_i$.

For the inductive step, assume that the lemma holds for all subterms of Q. There are the following cases.

- Suppose Q has the form $\mathrm{f}(\vec{E})$, where "f" is a trait function. Since Q has nominal sort `U`, it must be that the tuple $\vec{E}$ has nominal sort $\vec{\tau}$ and $\mathit{ResSort}(\mathrm{f}, \vec{\tau}) = \mathtt{U}$. By the inductive hypothesis, the tuple $\vec{E}[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ has a nominal sort $\vec{\sigma}$ such that $\vec{\sigma} \leq \vec{\tau}$. Since *ResSort* is monotonic, $\mathit{ResSort}(\mathrm{f}, \vec{\sigma})$ is defined. By the sort inference rule [tfinvoc], $\mathrm{f}(\vec{E})[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ has nominal sort $\mathit{ResSort}(\mathrm{f}, \vec{\sigma})$. By the monotonicity of *ResSort*, $\mathit{ResSort}(\mathrm{f}, \vec{\sigma}) \leq \mathit{ResSort}(\mathrm{f}, \vec{\tau}) = \mathtt{U}$.

[4] This means "induction on the abstract syntax of terms", since we shall never again refer to their concrete syntax.

- Suppose Q has the form $E_1 = E_2$. Then the nominal sort of Q is `Bool`. Since Q is subtype-constraining, both E_1 and E_2 have visible sorts. By the inductive hypothesis, the nominal sort of $E_1[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ is a subtype of the nominal sort of E_1. Since the only subtype of a visible sort is itself, the sorts of $E_1[\vec{\mathtt{x}}/\vec{\mathtt{v}}]$ and E_1 are the same. The same holds for E_2. So the nominal sorts are the same as the original sorts, and thus the same as each other, so by the sort inference rule [=] the nominal sort of $Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ is `Bool`. Since $\leq$ is reflexive, the result follows.

■

3.2.3 Evaluation of Assertions in the Presence of Subtyping

Informally, the meaning of an assertion is given by dynamic overloading of the trait functions, in a way that is similar to message-passing. This allows inheritance of specifications by subtypes. For example, we could specify `Interval` by omitting the operation specification of `elem`. Then to understand a message send such as `elem(iv,2)` one would use the specification of `elem` in Figure 7. If `iv` has nominal type `Interval` and abstract value "[1,3]" then because of the overloaded trait functions, a description of the result is obtained by substituting "[1,3]" for `s` and "2" for `i` in the post-condition of `elem`'s operation specification in Figure 7; this gives the assertion "b = (3 ∈ [1,3])." Since $\_ \in \_$ is defined for `Interval` abstract values, this assertion makes sense.

To formally define the meaning of an assertion, each identifier denotes some abstract value in the carrier set of an algebra. The mapping from identifiers to abstract values is called an environment. (Since the identifiers are either formal arguments or formal results, they have types, not just sorts.)

Definition 3.11 (Σ-environment) *Let Σ be a signature whose set of type symbols is TYPES and whose subtype relation is ≤. Let A be a Σ-algebra. Let X be a set of typed identifiers, whose types are in TYPES.*

Then a mapping $\eta: X \to |A|$ is a Σ-environment *if and only if for every type* $\mathtt{T} \in TYPES$ *and for every* $\mathtt{x} : \mathtt{T}$ *in X, $\eta(\mathtt{x})$ has a type* `S` *such that* $\mathtt{S} \leq \mathtt{T}$.

The most unusual feature of environments is that they may map identifiers of one type to values of another type. That is, an environment may map an identifier $\mathtt{x} : \mathtt{T}$ to an abstract value that has any subtype of `T`. When we want to emphasize this property for some environment η, we say that η *obeys* ≤. Standard environments, which can only map $\mathtt{x} : \mathtt{T}$ to a value of type `T`, are called "nominal", since the nominal type of `x` is the same as the type of its value.

Definition 3.12 (nominal Σ-environment) *Let Σ be a signature. A Σ-environment, $\eta: X \to |A|$, is a* nominal Σ-environment *if and only if for each* $\mathtt{x} : \mathtt{T} \in X$, *$\eta(\mathtt{x})$ has type* `T`.

An environment is *proper* if its range does not include ⊥. In standard semantics, what we call a proper and nominal environment is often called an assignment, when used to give meaning to terms.

If `x` has nominal type `T` and q has some subtype of `T`, then the following shorthand can be used for adding a binding to an environment:

$$\eta[q/\mathtt{x}] \stackrel{\text{def}}{=} \lambda l.\ \text{if } l = \mathtt{x} \text{ then } q \text{ else } \eta(l). \tag{3}$$

The denotation of an identifier forms one basis for the inductive definition of the evaluation of assertions. (The other basis is the meaning of a constant, which is directly interpreted by an algebra.) We inductively extend the environment to a map from terms to abstract values in the standard way [18, Section 1.10]. The notation $\overline{\eta}$ means the extension of the Σ-environment $\eta : X \to |A|$ to a mapping from Σ-terms to $|A|$. This extension uses dynamic overloading of the trait functions of the algebra in the environment's range when applying trait function symbols and uses the environment itself to evaluate free identifiers. Rather than repeat the standard definition we give examples. Recall that the use of dynamic overloading for trait functions is signalled by our notation $\mathtt{f}^A$. Suppose `s` has nominal sort `IntSet`, `i` : `Int`, $\eta(\mathtt{s}) = \{1,2,3\}$, $\eta(\mathtt{i}) = 1$, and for each i, $\_\_ \in \_\_^A(e_i, \{e_1, \ldots, e_n\}) = \mathit{true}$, then

$$\begin{aligned} \overline{\eta}[\![\mathtt{i} \in \mathtt{s}]\!] &= \_\_ \in \_\_^A(\eta(\mathtt{i}), \eta(\mathtt{s})) \\ &= \_\_ \in \_\_^A(1, \{1,2,3\}) \\ &= \mathit{true}. \end{aligned}$$

Recall that all trait functions are strict, and that terms cannot contain quantifiers. The equals sign, "=", is interpreted in the standard way as equality in the carrier set of the algebra. An equation evaluates to either *true* or *false* (not $\perp$).

$$\overline{\eta}[\![E_1 = E_2]\!] \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } \overline{\eta}[\![E_1]\!] = \overline{\eta}[\![E_2]\!] \\ \mathit{false} & \text{otherwise.} \end{cases} \tag{4}$$

Thus if both E_1 and E_2 are $\perp$, then $\overline{\eta}[\![E_1 = E_2]\!] = \mathit{true}$. If E_1 and E_2 have different sorts, they may easily be either not equal or equal in different algebras. For example, in an algebra where the carrier set for each sort is disjoint from the other carrier sets, $\overline{\eta}[\![[3,3] = \{3\}]\!]$ would be *false*. In an algebra where the carrier set `Interval` is a subset of the carrier set for `IntSet`, $\overline{\eta}[\![[3,3] = \{3\}]\!]$ might be *true*. Thus the formula "[3,3] = {3}" is not valid in all algebras that satisfy the set of type specifications II; this is one reason that Larch/LOAL specifications are restricted to subtype-constraining assertions.

In evaluating a term, one might worry that, since an environment need not be nominal, a trait function might be applied outside its domain. That cannot happen, however, by the definition of the nominal sort of a term and the monotonicity of *ResSort*. This is stated formally in the following lemma, whose proof is by induction on the structure of terms.

Lemma 3.13 *Let Σ be a signature, whose subtype relation is $\leq$. Let P be a Σ-term whose set of free identifiers is X. Let H be a sort-environment that contains X. Let A be a Σ-algebra. Let $\eta: X \to |A|$ be a Σ-environment.*

If $\Sigma; H \vdash P$: `T`, then $\overline{\eta}[\![P]\!]$ has some sort `S` $\leq$ `T`. ∎

We can now define validity for assertions. An algebra-environment pair models an assertion when the assertion is true in that environment.

Definition 3.14 (models) *Let Σ be a signature. Let P be a Σ-assertion whose set of free identifiers is X. Let A be a Σ-algebra. Let Y be a set of typed identifiers such that $X \subseteq Y$. Let $\eta: Y \to |C|$ be a Σ-environment.*

Then (A, η) models *P, written $(A, \eta) \models P$, if and only if $\overline{\eta}[\![P]\!] = \mathit{true}$.*

For example, consider the assertion "$1 \in \mathtt{s}$" and the algebra A^{II} of Figure 15. Let Y be the set $\{\mathtt{s} : \mathtt{IntSet}\}$. Let $\eta{:}Y \to |A^{\mathrm{II}}|$ be the environment such that $\eta(\mathtt{s}) = \{1,2,3\}$. Since $\overline{\eta}[\![1 \in \mathtt{s}]\!] = \mathit{true}$, $(A^{\mathrm{II}}, \eta) \models 1 \in \mathtt{s}$. Since $\overline{\eta}[\![4 \in \mathtt{s}]\!] = \mathit{false}$, (A^{II}, η) does not model "$4 \in \mathtt{s}$."

The above definition of "models" specializes to the standard definition [19] when the subtype relation is equality (=).

3.2.4 Satisfaction for Type Specifications

We give a "loose" semantics to type specifications. That is, the meaning of a set of type specifications is a family of algebras with the same signature. These algebras must satisfy the specification's traits and the specification of each operation.

The first part of the definition of satisfaction is checking that the trait functions of an algebra satisfy the traits of the specification. For this it is convenient to free the trait functions from the program operations; hence the following definition.

Definition 3.15 (trait structure) *The* trait structure *of an algebra A is formed from A by throwing out the methods, deleting $\perp$ from each carrier set and restricting the trait functions to these domains.*

(The trait functions are well-defined on carrier sets without $\perp$ because they are strict and their result is only $\perp$ when one of their arguments is $\perp$.)

There is a standard notion of when something like a trait structure satisfies a set of sentences in second-order logic [19], and the formal semantics of the Larch Shared Language (LSL) provides a translation into such sentences [27]. (The **generated by** and **partitioned by** constructs in LSL are translated into second-order sentences.) Hence the following.

Definition 3.16 (satisfies the traits) *An algebra A* satisfies the traits *of a specification if and only if the trait structure of A satisfies the second-order sentences that are the meaning of those traits.*

Satisfaction for the methods is defined using the most specific applicable operation specification for a given set of arguments (Definition 3.2). The method must satisfy this specification in the sense that if the arguments have the appropriate types and satisfy the pre-condition, then the operation must halt, and can only return results that have the appropriate type and satisfy the post-condition. Note that the most applicable operation specification governs the behavior of the method only for arguments that have types for which it is the most specific applicable operation specification. For example, the specification of `choose` for `IntSet` arguments does not govern the behavior of `choose` for `Interval` arguments, since there is a more specific operation specification for `Interval` arguments. The definition of legal subtyping, however, does force such a relationship, but we did not want to build it into Larch/LOAL. This allows us to give sets of type specifications for which the claimed subtype relation is not legal, but still have the specification be meaningful.

Definition 3.17 (satisfies for methods) *Let SPEC be a set of type specifications. Let* $\mathtt{g}$ *be a message name of SIG(SPEC). Let A be an algebra whose signature is SIG(SPEC).*

A method $\mathtt{g}^A$ satisfies the specification of $\mathtt{g}$ *in SPEC if and only if for all tuples of types* $\vec{\mathtt{S}}$*, if ResSort*$(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{T}$*, then the following condition is met. Suppose the form of the most specific applicable operation specification for* $\vec{\mathtt{S}}$ *in SPEC is:*

op *g(*$\vec{\mathtt{x}} : \vec{\mathtt{U}}$*)* **returns***(*$\mathtt{y} : \mathtt{W}$*)*

requires R
ensures Q.

Then for all proper $\vec{q} \in A_{\vec{\mathtt{S}}}$*, for all SIG(SPEC)-environments* $\eta\colon \{\vec{\mathtt{x}} : \vec{\mathtt{U}}\} \to |A|$ *such that* $\eta(\mathtt{x}_i) = q_i$*, if* $(A, \eta) \models R$*, then for all possible results* $r \in \mathtt{g}^A(\vec{q})$*:* $r \neq \bot$*,* $(A, \eta[r/\mathtt{y}]) \models Q$*, and there is some* $\mathtt{V} \in TYPES$ *such that* $r \in A_{\mathtt{V}}$ *and* $\mathtt{V} \leq \mathtt{T}$*. Furthermore, whenever some argument to the operation is* $\bot$*, then the only possible result is* $\bot$*.*

For example, the method defined by

$$\mathtt{elem}^A(s, i) \stackrel{\text{def}}{=} \{\_ \in \_\_^A(i, s)\} \tag{5}$$

$$\mathtt{elem}^A(\bot, i) \stackrel{\text{def}}{=} \{\bot\} \tag{6}$$

$$\mathtt{elem}^A(s, \bot) \stackrel{\text{def}}{=} \{\bot\} \tag{7}$$

$$\mathtt{elem}^A(\bot, \bot) \stackrel{\text{def}}{=} \{\bot\} \tag{8}$$

(where s and i are proper) satisfies the specification of `elem` in II.

A method may satisfy a specification by being more deterministic than required by the specification. For example one might have $\mathtt{choose}^C(\{1, 2\}) = \{1\}$ in an II-algebra, C; whereas in A^{II}, $\mathtt{choose}^{A^{\mathrm{II}}}(\{1, 2\}) = \{1, 2\}$. Similarly, a method may be more "defined" than required by the specification. For example, one might have $\mathtt{choose}^C(\{\}) = \{0\}$, as opposed to $\mathtt{choose}^{A^{\mathrm{II}}}$ which on an empty set could either not terminate or return any integer.

The nullary messages that name class objects are implicitly specified as follows:

op T() **returns**(y:TClass)
ensures y eq T.

A method $\mathtt{T}^A$ satisfies this specification if its only possible result is the class object for `T`.

We summarize satisfaction for algebras in the following definition.

Definition 3.18 (satisfaction for algebras, *SPEC*-**algebra)** *Let SPEC be a set of type specifications. Let A be a SIG(SPEC)-algebra.*

Then A satisfies *SPEC if and only if the trait structure of A satisfies the traits of SPEC, and for each message name* `g` *of SIG(SPEC),* $\mathtt{g}^A$ *satisfies the specification of* `g` *in SPEC.*

*An algebra A is a SPEC-*algebra *if and only if A satisfies SPEC.*

For example, the algebra A^{II} of Figure 15 satisfies the specification II. Thus A^{II} is an II-algebra.

The *semantics of a set of type specifications SPEC* is the set of all *SPEC*-algebras.

3.3 Simulation Relations

What does it mean for one object to "behave like" another object? This notion figures prominently in most intuitive definitions of subtyping, and also in our definition.

Algebraically, q behaves like r, written $q R r$, means that in all contexts $P(\cdot)$, $P(q) R P(r)$. We are concerned with user-visible behavior, and thus are most concerned with program contexts; that is, contexts P that have visible types. For visible contexts, we want $P(q)$ to equal $P(r)$, because anything other than equality would be a "visible" difference.

The above story is slightly complicated when we consider observing objects in a language, like LOAL, with subtyping. In a typed language with subtyping, the observations that can

be applied to an object depend on what type one assumes for the object. For example, one can apply more methods to a triple than to a pair, hence whether two triples behave like each other depends on whether one observes them as pairs or triples. Therefore, simulation relations are families of relations that have one binary relation per sort. At each sort $\mathtt{T}$, a simulation relation can relate elements of all sorts $\mathtt{S} \leq \mathtt{T}$. The following abbreviation will be used to describe the set of all such elements in an algebra A.

$$Below(A, \mathtt{T}) \stackrel{\text{def}}{=} \bigcup_{\mathtt{U} \leq \mathtt{T}} A_{\mathtt{U}} \tag{9}$$

If A is a Σ-algebra, then the preorder $\leq$ used in this abbreviation is the preorder on the sorts of Σ.

Our extension of homomorphic relations to nondeterministic algebras was inspired by [51]. However, to make the analogy to the deterministic case clearer, we would like to deemphasize the nondeterminism in our notation. So we use another abbreviation when comparing sets of possible results with a relation $\mathcal{R}_{\mathtt{T}}$. If Q and R are sets of possible results, then

$$Q \; \mathcal{R}_{\mathtt{T}} \; R \stackrel{\text{def}}{=} \forall (q \in Q) \exists (r \in R) \;\; q \; \mathcal{R}_{\mathtt{T}} \; r. \tag{10}$$

For example, this abbreviation allows Formula (12) to look the same as it would for deterministic algebras. This overloading of $\mathcal{R}_{\mathtt{T}}$ applies only to sets of possible results; it does not apply when relating individual results (which might nonetheless be sets in our examples).

Definition 3.19 (simulation relation) *Let Σ be a signature. Let C and A be Σ-algebras. A SORTS-indexed family $\mathcal{R} = \{\mathcal{R}_T \subseteq (Below(C,T) \times Below(A,T)) | T \in SORTS\}$ is a* Σ-simulation relation between C and A, *if and only if the following properties hold:*

Substitution: *for all sorts $\mathtt{T}$, for all tuples of sorts $\vec{\mathtt{S}}$, and for all tuples $\vec{q} \in Below(C, \vec{\mathtt{S}})$, and $\vec{r} \in Below(A, \vec{\mathtt{S}})$:*

- *for all trait function symbols $\mathrm{f} \in TFUNS$, such that $ResSort(\mathrm{f}, \vec{\mathtt{S}}) = \mathtt{T}$,*

$$(\vec{q} \; \mathcal{R}_{\vec{\mathtt{S}}} \; \vec{r}) \Rightarrow \mathrm{f}^C(\vec{q}) \; \mathcal{R}_{\mathtt{T}} \; \mathrm{f}^A(\vec{r}), \tag{11}$$

- *for all message names $\mathtt{g} \in MN$, such that $ResSort(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{T}$,*

$$(\vec{q} \; \mathcal{R}_{\vec{\mathtt{S}}} \; \vec{r}) \Rightarrow \mathtt{g}^C(\vec{q}) \; \mathcal{R}_{\mathtt{T}} \; \mathtt{g}^A(\vec{r}). \tag{12}$$

Subsorting: *for all sorts $\mathtt{S}$ and $\mathtt{T}$:*

$$(\mathtt{S} \leq \mathtt{T}) \Rightarrow (\mathcal{R}_{\mathtt{S}} \subseteq \mathcal{R}_{\mathtt{T}}). \tag{13}$$

Coercion: *for all sorts $\mathtt{S}$ and $\mathtt{T}$:*

$$(\mathtt{S} \leq \mathtt{T}) \Rightarrow (\forall (q \in C_{\mathtt{S}}) \exists (r \in A_{\mathtt{T}}) \; q \; \mathcal{R}_{\mathtt{T}} \; r) \tag{14}$$

Bistrict: *for each sort $\mathtt{T}$, $\bot \; \mathcal{R}_{\mathtt{T}} \; \bot$, and whenever $q \; \mathcal{R}_{\mathtt{T}} \; r$ and one of q or r is $\bot$, then so is the other.*

V-identical: *for each $\mathtt{T} \in SORTS$, if $q \; \mathcal{R}_{\mathtt{T}} \; r$ and either q or r has a visible type, then $q = r$; for each $v \in V$, $\mathcal{R}_v$ contains the identity relation on the carrier set of v (which is the same in both C and A).*

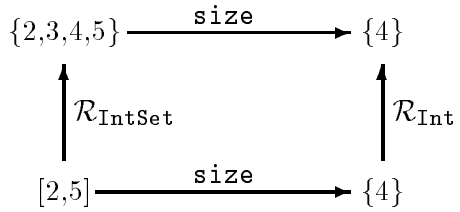

Figure 17: The substitution property for `size`.

The most important property is the "substitution" property, which says that simulation is preserved by both message-passing and by the trait functions. This property is often pictured in a commutative diagram, such as Figure 17, which depicts the following relationship.

$$\mathtt{size}([2,5]) \quad \mathcal{R}_{\mathtt{Int}} \quad \mathtt{size}(\{2,3,4,5\}) \tag{15}$$

In the "substitution" property, the notation $\vec{q}\,\mathcal{R}_{\vec{\mathtt{T}}}\,\vec{r}$ means that for each i, $q_i\,\mathcal{R}_{\mathtt{T}_i}\,r_i$ (assuming that $\vec{\mathtt{T}}$ is a tuple of types and $\vec{q}$ and $\vec{r}$ are tuples of objects the same length). The tuples $\vec{q}$ and $\vec{r}$ may be empty. Therefore, if $\mathcal{R}$ is a simulation relation between C and A, then for all nullary message names (i.e., type symbols) $\mathtt{T} \in MN$ and for all types `TClass` such that $\mathit{ResSort}(\mathtt{T}, \langle\rangle) = \mathtt{TClass}$,

$$\mathtt{T}^C() \,\mathcal{R}_{\mathtt{TClass}}\, \mathtt{T}^A() \tag{16}$$

The same holds for nullary trait function symbols.

The "subsorting" property is a technical requirement; it embodies the intuition that if one object simulates another at a subtype, then this simulation relationship should hold at each supertype, since no extra operations will be applicable at the supertype.

The "coercion" property is necessary for the soundness of verification. In verification one needs to relate, at each type `T`, each element of every subtype of `T` to some element of type `T`.

The "bistrict" property ensures that the meaning of $\perp$ is preserved. It is part of the definition of simulation relations because nontermination is (in some sense) visible.

The "V-identical" property has two parts. The first part says that distinct elements of the visible types cannot be related, at any type. (A visible type can be a subtype of some other type, but cannot have subtypes.) The second part says that, viewed at a visible type, identical elements of that type are related.

Example 3.20 *Let A^{II} be the algebra of Figure 15. Let $\mathcal{R}$ be the smallest bistrict sorted family of relations between A^{II} and A^{II} such that for all types `T`, if $q \in {A^{\mathrm{II}}}_{\mathtt{T}}$, then $q\,\mathcal{R}_{\mathtt{T}}\,q$, and for all proper $y \geq x$ in $A_{\mathtt{Int}}$,*

$$[x,y] \quad \mathcal{R}_{\mathtt{IntSet}} \quad \mathrm{toSet}^{A^{\mathrm{II}}}([x,y]) \tag{17}$$

$$[x,y] \quad \mathcal{R}_{\mathtt{IntSet}} \quad [x,y]. \tag{18}$$

This $\mathcal{R}$ is a simulation relation. ∎

Notice that $\mathcal{R}_{\texttt{IntSet}}$ is not symmetric, because an `IntSet` cannot be related to an `Interval`, because the `choose` operation is more nondeterministic on `IntSet` arguments. That is, if q is an `Interval`, then

$$\texttt{choose}^{A^{\mathrm{II}}}(q) = \{\texttt{leastElement}^{A^{\mathrm{II}}}(q)\}. \tag{19}$$

But for this algebra, if r is an `IntSet`,

$$\texttt{choose}^{A^{\mathrm{II}}}(r) = \{r_i \mid r_i \in r\}. \tag{20}$$

Having `choose` be more deterministic on `Interval` than on `IntSet` is desirable for several reasons. If one represents an `IntSet` by a linked list of integers, one might want to return the first element of the list as the result of `choose`. Since this has little to do with the abstract values, it will appear non-deterministic to clients. But it would be strange to specify `Interval` so that `choose` was *required* to be so non-deterministic. Indeed, making `choose` deterministic for `Interval` can be thought of as the record of a design decision. Similarly, one can think of non-determinism in a specification as leaving room for later design decisions, so subtypes should be allowed to be more deterministic.

Example 3.21 *It is not always possible to find a simulation between an algebra and itself. Consider an algebra C that is just like A^{II} of Figure 15, except that the* `choose` *operation is deterministic and returns the maximum element of a non-empty* `IntSet`*. Consider the relation $\mathcal{R}$ defined in example 3.20, with C substituted for A^{II}. This $\mathcal{R}$ is not a simulation relation between C and C, because $[1,3]\,\mathcal{R}_{\texttt{IntSet}}\,\{1,2,3\}$, but*

$$\texttt{choose}^{C}([1,3]) = \{1\} \tag{21}$$
$$\texttt{choose}^{C}(\{1,2,3\}) = \{3\} \tag{22}$$

and 1 is not related by $\mathcal{R}_{\texttt{Int}}$ to 3. Furthermore, no simulation relation exists between C and itself. ∎

There are two main theorems about simulation relations. The first is that simulation is preserved by LOAL expressions and programs, which is Theorem 5.16 below.

The other is that simulation is preserved by subtype-constraining terms. This theorem follows immediately, after a bit of notation.

To allow us to succinctly express that environments are related pointwise, we use the following convention. Given Σ-environments $\eta_B : X \to |B|$ and $\eta_A : X \to |A|$, the notation $\eta_B\,\mathcal{R}\,\eta_A$ means that for all sorts `T`, for all $\texttt{x} : \texttt{T} \in X$, $\eta_B(\texttt{x})\,\mathcal{R}_{\texttt{T}}\,\eta_A(\texttt{x})$.

Theorem 3.22 *Let Σ be a signature. Let C and A be Σ-algebras. Let X be a set of identifiers. Let* `S` *be a sort of Σ. Let Q be a Σ-term with free identifiers from X and nominal sort* `S`*.*

If Q is subtype-constraining, $\mathcal{R}$ is a Σ-simulation relation between C and A, and $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ are environments such that $\eta_C\,\mathcal{R}\,\eta_A$, then $\overline{\eta_C}[\![Q]\!]\,\mathcal{R}_{\texttt{S}}\,\overline{\eta_A}[\![Q]\!]$.

Proof: (by induction on the structure of terms).

By Lemma lemma-no-tf-app-outside-domain what we are trying to prove is well-defined.

For the basis there are two cases. If Q is an identifier $\texttt{x} : \texttt{S}$, then by the hypothesis $\eta_C(\texttt{x})\,\mathcal{R}_{\texttt{S}}\,\eta_A(\texttt{x})$. If Q is a nullary trait function, then the result follows by the substitution property for trait functions.

For the inductive step, assume that the result holds for all subterms of Q.

If Q has the form $\mathrm{f}(\vec{E})$, then by the inductive hypothesis, for each of the E_i, if the nominal sort of E_i is $\mathtt{S}_i$, then $\overline{\eta_C}[\![E_i]\!]\,\mathcal{R}_{\mathtt{S}_i}\,\overline{\eta_A}[\![E_i]\!]$. Thus the result follows by the substitution property for trait functions.

If Q has the form $E_1 = E_2$, then since Q is subtype-constraining, the nominal sort of E_1 is a visible sort, say $\mathtt{T}$. By the inductive hypothesis, for each of the E_i, $\overline{\eta_C}[\![E_i]\!]\,\mathcal{R}_{\mathtt{T}}\,\overline{\eta_A}[\![E_i]\!]$. Since $\mathtt{T}$ is visible, $\mathcal{R}_{\mathtt{T}}$ is the identity on $\mathtt{T}$. Since there can be no subtypes of a visible type, for each of the E_i, $\overline{\eta_C}[\![E_i]\!] = \overline{\eta_A}[\![E_i]\!]$. So if $\overline{\eta_C}[\![E_1]\!] = \overline{\eta_C}[\![E_2]\!]$, then $\overline{\eta_A}[\![E_1]\!] = \overline{\eta_A}[\![E_2]\!]$ and if $\overline{\eta_C}[\![E_1]\!] \neq \overline{\eta_C}[\![E_2]\!]$, then $\overline{\eta_A}[\![E_1]\!] \neq \overline{\eta_A}[\![E_2]\!]$. ∎

An important consequence of the above theorem is that if one environment simulates another, then the same set of subtype-constraining assertions is valid in each environment. This justifies reasoning about objects by using a simulation relation to coerce each abstract value to its nominal type. That is, one can always imagine that one is dealing with the abstract values of the nominal types (by use of an implicit simulation), and one is never misled about the value of a subtype-constraining assertion in such an environment. This is because the abstract values of the real objects simulate the imagined abstract values.

Lemma 3.23 *Let Σ be a signature. Let C and A be Σ-algebras. Let X be a set of identifiers. Let Q be a Σ-assertion with free identifiers from X.*

If Q is subtype-constraining, $\mathcal{R}$ is a Σ-simulation relation between C and A, and $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ are environments such that $\eta_C\,\mathcal{R}\,\eta_A$, then $(C, \eta_C) \models Q$ if and only if $(A, \eta_A) \models Q$. ∎

4 Legal Subtype Relations

4.1 Definition of Legal Subtype Relations

The semantic property that characterizes a legal subtype relation is the existence of a simulation relation. That is, informally, the reason `Interval` is a legal subtype of `IntSet` is that each instance of type `Interval` simulates some instance of `IntSet`. As pointed out by Example 3.21, the existence of such a simulation between an II-algebra and itself depends on how the `choose` operation is implemented in that algebra. So it is necessary to consider not just one algebra, but all algebras that satisfy a given specification. That is, whether a subtype relation holds or not depends on the semantics of a specification: a family of algebras. Considering the entire family of algebras in the definition of legal subtype relations allows our definition to work for incomplete specifications, such as II. The following is our formal definition of legal subtype relations for first-order, immutable, *abstract* types as characterized by a set of type specifications.

