Axiomatic Specifications in √eriFun

Andreas Schlosser, Christoph Walther, and Markus Aderhold

Fachgebiet Programmiermethodik, Technische Universität Darmstadt, Germany
http://www.informatik.tu-darmstadt.de/pm

Abstract. Axiomatic specifications stipulate properties of operations axiomatically and are used to reason about mathematical structures like groups, rings, fields, etc. on an abstract layer. Axiomatic specifications allow the organization of the mathematical structures under investigation in a modularized and hierarchical manner, thus supporting well-structured presentations. They also provide representations of non-freely generated data types like integers in a way which supports automated reasoning. A further advantage is that concrete implementations inherit the instances of all proven properties of a specification after it has been proved that the implementation satisfies the axioms of the specification. To utilize these benefits, the interactive verification tool √eriFun has been extended to support axiomatic specifications. We illustrate the expressiveness of our approach by several examples and compare the features provided with those known from other proposals.

1 Introduction

Most up-to-date programming languages allow modularization of programs to organize increasing amounts of lines of code. Modules provide interfaces for the communication with external code. The properties of a module are mostly described by additional textual comments. Programmers implementing the same interface in another module know in which way the procedures should behave and hopefully stick to the given comments. However, for formal program verification, the properties of procedures declared by an interface need to be represented formally. An interface used for program verification has to specify the operators and additionally some axioms which constrain the operators to specific properties. The formal notion is that of a *specification* which stipulates domains, operators defined on these domains, and axioms stating properties on these operators. The classic algebraic concept of axiomatic specifications is exposed in detail in [9, 10].

This paper is concerned with the integration of axiomatic specifications into the √eriFun system [1, 19, 20], an interactive theorem prover for the verification of programs written in the functional programming language $\mathcal{L}$ [18]. This language consists of a definition principle for freely generated polymorphic data types, a definition principle for procedures operating on these data types based on recursion, case analyses, and functional composition, and a definition principle

```
structure bool <= true, false
structure ℕ <= 0, +(-:ℕ)
structure list[@X] <= ø, [infix]::(hd:@X, tl:list[@X])

function occurs(i:@X, k:list[@X]):ℕ <=
if k = ø
  then 0
  else if i = hd(k) then +(occurs(i, tl(k))) else occurs(i, tl(k)) end
end

function ordered(k:list[ℕ]):bool <=
if k = ø
  then true
  else if tl(k) = ø
    then true
    else if hd(k) > hd(tl(k)) then false else ordered(tl(k)) end
  end
end

function insert(n:ℕ, k:list[ℕ]):list[ℕ] <=
if k = ø
  then n :: ø
  else if n > hd(k) then hd(k) :: insert(n, tl(k)) else n :: k end
end

function isort(k:list[ℕ]):list[ℕ] <=
if k = ø then ø else insert(hd(k), isort(tl(k))) end

lemma isort sorts <= ∀k:list[ℕ] ordered(isort(k))

lemma isort permutes <= ∀n:ℕ, k:list[ℕ] occurs(n, k) = occurs(n, isort(k))
```

Fig. 1. Insertion sort over domain $(\mathbb{N}, >)$

for statements (called "lemmas" in $\mathcal{L}$) about the data types and the procedures. Lemmas are defined by universal quantifications using case analyses and the truth values to represent connectives. The free data types *bool* with constructors *true*, *false* and $\mathbb{N}$ for natural numbers with constructors 0 and $^{+}(\ldots)$ for the successor function are predefined in $\mathcal{L}$, cf. Fig. 1.[1] Natural numbers are compared by the predefined procedure `function` $>(x, y{:}\mathbb{N}){:}bool$ <= ... which decides whether x is strictly greater than y. Since well-foundedness of $>$ is *assumed* in ✓eriFun, termination hypotheses for procedures `function` $f(x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau$ <= ... may be based on $>$ by stipulating $C \rightarrow \rho(x_1, \ldots, x_n) > \rho(t_1, \ldots, t_n)$ for recursive calls $f(t_1, \ldots, t_n)$ under conditions C using arguments $t_1, \ldots, t_n$ and a *termination function* ρ. Figure 1 shows a simple $\mathcal{L}$-program for sorting lists of

