Polymorphism in ACL2

Ruben Gamboa Mark Patterson
Computer Science Department
University of Wyoming
{ruben,mark}@cs.uwyo.edu

Abstract

The logic of ACL2 is descended from λ-calculus via Common LISP. It is well-known that this logic is sufficient to reason about arbitrary computations. However, λ-calculus is not today's dominant programming paradigm. To reason about today's programs, as opposed to today's computations, we need a logic that supports modern programming practices. In this paper, we present an approach that allows ACL2 to support polymorphism, a cornerstone of object-oriented programming. We will also show how polymorphism provides a convenient alternative to `encapsulate` for reasoning about classes of functions. At present, the solution is based on a translator which replaces ACL2 proof scripts including polymorphism with regular ACL2 proof scripts. In the future, we plan a version of ACL2 that supports polymorphism directly.

1 Introduction

ACL2 is an industrial-strength theorem prover [6]. Based on Nqthm [2], also known as the Boyer-Moore theorem prover, ACL2 allows the user to define programs in the purely functional or applicative subset of Common Lisp and to prove theorems about such programs. This approach has yielded impressive results. ACL2 has been used to verify aspects of commercial microprocessors, floating-point arithmetic units, program translators (compilers), and programs in a Java-like virtual machine [9, 8, 5]. A distinct advantage of ACL2 is that programs written in ACL2 can be executed directly. A consequence is that a processor model written in ACL2 is not just a mathematical object whose properties can be verified formally; such a model is also an executable simulator of the processor, which can be used, for example, in a traditional testing scenario.

It is hard to argue with success. ACL2 is a proven tool in the application of formal methods to both hardware and software verification, so why should we modify it to support polymorphism? The class of programs expressible in ACL2 is purely functional. While this class of programs has proven sufficient to reason about microprocessor and hardware design, much of current software follows the object-oriented model. As it stands, it is not possible to reason about an object-oriented program directly in ACL2.

There are two aspects to object-oriented programming that do not fit well with ACL2's functional style. First, the object-oriented paradigm revolves around state. Each object contains some state, and its state changes as the object responds to messages or methods. Second, different objects may behave differently when responding to the same message. This difference in behavior is referred to as polymorphism. Informally, we say that objects encapsulate both state and behavior.

A logic of object-oriented programs needs to support both of these notions: state and polymorphism. We will show how ACL2's stobjs can be used to hold the state of an object-oriented program. In this paper, we will concentrate on an approach to polymorphism. The basic idea is that, *at any given time*, each message or method can be thought of as a simple function whose behavior depends on the type of its arguments. As new types or classes are introduced, it becomes necessary to modify the definition of this simple function, and the modifications are always of a simple form. Each time a new class is added, we must extend the prior definitions by adding a case corresponding to the new class.

The remainder of this paper is structured as follows. In section 2, we will examine how CLOS incorporates polymorphism into Common LISP and discuss why we can not take a similar approach in ACL2. Our extensions to the ACL2 language are described in section 3, and section 4 discusses why these extensions are sound relative to ACL2. The basic soundness argument uses a translator that translates ACL2 with polymorphism into traditional ACL2. In section 5 we describe a translator we are using currently to experiment with polymorphism in ACL2. In section 6, we give an example of polymorphism in ACL2. We will pay special attention to the way in which polymorphism can be viewed as an executable alternative to `encapsulate`. We present conclusions and future directions in section 7.

2 Polymorphism in CLOS

It is an explicit goal of ACL2 to retain the syntax and semantics of the applicative subset of Common LISP. It makes sense, therefore, to look at Common LISP's object system (CLOS) to see how it handles polymorphism, and possibly to adopt CLOS directly on ACL2. However, CLOS does not provide a good avenue for polymorphism in ACL2.

The reason is that the semantics of CLOS are inconsistent with classical logic. In particular, the semantics of CLOS is non-monotonic; as more functions are defined, the value of *previously defined* expressions changes. This is completely inconsistent with the semantics of ACL2 (not to mention good programming sense).