Definition 4.1 (legal subtype relation) *Let SPEC be a set of type specifications. Let* $\leq$ *be the subtype relation of SIG(SPEC).*

Then $\leq$ *is a* legal subtype relation on the types of *SPEC if and only if for all SPEC-algebras C, there is some SPEC-algebra A such that there is a SIG(SPEC)-simulation relation between C and A.*

We sometimes say that a subtype relation "is legal" instead of saying that it is a legal subtype relation.

4.2 Examples of Legal Subtype Relations

A trivial example of a legal subtype relation is the identity relation on types. When $\leq$ is the identity on types, every algebra simulates itself.

Example 4.2 *The subtype relation of the specification II is legal.*

The proof sketch below shows how we use the **simulates by** clauses in type specifications to prove a subtype relation is legal. This example also shows that a subtype, such as `Interval`, can be more defined and more deterministic than its supertype, `IntSet`. Recall that the result of applying `choose` to an empty `IntSet` is undefined and the possible results of applying `choose` to a non-empty set can be any element of the set. Recall also that the result of applying `choose` to an `Interval` is its least element.

Proof Sketch: Let C be an II-algebra. Let A be an algebra that is the same as C, except that its `choose` operation exhibits all the nondeterminism allowed by its specification. Then A is an II-algebra.

A simulation relation $\mathcal{R}'$ between C and A is constructed from the specification of II, as described in Section 2.4. This construction ensures that $\mathcal{R}'$ is such that, for all environments η_C over C and η_A over A:

$$(\eta_C(\mathtt{l})\ \mathcal{R}'_{\mathtt{Int}}\ \eta_A(\mathtt{l}) \wedge \eta_C(\mathtt{u})\ \mathcal{R}'_{\mathtt{Int}}\ \eta_A(\mathtt{u})) \Rightarrow \overline{\eta_C}[\![[\mathtt{l},\mathtt{u}]]\!]\ \mathcal{R}'_{\mathtt{IntSet}}\ \overline{\eta_A}[\![\mathrm{toSet}([\mathtt{l},\mathtt{u}])]\!]. \qquad (23)$$

It is easy to show that $\mathcal{R}'$ is a simulation relation. For example, to show the substitution property for the method `choose`, suppose $q\ \mathcal{R}'_{\mathtt{IntSet}}\ r$; then a possible result of $\mathtt{choose}^C(q)$ must be a possible result of $\mathtt{choose}^A(r)$, because q and r have the same elements and A's `choose` operation is maximally nondeterministic. ∎

Example 4.3 `IntSet` *cannot be a legal subtype of* `Interval`.

Proof Sketch: There are several reasons for this. To start with the most mundane, the specification II does not state that `IntSet` is a subtype of `Interval`. Even if it did, the operations `ins` and `remove` in the specification `Interval` would have the wrong type. Even if those operations were deleted, the trait functions "insert", "delete", "$\cup$", and "$\cap$" would have the wrong signature to make *ResSort* satisfy the restrictions on signatures, and the trait functions "leastElement" and "greatestElement" would have to be defined for `IntSet` arguments in the trait `IntSetTrait`. However, imagine a specification II', where all these changes were made, it would still be impossible for `IntSet` to be a legal subtype of `Interval`.

To see that even for II', `IntSet` cannot be a legal subtype of `Interval`, let C be a II'-algebra such that $\texttt{choose}^C$ has as its possible results each element of an argument of sort `IntSet`. Such algebras are allowed by the specification II'. For the sake of contradiction, suppose that there is some II'-algebra A, and some simulation relation $\mathcal{R}'$ from C to A.

One reason such a simulation relation cannot exist is that it would necessarily violate the coercion property of simulation relations. Let q be the abstract value denoted by the term "{}" in $C_{\texttt{IntSet}}$. By the coercion property, q must be related to some r in $A_{\texttt{Interval}}$. That is, $q\ \mathcal{R}'_{\texttt{Interval}}\ r$, for some r in $A_{\texttt{Interval}}$. Then by the substitution property

$$\text{isEmpty}^C(q)\ \mathcal{R}'_{\texttt{Bool}}\ \text{isEmpty}^A(r). \tag{24}$$

But $\text{isEmpty}^C(q) = \mathit{true}$ and $\text{isEmpty}^A(r) = \mathit{false}$, so this is a contradiction.

Another reason such a simulation relation cannot exist is that it would violate the substitution property of message names. That is so even if we change the specification of `IntSet` in II' so that there are no empty sets, because the `choose` operation of `IntSet` can be nondeterministic, while the `choose` operation of `Interval` is deterministic. To see this, let s_{23} be the abstract value denoted by "insert(insert({}, 2), 3)" in $C_{\texttt{IntSet}}$. By the coercion property and the assumption that $\texttt{IntSet} \leq_{II'} \texttt{Interval}$, there is some $i_{23} \in A_{\texttt{Interval}}$ such that $s_{23}\ \mathcal{R}'_{\texttt{Interval}}\ i_{23}$. Then by the substitution property, it would have to be true that

$$\texttt{choose}^C(s_{23})\ \mathcal{R}'_{\texttt{Int}}\ \texttt{choose}^A(i_{23}) \tag{25}$$

Since C is maximally nondeterministic,

$$\texttt{choose}^C(s_{23}) = \{2, 3\}. \tag{26}$$

That is, both 2 and 3 are possible results. However, by the definition of satisfaction for operations (Definition 3.17), if $\eta_A(\texttt{s}) = i_{23}$, then for each $r \in \texttt{choose}^A(i_{23})$,

$$(A, \eta_A[r/\texttt{i}]) \models \texttt{i} = \text{leastElement}(\texttt{s}). \tag{27}$$

Thus, by the specification of "leastElement", the only possible result is 2. By the V-identical property, $\mathcal{R}'_{\texttt{Int}}$ is the identity relation on the integers. So the possible result 3 of $\texttt{choose}^C(s_{23})$ is not related by $\mathcal{R}'_{\texttt{Int}}$ to any possible result of $\texttt{choose}^A(i_{23})$. Thus Formula 25 is false, and so $\mathcal{R}'$ cannot be a simulation relation. ∎

4.3 Discussion of Legal Subtype Relations

The definition of legal subtype relations says more than the intuition that "each object of the subtype simulates some object of the supertype." The deep problem with this intuition

op choose(s:Crowd) **returns**(i:Int)
requires ¬ isEmpty(s)
ensures i = choice(s)

Figure 18: Specification of the choose operation of the type `Crowd`.

CrowdTrait: **trait**
imports IntSetTrait(eqCrowd **for** eqSet)
introduces
choice: C → Int

Figure 19: The trait CrowdTrait.

is that it does not consider the behavior of two or more objects acting together. That is, each of two objects of a subtype may simulate some object of a supertype, but it may be that both of these supertype objects cannot appear in the same program at the same time. Algebraically, the problem is that for some specifications there is no "best" algebraic model.

To see this, consider a type `Crowd` that is the same as `IntSet`, except that its `choose` operation, when applied to a nonempty `Crowd` object, is required to be deterministic. That is, `choose` applied to the `Crowd` containing 2 and 3 must return either 2 or 3; the choice would have to be a function on the abstract values of `Crowd` objects, but the exact function could vary from algebra to algebra. This can be specified as in Figure 18. The post-condition says that the result must be the same as the trait function "choice" applied to the argument `s`. Since trait functions must be mathematical (i.e., deterministic) functions, there can only be one possible result. The specification of `Crowd` uses the trait CrowdTrait (Figure 19), which simply adds the signature of the trait function "choice" to IntSetTrait. Thus the abstract values of crowd objects are generated (by the trait functions "{}" and "insert"). This means, for example, that there cannot be two different abstract values that contain just the integers 2 and 3, so that `choose` must really be a function on abstract values.

Consider also another type that is like `IntSet`, the type `PSchd` of "priority schedulers." The priority scheduler has "jobs" represented by integers, and a `choose` operation that returns either the least or the greatest "job," depending on a "priority" that is set when the scheduler is created. The specification of `PSchd` is given in Figure 20. Abstractly, the priority is represented as a pair whose first element is a boolean, and whose second is a set of integers. The boolean "true" means that the `choose` operation should return the least element. Figure 22 gives the details, using the shorthand notation for traits in which trait functions that are not explicitly overloaded are defined by the coercion function "toCrowd."

Example 4.4 *The subtype relation of the specification* `Crowd + PSchd` *is not legal. That is,* `PSchd` *is not a legal subtype of* `Crowd`.

Proof Sketch: One might think that `PSchd` could be a legal subtype of `Crowd`, as the specification claims, because each `PSchd` simulates some `Crowd` object. However, a least-first `PSchd` object, one with abstract value "[true, {2,3}]" simulates a `Crowd` object with

PSchd **immutable type**
subtype of Crowd **by** c **simulates** toCrowd(c)
class ops [new] **instance ops** [ins, elem, choose, size, remove, leastFirst]
based on sort C **from** PSchdTrait
op new(c:PSchdClass, b:Bool) **returns**(p:PSchd)
ensures p eqPSchd [b, {}]
op ins(p:PSchd, i:Int) **returns**(r:PSchd)
ensures (r eqPSchd insert(p, i))
op elem(p:PSchd, i:Int) **returns**(b:Bool)
ensures b = i ∈ p
op choose(p:PSchd) **returns**(i:Int)
requires ¬ isEmpty(p)
ensures i ∈ p
∧ (p.first ⇒ lowerBound?(p.second,i))
∧ ((¬p.first) ⇒ upperBound?(p.second,i))
op size(p:PSchd) **returns**(i:Int)
ensures i = size(p)
op remove(p:PSchd, i:Int) **returns**(r:PSchd)
ensures (r eqPSchd delete(p,i))
op leastFirst(p:PSchd) **returns**(b:Bool)
ensures b = p.first

Figure 20: Specification of the priority scheduler type, `PSchd`.

OrderedIntSet: **trait**
imports IntSetTrait
assumes Ordered(Int)
introduces
lowerBound?, upperBound?: C, Int → Bool
leastElement, greatestElement: C → Int
asserts ∀ s: C, i, j: Int
lowerBound?({},i) == true
lowerBound?(insert(s,j),i) == (($i \leq j$) ∧ lowerBound?(s,i))
upperBound?({},i) == true
upperBound?(insert(s,j),i) == (($i \geq j$) ∧ upperBound?(s,i))
leastElement(insert(s,j)) ∈ insert(s,j)
lowerBound?(insert(s,j), leastElement(insert(s,j)))
greatestElement(insert(s,j)) ∈ insert(s,j)
upperBound?(insert(s,j), greatestElement(insert(s,j)))

Figure 21: The trait OrderedIntSet, included by PSchdTrait.

PSchdTrait: **trait**
subtrait of CrowdTrait(Crowd **for** C) **by** toCrowd **subsort** C **supersort** Crowd
imports OrderedIntSet(Crowd **for** C)
C **tuple of** first: Bool, second: Set
introduces toCrowd: C → Crowd
insert, delete: C,Int → C
∪, ∩: C,Crowd → C
∪, ∩: Crowd,C → C
__eqPSchd__: C,C → Bool
asserts ∀ p, p_2: C, s: Crowd, b: Bool, i, j: Int
toCrowd(p) == p.second
insert(p, i) == [p.first, insert(toCrowd(p), i)]
delete(p, i) == [p.first, delete(toCrowd(p), i)]
$(p \cup s)$ == ([p.first, toCrowd(p) ∪ s])
$(s \cup p)$ == ([p.first, s ∪ toCrowd(p)])
$(p \cap s)$ == ([p.first, toCrowd(p) ∩ s])
$(s \cap p)$ == ([p.first, s ∩ toCrowd(p)])
(p eqPSchd p_2) == ($p = p_2$)

Figure 22: The trait PSchdTrait, which includes the trait OrderedIntSet from Figure 21.

abstract value "{2,3}" in an algebra where `choose` returns the least element, 2. On the other hand, a greatest-first `PSchd` object, one with abstract value "[false, {2,3}]" simulates a `Crowd` object with abstract value "{2,3}", but in a necessarily different algebra—one where the `choose` operation returns the greatest element, 3. But by the definition of legal subtype relations, all the objects in a given model of `PSchd` would have to simulate `Crowd` objects in a single algebra. But to do so would violate the coercion property, as only one of "[true, {2,3}]" and "[false, {2,3}]" can simulate a crowd object in a given algebra. So there cannot be a simulation relation such that the method `choose` satisfies the substitution property. So `PSchd` cannot be a legal subtype of `Crowd` [34, Section 4.1.2]. ∎

The above example shows the failing of the informal motto, "`S` is a subtype of `T` if every object of type `S` acts like some object of type `T`," in the face of incomplete specifications. The type `Crowd` is incompletely specified, since the trait function "choice" and thus the exact behavior of `choose` is left to implementations. The informal motto would say that `PSchd` is a legal subtype of `Crowd`, but it is not. However, the reader may question whether it is our formal definition that is at fault instead of the informal motto.

We argue that taking the informal motto at face value leads to a less expressive notion of specification. The problem is that the specifier would not be able to say that all the types and subtypes involved in the program must, together, "act as a single implementation." That is, the reader of a specification should be able to assume that all the objects that can be used as objects of a given type, together satisfy the type specification.

We believe that when one reads a type specification, one takes the point of view of a client. That is, one implicitly assumes that everything in a program that is involved in implementing that specification, taken as a whole, satisfies that specification. If there are separate code modules, that makes no difference from the point of view of one reading the specification. If there are subtypes, this also makes no difference, as we wish to reason

based on the specification at hand, ignoring subtypes [38].[5] For example, this allows one to conclude that, because the `choose` operation is specified to be deterministic on `Crowd` abstract values, when one has two `Crowd` objects with abstract value "{2,3}", `choose` returns the same integer when applied to each object. Thus if a program observes that a `Crowd` object has two elements (by using `size`) and that it contains both 2 and 3 (by using `elem`) then one can conclude that its abstract value is "{2,3}", and expect it to behave accordingly.

Technically, the specifier states that there are not two distinct `Crowd` abstract values of size 2 containing 2 and 3 by specifying in the trait `CrowdTrait` that the abstract values of `Crowd` objects are generated[6]; i.e., there is "no junk" [7]. Our simulation relations preserve a specification of "no junk" because they must relate every element of a subtype to some element of each supertype *in one algebra*. If the specifier wants to allow junk, this can be done by not saying that the abstract values are generated, which allows other abstract values of the same sort to exist in a single algebra. So by restricting simulation so that it relates all subtype abstract values to supertype abstract values in a single algebraic model, our semantics allows either intention to be expressed. On the other hand, if the definition of legal subtype relations allowed abstract values to simulate others in different algebras, then the intention of "no junk" would be impossible to express.

The above example also shows the difference between our models, which allow disjoint carrier sets for subtypes, and models that require the carrier set of a supertype to include the carrier set of each of its subtypes [8] [21]. Such models by their very nature cannot enforce the restriction that a carrier set must be generated; that is, they always allow "junk" in carrier sets. (The "junk" elements are the abstract values of subtype objects.) By not allowing "junk" in carrier sets of types that are specified to be generated, we can have faithful models of incomplete specifications such as `Crowd`. Because we are forced to model such generated carrier sets without junk, we are forced to allow models where the carrier set of a subtype is not a subset of the carrier set for the supertype; that is, disjoint carrier sets seem necessary to faithfully model specifications where the carrier sets are specified to be generated.

Example 4.5 *Consider the type specification* `LFPSchd` *(in Figure 23) of least-first priority schedulers. The claimed subtype relationship of* `LFPSchd` + `Crowd`*, is not legal.*

Proof Sketch: We will show that the specification `LFPSchd` + `Crowd` does not have a legal subtype relation. To do this, we construct an algebra, C, where the operation `choose` returns the least element of a `LFPSchd` and the greatest element of a `Crowd` object; thus C cannot simulate any model of the specification. This construction might seem to be prohibited by PSchdTrait, which, because it is a subtrait of CrowdTrait, forces the trait function "choice" to do the same thing for a `LFPSchd` value and its image under the coercion function "toCrowd". The trait function "choice" must do the same thing, but because "choice" is not used in the specification of `LFPSchd`'s `choose` operation, the construction still produces a (`LFPSchd` + `Crowd`)-algebra.

[5] The implementor may have good reasons for wanting multiple code modules to be used in an implementation [14], as in Smalltalk where the Boolean is implemented by 3 classes [22]. Our point is that these techniques are of no more concern to a client than the data structures used to implement the specification [12].

[6] `CrowdTrait` specifies that the abstract values are generated because it includes the trait IntSetTrait, which includes the trait SetBasics from the LSL Handbook [26, Page 166], which has a **generated by** clause.

LFPSchd **immutable type**
subtype of Crowd **by** c **simulates** toCrowd(c)
class ops [new] **instance ops** [ins, elem, choose, size, remove, leastFirst]
based on sort C **from** PSchdTrait
op new(c:LFPSchdClass) **returns**(p:LFPSchd)
ensures p eqPSchd [true, {}]
op ins(p:LFPSchd, i:Int) **returns**(r:LFPSchd)
ensures (r eqPSchd insert(p, i))
op elem(p:LFPSchd, i:Int) **returns**(b:Bool)
ensures b = i ∈ p
op choose(p:LFPSchd) **returns**(i:Int)
requires ¬ isEmpty(p)
ensures i ∈ p ∧ lowerBound?(p.second,i)
op size(p:LFPSchd) **returns**(i:Int)
ensures i = size(p)
op remove(p:LFPSchd, i:Int) **returns**(r:LFPSchd)
ensures (r eqPSchd delete(p,i))
op leastFirst(p:LFPSchd) **returns**(b:Bool)
ensures b = true

Figure 23: Specification of the least-first priority scheduler type, `LFPSchd`.

Let C be a (`LFPSchd` + `Crowd`)-algebra, such that for all $l \in C_{\texttt{LFPSchd}}$, $\text{choice}^C(l) = \text{greatestElement}^C(l)$, and for all $c \in C_{\texttt{Crowd}}$, if both 2 and 3 are in c (using $\_\_ \in \_\_^C$), then $\text{choice}^C(c) = 3$. This algebra satisfies the set of type specifications `LFPSchd` + `Crowd`, because "choice(l) = choice(toCrowd(l))", and because there are no axioms governing the trait function "choice". Note that this implies that $\texttt{choose}^C$ returns the greatest element of a `Crowd` in C.

Suppose, for the sake of contradiction, that there is some (`LFPSchd` + `Crowd`)-algebra A and some simulation relation $\mathcal{R}'$ between C and A. Let $l \in C_{\texttt{LFPSchd}}$ be such that l has size 2, and both 2 and 3 are in l (as determined by $\_\_ \in \_\_^C$). Similarly let $c \in C_{\texttt{Crowd}}$ be such that c has size 2, and both 2 and 3 are in c.

By the coercion property, there is some $c' \in A_{\texttt{Crowd}}$, such that $c\ \mathcal{R}'_{\texttt{Crowd}}\ c'$ and some $l' \in A_{\texttt{Crowd}}$, such that $l\ \mathcal{R}'_{\texttt{Crowd}}\ l'$. So by the substitution property it must be that in A, both l' and c' contain 2 and 3 and have size 2. Since the abstract values of `Crowd` in an algebra must be generated (according to CrowdTrait), it must be that $c' = l'$. However, also by the substitution property:

$$\{2\} = \texttt{choose}^C(l) = \texttt{choose}^A(l') = \texttt{choose}^A(c') = \texttt{choose}^C(c) = \{\text{choice}^C(c)\} = \{3\} \quad (28)$$

which is a contradiction. ∎

Intuitively, the reason that `LFPSchd` is not a legal subtype of `Crowd` is that objects of these types may have the same elements (thus look the same as `Crowd`s), but differ in their response to the `choose` message. This can be observed in a program, and would be surprising to someone who was using supertype abstraction (i.e., using the specification of `Crowd`) to reason about the program. For the same reason, the **simulates** clause in Figure 23 does not give a simulation.

In view of the previous two examples we believe that a better informal motto for legal subtyping is: "subtyping means no surprises" [40] [38] [35] [36]. The idea behind this motto is that if one has a legal subtype relation, then a program cannot observe anything that would not be expected based on the specification of the supertypes. This is, we believe, the ultimate justification for a definition of "legal subtype relations". This justification is the purpose of the next two sections. The next section describes the programming language LOAL, and Section 6 describes Hoare-style verification for LOAL. In Section 6 we prove that supertype abstraction is a sound reasoning principle for LOAL, which justifies our definition of legal subtype relations.

Example	Kind of name
inBoth	LOAL function identifier
`choose`	message name (ADT operation name)

Table 1: Font conventions for names in LOAL programs.

5 An Applicative Language

In this section we define the programming language LOAL. The main purpose of LOAL is to have a vehicle to demonstrate supertype abstraction in program verification for an object-oriented language with message passing and subtyping. To do that, we need a programming language that can observe objects of immutable abstract types by message passing. We formally define the syntax and semantics of LOAL, show how to specify LOAL functions in Larch/LOAL, and show that simulation relations are preserved by LOAL programs.

LOAL is an extension of the simply-typed, applicative-order lambda calculus. LOAL is only half of an object-oriented language, since it does not have classes or other modules for implementing abstract data types. This is in accord with our purpose for LOAL, which is to manipulate the objects of existing abstract data types. Instead, a LOAL program is "linked" to an algebra and computes over that algebra; that is, identifiers in a LOAL program denote elements of the carrier set of an algebra, and when the program needs the set of possible results from a message send it consults that algebra. For example, in the LOAL functions of Figure 2, `s1` and `s2` denote elements of some algebra's carrier set, and `choose`, `remove`, and `elem` are evaluated by using the algebra's operations. So a LOAL program consists entirely of client code that manipulates some abstract data types, as modeled by an algebra.

5.1 LOAL Concrete Syntax

The concrete syntax of LOAL is given by both Figures 24 and 25. The syntax uses **fun** instead of λ. The nonterminal ⟨type⟩ denotes a type symbol, such as `IntSet`. The nonterminal ⟨message name⟩ denotes a message name, such as `choose`. The syntax of ⟨identifier⟩s, ⟨constant⟩s, and ⟨function identifier⟩s is left unspecified. A message name is the name of an operation of an abstract data type, which is used but not described in a LOAL program; in contrast, a function identifier comes from a recursive function definition given in the preamble of a LOAL program. We use a slanted font for function identifiers to distinguish them from message names, e.g., *inBoth* vs. `choose`, because function identifiers are statically bound to function denotations and message names are bound to the operations of an algebra, which models dynamic binding. Table 1 summarizes these conventions.

5.2 Informal Overview of LOAL Semantics

A program consists of a set of mutually recursive function definitions, the only part shown in Figure 2, followed by a ⟨prog expr⟩. The ⟨prog expr⟩ declares the program's inputs and the type of its result. A complete program is given in Figure 26. In this figure, f is a function identifier, and `choose` is a message name. The type of a program's result must be a *visible* type — either `Bool` or `Int`. The type of the result of the program in Figure 26 is `Int`. The types of a program's arguments need not be visible. So one may think of a LOAL

```
⟨program⟩ ::= ⟨prog expr⟩ | ⟨rec fun def⟩ ⟨program⟩
⟨prog expr⟩ ::= prog ( ⟨decls⟩ ) : ⟨type⟩ = ⟨expr⟩
⟨decls⟩ ::= ⟨decl list⟩ | ⟨empty⟩
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩
⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩
⟨empty⟩ ::=
⟨expr⟩ ::= ⟨identifier⟩ | bottom [ ⟨type⟩ ]
    | ⟨message name⟩ ( ⟨exprs⟩ ) | ⟨function identifier⟩ ( ⟨exprs⟩ )
    | ( ⟨function abstract⟩ ) ( ⟨exprs⟩ )
    | if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩ fi | isDef?  ( ⟨expr⟩ )
⟨exprs⟩ ::= ⟨expr list⟩ | ⟨empty⟩
⟨expr list⟩ ::= ⟨expr⟩ | ⟨expr list⟩ , ⟨expr⟩
⟨function abstract⟩ ::= fun ( ⟨decls⟩ ) ⟨expr⟩
⟨rec fun def⟩ ::= fun ⟨function identifier⟩ ( ⟨decl list⟩ ) : ⟨type⟩ = ⟨expr⟩ ;
```

Figure 24: Abstract syntax of the programming language LOAL.

```
⟨decl⟩ ::= ⟨identifier list⟩ : ⟨type⟩
⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩ , ⟨identifier⟩
⟨expr⟩ ::= ⟨constant⟩ | ⟨type⟩ | ( ⟨expr⟩ )
```

Figure 25: Additional concrete syntax for LOAL (syntactic sugars).

```
fun f(x:Int): Int = add(x,x);
prog (s:IntSet): Int = f(choose(s)).
```

Figure 26: A LOAL program. This program shows the difference between LOAL's lazy evaluation semantics and call by name.

program as an abstraction of the part of a "real" program that processes objects after they have been constructed from the "real" program's input.

Following Smalltalk-80, there are *class objects* that are used to represent types at run-time. Thus in the concrete syntax, a ⟨type⟩ is an ⟨expr⟩. For example, one would write `null(IntSet)` to create an empty `IntSet` object.

LOAL programs and functions may be nondeterministic, since the operations of an abstract type may be nondeterministic. Although there are no facilities in LOAL itself for introducing nondeterminism, the addition of such facilities does not invalidate the results of this paper [34].

LOAL uses lazy evaluation for evaluating function arguments [56, Page 181] [4]. Because of lazy evaluation, functions need not be strict. However, each actual parameter is only evaluated once; hence formal parameters are not sources of nondeterminism. That is, if a formal argument is mentioned twice in the body of a function abstract, the same value will be substituted in each instance. The program in Figure 26 demonstrates the difference between lazy evaluation and call-by-name. In LOAL, if that program is passed an `IntSet` with value {0,1}, then it has as possible results both 0 and 2; a result of 1 is not possible.

Since LOAL functions are non-strict, the primitive `isDef?` is provided to allow one to write strict functions. It has value *true* if its argument is defined, but has no result if evaluation of its argument does not terminate.

For a given type `T`, the expression `bottom[T]` is an infinite loop; it is used in giving a semantics to recursive functions. The expression `bottom[T]` has type `T`.

5.3 Syntactic Sugars and Abstract Syntax

To simplify the formal semantics of LOAL, we use an abstract syntax for LOAL that restricts both declaration lists and expressions. The abstract syntax for LOAL declaration lists and expressions is presented in Figure 24. In the abstract syntax, class objects are denoted by expressions of the form `IntSet()`; that is, we model the class objects with nullary message names, and the concrete syntax `null(IntSet)`, is considered sugar for the abstract syntax `null(IntSet())`.

Constants are also modeled with message names, which take a class object as an argument. For example, the concrete syntax's `true` is considered sugar for the abstract syntax `true(Bool())`.

Finally, a concrete syntax declaration such as `f,s: Int` is considered sugar for the abstract syntax declaration list `f: Int, s: Int`.

In the following we present LOAL examples in the concrete syntax, but we only consider the abstract syntax in the semantics and proofs.

5.4 Type Checking and Nominal Types

Type checking for LOAL is based on subtyping, using techniques from Reynolds's category sorted algebras [53] [54]. Each type-safe expression is statically assigned a "nominal type", determined from the information given in type specifications and program declarations. Thus the nominal type of an expression is just the expression's static type; for example, the nominal type of an identifier is given in its declaration. Nominal types play a crucial role in program verification.

An expression's nominal type is an upper bound on the types of the objects it can denote. That is, an expression with nominal type `T` can only denote an object whose type is a subtype

of `T`. For example, `ins(s,3)` has nominal type `IntSet` if `s` has nominal type `IntSet`, but the expression may return an `Interval` (when `s` denotes an `Interval` that contains 3 at run-time). The notion of an expression's nominal type is similar to Reynolds's notion of the minimal type of an expression [53] [54]. Like Reynolds, each type-safe expression is given a single nominal type. This is in contrast to type systems with a rule of subsumption, such as Cardelli's [8], where expressions have multiple types. As with Reynolds's system, the nominal type of an `if` expression is the least upper bound of the nominal types of the arms, if the least upper bound exists. In Reynolds's system, there is a "nonsense" type that is a supertype of all other types, but the LOAL type system has no such type.

To ensure that nominal types can be thought of as upper bounds and that message names defined on supertypes may be applied to subtypes, *ResSort* must be monotone as described in Definition 3.1. For example, since the message name `ins` is defined for an `IntSet` argument, it must also be defined for an `Interval` argument; furthermore, the nominal result sort must be a subtype of `IntSet`.