[1] $^{-}(\ldots)$ stands for the predecessor function in $\mathcal{L}$.

```
specification TotalOrder
  domain @O
  operator le : @O, @O → bool
  axiom reflexivity <= ∀x:@O le(x, x)
  axiom antisymmetry <= ∀x, y:@O le(x, y) ∧ le(y, x) → x = y
  axiom transitivity <= ∀x, y, z:@O le(x, y) ∧ le(y, z) → le(x, z)
  axiom totality <= ∀x, y:@O le(x, y) ∨ le(y, x)
```

Fig. 2. Axiomatic specification of a total order

natural numbers by the *insertion sort* principle as well as the lemmas stating the *correctness* properties for sorting, i.e. that *isort* returns an ordered permutation of its input list.

The remainder of this paper is organized as follows. Sections 2 and 3 describe the capabilities of specifications in ✓eriFun. The semantics of programs with specifications and lemmas stating program properties is defined in Sect. 4. Section 5 gives an survey of other systems which support axiomatic specifications and Sect. 6 concludes with the contributions of this paper and future work.

2 Specification Hierarchies

2.1 Simple Specifications

Lemmas *isort sorts* and *isort permutes* specify the requirements for *isort* (which must hold for *any* sorting algorithm). Sorting is specified *constructively* here by using a specific ordered domain, viz. $(\mathbb{N}, >)$, and procedures, viz. *ordered* and *occurs*, which decide list properties. However, sorting lists can be specified abstractly without relying on $(\mathbb{N}, >)$ only. Figure 2 displays the definition of an ordered domain by the axiomatic specification *TotalOrder*. This definition gives the *domain* @O of the specification, an operator *le* over @O, and the axioms defining properties of *le*.

The abstract variant of insertion sort for ordered domains is given in Fig. 3. *Importing* a specification into a procedure (denoted by [...]) extends the procedure's signature by the symbols defined in the specification, thus making these symbols available in the procedure body. For example, procedure *ordered*, importing specification *TotalOrder*, now is defined upon lists over domain @O and uses the operator *le* for comparing list elements. By this import, the signature of *ordered* is extended by the additional specification parameter O of "type" *TotalOrder*, an actual instance of which has to be provided each time procedure *ordered* is called, cf. lemma *isort sorts* in Fig. 3. In this lemma the specification is imported, too, and passed to the calls of *ordered* and *isort*. Importing a specification into a lemma definition does not only make the domains and operators available in the lemma body, but also imports the axioms given for the imported operators. These axioms may be used for the proof of the lemma like verified

```
function ordered[O:TotalOrder](k:list[@O]):bool <=
if k = ø
  then true
  else if tl(k) = ø
    then true
    else if le(hd(k), hd(tl(k))) then ordered[O](tl(k)) else false end
  end
end

function insert[O:TotalOrder](n:@O, k:list[@O]):list[@O] <=
if k = ø
  then n :: ø
  else if le(n, hd(k)) then n :: k else hd(k) :: insert[O](n, tl(k)) end
end

function isort[O:TotalOrder](k:list[@O]):list[@O] <=
if k = ø then ø else insert[O](hd(k), isort[O](tl(k))) end

lemma isort sorts[O:TotalOrder] <= ∀k:list[@O] ordered[O](isort[O](k))

lemma isort permutes[O:TotalOrder] <= ∀n:ℕ, k:list[@O]
  occurs(n, k) = occurs(n, isort[O](k))
```