To illustrate the point, consider the following example. There are two classes in the hierarchy, `super` and `sub`, and `sub` is a subclass of `super`:

```
(defclass super nil)
(defclass sub (super))
```

We now define the method `size` of `super` as follows:

```
(defmethod size ((super s))
   0)
```

Now consider the following expression:

```
(size (make-instance sub))
```

Since the method `size` has not been defined for the class `sub` at this point, this expression uses the method defined for the parent class, `super`. This is, after all, the whole point of inheritance. In a hypothetical ACL2+CLOS system, we would now be able to prove the following theorem:

```
(defthm size-sub
   (implies (subp s)
            (equal (size s) 0)))
```

However, suppose we proceed by defining the `size` method for the subclass:

```
(defmethod size ((sub s))
   1)
```

Suddenly, `(size (make-instance sub))` is equal to 1 instead of 0, so the theorem `size-sub` has become false.

The problem, of course, is that CLOS allows the definition of a new type to be spread out throughout the program. When the type `sub` is defined, its behavior is not fully specified. Expressions involving `sub` at this point will use the default behavior (i.e., that provided by inheritance). But this behavior changes as more details of `sub` are provided, i.e. by overriding methods. What we need is a way to define the entire behavior of a new type atomically, as is provided by the `class` primitive of other object-oriented languages. We give such a proposal in the next section.

3 Adding Polymorphism to ACL2

We introduce classes into ACL2 using the `defclass` event, which is similar in many ways to `encapsulate`. Like `encapsulate`, it contains embedded events, such as the definition of the methods of the class. It may also contain theorems that serve as constraints on future subclasses of the class. Syntactically, `defclass` looks as follows:

```
(defclass classname superclass
  state
  methods
  constraints)
```

The *classname* and *superclass* are identifiers naming the new class and its parent class, respectively. If the new class does not inherit from any other class, *superclass* will be `nil`. The *state* specifies the state held by the object, using the same notation as `defstobj`. The *methods* specify the behavior of the objects in the class. Each method is defined in a `defmethod` event, similar to a `defun` event, but with the restriction that the first argument of the function must be an object of the given class (or one of its subclasses). The *constraints* are theorems that serve to limit future subclasses. For example, a constraint of a `shape` class might be that the `area` method return a positive real. As with `encapsulate`, the constraint must be satisfied by the current definition of the class. It must also be satisfied every time a new subclass is introduced. In the language of ACL2, classes are inherently constrained, and subclasses must be functional instances of their superclass.

3.1 Object State

There are two different entities that we must include in our model to support object-oriented programs: objects and references. Objects contain state information, while references point to objects. It is possible for more than one reference to point to the same object. This is consistent with most formalizations of object-oriented programs [1, 3, 4].

We store objects in a global registry, implemented as a stobj. Our current approach is to encode all the state for an object into a single ACL2 object, i.e., a list. The structure of this object is given by the state declaration of the `defclass`, which uses the same syntax as `defstobj`. The ACL2 object is placed in the registry, which is an array of cells.

References are essentially indices into the global registry. We have found it convenient to implement references as pairs that also include the type of the object, although this is not necessary.

References correspond to objects in typical object-oriented programs. For example, when a Java program creates a new object with `new`, the value returned is a *reference*; the object itself is somewhere in main memory, and its precise location is of no relevance to a Java program.

It should be noted that references in a polymorphic ACL2 are first-class ACL2 objects, not subject to the "draconian" restrictions placed on stobjs. What this means is that it is possible to place references in other ACL2 objects, e.g., lists. It is also possible to place a reference inside an object, which is necessary to implement object-oriented data structures. Of course, this is only possible because we use a global stobj to hold all object state.

3.2 Object Behavior

The behavior of objects is defined by both the methods and constraints included in the class. The methods define the behavior of objects of this specific class. It is assumed that objects of a subclass may implement different behavior. However, both the current class and all subclasses must abide by the given constraints.

To make this explicit, consider the following example:

```
(defclass measurable nil
  ((v :type integer :initially 0))
  (defmethod measure (x memory)
    (let ((xval (measurable..v x memory)))
      (if (< xval 0)
          (- xval)
        xval)))
  (defthm measure-is-non-negative-real
    (implies (measurable-p x)
             (and (realp (measure x memory))
                  (<= 0 (measure x memory)))))
  ...)
```

This introduces a class called `measurable` and a method `measure`. Notice that `measure` accepts two arguments. The first argument, `x`, must belong to the class `measurable` or one of its subclasses. It is the object receiving the message. The last argument, `memory`, is the stobj representing the global object state.