The nominal type of an expression is defined recursively. At the base, the nominal type of an identifier is given in its declaration. The nominal type of a function call is given by that function's declaration. To support subtype polymorphism, an actual argument may have a nominal type that is a subtype of the corresponding formal argument's nominal type, and thus the actual argument expressions may have nominal types that are subtypes of the corresponding formal argument types. Similarly, the body of a function may have a nominal type that is a subtype of the nominal result type of the function. The nominal type of a message send is determined by the *ResSort* function, applied to the nominal types determined (recursively) for the arguments.

Figure 27 shows the type inference rules for LOAL. These rules precisely define the nominal type of each LOAL expression. In the figure, H is a type environment that maps identifiers to types and function identifiers to nominal signatures. A type environment H can be thought of as a set of type assumptions, which are the pairs of the mapping. An assumption of the form `x : T` means that the identifier `x` has nominal type `T`. The same notational conventions are used as in Figure 16, with the following additions. An assumption of the form $f : \vec{\mathtt{S}} \rightarrow \mathtt{T}$ means that the function identifier f has nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. Note that the [prog] rule puts these assumptions for the program's system of function definitions in the type environment. The notation $\Sigma \vdash \mathrm{lub}(\mathtt{S},\mathtt{U}) = \mathtt{T}$ means that the least upper bound in Σ's presumed subtype relation, $\leq$, of `S` and `U` exists and is equal to `T`. The notation $\Sigma \vdash \vec{\sigma} \leq \vec{\mathtt{S}}$ means that for each i, $\sigma_i \leq \mathtt{S}_i$, where $\leq$ is the presumed subtype relation of Σ.

The only rules that allow one to exploit the presumed subtype relation $\leq$ are [mp], [fcall] and [comb]. These rules allow the nominal type of an actual argument expression to be a presumed subtype of the formal's type.

We can now make the following definitions formally.

Definition 5.1 (nominal type, type safe) *Let Σ be a signature. Let H be a type environment.*

A LOAL expression E has nominal type `T` with respect to Σ and H *if and only if* $\Sigma; H \vdash E : \mathtt{T}$.

The set of type safe LOAL expressions over Σ and H *is the set of all LOAL expressions E that have a nominal type with respect to Σ and H.*

These definitions also apply to LOAL programs, but without any need to mention a

[ident] $\Sigma; H, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}$

[fident] $\Sigma; H, f : \vec{\mathtt{S}} \to \mathtt{T} \vdash f : \vec{\mathtt{S}} \to \mathtt{T}$

[bot] $\Sigma; H \vdash \mathtt{bottom[T]} : \mathtt{T}$

[mp] $$\frac{\Sigma; H \vdash \vec{E} : \vec{\sigma},\ \mathit{ResSort}(\mathtt{g}, \vec{\sigma}) = \mathtt{T}}{\Sigma; H \vdash \mathtt{g}(\vec{E}) : \mathtt{T}}$$

[fcall] $$\frac{\Sigma; H \vdash f : \vec{\mathtt{S}} \to \mathtt{T},\ \Sigma; H \vdash \vec{E} : \vec{\sigma},\ \Sigma \vdash \vec{\sigma} \leq \vec{\mathtt{S}}}{\Sigma; H \vdash \mathtt{f}(\vec{E}) : \mathtt{T}}$$

[comb] $$\frac{H, \vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{T},\ \Sigma; H \vdash \vec{E} : \vec{\sigma},\ \Sigma \vdash \vec{\sigma} \leq \vec{\mathtt{S}}}{\Sigma; H \vdash (\mathtt{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (\vec{E}) : \mathtt{T}}$$

[if] $$\frac{\Sigma; H \vdash E_1 : \mathtt{Bool},\ \Sigma; H \vdash E_2 : \mathtt{S}_2,\ \Sigma; H \vdash E_3 : \mathtt{S}_3,\ \Sigma \vdash \mathrm{lub}(\mathtt{S}_2, \mathtt{S}_3) = \mathtt{T}}{\Sigma; H \vdash (\mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi}) : \mathtt{T}}$$

[isDef] $$\frac{\Sigma; H \vdash E : \mathtt{S}}{\Sigma; H \vdash \mathtt{isDef?}(E) : \mathtt{Bool}}$$

[prog] $$\frac{\begin{array}{c}\Sigma; f_1 : \vec{\mathtt{S}_1} \to \mathtt{T}_1, \ldots, f_m : \vec{\mathtt{S}_m} \to \mathtt{T}_m, \vec{\mathtt{x}_1} : \vec{\mathtt{S}_1} \vdash E_1 : \mathtt{T}_1, \\ \vdots \\ \Sigma; f_1 : \vec{\mathtt{S}_1} \to \mathtt{T}_1, \ldots, f_m : \vec{\mathtt{S}_m} \to \mathtt{T}_m, \vec{\mathtt{x}_m} : \vec{\mathtt{S}_m} \vdash E_m : \mathtt{T}_m, \\ \Sigma; \vec{\mathtt{y}} : \vec{\mathtt{U}}, f_1 : \vec{\mathtt{S}_1} \to \mathtt{T}_1, \ldots, f_m : \vec{\mathtt{S}_m} \to \mathtt{T}_m \vdash E : \mathtt{T}, \\ \mathtt{T} \in V\end{array}}{\Sigma \vdash \left(\begin{array}{l}\mathtt{fun}\ f_1\ (\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1\ \mathtt{=}\ E_1\mathtt{;} \\ \vdots \\ \mathtt{fun}\ f_m\ (\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m\ \mathtt{=}\ E_1\mathtt{;} \\ \mathtt{program}\ (\vec{\mathtt{y}} : \vec{\mathtt{U}}) : \mathtt{T}\ \mathtt{=}\ E\end{array}\right) : \mathtt{T}}$$

Figure 27: Type Inference Rules for LOAL

type environment.

5.5 Modularity of Type Checking

How is the nominal type of an expression affected by adding new types to a program? This question is important for modularity of program verification, since type checking is part of the verification process. Adding new types may change the nominal types of expression, but if the new types are added in such a way as to make the original signature a subsignature of the new signature, the new nominal types will only be subtypes of the original nominal types. When the new nominal type is guaranteed to be a subtype of the original nominal type, there can be no problem, as the type system always allows a subtype to be used in place of a supertype. Hence there is no need to redo type-checking when adding new types, if we can (as we do) prove that the new nominal type cannot grow larger (in the $\leq$ ordering) or become undefined.

How could such problems occur? One way that adding new types could cause an expression to cease to have a nominal type would be if the least upper bounds used to assign the nominal type to an **if**-expressions no longer existed. That could happen if a new supertype of an existing type were added to a program. The nominal type of an expression could grow larger if one were to add a new subtype relationship, so that the least-upper bound used to compute the nominal type of an **if** expression became larger than it was originally. Both of these problems are prevented by requiring that the new types are added in such a way as to make the original signature a subsignature of the new signature. Hence the following lemma is the source of the restrictions in the definition of subsignature which prohibit relating previously unrelated types by $\leq$, and the restrictions on least upper bounds in $\leq$. The restrictions on least upper bounds ensure that adding new types does not cause the least upper bounds of expressions to become undefined or larger than expected in the original signature.

Lemma 5.2 *Let Σ' and Σ be signatures. Let $\leq$ be the subtype relation of Σ. Let H be a type environment. Let $TYPES'$ and $TYPES$ be the type symbols of Σ' and Σ. Let $\mathtt{T} \in TYPES'$ be a type symbol. Let E be a LOAL expression.*

If Σ' is a subsignature of Σ and $\Sigma'; H \vdash E : \mathtt{T}$, then there is some type $\mathtt{S} \in TYPES$ such that $\Sigma; H \vdash E : \mathtt{S}$ and $\mathtt{S} \leq \mathtt{T}$.

Proof: (by induction on the length of the proof of $\Sigma'; H \vdash E : \mathtt{T}$).

Suppose that $\Sigma'; H \vdash E : \mathtt{T}$.

For the basis, if the proof has one step, then it must consist of an instance of one of the axiom schemes [ident] or [bot]. The conclusion then follows immediately.

Suppose that the proof of $\Sigma'; H \vdash E : \mathtt{T}$ takes $n > 1$ steps. The inductive hypothesis is that if $\Sigma'; H \vdash E_1 : \mathtt{T}_1$ is any step of the proof but the last, then there is some type $\mathtt{S}_1 \leq \mathtt{T}_1$ such that $\Sigma; H \vdash E_1 : \mathtt{S}_1$. There are several cases.

- If the last step of the proof is an the conclusion of the rule [mp], then E has the form $\mathtt{g}(\vec{E})$. There must be earlier steps of the form $\Sigma'; H \vdash \vec{E} : \vec{\tau}$ and $\Sigma' \vdash \mathit{ResSort}'(\mathtt{g}, \vec{\tau}) = \mathtt{T}$. By the inductive hypothesis, $\Sigma; H \vdash \vec{E} : \vec{\sigma}$, where $\vec{\sigma} \leq \vec{\tau}$. Since Σ' is a subsignature of Σ, $\mathit{ResSort}(\mathtt{g}, \vec{\tau}) = \mathtt{T}$. By the monotonicity of *ResSort*, it follows that for some $\mathtt{S} \leq \mathtt{T}$, $\Sigma \vdash \mathit{ResSort}(\mathtt{g}, \vec{\sigma}) = \mathtt{S}$.

- If the last step of the proof is an instance of [fcall] the conclusion follows as for the previous case.

- If the last step of the proof is an instance of [comb] there must be earlier steps in the proof of the form $\Sigma'; H, \vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{T}$, $\Sigma'; H \vdash \vec{E} : \vec{\sigma}$, and $\Sigma' \vdash \vec{\sigma} \leq' \vec{\mathtt{S}}$. By the inductive hypothesis, there is some type $\mathtt{U} \leq \mathtt{T}$ such that $\Sigma; H, \vec{\mathtt{x}} : \vec{\mathtt{S}} \vdash E_0 : \mathtt{U}$. By the inductive hypothesis, there is some $\vec{\tau} \leq \vec{\sigma}$ such that $\Sigma; H \vdash \vec{E} : \vec{\tau}$. Since Σ' is a subsignature of Σ, $\vec{\sigma} \leq \vec{\mathtt{S}}$. Since $\leq$ is transitive, $\vec{\tau} \leq \vec{\mathtt{S}}$.

- If the last step of the proof is an instance of [if] there must be earlier steps in the proof of the form: $\Sigma'; H \vdash E_1 : \mathtt{Bool}$, $\Sigma'; H \vdash E_2 : \mathtt{S}'_2$, $\Sigma'; H \vdash E_3 : \mathtt{S}'_3$, and $\Sigma' \vdash \mathrm{lub}(\mathtt{S}'_2, \mathtt{S}'_3) = \mathtt{T}$. Since there can be no subtypes of `Bool`, by the inductive hypothesis, $\Sigma; H \vdash E_1 : \mathtt{Bool}$. By the inductive hypothesis there are types $\mathtt{S}_2 \leq \mathtt{S}'_2$ and $\mathtt{S}_3 \leq \mathtt{S}'_3$ such that $\Sigma; H \vdash E_2 : \mathtt{S}_2$ and $\Sigma; H \vdash E_3 : \mathtt{S}_3$. Since Σ' is a subsignature of Σ, there is a sort $\mathtt{U} \leq \mathtt{T}$ that is a least upper bound for $\mathtt{S}_2$ and $\mathtt{S}_3$.

- If the last step of the proof is an instance of [isDef], then the result follows directly from the inductive hypothesis.

■

5.6 LOAL Semantics

The meaning of a LOAL program is an *observation*, which is a mapping from an algebra-environment pair to a set of possible results. The meaning is given for a particular signature, which in this section we fix as $\Sigma = (SORTS, TYPES, V, \leq, TFUNS, MN, ResSort)$. So instead of writing $\mathcal{M}_\Sigma[\![E]\!]$, we will write $\mathcal{M}[\![E]\!]$, for the meaning of an expression E.

5.6.1 Semantics of LOAL Expressions

Formally, $\mathcal{M}[\![E]\!]$ is a mapping that takes a Σ-algebra A, and a Σ-environment $\eta : Y \to |A|$ such that Y contains the free variables of E, and returns a set of possible results from $|A|$. Figure 28 gives the type of $\mathcal{M}[\![E]\!](A, \eta)$, and the denotation of each LOAL expression in an algebra A and environment $\eta : X \to |A|$. For convenience, we assume that η also maps typed function identifiers to the denotations of recursively-defined LOAL functions. In the figure, each expression has as its denotation a set of possible results. For example, for each type `T`, the only possible result of the expression `bottom[T]` is $\perp$. In the denotations of $\mathtt{g}(\vec{E})$ and $f(\vec{E})$, if $\vec{E}$ is empty, then $\mathcal{M}[\![\vec{E}]\!](A, \eta) = \{\langle\rangle\}$. So for example, $\mathcal{M}[\![\mathtt{g()}]\!](A, \eta) = \mathtt{g}^A()$.

5.6.2 Semantics of Recursive Function Definitions

Since LOAL "functions" can be nondeterministic, for a given algebra a function definition's denotation, written $\mathcal{F}[\![f]\!](A)$, is a mapping that takes a tuple of arguments and returns a set of possible results. However, this is not quite accurate, because in general systems of LOAL functions can be mutually recursive, and so in general one must assign meaning to the system as a whole, and extract the meaning of a particular function from it. The notation $\mathcal{F}[\![\vec{F}]\!]_j$ stands for the denotation of the j-th function in the system $\vec{F}$.

$\mathcal{F}[\![\vec{F}]\!]_j(A) : |A|^* \to PowerSet(|A|)$.

If the j-th function is named f, then the abbreviation $\mathcal{F}[\![f]\!]$ means $\mathcal{F}[\![\vec{F}]\!]_j$. For additional clarity, we will often consider the j-th function to be named f_j.

The semantics of a system of function definitions does not depend on the environment, because in the body of a recursively defined LOAL function, there can be no free identifiers

$\mathcal{M}[\![E]\!](A,\eta) : PowerSet(|A|)$

$\mathcal{M}[\![\mathtt{x}]\!](A,\eta) \stackrel{\text{def}}{=} \{\eta(\mathtt{x})\}$
$\mathcal{M}[\![\mathtt{bottom[T]}]\!](A,\eta) \stackrel{\text{def}}{=} \{\perp\}$
$\mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} \mathtt{g}^A(\vec{q})$
$\mathcal{M}[\![f(\vec{E})]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} (\eta(f))(\vec{q})$
$\mathcal{M}[\![\ (\mathbf{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)(\vec{E})]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{\vec{q}\in\mathcal{M}[\![\vec{E}]\!](A,\eta)} \mathcal{M}[\![E_0]\!](A, \eta[\vec{q}/\vec{\mathtt{x}}])$
$\mathcal{M}[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3\ \mathbf{fi}]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{q\in\mathcal{M}[\![E_1]\!](A,\eta)} \begin{cases} \mathcal{M}[\![E_2]\!](A,\eta) & \text{if } q = \mathit{true} \\ \mathcal{M}[\![E_3]\!](A,\eta) & \text{if } q = \mathit{false} \\ \{\perp\} & \text{otherwise} \end{cases}$
$\mathcal{M}[\![\mathtt{isDef?(}E\mathtt{)}]\!](A,\eta) \stackrel{\text{def}}{=} \bigcup_{q\in\mathcal{M}[\![E]\!](A,\eta)} \begin{cases} \{\mathit{true}\} & \text{if } q \neq \perp \\ \{\perp\} & \text{otherwise} \end{cases}$

Figure 28: The semantics of LOAL expressions.

or function identifiers, besides those of the other recursively defined functions and the function's formal arguments.

The semantics of systems of mutually recursive LOAL functions is given by a sequence of approximations. Approximations are obtained by textually expanding recursive calls [4, Page 20].

Fix an algebra A. Let

$$\begin{aligned} &\mathtt{fun}\ f_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1; \\ &\vdots \\ &\mathtt{fun}\ f_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m \end{aligned}$$

be a mutually recursive system of LOAL function definitions.

Definition 5.3 (unrolling family) *A family $D_{(j,i)}$ of expressions is called an* unrolling family *for the system of f_j if and only if for each j, $D_{(j,0)}$ is E_j, and $D_{(j,i+1)}$ is $D_{(j,i)}$ with, for all k, the function abstracts* $(\mathtt{fun}\ (\vec{\mathtt{x}_k} : \vec{\mathtt{S}_k})\ E_k)$ *simultaneously substituted for the f_k throughout $D_{(j,i)}$.*

The expression $D_{(j,i+1)}$ differs from $D_{(j,i)}$ in that one more level of recursion is unrolled.

As usual, an everywhere-$\perp$ function is the first approximation to recursive invocations in the unrolling family.

Definition 5.4 ($\perp_j$) *For each j, $\perp_j$ is a function abstract of the following form.*

$(\mathtt{fun}\ (\vec{\mathtt{x}_j} : \vec{\mathtt{S}_j})\ \mathtt{bottom[T}_j\mathtt{]})$

The least upper bounds of sequences in $\sqsubseteq$ (see Section 3.1.4) of approximate results are used to define the meaning of a function for a given algebra.

Definition 5.5 (sequence of approximate results) *Given an unrolling family $D_{(j,i)}$, a* sequence of approximate results, *$Q_j(A)(\vec{q})$, is a sequence in $\sqsubseteq$ such that, if $\eta(\vec{\mathtt{x}_j}) = \vec{q}$, then*

the i-th element of the sequence is a possible result of the i-th unrolling of f_j*:*

$$Q_j(A)(\vec{q})_i \in \mathcal{M}[\![D_{(j,i)}[\vec{\perp}/\vec{f}]]\!](A,\eta).$$

As Broy notes, there are such sequences in $\sqsubseteq$ because the language constructs (and each operation of A) are monotonic and because $D_{(j,i+1)}$ is derived from $D_{(j,i)}$ by unrolling another recursion. Note that $D_{(j,i)}[\vec{\perp}/\vec{f}]$ is recursion-free.

For each function index j, the set of all sequences is denoted $DD_j(A)(\vec{q})$.

$$DD_j(A)(\vec{q}) \stackrel{\text{def}}{=} \{Q_j(A)(\vec{q}) \mid Q_j(A)(\vec{q}) \text{ is a sequence of approximate results for } D_{(j,i)}\} \quad (29)$$

The denotation $\mathcal{F}[\![f_j]\!]$ of a function definition is defined as follows.

$$\mathcal{F}[\![f_j]\!](A)(\vec{q}) \stackrel{\text{def}}{=} \{\text{lub}(Q_j(A)(\vec{q})) \mid Q_j(A)(\vec{q}) \in DD_j(A)(\vec{q})\} \quad (30)$$

That is, $\mathcal{F}[\![f_j]\!](A)(\vec{q})$ is the set of all the least upper bounds of all sequences in $\sqsubseteq$ from $DD_j(A)(\vec{q})$.

5.6.3 Semantics of LOAL Programs

The notation $\mathcal{M}[\![P]\!]$ is also used for the meaning of a program P, which is an observation. To ensure that all possible results of a program have a visible type, we use the following function:

$$\mathit{Visible}(q) \stackrel{\text{def}}{=} \begin{cases} q & \text{if } q \in A_{\mathtt{T}} \text{ and } \mathtt{T} \in V \\ \perp & \text{otherwise.} \end{cases} \quad (31)$$

When carrier sets overlap, it may be that an element has both a visible type and a non-visible type. According to the definition, if $q \in A_{\mathtt{T}} \cap A_{\mathtt{S}}$ and $\mathtt{T}$ is a visible type, but $\mathtt{S}$ is not, then $\mathit{Visible}(q) = q$. The function *Visible* is extended pointwise to sets of possible results.

Consider the program: $\vec{F}$; **prog** $(\vec{\mathtt{x}} : \vec{\mathtt{S}})$:$\mathtt{T}$ = E with a system of recursive function definitions $\vec{F}$ and an expression E whose free identifiers are in the set $X = \{\vec{\mathtt{x}} : \vec{\mathtt{S}}\}$ declared following **prog**. Let $\eta : Y \to A$ be an environment such that $X \subseteq Y$. Then the meaning of the program is the meaning of E in the environment η extended by the meaning of each function in A, $\mathcal{F}[\![\vec{F}]\!](A)$, to the function identifiers $\vec{f}$ defined in $\vec{F}$:

$$\mathcal{M}[\![\vec{F}; \textbf{ prog } (\vec{\mathtt{x}} : \vec{\mathtt{S}})\text{:}\mathtt{T} = E]\!](A,\eta) \stackrel{\text{def}}{=} \mathit{Visible}(\mathcal{M}[\![E]\!](A, \eta[\mathcal{F}[\![\vec{F}]\!](A)/\vec{f}])).$$

5.7 LOAL Programs Obey the Subtype Relation

The following lemmas connect the LOAL semantics and its type system.

The first lemma says that a LOAL expression obeys the subtype relation (of the algebra over which it computes), in the sense that its possible results all have a type that is a subtype of the expression's nominal type.

Lemma 5.6 *Let* Σ *be a signature. Let* A *be a* Σ*-algebra. Let* $\vec{F}$ *be system of mutually recursive LOAL function definitions from a LOAL program that is type-safe with respect to* Σ*. Let* E *be a LOAL expression whose set of free identifiers is* X*. Let* H *be the type environment appropriate for* X*. Let* $\eta : X \to |A|$ *be a* Σ*-environment. Let* η' *be* $\eta[\mathcal{F}[\![\vec{F}]\!](A)/\vec{f}]$*.*

If $\Sigma; H \vdash E : \mathtt{T}$*, then each possible result of* $\mathcal{M}[\![E]\!](A,\eta')$ *has a type* $\tau \leq \mathtt{T}$*.*

⟨function specification⟩ ::= **fun** ⟨nominal signature⟩
requires ⟨pre-condition⟩
ensures ⟨post-condition⟩

Figure 29: Syntax of Function Specifications.

Proof Sketch: If the expression does not involve function calls, the result can be shown by induction on the structure of such expressions. For expressions that involve function calls, each possible result may be $\perp$ or is the result of an expression that does not involve function calls obtained by expanding the recursive calls as much as needed to compute the result. Since $\perp$ is in each carrier set, the result also holds in that case. ∎

The following lemma states that programs as a whole also obey the subtype relation.

Lemma 5.7 *Let Σ be a signature. Let P be a LOAL program whose formal arguments are $\vec{\mathtt{x}} : \vec{\mathtt{S}}$. Let $X = \{\mathtt{x} : \mathtt{S}\}$ and let $\eta : Y \to A$ be a Σ-environment such that $X \subseteq Y$.*

If $\Sigma \vdash P : \mathtt{T}$, then each possible result of $\mathcal{M}[\![P]\!](A, \eta)$ has a type $\tau \leq \mathtt{T}$. ∎

5.8 LOAL Function Specifications

Larch/LOAL interface specifications of LOAL functions are illustrated by the specification of *inBoth* in Figure 1. Function specifications are just like operation specifications (see Section 2.5). That is, if the arguments satisfy the pre-condition, then the function must terminate and each possible result must satisfy the post-condition. If the arguments do not satisfy the pre-condition, the function may fail to terminate or give any result (of the appropriate type). As in type specifications, the assertions in the pre- and post-conditions must be subtype-constraining (see Section 3.2.2). Subtypes such as `Interval` are not mentioned, but may be passed as arguments and returned as results whenever the corresponding supertype is mentioned.

The syntax of function specifications is given in Figure 29. Nonterminals not described there are as in Figure 13.

An omitted pre-condition is syntactic sugar for "**requires** true".

We give semantics to LOAL function specifications in a "specification environment" that maps type names to type specifications.

Definition 5.8 (base specification set) *The* base specification set *of a function specification is the set of all ⟨type spec⟩s for the non-visible types mentioned, either directly or indirectly, in the nominal signature of that function specification.*

For example, the base specification set of *inBoth* includes `IntSet` but not `Interval`.

Recall that a LOAL function denotation is a curried function that takes an algebra as an argument, and returns a function that takes a tuple of arguments from the carrier set of that algebra, producing a set of objects in the carrier set of that algebra (representing the set of possible results). Subsignatures are defined in Definition 3.3.

fun is2in(s:IntSet) **returns**(b:Bool)
ensures b = (2 ∈ s)

Figure 30: The function specification *is2in*.

Definition 5.9 (satisfies for functions) *Let S_f be a function specification with the following form:*

fun $f(\mathtt{x}_1 : \mathtt{S}_1, \ldots, \mathtt{x}_n : \mathtt{S}_n)$ **returns**$(\mathtt{v} : \mathtt{T})$
requires R
ensures Q.

Let Σ' be the signature of the base specification set of S_f. Let SPEC be a set of type specifications that includes the base specification set and such that Σ' is a subsignature of SIG(SPEC). Let the subtype relation of SPEC be $\leq$. A function denotation f satisfies *S_f* with respect to *SPEC if and only if for all SPEC-algebras A, for all proper SIG(SPEC)-environments η: $\{\mathtt{x}_1 : \mathtt{S}_1, \ldots, \mathtt{x}_n : \mathtt{S}_n\} \to |A|$, the following condition holds. If $(A, \eta) \models R$, then for all possible results $q \in f(A)(\eta(\vec{\mathtt{x}}))$: $q \neq \bot$, $(A, \eta[q/\mathtt{v}]) \models Q$, and there is some $\mathtt{U} \in$ TYPES such that $q \in A_{\mathtt{U}}$ and $\mathtt{U} \leq \mathtt{T}$. Furthermore, whenever some argument to the operation is $\bot$, then the only possible result is $\bot$.*

Example 5.10 *As an example of function satisfaction, consider the specification of the function is2in given in Figure 30. Let f_2 be the following function denotation:*

$$f_2 \stackrel{\text{def}}{=} \lambda A.\lambda s.\mathtt{elem}^A(s, 2). \tag{32}$$

Then f_2 satisfies the specification is2in given above with respect to II.

To see this, let C be an II-algebra, and let η_C:$\{\mathtt{s} : \mathtt{IntSet}\} \to |C|$ be a proper environment. Then $(C, \eta_C) \models$ true, so the pre-condition is satisfied. Let $r \in f_2(C)(\eta_C(s), 2) = \mathtt{elem}^C(\eta_C(s), 2)$ be a possible result. Then r is proper, has type `Bool` and by definition of an II-algebra is such that

$$(C, \eta_C[r/\mathtt{b}]) \models \mathrm{b} = (2 \in \mathrm{s}). \tag{33}$$

5.9 Simulation is Preserved by LOAL Programs

The following lemmas are used to show that simulation is preserved by LOAL programs, not just by single invocations of program operations. This property is analogous to the "fundamental theorem of logical relations" [59].

We show that simulation is preserved by all type-safe LOAL expressions in two steps. The first step, Lemma 5.11, assumes that the denotations of LOAL functions are related by a simulation relation (in a way described below), and shows that the possible results of an expression preserve simulation. The second step, Lemma 5.15, justifies the assumption used to prove Lemma 5.11 by showing that the meaning of a function definition is appropriately related in the related algebras. This proof technique is not circular, because in the proof of Lemma 5.15, only expressions that do not involve function calls are used.

For a given algebra, the denotation of a LOAL function is a mapping from tuples of arguments to sets of possible results. Such mappings are related by analogy to the definition

of logical relations [59] [49]. That is, if $\mathcal{R}$ is family of sorted relations, it is extended to the signatures of LOAL function identifiers as follows:

$$\mathcal{R}_{\vec{\mathtt{S}}\to\mathtt{T}} \stackrel{\text{def}}{=} \left\{(f_1, f_2) \mid \vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r} \Rightarrow f_1(\vec{q})\,\mathcal{R}_{\mathtt{T}}\,f_2(\vec{r})\right\}. \tag{34}$$

That is, for all f_1 and f_2, f_1 is related by $\mathcal{R}_{\vec{\mathtt{S}}\to\mathtt{T}}$ to f_2 if and only if whenever $\vec{q}\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$, then for every $q' \in f_1(\vec{q})$, there is some $r' \in f_2(\vec{r})$ such that $q'\,\mathcal{R}_{\mathtt{T}}\,r'$. Notice that this extension of $\mathcal{R}$ preserves the substitution property of simulation relations. Since operations are not first-class objects in LOAL, it is not necessary to show that this extension has all the properties of a simulation relation at each function signature. (See [36] for such an extension and the corresponding fundamental theorem of logical relations.)

To deal with LOAL expressions that have free function identifiers, environments are allowed to map typed function identifiers to their denotations in the algebra that is the environment's range (i.e., to set-valued functions).

For brevity throughout the rest of this section, fix a signature Σ and Σ-algebras C and A. As usual Σ is such that:

$$\Sigma = (SORTS, TYPES, V, \leq, TFUNS, POPS, ResSort).$$

Informally, the following lemma says that simulation is preserved by LOAL expressions if it is preserved by each recursively defined function.