Fig. 3. Insertion sort over total ordered domains

```
specification PriorityQueue[O:TotalOrder]
  domain Q[@O]
  operator new : Q[@O]
  operator ins : @O, Q[@O] → Q[@O]
  operator min : Q[@O] → @O
  operator dm : Q[@O] → Q[@O]
  operator size : Q[@O] → ℕ
  axiom min ins new <= ∀v:@O min(ins(v, new)) = v
  axiom dm ins new <= ∀v:@O dm(ins(v, new)) = new
  axiom min ins notnew <= ∀v:@O, q:Q[@O]
    q ≠ new → if{le(min(q), v), min(ins(v, q)) = min(q), min(ins(v, q)) = v}
  axiom dm ins notnew <= ∀v:@O, q:Q[@O]
    q ≠ new → if{le(min(q), v), dm(ins(v, q)) = ins(v, dm(q)), dm(ins(v, q)) = q}
  axiom ins not new <= ∀v:@O, q:Q[@O] ins(v, q) ≠ new
  axiom ins min dm <= ∀q:Q[@O] q ≠ new → q = ins(min(q), dm(q)))
  axiom size definition <= ∀v:@O, q:Q[@O]
    (q = new → size(q) = 0)   ∧
    (size(ins(v, q)) = ⁺(size(q)))   ∧
    (q ≠ new → size(q) ≠ 0 ∧ size(dm(q)) = ⁻(size(q)))
```

Fig. 4. Specification of a priority queue

```
function makeQueue[P:PriorityQueue](l:list[@O]):Q[@O] <=
if l = ø then new else ins(hd(l), makeQueue[P](tl(l))) end

function makeList[P:PriorityQueue](q:Q[@O]):list[@O] <=
if q = new then ø else min(q) :: makeList[P](dm(q)) end

function sort[P:PriorityQueue](l:list[@O]):list[@O] <=
makeList[P](makeQueue[P](l))

lemma sort sorts[P:PriorityQueue] <= ∀l:list[@O] ordered[O(P)](sort[P](l))

lemma sort permutes[P:PriorityQueue] <= ∀n:@O, l:list[@O]
  occurs(n, l) = occurs(n, sort[P](l))
```

Fig. 5. A sorting algorithm using priority queues

lemmas of a program. Likewise, by importing a specification S into a procedure f, the axioms of S are also available in f to support a termination proof for f.

2.2 Parameterization

Specifications may not only be imported into procedures and lemmas, but into other specifications as well—called *parameterization.* Figure 4 shows the definition of a priority queue which is parameterized by a total order, i.e. the priority queue specification uses a totally ordered domain given by specification *TotalOrder* of Fig. 2. For defining the domain of *PriorityQueue*, a *type operator variable* Q for a container data type like *list* containing elements of type $@O$ (imported from *TotalOrder*) is used. Next, the operators for the priority queue are defined: Operator *new* creates a new queue, *ins* inserts an element into the queue, *min* yields the minimal element of the queue, and *dm* deletes the minimal element from the priority queue. Finally, the properties of the operators are given by the axioms. Specification *PriorityQueue* also defines an operator *size* which is used to justify termination of procedures which operate on *PriorityQueues*.

Elements are removed from the queue wrt. the given ordering, i.e. smallest element first. This property can be used to implement a sorting algorithm for lists over an ordered domain: First all elements are inserted into the queue, and then the elements are removed and added to an initially empty list in the order in which they were removed from the queue. The implementation of this algorithm is given in Fig. 5. Procedure *makeQueue* inserts all elements of a list into a new queue and the converse procedure *makeList* takes a queue as input and returns a list containing all elements of the queue by successively removing elements from the queue and adding them to the end of the list. The sorting algorithm itself just creates a priority queue from the given list l and converts it back into a list.