The function `measurable..v` is generated automatically by `defclass`. It is an accessor to the instance variable `v` of this class. When its argument is not a reference to a `measurable` object, `measurable..v` returns a default value (e.g., through a completion axiom). The effect of the `defmethod` event is to add the following definitional axiom to the ACL2 theory:

$$measure(x) = \begin{cases} -v(x), & \text{if } x \text{ is a strict measurable and } v(x) < 0 \\ v(x), & \text{if } x \text{ is a strict measurable and } v(x) \not< 0 \end{cases}$$

where $v(x)$ denotes the current value of the real number in the object x.

Notice in particular that nothing is said of the behavior of `measure` unless its argument is of type `measurable`, not including any subclasses of `measurable`. We differentiate between objects that satisfy `measurable-p`, which includes all objects of type `measurable` and any of its subclasses, and those which satisfy `strict-measurable-p`, which includes all objects of type `measurable` but not one of its subclasses. After this event, the following theorem will be valid:

```
(defthm strict-measurable-measure
  (implies (strict-measurable-p x)
           (or (equal (measure x memory)
                      (measurable..v x memory))
               (equal (measure x memory)
                      (- (measurable..v x memory))))))
```

However, the following theorem will *not* be provable:

```
(defthm measurable-measure
  (implies (measurable-p x)
           (or (equal (measure x memory)
```

```
                    (measurable..v x))
            (equal (measure x memory)
                   (- (measurable..v x))))))
```

The reason is that the behavior of `measure` for `measurable` objects is not completely known; subclasses may override the default behavior.

This is where constraints come into play. The constraint `measure-is-non--negative-real` guarantees that `measure` always returns a non-negative real for any `measurable` argument. Before the `defclass` event is admitted, these constraints are checked to make sure they work for the current class. In particular, the following proof obligation is established:

```
(implies (or (strict-measurable-p x))
         (implies (measurable-p x)
                  (and (realp (measure x memory))
                       (<= 0 (measure x memory)))))
```

As new subclasses are defined, these constraints are reaffirmed. For example, consider the following subclass:

```
(defclass complex measurable
  ((a :type real :initially 0)
   (b :type real :initially 0))

  (defmethod measure (c memory)
    (let ((x (complex..a c memory))
          (y (complex..b c memory)))
      (acl2-sqrt (+ (* x x) (* y y)))))
  ...)
```

Before this event is admitted, the following proof obligation needs to be established:

```
(implies (or (strict-measurable-p x)
             (strict-complex-p x))
         (implies (measurable-p x)
                  (and (realp (measure x memory))
                       (<= 0 (measure x memory)))))
```

In other words, whenever a new class is introduced, the proof obligations assert that the constraints are satisfied by all *known* classes.

The constraints themselves are available to ACL2 as theorems. This is the only way in which we can reason about methods for all instances of a class, not just the strict instances. For example, the following theorem would be valid:

```
(defthm abs-of-measurable-measure
  (implies (measurable-p x)
           (equal (abs (measure x memory))
                  (measure x memory))))
```

Contrast this with the non-theorem `measurable-measure`.

Because they allow us to reason about the behavior of all instances of a class, constraints also impose a restriction on future subclasses. In this vein, they can be viewed as an implementation of the design-by-contract paradigm popularized by Eiffel [7].

4 Soundness

A feature of `defmethod` is that it allows a subclass to override a method defined in a parent class. That is, it allows the definitional axiom of a function symbol to be modified. Since ACL2 does not support function redefinition, this issue raises some troubling soundness questions. We address these issues in this section.

Observe: we are not talking about arbitrary redefinitions. A better word may be "refinement." What we are allowing is for a function to be defined by cases, and we are not requiring that all cases be specified at once.

Consider a total function defined by cases:

$$f(x) = \begin{cases} f_1(x), \text{ if } x \in S_1 \\ f_2(x), \text{ if } x \in S_2 \\ \vdots \\ f_n(x), \text{ if } x \in S_n \end{cases}$$

where the S_i are mutually disjoint and exhaustive. Let F_i be the function defined over the first i cases. Then F_i corresponds with f for arguments x in S_1, S_2, ..., S_i, but its behavior is undefined for other x.