Lemma 5.11 *Let X be a set of typed identifiers and function identifiers. Let $\eta_C : X \to |C|$ and $\eta_A : X \to |A|$ be Σ-environments.*

If $\mathcal{R}$ is a Σ-simulation relation between C and A and if $\eta_C\,\mathcal{R}\,\eta_A$, then for all types `T` *and for all LOAL expressions E of nominal type* `T` *whose free identifiers and function identifiers are a subset of X,*

$$\mathcal{M}[\![E]\!](C, \eta_C)\,\mathcal{R}_{\mathtt{T}}\,\mathcal{M}[\![E]\!](A, \eta_A).$$

Proof: (by induction on the structure of expressions).

For the basis, suppose that the expression is either an identifier or `bottom[T]`. If the expression is an identifier, then the result follows from $\eta_C\,\mathcal{R}\,\eta_A$. If the expression is `bottom[T]` for some type `T`, then the result follows from the bistrictness of $\mathcal{R}_{\mathtt{T}}$.

For the inductive step, assume that if $\eta_C\,\mathcal{R}\,\eta_A$, then the denotation of each subexpression of nominal type `T` in the environment η_C is related by $\mathcal{R}_{\mathtt{T}}$ to the denotation of the same subexpression in the environment η_A. There are several cases (see Figure 24).

- Suppose the expression is $\mathtt{g}(\vec{E})$. Since this expression has a nominal type, by the type inference rules it must be that $\vec{E} : \vec{\sigma}$, and $ResSort(\mathtt{g}, \vec{\sigma}) = \mathtt{T}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C, \eta_C)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_A)$ such that $\vec{q}\,\mathcal{R}_{\vec{\sigma}}\,\vec{r}$. Since $\vec{q}\,\mathcal{R}_{\vec{\sigma}}\,\vec{r}$ and $\mathcal{R}$ is a simulation relation, by the substitution property,

  $$\mathtt{g}^C(\vec{q})\,\mathcal{R}_{\mathtt{T}}\,\mathtt{g}^A(\vec{r}). \tag{35}$$

  Therefore, for each $q \in \mathcal{M}[\![\mathtt{g}(\vec{E})]\!](C, \eta_C)$ there is some $r \in \mathcal{M}[\![\mathtt{g}(\vec{E})]\!](A, \eta_A)$ such that $q\,\mathcal{R}_{\mathtt{T}}\,r$.

- Suppose the expression is $f(\vec{E})$ and f is a function identifier with nominal signature $\vec{\mathtt{S}} \to \mathtt{T}$. Since this expression has a nominal type, by the type inference rules it must be that $\vec{E}$ has nominal type $\vec{\sigma}$ and $\vec{\sigma} \leq \vec{\mathtt{S}}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C, \eta_C)$ be given. By the

inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_A)$ such that $\vec{q}\,\mathcal{R}_{\vec{\sigma}}\,\vec{r}$. Since $\vec{\sigma} \leq \vec{\mathtt{S}}$, by the subsorting property of a Σ-simulation relation, $\mathcal{R}_{\vec{\sigma}} \subseteq \mathcal{R}_{\vec{\mathtt{S}}}$, and so $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$. Since $\eta_C\ \mathcal{R}\ \eta_A$,

$$\eta_C(f)\ \mathcal{R}_{\vec{\mathtt{S}}\to\mathtt{T}}\ \eta_A(f), \tag{36}$$

and thus

$$(\eta_C(f))(\vec{q})\ \mathcal{R}_{\mathtt{T}}\ (\eta_A(f))(\vec{r}). \tag{37}$$

So, by definition of LOAL, for every possible result $q \in \mathcal{M}[\![f(\vec{E})]\!](C, \eta_C)$ there is some $r \in \mathcal{M}[\![f(\vec{E})]\!](A, \eta_A)$ such that $q\ \mathcal{R}_{\mathtt{T}}\ r$.

- Suppose the expression is $(\mathtt{fun}(\vec{\mathtt{x}} : \vec{\mathtt{S}})E_0)(\vec{E})$ and that the nominal type of the entire expression is $\mathtt{T}$. Let $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C, \eta_C)$ be given. By the inductive hypothesis, there is some $\vec{r} \in \mathcal{M}[\![\vec{E}]\!](A, \eta_A)$ such that $\vec{q}\,\mathcal{R}_{\vec{\sigma}}\,\vec{r}$, where $\vec{\sigma}$ is the nominal type of $\vec{E}$. Since the expression has a nominal type, by the type inference rules for LOAL it must be that $\vec{\sigma} \leq \vec{\mathtt{S}}$; thus $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$. It follows that if one binds $\vec{\mathtt{x}}$ to $\vec{q}$ in η_C and $\vec{\mathtt{x}}$ to $\vec{r}$ in η_A, then $(\eta_C[\vec{q}/\vec{\mathtt{x}}])\ \mathcal{R}\ (\eta_A[\vec{r}/\vec{\mathtt{x}}])$; thus the result follows by the inductive hypothesis (applied to E_0).

- Suppose the expression is $\mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{fi}$. Since $\mathtt{Bool}$ is a visible type and $\mathcal{R}$ is V-identical, the possible results from E_1 in η_C are a subset of those possible in η_A. Therefore the result follows from the inductive hypothesis applied to E_2 and E_3.

- If the expression is $\mathtt{isDef?}(E_1)$, then the result follows directly from the inductive hypothesis applied to E_1 and the bistrictness of $\mathcal{R}$.

■

The proof of the above lemma is the source of subsorting property required of simulation relations. This property guarantees that expressions related at a subtype are also related when a function call or a combination exploits subtype polymorphism. For example, if E has nominal type $\mathtt{S}$, $\mathtt{S}$ is a subtype of $\mathtt{T}$, and the function identifier f has nominal signature $\mathtt{T} \to \mathtt{U}$, then the expression $\mathtt{f}(E)$ is type-safe; furthermore, if the meanings of E in η_C and η_A are related at type $\mathtt{S}$ and if the meanings of f are also related at $\mathtt{T} \to \mathtt{U}$, then by the subsorting property the arguments are related at the nominal argument type $\mathtt{T}$, and so the results will be related at the nominal result type $\mathtt{U}$.

To show that the substitution property holds for LOAL programs one needs to show that simulation is preserved by recursively-defined LOAL functions.

To prove that, however, we need a technical result about LOAL functions and simulation relations: that simulation is preserved by recursively-defined LOAL functions. Because the semantics of systems of mutually recursive function definitions involve least upper bounds of sequences, it is convenient to first show that simulation relations are strongly monotonic and continuous. Recall that $q_1 \sqsubset q_2$ means $q_1 \sqsubseteq q_2$ and $q_1 \neq q_2$.

Definition 5.12 (strongly monotonic)
A binary relation $\mathcal{R}_{\mathtt{T}}$ between domains D_1 and D_2 is strongly monotonic *if and only if for all $q_1, q_2 \in D_1$ and for all $r_1, r_2 \in D_2$, whenever $q_1 \sqsubset q_2$, $q_1 \mathcal{R}_{\mathtt{T}} r_1$, and $q_2 \mathcal{R}_{\mathtt{T}} r_2$, then $r_1 \sqsubset r_2$.*

This definition is illustrated in Figure 31. A family of relations $\mathcal{R}$ is strongly monotonic if each $\mathcal{R}_{\mathtt{T}}$ is a strongly monotonic relation.

The following lemma says that each simulation relation is strongly monotonic. Recall from Section 3.1.4 that all carrier sets are assumed to be flat domains.

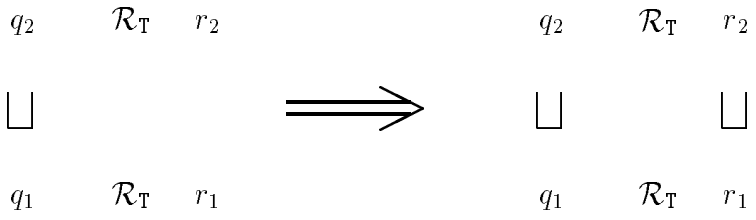

Figure 31: Strong monotonicity of $\mathcal{R}_{\mathtt{T}}$.

Lemma 5.13 *If $\mathcal{R}$ is a Σ-simulation relation between C and A, then $\mathcal{R}$ is strongly monotonic.*

Proof: Let $\mathtt{T}$ be a sort. Suppose $q_1 \sqsubset q_2$, $q_1 \mathcal{R}_{\mathtt{T}} r_1$, and $q_2 \mathcal{R}_{\mathtt{T}} r_2$. Since $q_1 \sqsubset q_2$, $q_1 = \bot$ and q_2 is proper. Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, $r_1 = \bot$ and r_2 is proper. So $r_1 \sqsubset r_2$. ∎

The following lemma says that each simulation relation is continuous. A family of relations is *continuous* if it is continuous at each type.

Lemma 5.14 *If $\mathcal{R}$ is a Σ-simulation relation between C and A, then $\mathcal{R}$ is continuous.*

Proof: Let $\mathtt{T}$ be a sort. Let Q be a sequence in $\sqsubseteq$ of elements of C. Let R be a sequence in $\sqsubseteq$ of elements of A, indexed by the same set as Q, such that for all indexes i, $q_i \mathcal{R}_{\mathtt{T}} r_i$.

If the only elements of Q are $\bot$, then the only elements of R are $\bot$, since $\mathcal{R}_{\mathtt{T}}$ is bistrict; thus $\mathrm{lub}(Q) = \bot \mathcal{R}_{\mathtt{T}} \bot = \mathrm{lub}(R)$.

Otherwise, Q contains some proper elements, Since Q contains elements from a flat domain, there is some index j such that $\mathrm{lub}(Q) = q_j \in Q$. Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, R also contains some proper elements and relates proper elements of Q to proper elements of R. Hence r_j is proper. Since R contains elements from a flat domain, $\mathrm{lub}(R) = r_j$. Thus $\mathrm{lub}(Q) = q_j \mathcal{R}_{\mathtt{T}} r_j = \mathrm{lub}(R)$. ∎

The lemma below shows that the substitution property holds for recursively defined functions.

Lemma 5.15 *Let*

$$\begin{aligned}&\mathtt{fun}\ f_1(\vec{\mathtt{x}_1} : \vec{\mathtt{S}_1}) : \mathtt{T}_1 = E_1;\\ &\vdots\\ &\mathtt{fun}\ f_m(\vec{\mathtt{x}_m} : \vec{\mathtt{S}_m}) : \mathtt{T}_m = E_m\end{aligned}$$

be a mutually recursive system of LOAL function definitions.

Suppose $\mathcal{R}$ is a Σ-simulation relation between Σ-algebras C and A Then for each j from 1 to m,

$$\mathcal{F}[\![f_j]\!](C)\, \mathcal{R}_{\vec{\mathtt{S}_j} \to \mathtt{T}_j}\, \mathcal{F}[\![f_j]\!](A). \tag{38}$$

Proof: Let $k \in \{1, \ldots, m\}$ be given. Let $\vec{q} \in C_{\vec{\mathtt{S}_k}}$ and $\vec{r} \in A_{\vec{\mathtt{S}_k}}$ be such that $\vec{q} \mathcal{R}_{\vec{\mathtt{S}_k}} \vec{r}$. Let $\eta_C : \{\vec{\mathtt{x}_k} : \vec{\mathtt{S}_k}\} \to |C|$ and $\eta_A : \{\vec{\mathtt{x}_k} : \vec{\mathtt{S}_k}\} \to |A|$ be such that $\eta_C(\vec{\mathtt{x}_k}) = \vec{q}$ and $\eta_A(\vec{\mathtt{x}_k}) = \vec{r}$. By construction, $\eta_C \mathcal{R} \eta_A$.

Let $D_{(j,i)}$ be an unrolling family, and let $Q_k(C)(\vec{q}) = \langle \hat{q}_i \rangle$ be a sequence of approximations for $D_{(j,i)}$. Let $DD_k(C)(\vec{q})$ denote the set of all such sequences $Q_k(C)(\vec{q})$ in $\sqsubseteq_C$; also let $DD_k(A)(\vec{r})$ be similarly defined to be the set of all such sequences $Q_k(A)(\vec{r}) = \langle \hat{r}_i \rangle$ in $\sqsubseteq_A$.

For each $Q_k(C)(\vec{q}) \in DD_k(C)(\vec{q})$, there is some $Q_k(A)(\vec{r}) \in DD_k(A)(\vec{r})$ such that for all i,

$$\hat{q}_i \; \mathcal{R}_{\mathtt{T}_k} \; \hat{r}_i. \tag{39}$$

Such a sequence $\langle \hat{r}_i \rangle$ can be found, because $D_{(k,i)}[\vec{\perp}/\vec{f}]$ is recursion-free, (thus Lemma 5.11 applies) and because $\mathcal{R}$ is strongly monotonic (by Lemma 5.13). Since $\mathcal{R}_{\mathtt{T}_k}$ is a continuous relation by Lemma 5.14, for such a $Q_k(A)(\vec{r})$,

$$\mathrm{lub}(Q_k(C)(\vec{q})) \; \mathcal{R}_{\mathtt{T}_j} \; \mathrm{lub}(Q_k(A)(\vec{r})). \tag{40}$$

We can thus calculate as follows.

$\quad \mathcal{F}[\![f_k]\!](C)(\vec{q})$
$= \quad \langle$by definition$\rangle$
$\quad \{\mathrm{lub}(Q_k(C)(\vec{q})) \mid Q_k(C)(\vec{q}) \in DD_k(C)(\vec{q})\}$
$\mathcal{R}_{\mathtt{T}_k} \; \langle$by the above, for all $Q_k(C)(\vec{q}) \in DD_k(C)(\vec{q})$, there is some $Q_k(A)(\vec{r}) \in DD_k(A)(\vec{r})\rangle$
$\quad \{\mathrm{lub}(Q_k(A)(\vec{r})) \mid Q_k(A)(\vec{r}) \in DD_k(A)(\vec{r})\}$
$= \quad \langle$by definition$\rangle$
$\quad \mathcal{F}[\![f_k]\!](A)(\vec{r})$

■

The main result of this section is the following theorem, which says that simulation is preserved by LOAL programs. In the study of the lambda calculus, this kind of theorem is known as the fundamental theorem of logical relations [59] [49] [37]. Showing such a theorem is another route to justifying the definition of legal subtype relations [34, Chapter 7] [37, Section 2.4], but one that is outside the scope of this paper. For the present paper, the following serves as another confirmation that our definition of simulation relations, on which the definition of subtype relations is reasonable.

Theorem 5.16 *Let Σ be a signature. Let $\leq$ be the presumed subtype relation of Σ. Let C and A be Σ-algebras.*

If $\mathcal{R}$ is a Σ-simulation relation between C and A, then for all sets of typed identifiers X, for all environments $\eta_C : X \to C$, and for all environments $\eta_A : X \to A$, if $\eta_A \; \mathcal{R} \; \eta_A$, then for all type-safe LOAL programs, P,

$$\mathcal{M}[\![P]\!](C, \eta_C) \subseteq \mathcal{M}[\![P]\!](A, \eta_A).$$

Proof: Suppose that $\mathcal{R}$ is a Σ-simulation relation between C and A. Let $X = \{\vec{\mathtt{x}} : \vec{\mathtt{U}}\}$ be a set of typed identifiers and let $\eta_C : X \to C$ and $\eta_A : X \to A$ be such that $\eta_C \; \mathcal{R} \; \eta_A$. Let P be a type-safe LOAL program of the form:

$$\begin{array}{l}
\mathtt{fun}\ f_1\ (\vec{\mathtt{z}_1} : \vec{\mathtt{S}_1})\mathtt{:}\ \mathtt{T}_1\ \mathtt{=}\ E_1\mathtt{;} \\
\vdots \\
\mathtt{fun}\ f_m\ (\vec{\mathtt{z}_m} : \vec{\mathtt{S}_m})\mathtt{:}\ \mathtt{T}_m\ \mathtt{=}\ E_m\mathtt{;} \\
\mathtt{program}\ (\vec{\mathtt{x}} : \vec{\mathtt{U}})\mathtt{:T}\ \mathtt{=}\ E\mathtt{.}
\end{array}$$

Let Z be the set of typed function identifiers that contains the f_j with their nominal signatures. Let $\eta'_C : Z \cup X \to C$ and $\eta'_A : Z \cup X \to A$ be defined so that for all $\mathtt{x}_i \in X$, $\eta'_C(\mathtt{x}_i) = \eta_C(\mathtt{x}_i)$, $\eta'_A(\mathtt{x}_i) = \eta_A(\mathtt{x}_i)$ and for all $f_j \in Z$, $\eta'_C(f_j)$ is $\mathcal{F}[\![f_j]\!](C)$ and $\eta'_A(f_j)$ is $\mathcal{F}[\![f_j]\!](A)$.

By Lemma 5.15, $\eta'_C \mathcal{R} \eta'_A$, since the denotations of recursively defined functions are related by $\mathcal{R}$. So by Lemma 5.11,

$$\mathcal{M}[\![E]\!](C, \eta'_C) \mathcal{R}_{\mathtt{T}} \mathcal{M}[\![E]\!](A, \eta'_A). \tag{41}$$

Recall that this means that for each $q \in \mathcal{M}[\![E]\!](C, \eta'_C)$, there is some $r \in \mathcal{M}[\![E]\!](A, \eta'_A)$ such that $q \mathcal{R}_{\mathtt{T}} r$. Since P is a program, the nominal type of E must be a visible type; that is, $\mathtt{T} \in V$. By Lemma 5.7, each such q and r has type $\mathtt{T}$. Since *Visible* is the identity on the visible types,

$$\mathcal{M}[\![E]\!](C, \eta'_C) = \mathcal{M}[\![P]\!](C, \eta_C) \tag{42}$$
$$\mathcal{M}[\![E]\!](A, \eta'_A) = \mathcal{M}[\![P]\!](A, \eta_A). \tag{43}$$

Since $\mathcal{R}$ is V-identical, for each $q \in \mathcal{M}[\![P]\!](C, \eta_C)$, there is some $r \in \mathcal{M}[\![P]\!](A, \eta_A)$ such that $q = r$; that is,

$$\mathcal{M}[\![P]\!](C, \eta_C) \subseteq \mathcal{M}[\![P]\!](A, \eta_A). \tag{44}$$

■

6 Hoare-style Verification for LOAL Programs

To verify a LOAL program that uses abstract data types, one reasons about expressions in much the same way one would as if there were no subtyping and message passing. That is, in verification one uses the nominal (static) type of each identifier and expression and the specification associated with each expression's nominal type.

Except for one proof rule that allows one to regard the type of an expression as a supertype of its nominal type, subtyping does not enter into the verification of a program directly. That is, most parts of a program's proof of correctness are unaffected by subtypes, and would be the same if LOAL did not have subtyping. However, the verifier must also prove that the specified relation $\leq$ is a legal subtype relation. We call this separation of concerns *supertype abstraction* [38] [35], because during verification one ignores the subtypes.

An unusual feature of LOAL verification is that we use a Hoare logic, despite the applicative nature of LOAL programs. Equational logics, which are preferred for reasoning about functional programs are difficult to use for LOAL, because the operations of the abstract data types can be nondeterministic. We also wanted to more easily adapt this research to the verification of imperative programs.

6.1 The Hoare Logic of LOAL Programs

Hoare-triples are written $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ and consist of a *pre-condition* P, a *result identifier* `v`, the result identifier's type `T`, an expression E, and a *post-condition* Q. In an applicative language, expressions have results but do not change the environment in which they execute. So the post-condition describes the environment that would result from binding the result identifier (`v:T`) to E's value, if the execution of E terminates. The type `T` must be a supertype of E's nominal type. Intuitively, $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ is true if whenever P holds and the execution of E terminates, then the value of E satisfies Q. Note that these are partial correctness triples. The name of the result identifier can be chosen at will, but cannot occur free in the pre-condition. Otherwise one might think that the execution of E changes the binding of the result identifier in the surrounding environment, whereas the notation only shows what identifier will be used to denote the possible results of E in the post-condition.

In the following formal definition, the notation $SIG(FSPEC)$ means a type environment that maps each function identifier in the set of function specifications $FSPEC$ to their nominal types. The pair $(SPEC, FSPEC)$ consists of a set of type specifications, $SPEC$, and a set of LOAL function specifications $FSPEC$. It is necessary to keep track of these specifications so that one can study the modularity of verification formally. In many examples below we use the pair (II, *inBoth*), where II is as in Example 2.1, and *inBoth* stands for the function specification of Figure 1.

Definition 6.1 (Hoare-triple) *Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let* `T` *be a type symbol of $SIG(SPEC)$. Let $\leq$ be the presumed subtype relation of $SIG(SPEC)$. Let E be a LOAL expression. Let H be a set of type assumptions that includes $SIG(FSPEC)$ and whose domain includes each free identifier in E.*

Then the formula $P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ Q$ is a Hoare-triple for $(SPEC, FSPEC)$ *if and only if* `y : T` *does not occur free in P, there is some type* `S` $\leq$ `T` *such that $SIG(SPEC); H \vdash E$: `S`, P and Q are $SIG(SPEC)$-assertions such that $SIG(SPEC); H \vdash P$: `Bool` and $SIG(SPEC); H, \mathtt{y} : \mathtt{T} \vdash Q$: `Bool`.*

[ident] $(SPEC, FSPEC) \vdash \text{true}\ \{\mathtt{v} : \mathtt{T} \leftarrow \mathtt{x}\}\ \mathtt{v} = \mathtt{x}$ $\quad \mathtt{x} : \mathtt{T}$

[bot] $(SPEC, FSPEC) \vdash \text{true}\ \{\mathtt{v} : \mathtt{T} \leftarrow \mathtt{bottom[T]}\}\ \text{false}$

[isDef] $(SPEC, FSPEC) \vdash \text{true}\ \{\mathtt{y} : \mathtt{Bool} \leftarrow \mathtt{isDef?}(E)\}\ \mathtt{y} = \text{true}$

[mp] $(SPEC, FSPEC) \vdash \text{Pre}(\mathtt{g}, \vec{\mathtt{S}})\ \{\mathtt{y} : \mathtt{T} \leftarrow \mathtt{g}(\vec{\mathtt{x}})\}\ \text{Post}(\mathtt{g}, \vec{\mathtt{S}})$
$\text{Formals}(\mathtt{g}, \vec{\mathtt{S}}) = \vec{\mathtt{x}} : \vec{\mathtt{S}}, \mathtt{y} : \mathtt{T}$
$\text{Pre}(\mathtt{g}, \vec{\mathtt{S}})$ sub.-con.
$\text{Post}(\mathtt{g}, \vec{\mathtt{S}})$ sub.-con.

[fcall] $(SPEC, FSPEC) \vdash \text{Pre}(f, \vec{\mathtt{S}})\ \{\mathtt{y} : \mathtt{T} \leftarrow f(\vec{\mathtt{x}})\}\ \text{Post}(f, \vec{\mathtt{S}})$
$\text{Formals}(f, \vec{\mathtt{S}}) = \vec{\mathtt{x}} : \vec{\mathtt{S}}, \mathtt{y} : \mathtt{T}$
$\text{Pre}(f, \vec{\mathtt{S}})$ sub.-con.
$\text{Post}(f, \vec{\mathtt{S}})$ sub.-con.

Figure 32: Axiom Schemes for verification of LOAL Expressions.

The phrase "for $(SPEC, FSPEC)$" is omitted when clear from context. The type of the result identifier is sometimes also omitted.

Figures 32 and 33 summarize the axioms and inference rules for LOAL expressions. In these figures P, Q, and R are assertions, M is a term, and E, E_1, and so on are LOAL expressions. The notation $E[\vec{\mathtt{z}}/\vec{\mathtt{x}}]$ means the expression E with the $\mathtt{z}_i$ simultaneously replacing all free occurrences of the $\mathtt{x}_i$. The notation $(SPEC, FSPEC) \vdash H$, where H is a Hoare-triple, means that one can prove H using the proof rules, including the traits of $SPEC$ and specifications of $(SPEC, FSPEC)$. The notation $SPEC \vdash Q$ means that the formula Q is provable from the traits of $SPEC$. The writing of one or more triples over another, as in the [call] and [conseq] rules, means that to prove the triple on the bottom, it suffices to prove the triples on the top. The name of a rule appears to its left. To the right of some of the rules are conditions on types and identifiers.

- The notation $\text{Formals}(\mathtt{g}, \vec{\mathtt{S}}) = \vec{\mathtt{x}} : \vec{\mathtt{S}}, \mathtt{y} : \mathtt{T}$ means that the formal arguments of the relevant operation specification of $\mathtt{g}$ are $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ (that is, a list $\mathtt{x}_1 : \mathtt{S}_1, \ldots$) and the formal result is $\mathtt{y} : \mathtt{T}$. The same notation is used for LOAL functions.

- The notation $\text{Pre}(\mathtt{g}, \vec{\mathtt{S}})$ means the pre-condition of the operation specification from $SPEC$ named $\mathtt{g}$ with nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$, where $\mathtt{T} = \mathit{ResSort}(\mathtt{g}, \vec{\mathtt{S}})$. A set of type specifications $(SPEC)$ must have at most one such operation specification. Similarly, $\text{Post}(\mathtt{g}, \vec{\mathtt{S}})$ is the post-condition of the operation specification with nominal signature $\vec{\mathtt{S}} \rightarrow \mathtt{T}$. The same notation is used for LOAL functions.

- The conditions of some rules require assertions to be subtype-constraining, abbreviated by "sub.-con."

- An identifier is *fresh* if it is not in the set of free identifiers of either the desired pre-condition or the desired post-condition of the rule. Identifiers are required to be fresh to avoid name capture problems.

- The notation $\mathtt{y} \notin \vec{\mathtt{x}}$ for the [comb] rule means that the result identifier $\mathtt{y}$ must not be one of the $\mathtt{x}_i$. This is also necessary to avoid capture problems.

[rename] $$\frac{(SPEC, FSPEC) \vdash P[\vec{\mathtt{z}}/\vec{\mathtt{x}}]\ \{\mathtt{y} : \mathtt{T}[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \leftarrow E[\vec{\mathtt{z}}/\vec{\mathtt{x}}]\}\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}]}{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ Q} \quad \vec{\mathtt{x}} : \vec{\mathtt{S}},\ \vec{\mathtt{z}} : \vec{\mathtt{S}} \text{ fresh}$$

[call] $$\frac{(SPEC, FSPEC) \vdash P\ \left\{\mathtt{y} : \mathtt{T} \leftarrow (\textbf{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ g(\vec{\mathtt{x}}))\ (\vec{E})\right\}\ Q}{(SPEC, FSPEC) \vdash P\ \left\{\mathtt{y} : \mathtt{T} \leftarrow g(\vec{E})\right\}\ Q} \quad \vec{E} : \vec{\sigma},\ \vec{\sigma} \leq \vec{\mathtt{S}}$$

[comb] $$\frac{\begin{array}{c}(SPEC, FSPEC) \vdash R_1 \wedge \cdots \wedge R_n\ \{\mathtt{y} : \mathtt{T} \leftarrow E_0\}\ Q, \\ (SPEC, FSPEC) \vdash P\ \{\mathtt{x}_1 : \mathtt{S}_1 \leftarrow E_1\}\ R_1, \\ \vdots \\ (SPEC, FSPEC) \vdash P\ \{\mathtt{x}_n : \mathtt{S}_n \leftarrow E_n\}\ R_n\end{array}}{(SPEC, FSPEC) \vdash P\ \left\{\mathtt{y} : \mathtt{T} \leftarrow (\textbf{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (E_1, \ldots, E_n)\right\}\ Q} \quad \begin{array}{r}\vec{\mathtt{x}} \text{ fresh}, \\ \mathtt{y} \notin \vec{\mathtt{x}}\end{array}$$

[up] $$\frac{(SPEC, FSPEC) \vdash P\ \{\mathtt{v} : \mathtt{S} \leftarrow E\}\ Q[\mathtt{v}/\mathtt{x}]}{(SPEC, FSPEC) \vdash P\ \{\mathtt{x} : \mathtt{T} \leftarrow E\}\ Q} \quad \begin{array}{r}\mathtt{v} \text{ fresh}, \mathtt{S} \leq \mathtt{T}, \\ E : \sigma, \sigma \leq \mathtt{S}, \\ Q \text{ sub.-con.}\end{array}$$

[conseq] $$\frac{\begin{array}{c}SPEC \vdash P \Rightarrow P_1, \\ (SPEC, FSPEC) \vdash P_1\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ Q_1, \\ SPEC \vdash Q_1 \Rightarrow Q\end{array}}{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ Q} \quad \begin{array}{r}P, P_1, Q_1, Q \\ \text{sub.-con.}\end{array}$$

[carry] $$\frac{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ Q}{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ P \wedge Q}$$

[equal] $$\frac{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ \mathtt{y} = N}{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ M[\mathtt{y}/\mathtt{z}] = M[N/\mathtt{z}]} \quad \mathtt{z} : \mathtt{T}$$

[if] $$\frac{\begin{array}{c}(SPEC, FSPEC) \vdash P \wedge R_1\ \{\mathtt{v} : \mathtt{Bool} \leftarrow E_1\}\ \mathtt{v} = \text{true}, \\ (SPEC, FSPEC) \vdash P \wedge R_1\ \{\mathtt{y} : \mathtt{T} \leftarrow E_2\}\ Q, \\ (SPEC, FSPEC) \vdash P \wedge R_2\ \{\mathtt{v} : \mathtt{Bool} \leftarrow E_1\}\ \mathtt{v} = \text{false}, \\ (SPEC, FSPEC) \vdash P \wedge R_2\ \{\mathtt{y} : \mathtt{T} \leftarrow E_3\}\ Q, \\ (SPEC, FSPEC) \vdash P \wedge R_3\ \{\mathtt{y} : \mathtt{T} \leftarrow E_2\}\ Q, \\ (SPEC, FSPEC) \vdash P \wedge R_3\ \{\mathtt{y} : \mathtt{T} \leftarrow E_3\}\ Q, \\ SPEC \vdash (R_1 \vee R_2 \vee R_3) = \text{true}\end{array}}{(SPEC, FSPEC) \vdash P\ \{\mathtt{y} : \mathtt{T} \leftarrow \textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3\ \textbf{fi}\}\ Q}$$

Figure 33: Inference rules for verification of LOAL expressions.