To check the correctness of the sorting algorithm of Fig. 5 it has to be proved that *sort* returns an ordered permutation of its input list. The elements of a priority queue P are ordered wrt. the total order O of priority queue P, denoted by

```
specification Monoid
  domain @M
  operator op : @M, @M → @M
  operator neut : @M
  axiom left neut <= ∀x:@M op(neut, x) = x
  axiom right neut <= ∀x:@M op(x, neut) = x
  axiom op assoc <= ∀x, y, z:@M op(op(x, y), z) = op(x, op(y, z))

specification Group[M:Monoid(@G, op, neut)]
  operator inv : @G → @G
  axiom inv op right <= ∀x:@G op(x, inv(x)) = neut
  lemma inv op left <= ∀x:@G op(inv(x), x) = neut
  lemma inv inv <= ∀x:@G inv(inv(x)) = x

specification GroupHomomorphism[G1:Group(@G1, op1, neut1, inv1),
                                G2:Group(@G2, op2, neut2, inv2)]
  operator h : @G1 → @G2
  axiom homomorphism <= ∀x, y:@G1 op2(h(x), h(y)) = h(op1(x, y))
  lemma h keeps neut <= h(neut1) = neut2
  lemma h keeps inv <= ∀x:@G1 h(inv1(x)) = inv2(h(x))

specification RingUnit[G:Group(@R, plus, zero, minus),
                       M:Monoid(@R, mult, one)]
  axiom plus commutativity <= ∀x, y:@R plus(x, y) = plus(y, x)
  axiom left distributivity <= ∀x, y, z:@R
    mult(x, plus(y, z)) = plus(mult(x, y), mult(x, z))
  axiom right distributivity <= ∀x, y, z:@R
    mult(plus(y, z), x) = plus(mult(y, x), mult(z, x))
```

Fig. 6. Axiomatic specifications of algebraic structures

$O(P)$, by which specification *PriorityQueue* is parameterized. Hence, procedure *ordered* of Fig. 3 is called with $O(P)$ in the lemma *sort sorts* of Fig. 5, thus *referencing* the total order of the priority queue specification.

2.3 Inheritance, Renaming, and Lemmas

The priority queue specification uses the total order specification by adding an additional domain and operators over the new domain. Another kind of using a specification is *inheritance*. When inheriting from other specifications, these specifications are *refined* by adding new operators on the imported domains and/or restricting the imported operators by additional axioms.[2] Inheritance of specifications is related to inheritance in object oriented languages (OO) where a

[2] There is no syntactical or semantical difference between importing, parameterizing, and inheriting. The different namings only indicate different purposes of utilizing specifications.

class is extended by additional fields and methods (whereas overriding methods is not possible with specifications). An inherited specification $S[T{:}U(\ldots)]$ can also be used in contexts where the imported specification U is expected by simply referencing it with $T(S)$—corresponding to subtyping in OO. An example for inheritance is the extension of a *monoid* to a *group*. A monoid is a structure with an associative operator and a neutral element. By adding an inverse operator *inv*, a group *inherits* from a monoid like in specification *Group* of Fig. 6.

When importing a specification, domains and operators may be *renamed*. For instance, the domain of *Monoid* is renamed to $@G$ upon inheritance into specification *Group*. Renaming of local domains and operators of specifications particularly allows the import of *multiple instances* of a *single* specification into another specification. For example, specification *GroupHomomorphism* of Fig. 6 displays a group homomorphism which is defined by importing two separate instances of specification *Group*. By renaming the domains and all operators of *Group*, both instances are distinguished in specification *GroupHomomorphism*. As another benefit, renaming is also used to *link* imported specifications together by sharing domains and/or operators. For instance, Fig. 6 shows how a ring with unit element inherits from a group and a monoid by renaming the domains of *Group* and *Monoid* to $@R$, thus *identifying* both domains. Therefore all operators imported from *Monoid* and *Group* are defined on the *same* domain in the context of specification *RingUnit*.

As an additional feature, a specification S may also contain *lemmas* which are *valid* iff they are logically implied by the axioms of S plus the axioms of the specifications imported by S. If a specification S provides lemmas, it is demanded that all these lemmas be proved before starting to verify a lemma which depends on S. For example, both specifications *Group* and *GroupHomomorphism* of Fig. 6 provide lemmas. As *GroupHomomorphism* imports *Group*, the lemmas of *Group* have to be proven before starting to prove the lemmas of *GroupHomomorphism*. Having succeeded in these proofs, the lemmas of the imported specification S (as well as the lemmas of the specifications which S imports) are available in the importing entity like the axioms of S (as well as the axioms of the specifications which S imports). For example, specification *GroupHomomorphism* inherits the (renamed) lemmas of *Group* to be available for proving the lemmas stating properties about the group-homomorphism function h of *GroupHomomorphism*.