Thinking of definitions as introducing axioms, we have the following view. The defining axiom for f is given by

$$f \equiv \wedge_{i=1}^{n} x \in S_i \Rightarrow f(x) = f_i(x).$$

Since the S_i are mutually disjoint, this axiom is consistent (as long as the f_i are valid functions). Since the S_i partition the universe, the axiom provides a total definition — i.e., it tells us what the value of $f(x)$ is for any x.

The story is not much different for the F_j:

$$F_j \equiv \wedge_{i=1}^{j} x \in S_i \Rightarrow f(x) = f_i(x).$$

This axiom is clearly consistent, since it is weaker than the defining axiom for f. Moreover, for $i \leq j$ we have that $F_j \Rightarrow F_i$, in particular $f \Rightarrow F_i$ for any $1 \leq i \leq n$.

Now, consider an ACL2 event history $h = (e_1, e_2, \ldots, e_m)$, and suppose that there are indices $1 \leq i_i < i_2 < \ldots < i_n \leq m$ so that $e_{i_j} = F_j$ for $1 \leq j \leq n$ and none of the other e_i events are partial definitions (i.e., all of the other events are ACL2 events not involving polymorphism). Construct a new history h' by introducing the defining axiom for f immediately before e_{i_1} and replacing the e_{i_j} defining axioms with the identical theorems (provable from the definition of

f). Notice that h' is a valid ACL2 history. In it, method definitions have been replaced by theorems about an ideal function f. We define the semantics of our polymorphic extensions to ACL2 by this construction. I.e., polymorphic ACL2 programs can always be translated into regular ACL2 programs.

So as long as we require that there exist a real total function f that is being defined by disjoint cases, the principle of redefinition we utilize is sound. In a practical sense, the construction outlined above could be used to create the new history h', disabling the definition of f except in the proof of the F_i — we use that approach in our translator, although we would like to see support for polymorphism built into the theorem prover.

This construction depends on knowing the function f we are defining a priori, or equivalently on knowing all subclasses that will ever be defined of a given class. But what if we don't? Suppose we have to consider the history h without knowledge of the function f. Moreover, suppose the events e_{i_j} defining f in h are not exhaustive. We can still generate h' as follows: first, complete the definition of f by (arbitrarily) setting it to `nil` for any argument x that is not in the cases covered by the e_{i_j}, and then introduce f before e_{i_1} as before.

However, as we add new events to h, the parallel construction of h' may result in different completions of f. Each history h may be justified by a different function f, but notice that for soundness we only need to have *some* f that satisfies the defining axioms. But we insist that f be constructed by considering *all* redefinitions in h, whether or not they are visible at the end of h. That is, we disallow `defclass` events inside `local`.

To summarize the logical perspective of polymorphism: We introduce a new logical notion, the definition of functions by mutually disjoint cases. We allow the cases to be presented in different events in a history. The resulting history violates the *current* rules for ACL2 histories. However, this history can be mechanically converted into a valid ACL2 history by collecting the definitions and placing them at the beginning (possibly requiring the function to be completed first). Hence this "new" principle is sound, since it can be viewed as nothing more than syntactic sugar. What this means is that we are *not changing* the semantics of ACL2 at all. We are merely introducing convenient syntax.

5 The Translator

The translator parses an existing ACL2 file containing the definitions of a class hierarchy and then writes a translation of that class hierarchy in traditional ACL2 to a new file. The translator, written in ACL2, is largely experimental in nature. Currently, the translator is far from a finished product. It was written simply to demonstrate that polymorphism may be expressed in traditional ACL2 by employing stobjs to hold the state of an object-oriented program. The current translator handles most aspects of ACL2 with polymorphism, though some aspects are still handled manually. For instance, the translator operates on a single file at a time, so it is not practical to write large polymorphic programs in ACL2.

Moreover, as part of removing polymorphism, the translator needs to collect all `defmethod` events defining a given method and combine their definition into a single function. However, it is possible for a `defmethod` to use a function (or method) that is defined after the initial parent class is written. This makes it necessary to move not only the `defmethod` events, but also the intervening definitions, and possibly some theorems to allow ACL2 to accept the definition events. Although the translator does some of this automatically, it does not offer a general solution. Instead, we often find it necessary to modify the translated files.