As usual, the specifications of each type's operations and the specifications of the LOAL functions are taken as axioms. The instance of the axiom scheme [mp] used for a particular message send is determined by the nominal types of the message send's arguments. Such static overloading resolution is the only choice for verification, even for programs that use dynamic binding, because verification should be possible before the program runs.

The axiom schemes [mp] and [fcall] are derived from the specifications of operations and LOAL functions (respectively). For example, the following are two instances of the axiom scheme [mp] for the message name `choose`: the first from the specification of type `IntSet`, the second from the specification of the type `Interval`.

$$(II, inBoth) \;\vdash\; \neg\text{isEmpty}(\texttt{s})\ \{\texttt{i} : \texttt{Int} \leftarrow \texttt{choose(s)}\}\ \texttt{i} \in \texttt{s} \tag{45}$$

$$(II, inBoth) \;\vdash\; \text{true}\ \{\texttt{i} : \texttt{Int} \leftarrow \texttt{choose(s)}\}\ \texttt{i} = \text{leastElement}(\texttt{s}). \tag{46}$$

In the first, the type of `s` is `IntSet`, and in the second it is `Interval`. Similarly, the following is an instance of the axiom scheme [fcall] for the function name *inBoth*:

$$(\text{II}, inBoth) \vdash \neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2}))\ \{\texttt{i} : \texttt{Int} \leftarrow inBoth(\texttt{s1},\texttt{s2})\}\ (\texttt{i} \in \texttt{s1}) \wedge (\texttt{i} \in \texttt{s2}). \tag{47}$$

As they stand, these axioms only apply when the actual argument expressions and the result identifier are the same as the formals used in the specifications; i.e., they are the same identifier. To reason about message passing and operations in more general situations one uses the inference rules [call], [comb], [up] and [rename].

Renaming of identifiers without changing their types is handled by the inference rule [rename]. For example, if `s1` has nominal type `IntSet`, then the proof of

$$(\text{II}, inBoth) \vdash \neg\text{isEmpty}(\texttt{s1})\ \{\texttt{j} : \texttt{Int} \leftarrow \texttt{choose(s1)}\}\ \texttt{j} \in \texttt{s1} \tag{48}$$

follows directly from Formula (45), since one can replace `s` for `s1` and `i` for `j` in the above to obtain that axiom.

To reason about function calls or message passing expressions whose argument expressions are not identifiers, one first uses the inference rule [call], to convert the expression into a combination (the applications of a function abstract), and then uses the rule [comb] to deal with the combination. For example, to prove the following

$$(\text{II}, \emptyset) \vdash \text{true}\ \{\texttt{r} : \texttt{IntSet} \leftarrow \texttt{ins(null(IntSet),3)}\}\ \texttt{r}\ \text{eqSet}\ \{3\} \tag{49}$$

it suffices to prove the following triple (which is displayed vertically).

$$(\text{II}, \emptyset) \vdash \begin{array}{l} \text{true} \\ \{\texttt{r} \leftarrow \textbf{fun}\ \texttt{(s:IntSet,i:Int) ins(s,i)) (null(IntSet), 3)}\} \\ \texttt{r}\ \text{eqSet}\ \{3\} \end{array} \tag{50}$$

The combination in Formula 50 is proved by using the rule [comb]. To ensure the proper scope for the formal of the function abstract (the $\texttt{x}_i$ in the rule for [comb]), in general one first must use the rule [rename] to hide any bindings of the $\texttt{x}_i$ in the outer scope before using [comb]. The R_i in the [comb] rule are chosen so that they characterize the argument values and so that their conjunction is sufficient to prove the desired post-condition, Q, from the body of the function abstract. For example, to prove Formula (50), it suffices to prove the following triples

$$(\text{II}, \emptyset) \;\vdash\; (\texttt{s}\ \text{eqSet}\ \{\}) \wedge (\texttt{i} = 3)\ \{\texttt{r} \leftarrow \texttt{ins(s,i)}\}\ \texttt{r}\ \text{eqSet}\ \{3\} \tag{51}$$

$$(\text{II}, \emptyset) \;\vdash\; \text{true}\ \{\texttt{s} \leftarrow \texttt{null(IntSet)}\}\ \texttt{s}\ \text{eqSet}\ \{\} \tag{52}$$

$$(\text{II}, \emptyset) \;\vdash\; \text{true}\ \{\texttt{i} \leftarrow \texttt{3}\}\ \texttt{i} = 3. \tag{53}$$

The rule [up] is similar to the rule [rename], except that in [up] only the result identifier can be changed, and one can change not just its name, but also its type. For example, to show

$$(\mathrm{II}, \mathit{inBoth}) \vdash \text{true} \ \{\texttt{s2 : IntSet} \leftarrow \texttt{create(Interval,2,5)}\} \ \texttt{s2} \ \text{eqSet} \ [2,5] \tag{54}$$

it suffices to show the following

$$(\mathrm{II}, \mathit{inBoth}) \vdash \text{true} \ \{\texttt{iv : Interval} \leftarrow \texttt{create(Interval,2,5)}\} \ \texttt{iv} \ \text{eqSet} \ [2,5]. \tag{55}$$

Recall that the trait function "eqSet" is defined for combinations of `IntSet` and `Interval` arguments. So both triples make sense. The reason why Formula (54) follows from Formula (55) is because whenever the assertion "`iv` eqSet [2,5]" holds, and `s2` denotes the same abstract value as `iv`, then "`s2` eqSet [2,5]" holds. The [up] rule allows one to view an object of a subtype, `iv : Interval`, at a more abstract level, as an object of a supertype, `s2 : IntSet`.

There are two reasons why the [up] rule is valid in general. The first is that all trait functions defined on a supertype must also be defined for the subtype. Hence the [up] rule in effect requires that the desired post-condition, Q, is expressible in the language of the supertype; that is, so that Q sort checks with `x : T`. Since Q has a nominal sort in an environment where `x : T` is a type assumption, the assertion $Q[\texttt{v/x}]$ will sort-check if Q is subtype-constraining, `v : S`, and `S` $\leq$ `T`. To see why Q must be subtype-constraining, consider the following example. Suppose `iv : Interval` and Q is "`iv` = [3,3]". Then Q sort-checks, but if `s2 : IntSet`, then $Q[\texttt{s2/iv}]$, which is "`s2` = [3,3]", does not. However, such syntactic restrictions are not enough, and there are further restrictions placed on legal subtype relations to ensure that whenever $Q[\texttt{v/x}]$ holds and `v` and `x` denote the same abstract value, then Q holds as well.

If an assertion about a subtype object is not expressed using the trait functions of the supertype, then one must use the rule of consequence, [conseq], to rewrite it. The rule [conseq] is restricted in this Hoare logic in that the assertions involved must be subtype-constraining. This restriction is necessary, because identifiers may denote objects of a subtype of their nominal type. The following example shows why the restriction is needed. Consider the implication

$$((\text{size}(\texttt{s}) = 1) \wedge (3 \in \texttt{s})) \Rightarrow (\texttt{s} = \{3\}), \tag{56}$$

where `s` has nominal type `IntSet`. This implication can be proved from the axioms of the trait IntSetTrait, which means that if `s` denotes an `IntSet`, then the implication is valid. However, it is not valid if `s` denotes the `Interval` with abstract value [3,3], because [3,3] and {3} may be distinct abstract values. The fallacies that could arise because such of implications, those that appear to be true but do not work in the presence of subtyping, are avoided in this example by using "eqSet" instead of the second "=" to obtain a subtype-constraining assertion. Thus, in general subtype constraining assertions are necessary for the validity of the [conseq] rule.

An example of the use of [conseq] is the following. From the formulas

$$\mathit{II} \ \vdash \ ((\texttt{s} \ \text{eqSet} \ \{\}) \wedge (\texttt{i} = 3)) \Rightarrow \text{true} \tag{57}$$

$$(\mathit{II}, \emptyset) \ \vdash \ \text{true} \ \{\texttt{r} \leftarrow \texttt{ins}(\texttt{s}, \texttt{i})\} \ \texttt{r} \ \text{eqSet} \ (\texttt{s} \cup \{\texttt{i}\}) \tag{58}$$

$$\mathit{II} \ \vdash \ (\texttt{r} \ \text{eqSet} \ (\texttt{s} \cup \{\texttt{i}\})) \Rightarrow (\texttt{r} \ \text{eqSet} \ (\texttt{s} \cup \{\texttt{i}\})) \tag{59}$$

one can conclude

$$(II, \emptyset) \vdash (\mathtt{s}\ \text{eqSet}\ \{\}) \wedge (\mathtt{i} = 3)\ \{\mathtt{r} \leftarrow \mathtt{ins(s,i)}\}\ \mathtt{r}\ \text{eqSet}\ (\mathtt{s} \cup \{\mathtt{i}\}) \tag{60}$$

In later examples we omit trivial implications like Formula (59).

If we wish to prove Formula (51), we can use Formula (60), but must somehow get the assertion $\mathtt{i} = 3$ into the right hand side. Since LOAL does not have side-effects, the expression execution cannot invalidate the assertion $\mathtt{i} = 3$. So in the Hoare logic, the rule [carry] allows one to carry the pre-condition into the post-condition. For example, from Formula (51), one can conclude that

$$(\mathtt{s}\ \text{eqSet}\ \{\}) \wedge (\mathtt{i} = 3)\ \{\mathtt{r} \leftarrow \mathtt{ins(s,i)}\}\ \mathtt{r}\ \text{eqSet}\ (\mathtt{s} \cup \{\mathtt{i}\}) \wedge (\mathtt{s}\ \text{eqSet}\ \{\}) \wedge (\mathtt{i} = 3). \tag{61}$$

Formula (51) then follows by the rule [conseq].

The other rules of the logic are fairly straightforward and their peculiarities have more to do with using a partial correctness Hoare-logic to reason about nondeterministic expressions than with subtyping and message passing. The inference rule [equal] allows one to draw conclusions from equations in post-conditions; this ability is sometimes needed to weaken a post-condition that results from using the [ident] rule to one that is subtype constraining, because the rule [conseq] only permits one to use subtype-constraining assertions. See [34] for details of the other rules.

6.2 Verification Examples

Example 6.2 *The way that the logic handles explicit use of subtyping is shown by the proof of the formula:*

$$(\text{II}, \mathit{inBoth}) \vdash (\mathtt{s}\ \text{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \text{eqSet}\ [2,5])\ \{\mathtt{i : Int} \leftarrow \mathit{inBoth}\mathtt{(s,iv)}\}\ \mathtt{i} = 3 \tag{62}$$

where II is as in Example 2.1, `s` *has nominal type* `IntSet`*, and* `iv` *has nominal type* `Interval`*.*

Both type specifications are used in the verification below, because the type `Interval` is used explicitly. In the proof below and in other such verifications, we work "backwards", that is, from the formula to be proved to the axioms.

Proof: Since the second argument expression (`iv`) has a different nominal type from the formal specified for *inBoth*, the rule [call] is used first to obtain the following goal triple.

$$\begin{array}{rl} & (\mathtt{s}\ \text{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \text{eqSet}\ [2,5]) \\ (\text{II}, \mathit{inBoth}) \vdash & \{\mathtt{i} \leftarrow \mathtt{(\textbf{fun}\ (s1,s2:IntSet)}\ \mathit{inBoth}\mathtt{(s1,s2))\ (s,iv)}\} \\ & \mathtt{i} = 3 \end{array} \tag{63}$$

The rule [comb] then gives the following subgoals:

$$(\text{II}, \mathit{inBoth}) \vdash (\mathtt{s1}\ \text{eqSet}\ \{3\}) \wedge (\mathtt{s2}\ \text{eqSet}\ [2,5])\ \{\mathtt{i} \leftarrow \mathit{inBoth}\mathtt{(s1,s2)}\}\ \mathtt{i} = 3 \tag{64}$$

$$(\text{II}, \mathit{inBoth}) \vdash (\mathtt{s}\ \text{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \text{eqSet}\ [2,5])\ \{\mathtt{s1} \leftarrow \mathtt{s}\}\ \mathtt{s1}\ \text{eqSet}\ \{3\} \tag{65}$$

$$\begin{array}{rl} & (\mathtt{s}\ \text{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \text{eqSet}\ [2,5]) \\ (\text{II}, \mathit{inBoth}) \vdash & \{\mathtt{s2 : IntSet} \leftarrow \mathtt{iv}\} \\ & \mathtt{s2}\ \text{eqSet}\ [2,5] \end{array} \tag{66}$$

The proof of the third subgoal is the most interesting of the three subgoals. The rule [up] is used to generate the following goal

$$(\mathrm{II}, \mathit{inBoth}) \vdash (\mathtt{s}\ \mathrm{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \mathrm{eqSet}\ [2,5])\ \{\mathtt{iv2} : \mathtt{Interval} \leftarrow \mathtt{iv}\}\ \mathtt{iv2}\ \mathrm{eqSet}\ [2,5] \tag{67}$$

By the traits of II, the post-condition follows from the precondition conjoined with "`iv2` eqSet `iv`"

$$(\mathtt{s}\ \mathrm{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \mathrm{eqSet}\ [2,5]) \wedge (\mathtt{iv2}\ \mathrm{eqSet}\ \mathtt{iv}) \Rightarrow (\mathtt{iv2}\ \mathrm{eqSet}\ [2,5]) \tag{68}$$

so by the rule [conseq] it suffices to prove Formula (67) with the antecedent in the above implication substituted for the post-condition. Using the rule [carry] "backwards", the conjunction of the pre-condition can be dropped from the post-condition, so it suffices to prove the following.

$$(\mathrm{II}, \mathit{inBoth}) \vdash (\mathtt{s}\ \mathrm{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \mathrm{eqSet}\ [2,5])\ \{\mathtt{iv2} : \mathtt{Interval} \leftarrow \mathtt{iv}\}\ \mathtt{iv2}\ \mathrm{eqSet}\ \mathtt{iv} \tag{69}$$

By the traits of II

$$\mathrm{II} \vdash ((\mathtt{s}\ \mathrm{eqSet}\ \{3\}) \wedge (\mathtt{iv}\ \mathrm{eqSet}\ [2,5])) \Rightarrow \mathrm{true} \tag{70}$$

so by the rule [conseq], it suffices to prove the following.

$$(\mathrm{II}, \mathit{inBoth}) \vdash \mathrm{true}\ \{\mathtt{iv2} : \mathtt{Interval} \leftarrow \mathtt{iv}\}\ \mathtt{iv2}\ \mathrm{eqSet}\ \mathtt{iv} \tag{71}$$

By the traits of II

$$\mathrm{II} \vdash \mathtt{iv2}\ \mathrm{eqSet}\ \mathtt{iv} = \mathtt{iv}\ \mathrm{eqSet}\ \mathtt{iv} \Rightarrow \mathtt{iv2}\ \mathrm{eqSet}\ \mathtt{iv} \tag{72}$$

since "`iv` eqSet `iv`" is identically "true". So by the inference rule [conseq] it suffices to prove

$$(\mathrm{II}, \mathit{inBoth}) \vdash \mathrm{true}\ \{\mathtt{iv2} : \mathtt{Interval} \leftarrow \mathtt{iv}\}\ \mathtt{iv2}\ \mathrm{eqSet}\ \mathtt{iv} = \mathtt{iv}\ \mathrm{eqSet}\ \mathtt{iv} \tag{73}$$

By the inference rule [equal], with M as "`z` eqSet `iv`", it suffices to prove the following.

$$(\mathrm{II}, \mathit{inBoth}) \vdash \mathrm{true}\ \{\mathtt{iv2} : \mathtt{Interval} \leftarrow \mathtt{iv}\}\ \mathtt{iv2} = \mathtt{iv} \tag{74}$$

which is an instance of the axiom scheme [ident].

The second subgoal, Formula (65), follows in the same way as the third subgoal, except that [up] need not be used.

The first subgoal follows from the axiom [fcall] for *inBoth* and the subgoal's pre-condition. The key observation is that, by the traits of II,

$$\mathit{II} \vdash ((\mathtt{i} \in \mathtt{s1}) \wedge (\mathtt{i} \in \mathtt{s2}) \wedge (\mathtt{s1}\ \mathrm{eqSet}\ \{3\}) \wedge (\mathtt{s2}\ \mathrm{eqSet}\ [2,5])) \Rightarrow (\mathtt{i} = 3) \tag{75}$$

so by the rule [conseq] it suffices to prove the first subgoal with the antecedent of the above implication replacing the post-condition. ∎

The method for verifying the partial correctness of an entire program is "divide and conquer"; first one specifies and verifies the function definitions that appear in the program. Then one uses the Hoare logic to prove the desired Hoare-triple, using the specifications of the recursively defined functions as axioms.

fun testFor(i:Int, s1,s2: IntSet) **returns**(j:Int)
requires (i ∈ s1) ∧ (¬(isEmpty(s1 ∩ s2)))
ensures (j ∈ s1) ∧ (j ∈ s2)

Figure 34: Specification of the function *testFor*.

To verify the partial correctness of a system of LOAL function definitions, one shows that for each function f,

$$(SPEC, FSPEC) \vdash \mathrm{Pre}(f, \vec{\mathtt{S}})\ \{\mathtt{y} : \mathtt{T} \leftarrow E\}\ \mathrm{Post}(f, \vec{\mathtt{S}})$$

follows from the proof rules, where E is the body of f, $\mathtt{y} : \mathtt{T}$ is the formal result identifier from the specification of f, $\mathrm{Pre}(f, \vec{\mathtt{S}})$ is the pre-condition from the specification of f in *FSPEC*, and $\mathrm{Post}(f, \vec{\mathtt{S}})$ is its post-condition. During this proof one can use the axiom scheme [fcall], which assumes that each recursively defined function is partially correct. It is beyond the scope of this paper to provide a formal method for verifying termination.

As an example of recursive function verification, we verify the partial correctness of the implementation of *inBoth* given in Figure 2 against the specification given in Figure 1.

Since *inBoth* calls the function *testFor*, it is necessary to specify *testFor* as well. The specification of *testFor* is given in Figure 34.

During the verification of *inBoth* the specification of *testFor* is used as an axiom, to establish its partial correctness. The verifications use the type specification `IntSet`, since that is the only non-visible type mentioned. This shows how the proof system is modular: the type `Interval` does not even enter into the proof.

Example 6.3 *The implementation of inBoth is partially correct, that is:*

$$(\mathtt{IntSet}, \mathit{testFor}) \vdash \begin{array}{l} \neg(\mathrm{isEmpty}(\mathtt{s1} \cap \mathtt{s2})) \\ \{\mathtt{i} \leftarrow \mathit{testFor}(\mathtt{choose(s1),\ s1,\ s2})\}\ . \\ (\mathtt{i} \in \mathtt{s1})\ \wedge\ (\mathtt{i} \in \mathtt{s2}) \end{array} \tag{76}$$

Proof: To avoid name clashes with the result identifier and the formals of *testFor*, the rule [rename] is used to generate the following goal.

$$(\mathtt{IntSet}, \mathit{testFor}) \vdash \begin{array}{l} \neg(\mathrm{isEmpty}(\mathtt{t1} \cap \mathtt{t2})) \\ \{\mathtt{j} \leftarrow \mathit{testFor}(\mathtt{choose(t1),\ t1,\ t2})\} \\ (\mathtt{j} \in \mathtt{t1})\ \wedge\ (\mathtt{j} \in \mathtt{t2}) \end{array} \tag{77}$$

Since the expression in question is a function call, one must use the rule [call]. This gives a goal with the same pre- and post-conditions as above, but whose expression is:

`(fun (i:Int,s1,s2:IntSet)` *testFor*`(i,s1,s2)) (choose(t1), t1, t2).`

Since the expression above is a combination, the [comb] rule is used to give the following subgoals:

$$(\mathtt{IntSet}, \mathit{testFor}) \vdash R_1 \wedge R_2 \wedge R_3\ \{\mathtt{j} \leftarrow \mathit{testFor}(\mathtt{i,s1,s2})\}\ (\mathtt{j} \in \mathtt{t1}) \wedge (\mathtt{j} \in \mathtt{t2}) \tag{78}$$

$$(\mathtt{IntSet}, \mathit{testFor}) \vdash \neg(\mathrm{isEmpty}(\mathtt{t1} \cap \mathtt{t2}))\ \{\mathtt{i} \leftarrow \mathtt{choose(t1)}\}\ \mathtt{i} \in \mathtt{t1} \tag{79}$$

$$(\texttt{IntSet}, \textit{testFor}) \vdash \begin{array}{l} \neg(\text{isEmpty}(\texttt{t1} \cap \texttt{t2})) \\ \{\texttt{s1} \leftarrow \texttt{t1}\} \\ (\texttt{s1}\ \text{eqSet}\ \texttt{t1}) \wedge (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{t2}))) \end{array} \tag{80}$$

$$(\texttt{IntSet}, \textit{testFor}) \vdash \begin{array}{l} \neg(\text{isEmpty}(\texttt{t1} \cap \texttt{t2})) \\ \{\texttt{s2} \leftarrow \texttt{t2}\} \\ (\texttt{s2}\ \text{eqSet}\ \texttt{t2}) \wedge (\neg(\text{isEmpty}(\texttt{t1} \cap \texttt{s2}))) \end{array}, \tag{81}$$

where the subtype-constraining R_i are:

$$\begin{aligned} R_1 &= (\texttt{i} \in \texttt{t1}) \\ R_2 &= ((\texttt{s1}\ \text{eqSet}\ \texttt{t1}) \wedge (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{t2})))) \\ R_3 &= ((\texttt{s2}\ \text{eqSet}\ \texttt{t2}) \wedge (\neg(\text{isEmpty}(\texttt{t1} \cap \texttt{s2})))). \end{aligned}$$

The first subgoal is shown as follows. By the traits of `IntSet`, if follows that

$$\texttt{IntSet} \vdash (\texttt{j} \in \texttt{s1}) \wedge (\texttt{j} \in \texttt{s2}) \wedge R_1 \wedge R_2 \wedge R_3 \Rightarrow (\texttt{j} \in \texttt{t1}) \wedge (\texttt{j} \in \texttt{t2}). \tag{82}$$

So by [conseq], it suffices to prove the following.

$$(\texttt{IntSet}, \textit{testFor}) \vdash \begin{array}{l} R_1 \wedge R_2 \wedge R_3 \\ \{\texttt{j} \leftarrow \textit{testFor}(\texttt{i},\texttt{s1},\texttt{s2})\} \\ (\texttt{j} \in \texttt{s1}) \wedge (\texttt{j} \in \texttt{s2}) \wedge R_1 \wedge R_2 \wedge R_3 \end{array} \tag{83}$$

By the rule [carry], it suffices to prove the above with the conjunct $R_1 \wedge R_2 \wedge R_3$ dropped from the post-condition. By the traits of `IntSet`, one can prove the following:

$$\texttt{IntSet} \vdash (R_1 \wedge R_2 \wedge R_3) \Rightarrow ((\texttt{i} \in \texttt{s1}) \wedge (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})))), \tag{84}$$

where the right-hand side of the implication is the pre-condition of *testFor*. So by [conseq] it suffices to prove

$$(\texttt{IntSet}, \textit{testFor}) \vdash \begin{array}{l} ((\texttt{i} \in \texttt{s1}) \wedge (\neg(\text{isEmpty}(\texttt{s1} \cap \texttt{s2})))) \\ \{\texttt{j} \leftarrow \textit{testFor}(\texttt{i},\texttt{s1},\texttt{s2})\} \\ (\texttt{j} \in \texttt{s1}) \wedge (\texttt{j} \in \texttt{s2}) \end{array} \tag{85}$$

which is the axiom [fcall] for *testFor*.

The second subgoal is shown by using [conseq] and the following formula,

$$\texttt{IntSet} \vdash \neg(\text{isEmpty}(\texttt{t1} \cap \texttt{t2})) \Rightarrow \neg(\text{isEmpty}(\texttt{t1})) \tag{86}$$

along with the rules [rename] and the [mp] axiom for `choose`.

The third and fourth subgoals follow from the rules [conseq], [equal] and the axiom [ident]. ∎

The verification of the implementation of *testFor* (in Figure 2) is left as a (long, but straightforward) exercise for the reader.

6.3 Formal Semantics of Hoare-Triples

The model theory which we will use to prove the soundness of the Hoare logic is simple, given all the machinery we have already assembled. In this section we present the model theory of Hoare triples, by defining when they are modeled by an algebra-environment pair

with an extended environment, and when they are valid. The extension to the environment of an algebra-environment pair, maps function names to their denotations. These function denotations are models of the assumed function specifications.

The formal definition of the semantics of a Hoare-triple is similar to the definition of when a method or LOAL function satisfies its specification, but allows non-termination.

Definition 6.4 (models) *Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ be a Hoare-triple for $(SPEC, FSPEC)$. Let X be a set of free identifiers that contains all the free identifiers of P, E, and Q except $\mathtt{v} : \mathtt{T}$. Let A be a $SPEC$-algebra, and let $\eta : X \to |A|$ be a proper $SIG(SPEC)$-environment. For each f in the domain of $SIG(FSPEC)$, let $\mathcal{F}[\![f]\!]$ be a function denotation with signature $SIG(FSPEC)(f)$. Let $\eta' \stackrel{\text{def}}{=} \eta[\mathcal{F}[\![\vec{f}]\!](A)/\vec{f}]$*

Then (A, η') models *$P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, written $(A, \eta') \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, if and only if whenever $(A, \eta) \models P$, then for all possible results $r \in \mathcal{M}[\![E]\!](A, \eta')$, if r is proper then $(A, \eta[r/\mathtt{v}]) \models Q$ and there is some type $\mathtt{S}$ such that $r \in \mathtt{S}^A$ and $\mathtt{S} \leq \mathtt{T}$.*

Using the above definition, one can define when a Hoare-triple is valid. Note that to be valid a Hoare-triple for $(SPEC, FSPEC)$ has to be modeled not just by algebra-environment pairs that include all $SPEC$-algebras, but all extended environments in which the meaning of each function that satisfies $FSPEC$ is included.

Definition 6.5 (valid) *The Hoare-triple $P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ is* valid for *$(SPEC, FSPEC)$, written $(SPEC, FSPEC) \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, if and only if for all $SPEC$-algebras A, for all proper $SIG(SPEC)$-environments, $\eta : X \to |A|$ such that X contains the free identifiers of P, E, and Q except $\mathtt{v} : \mathtt{T}$, and for all extensions η' of η that bind each free function identifier f of E to $\mathcal{F}[\![f]\!](A)$, where $\mathcal{F}[\![f]\!]$ is a denotation that satisfies the specification of f in $FSPEC$ with respect to $SPEC$, $(A, \eta') \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$.*

6.4 Soundness Results

The major result in this subsection is a proof that the Hoare Logic for verifying the partial correctness of LOAL programs is sound. That is, we prove that the logic reaches valid conclusions when the specification's $\leq$ relation is a legal subtype relation. The soundness of the Hoare logic justifies our definition of legal subtype relations. Put another way, the reason we care about the soundness of the Hoare logic is that we want a definition of legal subtype relations that makes our style of modular verification work [38].

Completeness, the converse of soundness, is beyond the scope of this paper.

Informally, the soundness of the verification method rests on the syntactic restrictions on sets of type specifications, semantic restrictions, on both legal subtype relations and simulation relations, and the following technical results:

- If for some type $\mathtt{T}$, $q\ \mathcal{R}_{\mathtt{T}}\ r$, then a subtype-constraining assertion P characterizing the value of $\mathtt{x} : \mathtt{T}$ holds when $\mathtt{x}$ is bound to q if and only if P holds when $\mathtt{x}$ is bound to r (Lemma 3.23). This property is ensured by semantic restrictions on $\mathcal{R}$ and is important for the soundness of the rule [mp].

- An expression of nominal type $\mathtt{T}$ can only denote objects of a type $\mathtt{S} \leq \mathtt{T}$ (Lemma 5.6). This is ensured by type checking and the syntactic restrictions on type specifications.

- The soundness of the verification rule [up], which is ensured by dynamic overloading of trait functions (Lemma 6.7). (Type-checking for the [up] rule also depends on restricting the post-condition to be subtype-constraining.)
- Subtype-constraining assertions that can be proved from the traits used in a type specification remain valid when an identifier `x` is allowed to refer to the values of a subtype of the nominal type of `x` (Lemma 6.9). For example, the implication

$$\text{size}(\mathtt{s}) = 1 \wedge (1 \in \mathtt{s}) \Rightarrow (\mathtt{s}\ \text{eqSet}\ \{1\}), \tag{87}$$

  is valid even if the value of `s` is an `Interval`. This is ensured by semantic restrictions on legal subtype relations; it is important for the soundness of the rule [conseq].

After showing the remaining lemmas, we proceed to the main soundness result.

6.4.1 Assertions can be Lifted

It is crucial to the soundness of the [up] rule that when an assertion is true in a model, one can change the types of some its free identifiers to supertypes of their initial types, and the assertion will still be true. A converse also holds. The proviso is that the assertion must still sort-check when the types of the identifiers are changed; hence it must be subtype-constraining so that Lemma 3.10 applies. It is technically convenient to regard the process as moving renamings from a term into the environment; that is, the term (renamed with subtypes for the identifiers), is modeled by the environment, if and only if the environment (extended by binding the previous values to identifiers at the supertype) models the unrenamed term.

Lemma 6.6 *Let Σ be a signature. Let $\leq$ be the subtype relation of Σ. Let C be a Σ-algebra. Let X be a set of identifiers containing $\vec{\mathtt{x}} : \vec{\mathtt{T}}$. Let Y be a set of identifiers containing $\vec{\mathtt{v}} : \vec{\mathtt{S}}$ such that $Y \cup \{\mathtt{x}_i : \mathtt{T}_i\}_i \supseteq X$. Let Q be a Σ-term with free identifiers from X. Let $\eta : Y \to |C|$ be a proper Σ-environment.*

If Q is subtype-constraining and $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$, then $\overline{\eta}[\![Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]]\!] = \overline{\eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]}[\![Q]\!]$.

Proof Sketch: The proof is by induction on the structure of terms. The proof relies on the fact that the value of a term does not depend on nominal types, because of the dynamic overloading of trait functions. ∎

The following lemma is a direct corollary of the above.

Lemma 6.7 *Let Σ be a signature. Let $\leq$ be the subtype relation of Σ. Let C be a Σ-algebra. Let X be a set of identifiers containing $\vec{\mathtt{x}} : \vec{\mathtt{T}}$. Let Y be a set of identifiers containing $\vec{\mathtt{v}} : \vec{\mathtt{S}}$ such that $Y \cup \{\mathtt{x}_i : \mathtt{T}_i\}_i \supseteq X$. Let Q be a Σ-assertion with free identifiers from X. Let $\eta : Y \to |C|$ be a proper Σ-environment.*

If Q is subtype-constraining and $\vec{\mathtt{S}} \leq \vec{\mathtt{T}}$, then $(C, \eta) \models Q[\vec{\mathtt{v}}/\vec{\mathtt{x}}]$ if and only if $(C, \eta[\eta(\vec{\mathtt{v}})/\vec{\mathtt{x}}]) \models Q$. ∎

6.4.2 Provable and Subtype-Constraining Assertions are Valid

Assertions provable from the traits of a specification are only required to be valid in nominal environments, since that is the "standard definition of satisfaction" for traits. (See Sections 3.2.3 and 2.2.1 for more discussion on this point.)

The following lemma says that with a simulation one can construct a nominal environment. Its proof uses the coercion property of a simulation relation.

Lemma 6.8 *Let Σ be a signature. Let C and A be a Σ-algebras. Let X be a set of identifiers. If there is a Σ-simulation relation between C and A, then for all environments $\eta_C : X \to |C|$, there is a nominal environment $\eta_A : X \to |A|$ such that $\eta_C \mathcal{R} \eta_A$.* ∎

The following lemma shows that subtype-constraining assertions that are provable from a specification are valid in all environments, even those that admit subtyping, provided the subtype relation is legal. Recall that assertions are not Hoare-triples, so this is not *the* soundness theorem. Rather it shows that reasoning based on the theory of the supertype's trait is valid in any environment—even those that admit subtyping.

Lemma 6.9 *Let SPEC be a set of type specifications. Let X be a set of identifiers. Let Q be a $SIG(SPEC)$-assertion with free identifiers from X.*

If Q is subtype-constraining, $SPEC \vdash Q$, and $\leq$ is a legal subtype relation on the types of SPEC, then for all SPEC-algebras C and for all proper $SIG(SPEC)$-environments $\eta_C : X \to |C|$, $(C, \eta_C) \models Q$.

Proof: Suppose that $SPEC \vdash Q$ and that $\leq$ is a legal subtype relation.

Let C be a $SPEC$-algebra. Let $\eta_C : X \to |C|$, be a proper $SIG(SPEC)$-environment.

By definition of legal subtype relations, there is some $SPEC$-algebra A such that there is a $SIG(SPEC)$-simulation relation, $\mathcal{R}$, between C and A. By Lemma 6.8, there is some nominal environment $\eta_A : X \to |A|$ so that $\eta_C \mathcal{R} \eta_A$.

Since $SPEC \vdash Q$, and η_A is nominal, by definition of when an algebra satisfies its traits, $\overline{\eta_A}[\![Q]\!] = true$; hence $(A, \eta_A) \models Q$. Since Q is subtype-constraining, by Lemma 3.23, $(C, \eta_C) \models Q$. ∎

6.4.3 Soundness Theorems

The following lemma is the essential step in proving soundness for the Hoare logic. It says that if some Hoare-triple is provable, then it is valid. Soundness for program verification follows directly.

In the proof we only give full details for the interesting cases, that is those rules that are much different from standard Hoare-logic.

Lemma 6.10 *Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let $\leq$ be the subtype relation of $SIG(SPEC)$.*

If $\leq$ is a legal subtype relation on the types of SPEC, then every provable Hoare-triple for $(SPEC, FSPEC)$ is valid.

Proof: (by induction on the length of proof in the Hoare logic.)

Let $P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\} \ Q$ be a Hoare-triple for $(SPEC, FSPEC)$. Suppose that

$$(SPEC, FSPEC) \vdash P \ \{\mathtt{y} : \mathtt{T} \leftarrow E\} \ Q. \tag{88}$$

Let X be a set of identifiers such that X contains all the free identifiers of P and E and Q except $\mathtt{y}$.

For each function identifier f in the domain of $SIG(FSPEC)$, let $\mathcal{F}[\![f]\!]$ be a function denotation such that $\mathcal{F}[\![f]\!]$ satisfies the specification of f in $FSPEC$ with respect to $SPEC$. Given these function denotations and an algebra, B, $\mathcal{F}[\![f]\!](C)$ is a unique set-valued function, and so given any environment η' over B, there is a unique way to extend η' to an environment that is defined on the function identifiers in $FSPEC$; that is, construct $\eta'[\mathcal{F}[\![\vec{f}(B)]\!]/\vec{f}]$.

Since for the given function denotations, in each algebra this expansion is unique, it is not mentioned below.

Another part of the proof not mentioned in each case is that by Lemma 5.6, the type of each possible result is a subtype of the result identifier's nominal type.

Let C be a $SPEC$-algebra and $\eta_C : X \to |C|$ be a proper Σ-environment.

For the basis, the result must be shown for each of the axiom schemes.

- If the proof consists of one of the axiom schemes [ident], [bot], [isDef], or [fcall], then the result follows directly from the hypothesis.

- Suppose the proof consists of an instance of the axiom scheme [mp]:

$$\vdash \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{S}})\ \ \{\mathtt{y} : \mathtt{T} \leftarrow \mathtt{g}(\vec{\mathtt{x}})\}\ \ \mathrm{Post}(\mathtt{g}, \vec{\mathtt{S}}).$$

  Suppose further that $(C, \eta_C) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{S}})$.

  Since $\leq$ is a legal subtype relation, there is some $SPEC$-algebra A, such that there is a $SIG(SPEC)$-simulation relation, $\mathcal{R}$, between C and A. By Lemma 6.8 there is a nominal environment $\eta_A : X \to |A|$ such that $\eta_C\ \mathcal{R}\ \eta_A$.

  Since the assertion $\mathrm{Pre}(\mathtt{g}, \vec{\mathtt{S}})$ must be subtype-constraining, by Lemma 3.23

$$(A, \eta_A) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{S}}). \tag{89}$$

  By definition of LOAL,

$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](C, \eta_C) = \mathtt{g}^C(\eta_C(\vec{\mathtt{x}})) \tag{90}$$
$$\mathcal{M}[\![\mathtt{g}(\vec{\mathtt{x}})]\!](A, \eta_A) = \mathtt{g}^A(\eta_A(\vec{\mathtt{x}})). \tag{91}$$

  By construction of η_A, $\eta_C(\vec{\mathtt{x}})\ \mathcal{R}_{\vec{\mathtt{S}}}\ \eta_A(\vec{\mathtt{x}})$, and thus by the substitution property of simulation relations (see Figure 35):

$$\forall(q \in \mathtt{g}^C(\eta_C(\vec{\mathtt{x}})))\exists(r \in \mathtt{g}^A(\eta_A(\vec{\mathtt{x}})))q\ \mathcal{R}_{\mathtt{T}}\ r. \tag{92}$$

  By definition of when an operation satisfies its specification, for all possible results $r \in \mathtt{g}^A(\eta_A(\vec{\mathtt{x}}))$, $r \neq \bot$ and $(A, \eta_A[r/\mathtt{y}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{S}})$. Since $\mathcal{R}_{\mathtt{T}}$ is bistrict, for all $q \in \mathtt{g}^C(\eta_C(\vec{\mathtt{x}}))$, $q \neq \bot$. Finally, since $\mathrm{Post}(\mathtt{g}, \vec{\mathtt{S}})$ is subtype-constraining, since for each $q \in \mathtt{g}^C(\eta_C(\vec{\mathtt{x}}))$ there is some $r \in \mathtt{g}^A(\eta_A(\vec{\mathtt{x}}))$ such that $q\ \mathcal{R}_{\mathtt{T}}\ r$, and since all such r satisfy the post-condition, $(C, \eta_C[q/\mathtt{y}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{S}})$ by Lemma 3.23.

For the inductive step, suppose that the result holds for all proofs of length less than n. Consider a proof of length $n > 1$. The last step of the proof must be either an axiom or the conclusion of an inference rule. The axioms were covered above, so it remains to deal with the inference rules.

Let the nominal type of the expression in each rule (E) be $\mathtt{T}$. Let C be a $SPEC$-algebra and $\eta_C : X \to |C|$ be a proper environment.

- Suppose the last step is the conclusion of one of the rules: [rename], [call], [equal], or [if]. Then the result is straightforward from the induction hypothesis and the semantics of LOAL.

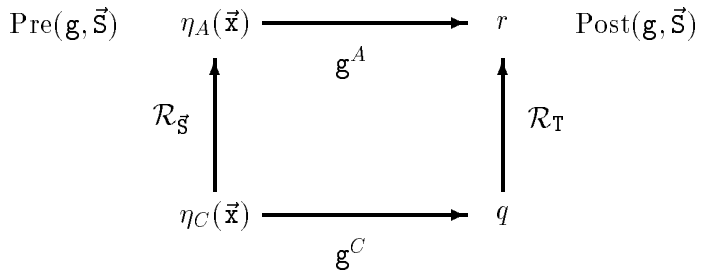

Figure 35: Soundness of the message passing axiom scheme.

- Suppose the last step is the conclusion of the rule [comb]:

$$\vdash P \ \left\{\mathtt{y} \leftarrow (\mathtt{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (E_1, \ldots, E_n)\right\}\ Q.$$

Suppose $(C, \eta_C) \models P$.

By the semantics of LOAL,

$$\mathcal{M}[\![(\mathbf{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (\vec{E})]\!](C, \eta_C) = \bigcup_{\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C,\eta_C)} \mathcal{M}[\![E_0]\!](C, \eta_C[\vec{q}/\vec{\mathtt{x}}]). \tag{93}$$

For each i from 1 to n, there are earlier steps in the proof of the form:

$$\vdash P\ \{\mathtt{x}_i \leftarrow E_i\}\ R_i.$$

By the inductive hypothesis, this Hoare-triple is valid in (C, η_C). Since $(C, \eta_C) \models P$ by hypothesis, for all possible results $q_i \in \mathcal{M}[\![E_i]\!](C, \eta_C)$, if q_i is proper, then $(C, \eta_C[q_i/\mathtt{x}_i]) \models R_i$. By the above for all proper $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C, \eta_C)$,

$$(C, \eta_C[\vec{q}/\vec{\mathtt{x}}]) \models R_1 \wedge \cdots \wedge R_n. \tag{94}$$

There must be some earlier step in the proof of the form

$$\vdash R_1 \wedge \cdots \wedge R_n\ \{\mathtt{y} \leftarrow E_0\}\ Q. \tag{95}$$

The above triple is valid, by the inductive hypothesis; so for all $\vec{q} \in \mathcal{M}[\![\vec{E}]\!](C, \eta_C)$ and for all $r \in \mathcal{M}[\![E_0]\!](C, \eta_C[\vec{q}/\vec{\mathtt{x}}])$, if r is proper then

$$(C, \eta_C[\vec{q}/\vec{\mathtt{x}}][r/\mathtt{y}]) \models Q. \tag{96}$$

Since the $\mathtt{x}_i$ are fresh, they do not appear free in Q, and thus can be dropped from the above formula. Furthermore, by the semantics of LOAL, all the r are possible results of the combination. Thus for all proper $r \in \mathcal{M}[\![(\mathtt{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (\vec{E})]\!](C, \eta_C)$,

$$(C, \eta_C[r/\mathtt{y}]) \models Q. \tag{97}$$

- Suppose the last step is the conclusion of the rule [up]:

$$P\ \{\mathtt{x} : \mathtt{T} \leftarrow E\}\ Q$$

Suppose $(C, \eta_C) \models P$.

There must be an earlier step in the proof of the form

$$P\ \{\mathtt{v} : \mathtt{S} \leftarrow E\}\ Q[\mathtt{v}/\mathtt{x}] \tag{98}$$

where Q is subtype-constraining and $\mathtt{S} \leq \mathtt{T}$. By the inductive hypothesis, this Hoare-triple is valid, so for all $q \in \mathcal{M}[\![E]\!](C, \eta_C)$, if q is proper, then

$$(C, \eta_C[q/\mathtt{v}]) \models Q[\mathtt{v}/\mathtt{x}]. \tag{99}$$

By Lemma 6.7, the renamings on Q can be moved to the environment; so for all proper $q \in \mathcal{M}[\![E]\!](C, \eta_C)$:

$$(C, \eta_C[q/\mathtt{v}][q/\mathtt{x}]) \models Q. \tag{100}$$

By the conditions on the use of [up], $\mathtt{v}$ is fresh; thus $\mathtt{v}$ does not occur free in Q, and so it can be dropped from the environment:

$$(C, \eta_C[q/\mathtt{x}]) \models Q. \tag{101}$$

- Suppose the last step is the conclusion of the rule [conseq]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q.$$

Suppose $(C, \eta_C) \models P$.

There must be steps in the proof of the form $SPEC \vdash P \Rightarrow P_1$ and $SPEC \vdash Q_1 \Rightarrow Q$. In general, P_1 and Q_1 may have more free identifiers than P and Q. For example, the formula "true $\Rightarrow$ $\mathtt{i} = \mathtt{i}$" and its converse are both valid. Let $\vec{\mathtt{z}} : \vec{\mathtt{S}}$ be a tuple of all the free identifiers of P_1 and Q_1 except for the result identifier $\mathtt{y} : \mathtt{T}$ that are not in X (i.e., that are not in the domain of η_C). Let $\vec{q} \in C_{\vec{\mathtt{S}}}$ be a tuple of proper elements. Since $SPEC \vdash P \Rightarrow P_1$ and since P and P_1 are subtype-constraining, by Lemma 6.9 $(C, \eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P \Rightarrow P_1$. (Recall that η_C is not necessarily a nominal environment, so the use of Lemma 6.9 and hence the assumption of subtype-constraining assertions are necessary.) Since $(C, \eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P$, it follows that

$$(C, \eta_C[\vec{q}/\vec{\mathtt{z}}]) \models P_1. \tag{102}$$

There must also be a step in the proof of the form $\vdash P_1\ \{\mathtt{y} \leftarrow E\}\ Q_1$. By the inductive hypothesis and the above, for all $r \in \mathcal{M}[\![E]\!](C, \eta_C[\vec{q}/\vec{\mathtt{z}}])$, if r is proper, then

$$(C, (\eta_C[\vec{q}/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q_1. \tag{103}$$

Since Q_1 and Q are subtype-constraining, by Lemma 6.9, for all proper possible results $r \in \mathcal{M}[\![E]\!](C, \eta_C[\vec{q}/\vec{\mathtt{z}}])$, $(C, (\eta_C[\vec{q}/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q_1 \Rightarrow Q$ and thus

$$(C, (\eta_C[\vec{q}/\vec{\mathtt{z}}])[r/\mathtt{y}]) \models Q. \tag{104}$$

Since the $\mathtt{z}_i$ are not free in Q, for such r,

$$(C, \eta_C[r/\mathtt{y}]) \models Q. \tag{105}$$

- Suppose the last step is the conclusion of the rule [carry]:

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ P \wedge Q.$$

Suppose $(C, \eta_C) \models P$. There must be an earlier step in the proof of the form

$$\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q.$$

By the inductive hypothesis, for all $r \in \mathcal{M}[\![E]\!](C, \eta_C)$, if r is proper then

$$(C, \eta_C[r/\mathtt{y}]) \models Q. \tag{106}$$

Since $(C, \eta_C) \models P$, and $\mathtt{y}$ is not free in P, for such proper r:

$$(C, \eta_C[r/\mathtt{y}]) \models P. \tag{107}$$

Therefore,

$$(C, \eta_C[r/\mathtt{y}]) \models P \wedge Q. \tag{108}$$

■

In the above lemma, and the definition of when a Hoare-triple is valid, one only considers LOAL function denotations that satisfy their specifications. So a first step in proving a program partially correct is to pick function specifications that are satisfied by the functions in a program. Given such specifications, the above result can be easily extended to the soundness of the Hoare logic for proving the partial correctness of LOAL programs.

The specification of a LOAL program of the form

$\vec{F}$; program $(\vec{\mathtt{x}} : \vec{\mathtt{S}})$:T = E

is a Hoare-triple of the form $R\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, where the $\vec{\mathtt{x}} : \vec{\mathtt{S}}$ may be used in the pre-condition R in the post-condition Q. To show the partial correctness of the program, one chooses some set of specifications, *FSPEC*, for the functions in $\vec{F}$, shows that each function satisfies its specification, and proves $(SPEC, FSPEC) \vdash R\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, where *SPEC* is a set of type specifications that includes at least all the types in $\vec{\mathtt{S}}$, T, the types explicitly mentioned in $\vec{F}$ and E, and the types used indirectly by the above.

Definition 6.11 (partial correctness of programs) *Let SPEC be a set of type specifications. Let p be a LOAL program of the form*

$\vec{F}$; program $(\vec{\mathtt{x}} : \vec{\mathtt{S}})$:T = E.

Then p is partially correct with respect to *SPEC* and $R\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$ *if and only if for all SPEC-algebras A, for all proper SIG(SPEC)-environments,* $\eta : X \to |A|$ *such that X contains the free identifiers of R, E, and Q except* $\mathtt{v} : \mathtt{T}$, $(A, \eta[\mathcal{F}[\![\vec{F}]\!](A)/\vec{f}]) \models P\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$.

The following corollary is the soundness result for program verification. It is a trivial consequence of the above lemma and Lemma 5.7.

Theorem 6.12 *Let (SPEC, FSPEC) be a pair of type and function specification sets. Let* $\leq$ *be the subtype relation of SIG(SPEC). Let p be a LOAL program of the form*

$\vec{F}$; program $(\vec{\mathtt{x}} : \vec{\mathtt{S}})$:T = E.

If $\leq$ *is a legal subtype relation on the types of SPEC, if the denotation of each function in* $\vec{F}$ *satisfies its specification in FSPEC with respect to SPEC, and if* $(SPEC, FSPEC) \vdash R\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$, *then p is partially correct with respect to SPEC and* $R\ \{\mathtt{v} : \mathtt{T} \leftarrow E\}\ Q$.
■

6.5 Modularity of Verification

The soundness results of the previous section do not completely vindicate the claim that the Hoare logic allows modular reasoning. The soundness result shows that one can, for a given set of type specifications, reason about a function or program using nominal type information without explicitly considering subtypes. Yet modularity demands that such verifications still be valid when new subtypes are added to a program. The precise notion of extension is given in the following definition.

Definition 6.13 (extends) *Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. The set $SPEC_2$* extends *$SPEC_1$ if and only if the type specifications $SPEC_1$ are included in $SPEC_2$ and $SIG(SPEC_1)$ is a subsignature of $SIG(SPEC_2)$.*

The following lemma shows that the verification of expressions is modular. It states that a verification using a smaller set of specifications necessarily is a verification using an extended set of specifications. In other words, the extended set's theory includes the smaller's theory.

Lemma 6.14 *Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. Let $FSPEC$ be a set of function specifications whose base specification set is contained in $SPEC_1$.*

Suppose that the set of type specifications $SPEC_2$ extends the set $SPEC_1$. Then for all Hoare-triples for $(SPEC_1, FSPEC)$, if

$$(SPEC_1, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\} \ Q,$$

then

$$(SPEC_2, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\} \ Q.$$

Proof: Suppose that $(SPEC_1, FSPEC) \vdash P \ \{\mathtt{y} \leftarrow E\} \ Q$. Since $SPEC_2$ extends $SPEC_1$, each axiom of the pair $(SPEC_1, FSPEC)$ is an axiom of $(SPEC_2, FSPEC)$.

Since the signature $SIG(SPEC_1)$ is a subsignature of $SIG(SPEC_2)$, by Lemma 5.2, the nominal type of each LOAL expression E with respect to $SIG(SPEC_1)$ and $SIG(FSPEC)$ is a supertype of the nominal type of the expression E's nominal type with respect to $SIG(SPEC_2)$ and $SIG(FSPEC)$. So each Hoare-triple for $(SPEC_1, FSPEC)$ is a Hoare-triple for $(SPEC_2, FSPEC)$.

It must also be checked that the type constraints of the Hoare-logic's inference rules are satisfied. The type constraints on the inference rules [rename] and [equal] only ensure that the nominal types of certain identifiers are the same. The nominal types of identifiers do not change when a signature changes. For the inference rule [call], the type constraint ensures that the actual arguments $\vec{E}$ have types $\vec{\sigma}$ with respect to $SIG(SPEC_1)$ and $SIG(FSPEC)$, such that $\vec{\sigma} \leq_1 \vec{\mathtt{S}}$, where $\leq_1$ is the subtype relation of $SIG(SPEC_1)$. By Lemma 5.2, the nominal types of the actual arguments $\vec{E}$ must be some $\vec{\tau} \leq_2 \vec{\sigma}$. Since $\leq_1 \subseteq \leq_2$, $\vec{\tau} \leq_2 \vec{\mathtt{S}}$. Similarly, in the rule [up], the type constraints guarantee that the nominal type of E, σ, is a subtype of $\mathtt{S}$; so by Lemma 5.2 the constraints are satisfied with respect to$SIG(SPEC_2)$. Therefore the proof for the triple is a proof in $(SPEC_2, FSPEC)$. ∎

The story for modularity is not, however, as simple as the above lemma would indicate. The complication is that the implementations of LOAL functions are verified using the smaller type specification set but not reverified using the expanded set of type specifications. (This, of course, is the very essence of modular verification.) Since the verification of recursively defined LOAL functions using the Hoare logic only shows partial correctness,

knowing that proof of partial correctness using the smaller specification set gives a proof of partial correctness for the expanded specification set is not enough to satisfy the conditions of Theorem 6.12. To avoid redoing the proof of termination of recursively defined LOAL functions one needs to know that if a LOAL function satisfies its specification with respect to the smaller set of type specifications, then it satisfies its specification with respect to an expanded set of type specifications.

The following lemma asserts that such problems do not occur for LOAL functions, provided that the new subtype relation is legal. The proof is the source of the restriction that function specifications may only use subtype-constraining assertions.

Lemma 6.15 *Let $SPEC_1$ and $SPEC_2$ be sets of type specifications. Let f be a function specification whose base specification set is contained in $SPEC_1$. Let f be the denotation of a LOAL function definition for f.*

Suppose that $SPEC_2$ extends $SPEC_1$. If the subtype relation $\leq_2$ of $SIG(SPEC_2)$ is a legal subtype relation on the types of $SPEC_2$ and if f satisfies the specification f with respect to $SPEC_1$, then f satisfies the specification f with respect to $SPEC_2$.

Proof: Suppose that f satisfies the specification f with respect to $SPEC_1$. Suppose that the subtype relation $\leq_2$ of $SIG(SPEC_2)$ is a legal subtype relation on the types of $SPEC_2$. Let C be a $SPEC_2$-algebra. Let X be a set of identifiers that includes the formal arguments from the specification of f. Let $\mathtt{S}$ be the nominal result type of f. Let $\eta_C : X \to |C|$ be a proper $SIG(SPEC_2)$-environment. Let R be the pre-condition of f, and let Q be its post-condition and $\mathtt{v}$ the formal result identifier. Suppose that

$$(C, \eta_C) \models R. \tag{109}$$

Let $q \in f(C)(\eta_C(\vec{\mathtt{x}}))$ be a possible result of f.

Since $SPEC_2$ extends $SPEC_1$ and since η_C is a $SIG(SPEC_2)$-environment, it must be that the nominal type of each $\mathtt{x}_i$ is some $\mathtt{T}_i$ and each $\eta_C(\mathtt{x}_i)$ has a type $\mathtt{U}_i$, such that $\mathtt{U}_i \leq \mathtt{T}_i$. Since $\leq_2$ is a legal subtype relation, there must be some $SPEC_2$-algebra A and a $SIG(SPEC_2)$-simulation relation, $\mathcal{R}$, from C to A. Let η_A be a nominal environment, that exists by Lemma 6.8, such that $\eta_C \,\mathcal{R}\, \eta_A$. Let A' be the $SIG(SPEC_1)$-reduct of A. Since η_A is nominal and the base specification of f is contained in $SPEC_1$, the nominal types of the formals of the $\mathtt{x}_i$, the $\mathtt{T}_i$, must be types in $SIG(SPEC_1)$; hence for each i, $\eta_A(\mathtt{x}_i) \in A'_{\mathtt{T}_i}$.

Since f is the denotation of a LOAL function definition and $\mathcal{R}$ is a simulation relation from C to A, by Lemma 5.15, there is some possible result $r \in f(A)(\eta_A(\vec{\mathtt{x}}))$, such that

$$q \;\mathcal{R}_{\mathtt{S}}\; r \tag{110}$$

(where $\mathtt{S}$ is the nominal result type of f). Since R is subtype-constraining, by Lemma 3.23,

$$(A, \eta_A) \models R. \tag{111}$$

Since the function f satisfies its specification with respect to $SPEC_1$, and since $\eta_A(\vec{\mathtt{x}})$ is in the carrier of the $SPEC_1$-algebra A',

$$(A', \eta_A[r/\mathtt{v}]) \models Q. \tag{112}$$

Since A' is a reduct of A, the same holds for A:

$$(A, \eta_A[r/\mathtt{v}]) \models Q. \tag{113}$$

Since Q is subtype-constraining and $\eta_C[q/\mathtt{v}] \mathcal{R} \eta_A[r/\mathtt{v}]$, by Lemma 3.23,

$$(C, \eta_C[q/\mathtt{v}]) \models Q. \tag{114}$$

■

The following corollary is the modularity result for program verification. It may seem that the corollary discusses adding new types to a program and then the new types are never used, because the program is unchanged. However, a LOAL program may take arguments of any type, and so it may have an argument whose nominal type is a supertype of a newly added type. Hence the old program may be passed objects of the new type, which is precisely what programmers are concerned with.