As it stands, the translator is comprised of approximately two thousand lines of code. The translator itself is comprised of seven modules and each module handles a particular aspect of the translation. The seven modules are: Preamble, Strict-p, Strict-object-p, Classrefs, Classobjs, Parentmethods, and Childmethods.

Each module addresses some aspect of the translation by creating and then writing to file the executable code associated with that aspect of the translation. Preamble creates the translated file and writes the code that establishes the global stobj (called `memory`) that will hold our program state, as well as functions to deal with the object lifecycle. Strict-p writes the specific recognizers for each of our hierarchical class references, i.e., the references to our class objects. Strict-object-p writes the specific recognizers for each of our hierarchical class objects. Classrefs writes the general recognizer for references to our parent class. Classobjs writes the general recognizer for objects in our parent class. This also writes the accessor methods for the class fields. Parentmethods writes ACL2 definitions of those non-polymorphic methods specific to the parent class. Childmethods writes ACL2 definitions of each method, both polymorphic and non-polymorphic, specific to the child classes.

The translator is a work in progress. The exercise of creating a translator has been valuable in helping us to identify and address those issues intrinsic to the problem of reasoning about the object-oriented paradigm in ACL2.

6 An Example

In this section we present a simple example that serves to illustrate most of the relevant points. We will start with a class that describes objects that can be compared:

```
(defclass comparable nil
  ((v :type integer :initially 0))
  (defmethod lorder (x y memory)
    (let ((xval (comparable..v x memory))
          (yval (comparable..v y memory)))
      (lexorder (cons (class-of x) xval)
                (cons (class-of y) yval))))
  (defthm lorder-boolean
    (implies (and (comparable-p x) (comparable-p y))
```

```
            (booleanp (lorder x y memory))))
(defthm lorder-total
  (implies (and (comparable-p x) (comparable-p y))
           (or (lorder x y memory) (lorder y x memory)))
  :rule-classes ...)
(defthm lorder-reflexive
  (implies (comparable-p x)
           (lorder x x memory)))
(defthm lorder-transitive
  (implies (and (comparable-p x)
                (comparable-p y)
                (comparable-p z)
                (lorder x y memory) (lorder y z memory))
           (lorder x z memory))))
```

The `class-of` function returns the name of the class to which an object belongs. We use it in the comparison function to make sure the ordering defined works well even when objects of different types are compared. Had we ignored subclasses now, it may have become impossible to prove one or more of the class constraints when making subclasses later[1]. The subtlety of checking the argument type when comparing objects will be familiar to Java programmers.

We can also create objects of the `comparable` class with the `comparable..new` function. This creates a default object, which can be modified with the method `comparable..update-v`, as in the following `mv-let` form:

```
(defun make-comparable (value memory)
  (declare (xargs  :stobjs (memory)))
  (mv-let (obj memory)
          (comparable..new memory)
          (let ((memory (comparable..update-v obj value memory)))
            (mv obj memory))))
```

We can define functions that work on `comparable` objects. It is possible, certainly, to define methods that do so. But it can be convenient to define traditional ACL2 functions, as the following version of minimum:

```
(defun min-lorder (x y memory)
  (declare (xargs  :stobjs (memory)))
  (if (lorder x y memory) x y))
```

Another obvious function would be `sort-lorder` which sorts lists of objects. Note again that we have the choice of writing a container class and making `sort` a method of that class, or of using an ACL2 container, e.g., lists.

Naturally, we can prove theorems about functions operating on objects, such as the following:

[1] In fact, we had a simpler definition of `lorder`, which assumed the second argument was of the same type as the first, but we discovered that `lorder-transitive` became false when we tried to introduce the `ordered-pair` subclass.

```
(defthm min-lorder-associative
  (implies (and (comparable-p x)
                (comparable-p y)
                (comparable-p z))
           (equal (min-lorder (min-lorder x y memory)
                              z
                              memory)
                  (min-lorder x
                              (min-lorder y z memory)
                              memory))))
```

Similarly, we could prove that `sort-lorder` returns an ordered permutation of its input.

It is worth noting again that the only reason these theorems are provable is that they follow from the constraints on the `comparable` class, not on the specific definition of `lorder` for that class. This is very similar to the situation with `encapsulate`, with a few differences.