Corollary 6.16 *Let p be a program specification, with pre-condition R, post-condition Q and nominal result type* `S`*. Let $(SPEC_1, FSPEC)$ be a pair of type and function specification sets. Let $SPEC_2$ be an extension of $SPEC_1$. Let $\leq_2$ be the subtype relation of $SIG(SPEC_2)$. Let P be the LOAL program $\vec{F}$;* `program (`$\vec{\mathtt{x}}$`:`$\vec{\mathtt{T}}$`):S =`E *where $\vec{F}$ is a system of mutually recursive LOAL functions whose names and nominal signatures match $SIG(FSPEC)$.*

Suppose that $\leq_2$ is a legal subtype relation on the types of $SPEC_2$ and the denotation of each function in $\vec{F}$ satisfies its specification with respect to $SPEC_1$. If $(SPEC_1, FSPEC) \vdash R\ \{\mathtt{y} \leftarrow E\}\ Q$, then the program P is partially correct with respect to $SPEC_2$ and p. ■

7 Proving Legal Subtyping from Specifications and Soundness of Supertype Abstraction

In this section we study the technical relationship between our definition of legal subtype relations and Meyer's [47, Section 11.1] [48, Sections 10.15 and 10.22] and America's [1] [2] definitions. (Similar definitions were proposed earlier by the designers of Trellis/Owl [55], and also figure in the recent work of Liskov and Wing [44] [45].) We first show how to adapt their technique for proving legal subtype relationships to our formalism. We then show the converse, that legal subtyping implies something like their conditions. The converse also says that supertype abstraction [38] [35] is sound. That is, one can reason about objects of subtypes as if they were objects of a supertype.

7.1 Proving Legal Subtype Relations from Specifications

In this section we show that our definition of legal subtype relations is implied by something like Meyer's and America's. We do this by adapting their proof technique to our formalism, and then showing that this technique is sound for proving legal subtype relations according to our definition.

Both America and Meyer only consider single-dispatch languages; whereas LOAL is a multiple-dispatch language. That is, in a language like CLOS or LOAL, message names are not uniquely tied to one type, since they may dispatch on arguments other than the first. However, the situation is clear for messages that take only one argument. Let $\mathtt{W} \leq \mathtt{U}$; that is, let `W` be a subtype of `U`. If a message name `g` is defined for the supertype `U`, then it will also be defined for the subtype `W`. Meyer and America would require that the post-condition of the subtype, `W`, imply the post-condition of the supertype, `U`. That is written in our notation as follows.

$$SPEC \models (\mathrm{Post}(\mathtt{g}, \mathtt{W})) \Rightarrow (\mathrm{Post}(\mathtt{g}, \mathtt{U})[\mathtt{w}, \mathtt{s}/\mathtt{u}, \mathtt{t}]) \tag{115}$$

Here, we assume that the formal arguments of the specification of g for the supertype are u : U and that the formal result of the supertype is t : T. The renaming of these to the formal arguments and result of the subtype (w : W and s : S) makes the two formulas be in the language of the subtype This renaming serves the same role as America's transfer functions, which we do not need because the trait functions of the supertype must be defined for the subtype. Their requirements for the pre-condition are similar, but in the opposite order.

$$SPEC \models (\mathrm{Pre}(\mathtt{g}, \mathtt{U})[\mathtt{w}/\mathtt{u}]) \Rightarrow (\mathrm{Post}(\mathtt{g}, \mathtt{W})) \tag{116}$$

That such implications only have to hold for objects of the subtype is key [61]. For an example, suppose we had specified the ins operation of the type Interval by including an invariant property of Interval objects in the pre-condition:

op ins(s:Interval, i:Int) **returns**(r:IntSet)
requires (leastElement(s) < greatestElement(s)) ⇒ ((leastElement(s) + 1) ∈ s)
ensures r eqSet (s ∪ {i})

This does not change the behavior of the operation, as the pre-condition is true of all Interval objects "s". Hence, Interval remains a legal subtype of IntSet (in our sense) with this specification. However, with this specification the pre-condition of the ins operation of IntSet (which is implicitly just "true") does not imply the pre-condition of the ins operation of Interval; that is, for all IntSet objects "s" (and for all integers j, k, and n), the following formula is not valid.

$$\mathrm{true} \Rightarrow ((\mathrm{leastElement(s)} < \mathrm{greatestElement(s)}) \Rightarrow ((\mathrm{leastElement(s)} + 1) \in \mathrm{s}))$$

(For a counter-example, let "s" denote the set "{3, 27, 703}".) However, this formula *is* valid for all Interval objects, "s".

To generalize the Meyer/America implications to multiple dispatch, we simply use a tuple of arguments, and hence of argument types: $\vec{\mathtt{W}} \leq \vec{\mathtt{U}}$ replaces $\mathtt{W} \leq \mathtt{U}$. Thus we would write the following for the postconditions.

$$SPEC \models (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{W}})) \Rightarrow (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}, \mathtt{s}/\vec{\mathtt{u}}, \mathtt{t}])$$

We call the formal statement of this requirement for both the pre- and post-conditions the method refinement condition for a set of type specifications (The notation $SPEC \models Q$ means that for all $SPEC$-algebras C, and for all proper environments $\eta : X \to |C|$, such that X includes the free variables of Q, $(C, \eta) \models Q$.)

Definition 7.1 (method refinement condition) *Let SPEC be a set of type specifications. Let* $\leq$ *be the subtype relation of SIG(SPEC).*

Then the method refinement condition holds for *SPEC if and only if for all tuples of sorts* $\vec{\mathtt{W}} \leq \vec{\mathtt{U}}$*, and for all* $\vec{\mathtt{w}} : \vec{\mathtt{W}}$*, for all message names* g*, if* g *is specified with nominal signatures* $\vec{\mathtt{W}} \to \mathtt{W}_{n+1}$ *and* $\vec{\mathtt{U}} \to \mathtt{U}_{n+1}$*, and* $\mathrm{Formals}(\mathtt{g}, \vec{\mathtt{W}}) = (\vec{\mathtt{w}} : \vec{\mathtt{W}}, \mathtt{r} : \mathtt{W}_{n+1})$ *and* $\mathrm{Formals}(\mathtt{g}, \vec{\mathtt{U}}) = (\vec{\mathtt{u}} : \vec{\mathtt{U}}, \mathtt{t} : \mathtt{U}_{n+1})$*, then*

$$SPEC \models (\mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}/\vec{\mathtt{u}}]) \Rightarrow \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{W}}) \tag{117}$$

$$SPEC \models (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{W}})) \Rightarrow (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}, \mathtt{r}/\vec{\mathtt{u}}, \mathtt{t}]). \tag{118}$$

In the above definition, the substitution of $\vec{\mathtt{w}}$ for $\vec{\mathtt{u}}$ and $\mathtt{r}$ for $\mathtt{t}$ makes the post-condition of the "supertype" talk about abstract values of the "subtype".

Instead of using a substitution of subtype identifiers for supertype identifiers, America's formulation of this proof technique uses a "transfer function" that maps the abstract values of a subtype to the abstract values of a supertype. In Larch/LOAL, one can formally specify the abstract values of types, and one can also specify what amounts to a transfer function; for example, the trait function "toSet" can be thought of as a transfer function from `Interval` to `IntSet` abstract values. Unlike America, we do not have to use transfer functions to translate assertions as our requirements on signatures already ensure that the assertions used in a supertype's specification are meaningful for subtype values. However, as the existence of such transfer functions is part of America's technique, the theorem below assumes that such are transfer functions exist.

In the theorem below the assumed transfer functions are named "to" with a subscript indicating what sort they translate to; that is, the theorem assumes that for each sort $\mathtt{S} \leq \mathtt{T}$, and for each q of sort $\mathtt{S}$, "$\mathrm{to}_{\mathtt{T}}(q)$" has sort $\mathtt{T}$. We also write $\mathrm{to}_{\vec{\mathtt{T}}}(\vec{q})$ for the tuple "$\langle \mathrm{to}_{\mathtt{T}_1}(q_1), \ldots, \mathrm{to}_{\mathtt{T}_n}(q_n) \rangle$".

Definition 7.2 (legal system of coercion functions) *Let SPEC be a set of type specifications. In SIG(SPEC), let $\leq$ be the subtype relation, let TFUNS be the set of trait function symbols, and let ResSort be the result sort function.*

Then SPEC has a legal system of coercion functions *if and only if*

- *for each pair of sorts $\mathtt{S} \leq \mathtt{T}$, there is a trait function symbol "$\mathrm{to}_{\mathtt{T}}$" such that*

$$ResSort(\mathrm{to}_{\mathtt{T}}, \mathtt{S}) = \mathtt{T},$$

- *for all $\mathtt{x} : \mathtt{T}$, $SPEC \models \mathrm{to}_{\mathtt{T}}(\mathtt{x}) = \mathtt{x}$, and*
- *for all tuples of sorts $\vec{\mathtt{W}}$ and $\vec{\mathtt{U}}$ such that $\vec{\mathtt{W}} \leq \vec{\mathtt{U}}$, and for all $\vec{\mathtt{w}} : \vec{\mathtt{W}}$, for all trait function symbols* f *such that $ResSort(\mathrm{f}, \vec{\mathtt{U}})$ is defined,*

$$SPEC \models \mathrm{to}_{ResSort(\mathrm{f},\vec{\mathtt{U}})}(\mathrm{f}(\vec{\mathtt{w}})) = \mathrm{f}(\mathrm{to}_{\vec{\mathtt{U}}}(\vec{\mathtt{w}})). \tag{119}$$

The requirement of a legal system of coercion functions resembles requirements used by Reynolds [53] and Bruce and Wegner [6]. America does not have these exact requirements, because his definition of subtyping is not concerned with modular specification, and thus makes no restrictions on how the abstract values of types are specified. Bruce and Wegner have an additional condition that the coercion functions must compose properly:

- for each triple of types $\mathtt{S} \leq \mathtt{T} \leq \mathtt{U}$, and for all $\mathtt{x} : \mathtt{S}$, $SPEC \models \mathrm{to}_{\mathtt{U}}(\mathrm{to}_{\mathtt{T}}(\mathtt{x})) = \mathrm{to}_{\mathtt{U}}(\mathtt{x})$.

However, we do not need this for the theorem below. Note also that since the "$\mathrm{to}_{\mathtt{T}}$" are trait functions, if A is a $SPEC$-algebra, $\mathtt{S} \leq \mathtt{T}$, $\mathtt{U} \leq \mathtt{T}$, and $q \in A_{\mathtt{S}} \cap A_{\mathtt{U}}$, then $\mathrm{to}_{\mathtt{T}}{}^{A}(q)$ is well-defined (by the definition of an algebra) and does not depend on which sort q is considered to have.

In Larch/LOAL, one specifies a simulation relation when one specifies a set of types (see Example 4.2). In many cases these simulation relationships are functional. For example, if in the specification of the type `Interval` we wrote the following

subtype of IntSet **by** $[l, u]$ **simulates** toSet($[l, u]$)

In this case "toSet" is a coercion function. In this section we systematically name such functions like "$\mathrm{to}_{\mathtt{IntSet}}$". The subscript on "$\mathrm{to}_{\mathtt{IntSet}}$" plays exactly the same role as the

subscript on a simulation relation, such as $\mathcal{R}_{\texttt{IntSet}}$. Recall that, since a simulation relation has a coercion property, it can be regarded as a generalization of a coercion function.

The construction in the following theorem's proof does not work for proving arbitrary subtype relationships. In essence, it only works for subtype relations where no supertype has more than one direct subtype. (A sort S is a *direct subtype* of T if $\texttt{S} \leq \texttt{T}$, $\texttt{S} \neq \texttt{T}$, and there is no other sort U such that $\texttt{U} \neq \texttt{S}$, $\texttt{U} \neq \texttt{T}$, and $\texttt{S} \leq \texttt{U} \leq \texttt{T}$.) The technical condition is stated in terms of operation specifications; we require that the operations of a specification are such that for each message name g, the nominal signatures of specifications of g must form chains in the $\leq$ ordering on the argument sorts.

Definition 7.3 (separate chains of operation specifications) *Let SPEC be a set of type specifications. Let ResSort and $\leq$ be the result sort mapping and subtype relation of SIG(SPEC).*

Then SPEC has separate chains of operation specifications *if and only if for each message name* g, *if* g *is specified with nominal signature* $\vec{\texttt{T}} \rightarrow \texttt{T}_{n+1}$, *then the set of all nominal signatures* $\vec{\texttt{S}} \rightarrow \texttt{S}_{n+1}$ *such that* g *is specified with nominal signature* $\vec{\texttt{S}} \rightarrow \texttt{S}_{n+1}$ *in SPEC,* $\vec{\texttt{S}} \leq \vec{\texttt{T}}$, *and* $\vec{\texttt{S}} \neq \vec{\texttt{T}}$ *is either empty or has a unique greatest element in the ordering $\leq$ applied to the tuple of argument sorts.*

The following theorem is the promised one that shows that if one can prove a subtype relation is legal according to Meyer's [47] [48] and America's definition [2], then the subtype relation is a legal subtype relation by our definition. As stated above, the proof is not completely general, but does give at least one way to construct the appropriate algebra and simulation relation from the assumed method refinement condition and the assumed coercion (transfer) functions.

Theorem 7.4 *Let SPEC be a set of Larch/LOAL type specifications; recall that the assertions in a Larch/LOAL specification are subtype-constraining. Let $\leq$ be the subtype relation of SIG(SPEC).*

If SPEC has separate chains of operation specifications, the method refinement condition holds for SPEC, and SPEC has a legal system of coercion functions, then $\leq$ is a legal subtype relation on the types of SPEC.

Proof: Let C be a *SPEC*-algebra. To show that $\leq$ is a legal subtype relation, we will construct a *SIG(SPEC)*-algebra A and a simulation relation from C to A. The construction uses the system of coercion functions to define the methods of A and to define a simulation relation. After that we show that A satisfies *SPEC*.

Let $|A| = |C|$; let the trait functions of A be the same as in C.

Let *ResSort* be the result sort mapping of *SIG(SPEC)*.

For each message name g, let $\texttt{g}^A$ be defined inductively as follows. By hypothesis there are separate chains of operation specifications of g. The inductive definition is by induction on each chain, starting from the bottom of the chain and going up. For the basis, suppose that the lowest specification in the chain for g has nominal signature $\vec{\texttt{W}} \rightarrow \texttt{W}_{n+1}$. Then for all $\vec{q} \in \mathit{Below}(A, \vec{\texttt{W}})$, let $\texttt{g}^A(\vec{q}) \stackrel{\text{def}}{=} \texttt{g}^C(\vec{q})$. For the inductive step, suppose that there is an operation specification of g with nominal signature $\vec{\texttt{W}} \rightarrow \texttt{W}_{n+1}$, and that $\texttt{g}^A$ has been defined for arguments in $\mathit{Below}(A, \vec{\texttt{W}})$, and the next highest operation specification of g in the chain has nominal signature $\vec{\texttt{U}} \rightarrow \texttt{U}_{n+1}$. (It follows that $\vec{\texttt{W}} \leq \vec{\texttt{U}}$.) Suppose $\vec{\texttt{V}} \neq \vec{\texttt{W}}$ is a tuple of types

such that $\vec{\mathtt{W}} \leq \vec{\mathtt{V}} \leq \vec{\mathtt{U}}$. Let $\vec{q} \in A_{\vec{\mathtt{V}}}$ be given. We define $\mathtt{g}^A(\vec{q})$ by the following.

$$\mathtt{g}^A(\vec{q}) \stackrel{\text{def}}{=} \begin{cases} \mathtt{g}^C(\vec{q}) & \text{if } \forall \vec{w} \in C_{\vec{\mathtt{W}}} \,.\, \mathrm{to}_{\vec{\mathtt{V}}}{}^C(\vec{w}) \neq \vec{q} \\ \bigcup_{\vec{w} \text{ s.t. } \mathrm{to}_{\vec{\mathtt{V}}}{}^C(\vec{w}) = \vec{q}} \mathrm{to}_{\mathit{ResSort}(\mathtt{g},\vec{\mathtt{V}})}{}^C(\mathtt{g}^C(\vec{w})), & \text{otherwise} \end{cases} \tag{120}$$

This completes the definition of the $SIG(SPEC)$-algebra A.

A simulation relation from C to A, $\mathcal{R}$, is defined as follows. For each sort $\mathtt{T}$, we define $\mathcal{R}_{\mathtt{T}}$ by the following.

$$q \,\mathcal{R}_{\mathtt{T}}\, r \stackrel{\text{def}}{\Leftrightarrow} (q \in C_{\mathtt{S}}) \wedge (r \in A_{\mathtt{U}}) \wedge (\mathtt{S} \leq \mathtt{T}) \wedge (\mathtt{U} \leq \mathtt{T}) \wedge (\mathtt{S} \leq \mathtt{U}) \wedge (\mathrm{to}_{\mathtt{U}}{}^C(q) = r) \tag{121}$$

Recall that $\mathrm{to}_{\mathtt{U}}{}^C(q) = r$ is in $A_{\mathtt{U}}$ because the carrier sets of C and A are the same.

By construction, $\mathcal{R}$ satisfies the subsorting, coercion, bistrict, and V-identical properties of simulation relations. To show that $\mathcal{R}$ satisfies the substitution property, let a sort $\mathtt{T}$, a tuple of sorts $\vec{\mathtt{S}}$, and tuples $\vec{q} \in \mathit{Below}(C, \vec{\mathtt{S}})$, and $\vec{r} \in \mathit{Below}(A, \vec{\mathtt{S}})$ be given. Suppose $\vec{q}\,\mathcal{R}_{\vec{\mathtt{S}}}\,\vec{r}$. Then by definition of $\mathcal{R}_{\vec{\mathtt{S}}}$, there are tuples of types, $\vec{\mathtt{W}} \leq \vec{\mathtt{S}}$ and $\vec{\mathtt{U}} \leq \vec{\mathtt{S}}$ such that $\vec{q} \in C_{\vec{\mathtt{W}}}$ and

$$\mathrm{to}_{\vec{\mathtt{U}}}{}^C(\vec{q}) = \vec{r}. \tag{122}$$

There are two cases.

- Let f be a trait function symbol such that $\mathit{ResSort}(\mathtt{f}, \vec{\mathtt{S}}) = \mathtt{T}$. We show that the substitution property holds for "f" by the following calculation. The first formula below holds by Formula (119) in the definition of a legal system of coercion functions.

  $\mathrm{to}_{\mathit{ResSort}(\mathtt{f},\vec{\mathtt{U}})}{}^C(\mathtt{f}^C(\vec{q})) = \mathtt{f}^C(\mathrm{to}_{\vec{\mathtt{U}}}{}^C(\vec{q}))$
  $\Leftrightarrow$ ⟨by Equation (122)⟩
  $\mathrm{to}_{\mathit{ResSort}(\mathtt{f},\vec{\mathtt{U}})}{}^C(\mathtt{f}^C(\vec{q})) = \mathtt{f}^C(\vec{r})$
  $\Leftrightarrow$ ⟨by construction $\mathtt{f}^C = \mathtt{f}^A$⟩
  $\mathrm{to}_{\mathit{ResSort}(\mathtt{f},\vec{\mathtt{U}})}{}^C(\mathtt{f}^C(\vec{q})) = \mathtt{f}^A(\vec{r})$
  $\Leftrightarrow$ ⟨by Formula (121), as $\mathit{ResSort}(\mathtt{f}, \vec{\mathtt{U}}) \leq \mathtt{T}$⟩
  $\mathtt{f}^C(\vec{q})\,\mathcal{R}_{\mathtt{T}}\,\mathtt{f}^A(\vec{r})$

- Let $\mathtt{g}$ be a message name such that $\mathit{ResSort}(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{T}$. Let $q' \in \mathtt{g}^C(\vec{q})$ be given. By the definition of $SIG(SPEC)$-algebras, q' has some type $\mathtt{V} \leq \mathtt{T}$. By Equation (122) and the construction of $\mathtt{g}^A$, $\mathrm{to}_{\mathtt{T}}(q') \in \mathtt{g}^A(\vec{r})$. So by Formula (121), $\mathtt{g}^C(\vec{q})\,\mathcal{R}_{\mathtt{T}}\,\mathtt{g}^A(\vec{r})$.

This completes the proof that $\mathcal{R}$ is a simulation relation.

We now show that A is a $SPEC$-algebra. The trait structure of A satisfies the traits of $SPEC$, because C is a $SPEC$-algebra and the trait structures of C and A are the same. So it remains to show that the methods of A satisfy their specifications.

Let $\mathtt{g}$ be message name. Let $\vec{\mathtt{S}}$ be a tuple of types such that $\mathit{ResSort}(\mathtt{g}, \vec{\mathtt{S}}) = \mathtt{T}$. Let the formal arguments of the most specific applicable specification for $\vec{\mathtt{S}}$ in $SPEC$ be $\vec{\mathtt{u}} : \vec{\mathtt{U}}$ and the formal result be $\mathtt{t} : \mathtt{T}$. Let $\vec{q} \in A_{\vec{\mathtt{S}}}$ be proper. Let the environment $\eta_A : \{\vec{\mathtt{u}} : \vec{\mathtt{U}}\} \to |A|$ be defined such that $\eta_A(\vec{\mathtt{u}}) \stackrel{\text{def}}{=} \vec{q}$. We proceed by induction on the chain used to define $\mathtt{g}^A$.

If $\vec{\mathtt{U}}$ is at the bottom of the chain used to define $\mathtt{g}^A$, then $\mathtt{g}^A(\vec{q}) = \mathtt{g}^C(\vec{q})$ and since C is a $SPEC$-algebra, if $(A, \eta_A) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})$, then for all possible results $r' \in \mathtt{g}^A(\vec{q})$: $r' \neq \bot$, $(A, \eta_A[r'/\mathtt{t}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$.

For the inductive step, there is some $\vec{\mathtt{W}} \leq \vec{\mathtt{U}}$ such that there is an operation specification of $\mathtt{g}$ with nominal signature $\vec{\mathtt{W}} \to \mathtt{W}_{n+1}$, formal arguments $\vec{\mathtt{w}} : \vec{\mathtt{W}}$, and formal result $\mathtt{r} : \mathtt{W}_{n+1}$. The inductive hypothesis is that $\mathtt{g}^A$ satisfies the operation specification with nominal signature $\vec{\mathtt{W}} \to \mathtt{W}_{n+1}$. There are two cases.

If there is no $\vec{w} \in C_{\vec{\mathtt{W}}}$ such that $\mathrm{to}_{\vec{\mathtt{S}}}(\vec{w}) = \vec{q}$, then by construction, $\mathtt{g}^A(\vec{q}) = \mathtt{g}^C(\vec{q})$ and again we are done.

Otherwise, let $\vec{w} \in C_{\vec{\mathtt{W}}}$ be such that $\mathrm{to}_{\vec{\mathtt{S}}}(\vec{w}) = \vec{q}$. Suppose

$$(A, \eta_A) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}}). \tag{123}$$

Our plan is to descend to C and the subtype, use the method refinement hypothesis to show the precondition for $\mathtt{g}$ there, use the assumption that C is a *SPEC*-algebra to conclude the post-condition, and then use the second part of the method refinement hypothesis.

Let the environment $\eta_C : \{\vec{\mathtt{u}} : \vec{\mathtt{U}}\} \to |C|$ be defined such that $\eta_C(\vec{\mathtt{u}}) \stackrel{\mathrm{def}}{=} \vec{w}$. Let $\eta'_C \stackrel{\mathrm{def}}{=} \eta_C[\vec{\mathtt{w}}/\vec{w}]$. The following calculation, starting from the assumption that the precondition of $\mathtt{g}$ for $\vec{\mathtt{U}}$ (the supertype) is modeled by (A, η_A), shows that the precondition for $\vec{\mathtt{W}}$ (the subtype) is modeled by (C, η'_C).

$(A, \eta_A) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})$
$\Leftrightarrow$ ⟨by Lemma 3.23, as $\eta_C \mathcal{R} \eta_A$ and $\mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})$ is subtype-constraining⟩
$(C, \eta_C) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})$
$\Leftrightarrow$ ⟨by the agreement of η_C and $\eta'_C[\eta'_C(\vec{\mathtt{w}})/\vec{\mathtt{u}}]$ on the free variables of $\mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})$⟩
$(C, \eta'_C[\eta'_C(\vec{\mathtt{w}})/\vec{\mathtt{u}}]) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})$
$\Leftrightarrow$ ⟨by Lemma 6.7, as the assertions are subtype-constraining⟩
$(C, \eta'_C) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}/\vec{\mathtt{u}}]$
$\Rightarrow$ ⟨by assumption that $(C, \eta'_C) \models (\mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}/\vec{\mathtt{u}}]) \Rightarrow \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{W}})$⟩
$(C, \eta'_C) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{W}})$

Since C satisfies *SPEC*, and $(C, \eta'_C) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{W}})$ for each $r \in \mathtt{g}^C(\vec{w})$ it must be that $r \neq \perp$, and

$$(C, \eta'_C[r/\mathtt{r}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{W}}). \tag{124}$$

In this case, by construction of $\mathtt{g}^A$, each $r' \in \mathtt{g}^A(\vec{q})$, is such that $r \mathcal{R}_{\mathtt{T}} r'$, for some such $r \in \mathtt{g}^C(\vec{w})$. Let such an r and r' be given.

Starting from Formula (124), we show that the post-condition of $\mathtt{g}$ for $\vec{\mathtt{U}}$ (the supertype) is modeled by $(A, \eta_A[r'/\mathtt{t}])$.

$(C, \eta'_C[r/\mathtt{r}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{W}})$
$\Rightarrow$ ⟨by assumption that $(C, \eta'_C[r/\mathtt{r}]) \models (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{W}})) \Rightarrow (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}, \mathtt{r}/\vec{\mathtt{u}}, \mathtt{t}])$⟩
$(C, \eta'_C[r/\mathtt{r}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})[\vec{\mathtt{w}}, \mathtt{r}/\vec{\mathtt{u}}, \mathtt{t}]$
$\Leftrightarrow$ ⟨by Lemma 6.7, as the assertions are subtype-constraining⟩
$(C, (\eta'_C[r/\mathtt{r}])[\eta'_C[r/\mathtt{r}](\vec{\mathtt{w}}), \eta'_C[r/\mathtt{r}](\mathtt{r})/\vec{\mathtt{u}}, \mathtt{t}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$
$\Leftrightarrow$ ⟨by $\eta'_C[r/\mathtt{r}](\mathtt{r}) = r$, $\eta'_C[r/\mathtt{r}](\vec{\mathtt{w}}) = \vec{w} = \eta_C(\vec{\mathtt{u}})$, and $\mathtt{r}$ is not free in $\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$⟩
$(C, \eta_C[r/\mathtt{t}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$
$\Leftrightarrow$ ⟨by Lemma 3.23 as $\eta_C[r/\mathtt{t}] \mathcal{R} \eta_A[r'/\mathtt{t}]$, and $\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$ is subtype-constraining⟩
$(A, \eta_A[r'/\mathtt{t}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$

Since the last formula holds for all $r' \in \mathtt{g}^A(\vec{q})$, A satisfies *SPEC*. ∎

7.2 Soundness of Supertype Abstraction

In this brief section we show that supertype abstraction [38] [35] is sound. That is, one can reason about objects of subtypes as if they were objects of a supertype.

The soundness of supertype abstraction also bears on the question of the extent to which our definition of legal subtype relations implies something like Meyer's [47] [48] and America's definitions [2] of subtypes? Are our definitions fundamentally equivalent?

We believe that the answers to the above questions are "yes". One way to try to show this is to try to prove an implication that joins the pre- and post-conditions of the subtype and the supertype, for each operation of the supertype. For example, let $\vec{\mathtt{W}} \leq \vec{\mathtt{U}}$, and let $\mathtt{g}$ be specified with nominal signature $\vec{\mathtt{U}} \to \mathtt{U}_{n+1}$; thus $\mathtt{g}$ must also be specified with a nominal signature $\vec{\mathtt{W}} \to \mathtt{W}_{n+1}$. Suppose that the formals of these operation specifications are $\text{Formals}(\mathtt{g}, \vec{\mathtt{W}}) = \vec{\mathtt{w}} : \vec{\mathtt{W}}, \mathtt{s} : \mathtt{W}_{n+1}$, and $\text{Formals}(\mathtt{g}, \vec{\mathtt{U}}) = \vec{\mathtt{u}} : \vec{\mathtt{U}}, \mathtt{t} : \mathtt{U}_{n+1}$. Then one might think that a translation of America's implications would be something like the following, for each such $\mathtt{g}$.

$$SPEC \models (\text{Pre}(\mathtt{g}, \vec{\mathtt{W}}) \Rightarrow \text{Post}(\mathtt{g}, \vec{\mathtt{W}})) \Rightarrow ((\text{Pre}(\mathtt{g}, \vec{\mathtt{U}}) \Rightarrow \text{Post}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}, \mathtt{s}/\vec{\mathtt{u}}, \mathtt{t}]), \tag{125}$$

A similar condition figures in the work of Liskov and Wing [44] (at our suggestion).

However, the following corollary does not prove Formula (125). The problem is that Formula (125) does not require $\mathtt{s}$ to be a possible result of a call of $\mathtt{g}$, and our model-theoretic conditions on legal subtype relations seem to require that. So the following corollary uses Hoare-triples, instead of just an implication. Translating Formula (125) into the appropriate Hoare-triples, one obtains something like the following inference rule, which one would like to be valid:

$$\frac{(SPEC, FSPEC) \models \text{Pre}(\mathtt{g}, \vec{\mathtt{W}}) \ \{\mathtt{s} : \mathtt{W}_{n+1} \leftarrow \mathtt{g}(\vec{\mathtt{w}})\} \ \text{Post}(\mathtt{g}, \vec{\mathtt{W}})}{(SPEC, FSPEC) \models (\text{Pre}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}/\vec{\mathtt{u}}] \ \{\mathtt{s} : \mathtt{W}_{n+1} \leftarrow \mathtt{g}(\vec{\mathtt{w}})\} \ (\text{Post}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}, \mathtt{s}/\vec{\mathtt{u}}, \mathtt{t}]} \tag{126}$$

However, as the hypothesis is an instance of the axiom scheme [mp], in view of Lemma 6.10 the validity of the rule reduces to the validity of the conclusion.[7] Thus the following corollary says that having a legal subtype relationship is similar to having the method refinement condition hold. Informally, another way to read it is that it says a legal subtype relationship allows one to use the specifications of supertypes to reason about subtypes; that is, supertype abstraction [38] is valid.

Corollary 7.5 *Let $(SPEC, FSPEC)$ be a pair of type and function specification sets. Let $\leq$ be the subtype relation of $SIG(SPEC)$. Let $\mathtt{g}$ be a message name of $SIG(SPEC)$. Let $\vec{\mathtt{W}}$ and $\vec{\mathtt{U}}$ be tuples of types, and let $\mathtt{W}_{n+1}$ and $\mathtt{U}_{n+1}$ be types such that: $\mathtt{g}$ is specified in SPEC with nominal signatures $\vec{\mathtt{W}} \to \mathtt{W}_{n+1}$ and $\vec{\mathtt{U}} \to \mathtt{U}_{n+1}$. Let $\text{Formals}(\mathtt{g}, \vec{\mathtt{W}}) = \vec{\mathtt{w}} : \vec{\mathtt{W}}, \mathtt{s} : \mathtt{W}_{n+1}$, and $\text{Formals}(\mathtt{g}, \vec{\mathtt{U}}) = \vec{\mathtt{u}} : \vec{\mathtt{U}}, \mathtt{t} : \mathtt{U}_{n+1}$. Suppose $\mathtt{s} : \mathtt{W}_{n+1}$ does not appear free in $\text{Post}(\mathtt{g}, \vec{\mathtt{U}})$, and is not equal to any of the $\mathtt{u}_i$ in $\vec{\mathtt{u}}$.*

If $\leq$ is a legal subtype relation on the types of SPEC, and if $\vec{\mathtt{W}} \leq \vec{\mathtt{U}}$, then

$$(SPEC, FSPEC) \models (\text{Pre}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}/\vec{\mathtt{u}}] \ \{\mathtt{s} : \mathtt{W}_{n+1} \leftarrow \mathtt{g}(\vec{\mathtt{w}})\} \ (\text{Post}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}, \mathtt{s}/\vec{\mathtt{u}}, \mathtt{t}]$$

Proof: Suppose $\leq$ is a legal subtype relation on the types of $SPEC$. Let A be a $SPEC$-algebra. Let $\eta : Y \to |A|$ be a $SIG(SPEC)$-environment, such that the free variables of above Formula are in Y.

[7] Also, it is interesting to note that if the method refinement condition does hold for $SPEC$, then the conclusion follows immediately from the soundness theorem and the [conseq] rule.

The conclusion is a Hoare-triple, that is, it type-checks, because of the assumptions in the statement of the theorem about the type of the formals. We proceed to show the definition of $\models$ for Hoare-triples.

Suppose the following holds (otherwise we are done).

$$(A, \eta) \models (\mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}/\vec{\mathtt{u}}] \tag{127}$$

Since the assertions in a type specification are subtype-constraining, by Lemma 6.7, the renamings on the precondition can be moved to the environment.

$$(A, \eta[\eta(\vec{\mathtt{w}})/\vec{\mathtt{u}}]) \models \mathrm{Pre}(\mathtt{g}, \vec{\mathtt{U}}) \tag{128}$$

Since $\leq$ is a legal subtype relation on the types of $SPEC$, by the soundness theorem it follows that for all $s \in \mathtt{g}^A(\eta(\vec{\mathtt{w}}))$, if s is proper, then

$$(A, (\eta[\eta(\vec{\mathtt{w}})/\vec{\mathtt{u}}])[s/\mathtt{t}] \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}}). \tag{129}$$

Since $\mathtt{s}$ does not appear free in $\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}})$, and $\mathtt{s}$ is not equal to any of the $\mathtt{w}_i$ in $\vec{\mathtt{w}}$ (since the formal result has to be a distinct variable for $\vec{\mathtt{W}}$) or the $\mathtt{u}_i$ in $\vec{\mathtt{u}}$, for all proper $s \in \mathtt{g}^A(\eta(\vec{\mathtt{w}}))$,

$$\begin{aligned} (\eta[\eta(\vec{\mathtt{w}})/\vec{\mathtt{u}}])[s/\mathtt{t}] &= \eta[\eta(\vec{\mathtt{w}}), s/\vec{\mathtt{u}}, \mathtt{t}] && (130)\\ &= (\eta[s/\mathtt{s}])[\eta[s/\mathtt{s}](\vec{\mathtt{w}}), \eta[s/\mathtt{s}](\mathtt{s})/\vec{\mathtt{u}}, \mathtt{t}]. && (131) \end{aligned}$$

Thus, it follows that:

$$(A, (\eta[s/\mathtt{s}])[\eta[s/\mathtt{s}](\vec{\mathtt{w}}), \eta[s/\mathtt{s}](\mathtt{s})/\vec{\mathtt{u}}, \mathtt{t}]) \models \mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}}) \tag{132}$$

Again since the assertions in a type specification are subtype-constraining, by Lemma 6.7, the renamings on the precondition can be out of the environment, and so the following holds.

$$(A, \eta[s/\mathtt{s}]) \models (\mathrm{Post}(\mathtt{g}, \vec{\mathtt{U}}))[\vec{\mathtt{w}}, \mathtt{s}/\vec{\mathtt{u}}, \mathtt{t}] \tag{133}$$

■

8 Discussion

The main contribution of this paper is a new technique for the modular verification of object-oriented programs that use message passing and subtype polymorphism.

For the practicing programmer, the main lesson of this work is that, with some discipline, one can reason about programs by using nominal (i.e., static) type information and letting supertypes stand for their subtypes. We call this kind of reasoning *supertype abstraction*. We believe that good object-oriented programmers use supertype abstraction to reason about their programs, hence they speak of protocols common to different types of objects [22].

The main pitfall in common use of supertype abstraction is failing to ensure that one's subtype relation is legal. Having a legal subtype relation allows one to safely use supertype abstraction in reasoning about a program that uses message passing. So programmers should check to be sure that they only use legal subtype relations, even if they only do so informally. Programmers who are not familiar with the ideas of specifications and abstract data types often equate the notions of inheritance and subtyping, which can lead to using a subtype relation that is illegal. The problem is that inherited code can be redefined and changed in ways that do not respect the interface specification of a superclass. On the other hand, a clear separation of the notions of inheritance and subtyping is a tool of great conceptual power in object-oriented programming, since it allows one to use inheritance to implement types in the most economical manner, and use subtyping to organize and reason about the use of types [58] [31] [30] [42] [15] [2].

The distinction between subtypes and subclasses is not just academic. If one passes an argument whose type is not a subtype of the expected formal argument type to a procedure, the procedure will not act as desired. So if one uses subclasses as if they always implemented subtypes, one's programs may behave in unexpected ways.

With supertype abstraction both informal reasoning about programs and formal verification is modular. That is, one can add new types to a program without rethinking or reverifying unchanged parts of the program [34] [39]. The only qualitative difference from standard program verification is that the verifier must also show that the specified subtype relation is legal.

For the practicing software designer, the main lesson of this work is that one can use subtype polymorphism to write polymorphic specifications. This style of specification does not require pre-planning for polymorphism. Our technique for modular specifications is to specify subtypes in such a way that the vocabulary used to state pre- and post-conditions in their supertypes is also meaningful for the subtypes. This ensures that when a new type is added to a specification, existing specifications do not have to be changed.

The main theoretical results in this work are our specification and verification techniques and the proof that the verification technique is sound and modular. These are the first such formal verification techniques for object-oriented programs with subtypes and message passing.

The most important property of the definition of legal subtype relations is that it allows *abstract* types to be compared, based on their specifications. This is in contrast with the work of Cardelli and others, which only described subtype relationships for a syntactically characterized set of built-in types [8].[8] Cardelli's landmark paper gives rules for what

[8] While *existential types* used by Mitchell, Cardelli and others [50] [10] [9] are sometimes said to describe abstract data types, they do not characterize the behavior of such types. An existential type only describes the syntactic interface of an abstract data type. Subtyping among the witness types (of existential types)

subtype relationships hold among the built-in types of a small programming language: immutable records, variants, and higher-order functions. Although we do not handle higher-order types, for first-order types our definition of legal subtype relations is more general, since it applies to abstract data types. (An extension of our definition to higher-order types is described in [37].) Like Cardelli, we have also attempted to give a definition of subtype relationships that is theoretically justified, instead of just appealing to intuition.

8.1 Limitations and Open Problems

Our theoretical results have certain formal limitations:

- The verification technique: only handles LOAL client programs (not the implementations of types), does not handle mutation or assignment, and has not been proved relatively complete.

- The specification language: cannot specify types with mutable objects, and cannot handle higher-order data types.

However, these formal limitations should not all be taken as fundamental weaknesses. We believe that our techniques can be extended to handle mutation and assignment, and work is in progress to do so [16] [17]. We have also extended these techniques to higher-order functions [37]. The main limitation to bear in mind is that the verification technique has not been proved relatively complete [13] [46, Section 8.2]. This means, for example, that ours may not be the only way to prove the correctness of object-oriented programs, and that, even for types with immutable objects, ours may not be the best definition of legal subtype relations. Indeed, a definition that says that there should be no way to get surprising behavior from a legal subtype is more general and certainly more fundamental [40] [38] [34, Chapter 7] [37]. Nevertheless, our soundness results and Corollary 7.5 show that a legal subtype cannot exhibit surprising behavior.

In addition to the removing the limitations mentioned above, we suggest the following as interesting open problems. There are several issues relating to classes that we ignored. For example, one would like to be able to specify the behavior of classes to compare them with types. One would also like to specify enough about a class so that a subclass could be programmed without looking at the code of the superclass (see [32]). In verification, one would like to use the fact that a subtype is implemented by a subclass (of a class that implements the supertype) to help prove legal subtyping.

8.2 Subtyping versus Refinement

One justification for subtype relationships might be that they are similar to refinement relationships among specifications. A type `S` is *a refinement of* `T` if every implementation of the specification of `S` is an implementation of the specification of `T`. For example, a type `LFPSchd`, which is like `IntSet` except that the `choose` operation is specified to return the least element of the set, would be a refinement of `IntSet`, because each implementation would also satisfy the specification of `IntSet`. However, a type `S` may be a subtype of `T`, even though `S` is not a refinement of `T`.

must be declared through the use of bounded existential types. When to declare such relationships is a question of legal subtyping in our sense.

The difference between a subtype relationship and a refinement relationship is due to the distinction between class and the instance operations[9]. For a subtype relationship only instance operations and the abstract values of objects that can be created by the class operations matter. This is because the behavior of an object, once it has been created, is only determined by its value as observed by its instance operations. For a refinement relationship, both the class and the instance operations matter, since an implementation of a type includes both kinds of operations. For example, `Interval` is a subtype of `IntSet`, even though the specification of `Interval` has a class operation named `create` instead of a class operation named `null` as in `IntSet`. However, because of this difference in class operations, `Interval` is not a refinement of `IntSet`. On the other hand, whenever `S` is a refinement of `T`, then `S` must be a subtype of `T`.

In his work on subtyping, America does not make a distinction between refinement and subtyping [2]. This is because in his language POOL-I, a type consists only of instance operations. Classes implement types and have class operations, but there is no hard and fast link between classes and types, since a type can be implemented by more than one class. Indeed, as POOL-I is not itself a specification language, there is no comparable semantics to ensure that all the classes that purport to implement a type do so collectively, not just individually. We believe that it is important to make a conceptual distinction between refinement and subtyping.

In Larch/LOAL, a type specification specifies two types, a type for the class operations and a type for the instance operations. The semantics of LOAL do not specify how types are implemented, but the semantics of Larch/LOAL lump together all implementations of a type found in a program. We believe that the close link between class operations and instance operations that our semantics and the definition of legal subtype relations requires will help one to reason about programs using datatype induction [43, Section 4.9.4], although this remains to be shown. Also remaining as future work is a formal connection between subtyping and refinement of the types of instances.

8.3 Related Work

Meyer, America, and Utting have discussed both subtyping and the specification and verification of object-oriented programs.

8.3.1 Meyer's work on Eiffel

Of authors who discuss specification and verification, Meyer's work on Eiffel is perhaps the best known [47] [48]. Meyer concentrates on specification and does not give a formal logic for the verification of Eiffel programs. In contrast to Larch/LOAL, Eiffel specifications may invoke methods. This can be used to give specifications an axiomatic flavor, where the methods and axioms mutually constrain each other. However, many Eiffel specifications do not call other methods, perhaps recognizing the potential problems with invoking operations that can mutate objects in the midst of assertions. Instead, assertions in Eiffel examples usually are expressed in terms of the values of an object's instance variables. But using instance variables also has problems, because implementations are no longer free to choose their own instance variables and because such instance variables must be visible to clients.

[9]Some object-oriented languages do not have classes, but are based on the notion of delegation [41] [60]. In such languages, specified class operations might be implemented by functions that clone prototypes, or by instance operations of prototypes. The point is that one can still design using class operations and subtyping, even if one programs in such a language.

The main problem with a specification language like Eiffel's is that one is sometimes forced to export more operations or instance variables than one would like in order to specify some types. For example, to specify a statistical database with instance operations `insert`, `mean`, and `variance`, one would also need to export operations that enumerate the elements to state the post-condition of `insert`. To see this, suppose such operations were hidden. Then how would a client understand the specification of `insert`? That is, a client cannot test whether `insert` satisfies its specification, because it cannot call `insert` and evaluate its post-condition. However, a designer may wish to prevent clients from enumerating the elements of the database for security reasons. In Larch/LOAL one can specify trait functions on the abstract values that allow the specification of `insert` without allowing clients to access individual elements of the database. Of course, since Larch/LOAL assertions cannot be executed, they are less useful for debugging.

The dynamic binding used in evaluating assertions at run-time in Eiffel is similar to our semantics for specifications, which also evaluates assertions based on the dynamic types of the objects to which the assertions refer. So like our specifications, the specifications in Eiffel are modular.

Meyer's definition of legal subtyping is based on implications between the pre- and post-conditions of corresponding operations of the subtype and supertype [47, Section 11.1]. (Similar definitions were proposed earlier by the designers of Trellis/Owl [55].) In the latest version of Eiffel, specifications of subtypes, if not inherited as is, must use a special form that automatically ensures that the pre-condition of the subtype is weaker than the pre-condition of the supertype, and that the post-condition of the subtype is stronger than the post-condition of the supertype [48, Sections 10.15 and 10.22]. The pre- and post-conditions of the sub- and supertypes can be combined into such formulas because the subtype inherits the instance variables and abbreviations from the supertype, and so the formulas are all in the language of the subtype.

8.3.2 America's formalization of Meyer's techniques

Meyer's specification technique and his definition of legal subtyping has been studied more formally by America [1] [2]. Although America and de Boer have done work on formal verification [3], their verification logic handles pointers, aliasing, and object creation, but explicitly excludes subtyping and message passing. It is America's definition of legal subtyping, based on behavioral specifications, which is most closely related to our work.

In America's work on subtyping, types are specified using abstract values, in a way similar to Larch/LOAL. America defines abstract values and their operations informally, but one could use LSL traits to define them. America specifies operations in POOL-I using pre- and post-conditions written in terms of the abstract values of the operation's formal arguments, just as in Larch/LOAL. However, America does not specify object creation in POOL-I, although that could be done using the same techniques.

America's requirements for legal subtypes take into account that the subtype and its supertypes may have different abstract values, and hence that their specifications may be expressed with different logical operations. His transfer functions, which map the abstract values of subtypes to the abstract values of supertypes, are similar to our simulation relations. Simulation relations, since they need not be functions, are more general than transfer functions [57], and the coercion functions of [53] and [6]. Larch/LOAL gives form to the expression of such relations on abstract values.

The technical relationship between our definition and America's is the subject of Sec-

tion 7.

8.3.3 Comparisons to America's Specification Techniques

America's work provides both a definition of legal subtyping and a specification method. The main differences between his specification method and our method (embodied in Larch/LOAL) are as follows.

- In Larch/LOAL we are limited to subtype-constraining assertions. America has no such limitation and can use "=" freely in assertions.

- America's use of a transfer function can be seen as a way to define the trait functions on arguments of the subtype; that is, the coercion function can be used in a shorthand such as Figure 6. Hence our technique of overloading the trait functions is more general.

- Because we do not require transfer *functions*, but allow relations, we can specify subtypes for which abstract values of the subtype simulate more than one abstract value of the supertype.

8.3.4 Utting's work

Utting's work on the specification and verification of object-oriented programs [62] [61] is in the framework of the refinement calculus. He also shows how to do modular reasoning. His work handles mutation, unlike ours which only deals with immutable objects. However, his work does not allow for change of object representations (data refinement).

Utting's notion of legal subtype relations is similar to refinement, like America's, but treats code that can make arbitrary observations on objects; that is, he treats code that can observe objects in other ways than just calling their methods. We do not include things like a `typeOf` operator in LOAL, because the expression `typeOf(x)` will give different results in environments where `x` denotes objects of different types. For us, such an operator destroys subtyping; but for Utting this is the main distinction between subtyping and refinement. The theoretical and practical implications of these distinctions is unknown, but we believe that programmers and language designers should be cautious in using operators like `typeOf`, as it will certainly complicate reasoning with supertype abstraction.

8.3.5 Liskov and Wing's work

Liskov and Wing [44] [45] formulate a definition of legal subtype relations that is similar to America's, except that it has explicit provisions to deal with aliasing for mutable types; their definition is thus more widely applicable than ours. While we believe that their definition has much to recommend it, their definition is only justified on the basis of intuition and examples; one would have to do the same kind of study as in this paper to justify their work formally. However, since their definition, like America's, is based on implications between assertions, it is more easily applied than ours.

Liskov and Wing do not give a formal logic for program verification. However, their approach to reasoning about programs is the same as our approach of supertype abstraction. (Note that this contradicts their discussion on page 138 of [44] of our work. We use simulation relations to coerce abstract values of subtypes to supertypes, not vice versa.)

8.4 Wider applications

The paradigm of supertype abstraction has wider application than just object-oriented programming. It is possible to view many table-driven programs, including compilers, in the same light. The essence of message passing is a combination of fetching a procedure from an object and invoking it. In a table-driven system the operations are fetched from a table based on some information in an object but the effect of dynamic binding is the same; hence the problems of specifying and verifying such systems are similar.

Our techniques might also be used to reason about type parameters in languages like CLU and Ada. The objects of the actual type parameter can be considered as a kind of subtype of the formal type parameter, especially if only instances of the actual parameter type are involved. Understanding more about the relationship between subtyping and refinement should help in this application of our work.

It may be that our technique of dynamic overloading for assertions would be helpful in proving more conventional refinement relationships and in avoiding information loss problems associated with any abstract views of objects.

We claim the broadest application not for our specific results, but for the concepts of supertpe abstraction and its benefits of modular specification and verification. Modularity is a worthy goal in formal methods for object-oriented programming.

Acknowledgements

Special thanks to two anonymous referees who carefully read earlier versions of this paper and gave many helpful and detailed comments.

Thanks to Barbara Liskov who was instrumental for encouraging this work at an early stage, and for continued technical discussions. Thanks to John Guttag for technical expertise, encouragement, the Larch approach to specification, and his suggestion that we use dynamic overloading of trait functions. Thanks to Jeannette Wing for pointing out related work by Reynolds that helped formalize Guttag's suggestion, for the Larch approach to specification, and for a number of technical discussions. Thanks to David McAllester, who crystallized an idea of ours by suggesting that we base the definition of subtype relations on an algebraic criterion instead of observations (as was done in Leavens' dissertation). Thanks to many others for stimulating technical discussions, especially: Yoonsik Cheon, Krishna Kishore Dhara, Don Pigozzi, Pierre America, Kim Bruce, Doug Lea, David Guaspari, Albert Baker, Val Breazu-Tannen, Albert Meyer, Tobias Nipkow, Qingming Ma, Luca Cardelli, Jim Horning, T.B. Dinesh, David Schmidt, John Mitchell, Martín Abadi, and William Cook. Thanks to Janet Leavens for providing moral support and encouragement.

References

[1] America, P. Inheritance and subtyping in a parallel object-oriented language. In Bezivin, J. et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, N.Y., June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[2] America, P. Designing an object-oriented programming language with behavioural subtyping. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[3] America, P. and de Boer, F. A sound and complete proof theory for SPOOL. Technical Report 505, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., May 1990.

[4] Broy, M. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45(1):1–61, 1986.

[5] Bruce, K. B. and Longo, G. A modest model of records, inheritance, and bounded quantification. In Gurevich, Y., editor, *Third Annual Symposium on Logic in Computer Science*, pages 38–51. IEEE, July 1988.

[6] Bruce, K. B. and Wegner, P. An algebraic model of subtype and inheritance. In Bançilhon, F. and Buneman, P., editors, *Advances in Database Programming Languages*, pages 75–96. Addison-Wesley, Reading, Mass., August 1990.

[7] Burstall, R. M. and Goguen, J. A. Algebras, theories and freeness: An introduction for computer scientists. In Broy, M. and Schmidt, G., editors, *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare*, volume 91 of *series C*, pages 329–348. D. Ridel, Dordrecht, Holland, 1982.

[8] Cardelli, L. A semantics of multiple inheritance. In G. Kahn, D. B. M. and Plotkin, G., editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, New York, N.Y., June 1984. A revised version of this paper appears in *Information and Computation*, volume 76, numbers 2/3, pages 138–164, February/March 1988.

[9] Cardelli, L. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 70–79. ACM, January 1988.

[10] Cardelli, L. and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[11] Chen, J. The Larch/Generic interface language. Technical report, Massachusetts Institute of Technology, EECS department, May 1989. The author's Bachelor's thesis. Available from John Guttag at MIT (guttag@lcs.mit.edu).

[12] Cheon, Y. Larch/Smalltalk: A specification language for Smalltalk. Technical Report 91-15, Department of Computer Science, Iowa State University, Ames, IA, June 1991. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[13] Cook, S. A. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7:70–90, 1978.

[14] Cook, W. R. Object-oriented programming versus abstract data types. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, N.Y., 1991.

[15] Cook, W. R., Hill, W. L., and Canning, P. S. Inheritance is not subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 125–135, January 1990. Also STL-89-17, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, Calif., July 1989.

[16] Dhara, K. K. Subtyping among mutable types in object-oriented programming languages. Master's thesis, Iowa State University, Department of Computer Science, Ames, Iowa, May 1992.

[17] Dhara, K. K. and Leavens, G. T. Subtyping for mutable types in object-oriented programming languages. Technical Report 92-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 1992. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[18] Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, New York, N.Y., 1985.

[19] Enderton, H. B. *A Mathematical Introduction to Logic.* Academic Press, Inc., Orlando, Florida, 1972.

[20] Goguen, J. A. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[21] Goguen, J. A. and Meseguer, J. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. Technical Report CSLI-87-92, Center for the Study of Language and Information, March 1987. Appears in Second Annual Symposium on Logic in Computer Science, Ithaca, NY, June, 1987, pages 18-29.

[22] Goldberg, A. and Robson, D. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Mass., 1983.

[23] Gratzer, G. *Universal Algebra.* Springer-Verlag, New York, N.Y., second edition, 1979.

[24] Guttag, J. V., Horning, J. J., and Wing, J. M. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1985. Order from src-report@src.dec.com.

[25] Guttag, J. Notes on type abstractions (version 2). *IEEE Transactions on Software Engineering*, SE-6(1):13–23, January 1980. Version 1 in *Proceedings Specifications of Reliable Software*, Cambridge, Mass., IEEE, April, 1979.

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

[27] Guttag, J. V., Horning, J. J., and Modet, A. 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.

[28] Guttag, J. V., Horning, J. J., and Wing, J. M. The Larch family of specification languages. *IEEE Software*, 2(4), September 1985.

[29] Hoare, C. A. R. Notes on data structuring. In Ole-J. Dahl, E. D. and Hoare, C. A. R., editors, *Structured Programming*, pages 83–174. Academic Press, Inc., New York, N.Y., 1972.

[30] LaLonde, W. R. Designing families of data types using exemplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, April 1989.

[31] LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An exemplar based Smalltalk. *ACM SIGPLAN Notices*, 21(11):322–330, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[32] Lamping, J. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[33] Lamport, L. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[34] Leavens, G. T. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[35] Leavens, G. T. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[36] Leavens, G. T. and Pigozzi, D. Typed homomorphic relations extended with subtypes. Technical Report 91-14, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, June 1991. Appears in the proceedings of *Mathematical Foundations of Programming Semantics '91*, Springer-Verlag, Lecture Notes in Computer Science, volume 598, pages 144-167, 1992.

[37] Leavens, G. T. and Pigozzi, D. Typed homomorphic relations extended with subtypes. In Brookes, S., editor, *Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 144–167. Springer-Verlag, New York, N.Y., 1992.

[38] Leavens, G. T. and Weihl, W. E. Reasoning about object-oriented programs that use subtypes (extended abstract). *ACM SIGPLAN Notices*, 25(10):212–223, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[39] Leavens, G. T. and Weihl, W. E. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report 92-28d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, August 1994. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[40] Leavens, G. T. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[41] Lieberman, H. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–223, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[42] Liskov, B. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[43] Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[44] Liskov, B. and Wing, J. M. A new definition of the subtype relation. In Nierstrasz, O. M., editor, *ECOOP '93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer-Verlag, New York, N.Y., July 1993.

[45] Liskov, B. and Wing, J. M. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[46] Loeckx, J. and Sieber, K. *The Foundations of Program Verification (Second edition)*. John Wiley and Sons, New York, N.Y., 1987.

[47] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., 1988.

[48] Meyer, B. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[49] Mitchell, J. C. Representation independence and data abstraction (preliminary version). In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 263–276. ACM, January 1986.

[50] Mitchell, J. C. *Lambda Calculus Models of Typed Programming Languages*. PhD thesis, Massachusetts Institute of Technology, August 1984.

[51] Nipkow, T. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22(16):629–661, March 1986.

[52] Nipkow, T. *Behavioural Implementation Concepts for Nondeterministic Data Types.* PhD thesis, University of Manchester, May 1987.

[53] Reynolds, J. C. Using category theory to design implicit conversions and generic operators. In Jones, N. D., editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, January 1980.

[54] Reynolds, J. C. Three approaches to type structure. In Ehrig, H., Floyd, C., Nivat, M., and Thatcher, J., editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[55] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An introduction to Trellis/Owl. *ACM SIGPLAN Notices*, 21(11):9–16, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[56] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[57] Schoett, O. Behavioural correctness of data representations. *Science of Computer Programming*, 14(1):43–57, June 1990.

[58] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[59] Statman, R. Logical relations and the typed λ-calculus. *Information and Control*, 65(2/3):85–97, May/June 1985.

[60] Stein, L. A., Lieberman, H., and Ungar, D. A shared view of sharing: The treaty of Orlando. In Kim, W. and Lochovsky, F. H., editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 3, pages 31–48. Addison-Wesley Publishing Co., Reading, Mass., 1989.

[61] Utting, M. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD thesis, University of New South Wales, Kensington, Australia, 1992. Draft of February 1992 obtained from the Author.

[62] Utting, M. and Robinson, K. Modular reasoning in an object-oriented refinement calculus. In Bird, R. S., Morgan, C. C., and Woodcock, J. C. P., editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June/July*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, New York, N.Y., 1992.

[63] Wing, J. M. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[64] Wing, J. M. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR92-28d
Submission Date: September 12, 1994