The first difference is that methods are executable. If we have a list of references to `comparable` objects, we can sort it with `sort-lorder`. This is most definitely not the case with `encapsulate`, since constrained functions in ACL2 are not executable.

The second difference is that functions operating on objects and theorems about these functions naturally extend to subclasses, without the need to create specialized versions of these functions and reprove the theorems with a `:functional-instance` hint.

To see this, consider the following subclass:

```
(defclass ordered-pair comparable
  ((v2 :type integer :initially 0))
  (defmethod lorder (x y memory)
    (let ((x1 (ordered-pair..v  x memory))
          (x2 (ordered-pair..v2 x memory))
          (y1 (ordered-pair..v  y memory))
          (y2 (ordered-pair..v2 y memory)))
      (lexorder (list (class-of x) x1 x2)
                (list (class-of y) y1 y2)))))
```

Recall that before this class definition is accepted, all constraints defined by the parent (or any ancestor) class are verified. In particular, the following proof obligations are generated:

```
(implies (and (or (strict-comparable-p x)
                  (strict-ordered-pair-p x))
              (or (strict-comparable-p y)
                  (strict-ordered-pair-p y)))
         (booleanp (lorder x y memory)))
```

```
(implies (and (or (strict-comparable-p x)
                  (strict-ordered-pair-p x))
              (or (strict-comparable-p y)
                  (strict-ordered-pair-p y)))
         (or (lorder x y memory) (lorder y x memory)))

(implies (or (strict-comparable-p x)
             (strict-ordered-pair-p x))
         (lorder x x memory))

(implies (and (or (strict-comparable-p x)
                  (strict-ordered-pair-p x))
              (or (strict-comparable-p y)
                  (strict-ordered-pair-p y))
              (or (strict-comparable-p z)
                  (strict-ordered-pair-p z))
              (lorder x y memory) (lorder y z memory))
            (lorder x z memory)))
```

Once the class is accepted, it is possible to use the function `min-lorder` or `sort-lorder` on arguments of type `ordered-pair` as well as `comparable`, without defining new functions. Moreover, the theorems about `min-lorder` and `sort-lorder` remain valid, without the need to state explicitly that they hold for objects of type `ordered-pair`.

In some ways, this makes polymorphism a more natural vehicle to reason about classes of functions than `encapsulate`. The drawback, of course, is that to use polymorphism requires dealing with the stobj `memory`, and using and reasoning about stobjs is more complicated than using and reasoning about regular ACL2 objects.

7 Conclusion

Since the dominant programming paradigm of the day is object-oriented programming, we believe it is important that ACL2 support reasoning about object-oriented programs. We face two major challenges in doing so: Object-oriented programs embody object state and behavior based on type. The first issue can be addressed with a global memory of objects, and ACL2 stobjs provide an efficient mechanism for this. To support the dynamic binding of behavior based on type requires us to address polymorphism.

We have seen how it is possible to provide support for polymorphism in ACL2. Our current implementation uses a translator which works with an entire ACL2 proof script. In particular, it is impossible to build the translation incrementally. In the future, we plan to add support for polymorphism directly into ACL2.

An interesting aspect of polymorphism is that it can be seen as a replacement for `encapsulate`. In this view, it offers some advantages; in particular,

it provides for executable constrained functions, and it does away with the need to rebuild special definitions and theorems for each functional instance of a constrained function. However, the major drawback is that our solution to polymorphism requires using stobjs. While the syntactic restrictions on the use of stobjs do not apply to our object references, we are still forced to use the somewhat awkward stobj notation to refer to the global memory.

Acknowledgments

We would like to thank John Cowles for helping us find the proper `mv-let` form to add new objects to the memory stobj.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. In *Proceedings of the ACM Conference on Principles of Programming Languages*, 1991.

[2] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, Orlando, 1979.

[3] S. Drossopoulou and S. Eisenbach. Describing the semantics of java and proving type soundness. In *Formal Syntax and Semantics of Java.* Springer, 1999.

[4] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java.* Springer, 1999.

[5] W. Georigk. Compiler verification revisited. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 15. Kluwer Academic Press, 2000.

[6] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[7] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[8] J Moore. Proving theorems about Java-like byte code. *Correct System Design — Issues, Methods and Perspectives*, 1999.

[9] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.