2.4 Instantiation

Before proving a lemma which imports a specification S, consistency of S has to be verified. To this effect, an instantiation of specification S with a structure satisfying the axioms of S has to be provided. Then the instantiated axioms have to be proved and—if successful—it is verified that S possesses a *model*. These models are quite simple in many cases, often using singleton domains. For example, the instantiation of *PriorityQueue* (also instantiating *TotalOrder*) is given in Fig. 7. The domain of specification *TotalOrder* is instantiated with the free data structure *singleton* consisting of the single constant constructor *single*. The domain of *PriorityQueue* is instantiated with the data structure *list*

```
structure singleton <= single

function leSingleton(x, y:singleton):bool <= true

function [outfix] | (l:list[@X]):ℕ <= if l = ø then 0 else +(| tl(l) |) end

instance PQInstance <= PriorityQueue(singleton, list, leSingleton, ø, ::, hd, tl, | · |)

function fold[M:Monoid](l:list[@M]):@M <=
if l = ø then neut else op(hd(l), fold[M](tl(l))) end

lemma fold append[M:Monoid] <= ∀l1, l2:list[@M]
  op(fold[M](l1), fold[M](l2)) = fold[M](append(l1, l2))

instance Plus <= Monoid(ℕ, +, 0)

function fold[Plus](l:list[ℕ]):ℕ <=
if l = ø then 0 else hd(l) + fold[Plus](tl(l)) end

lemma fold append[Plus] <= ∀l1, l2:list[ℕ]
  fold[Plus](l1) + fold[Plus](l2) = fold[Plus](append(l1, l2))
```

Fig. 7. Instantiations of specifications

of linear lists. Now all operators are instantiated in the order of their definition in the specification using the constant function *leSingleton*, the constructors and selectors of *list* and the length function $|\cdot|$. Similarly, all specifications of Fig. 6 can be instantiated by using singleton domains and constant functions.

Instantiations are also used to assert that a concrete structure *implements* an abstract structure given by an axiomatic specification. For instance, using the monoid structure given in Fig. 6, a procedure *fold* can be defined which folds the elements of a list by applying the monoid operation successively to all elements of the list, cf. Fig. 7. Given a further procedure *append* for list concatenation, it can be proved that applying *fold* to a concatenated list is the same as folding both parts separately and applying the monoid operation to the results, cf. lemma *fold append* in Fig. 7. Having proved that natural numbers with addition $+$ and 0, abbreviated by *Plus*, are an instance of *Monoid*, i.e. that *Plus implements* a monoid as given by *Monoid*, procedures using *Monoid* may be instantiated with *Plus* and all lemmas holding for monoids hold for *Plus* as well, cf. procedure *fold*[*Plus*] and lemma *fold append*[*Plus*] of Fig. 7 which are available *implicitly* after the instantiation *Plus* has been proved.

3 Specification of Non-Free Data Types

In the previous sections, specifications were used to build hierarchies of theories which then were used in generic algorithms and lemmas. Another benefit of axiomatic specifications is that they support the definition of non-free data types like *integer*. Such specifications are typically neither instantiated nor inherited from nor multi-used.

3.1 Integers

Integers are defined as a set $\mathbb{Z} := \mathbb{Z}^- \cup \{0\} \cup \mathbb{Z}^+$, where $\mathbb{Z}^- := \{-n \mid n \in \mathbb{N}, n > 0\}$ and $\mathbb{Z}^+ := \{n \mid n \in \mathbb{N}, n > 0\}$. This structure is modeled with specification *Integer* of Fig. 8. Here, integers are built with the operators *zero*, *succ*, and *pred* over some domain $@I$. Operators *succ* and *pred* are *non-free*, as non-trivial equations between them hold, e.g. that they are inverse to each other; both are injective, and do not possess a fixpoint. Additionally, the specification defines the algebraic sign *sign* and the absolute value *abs* of an integer.

The axioms for operators *sign*, *abs*, *pred*, and *succ* guarantee that $\mathbb{N}$ (represented by terms over 0 and $^+(\ldots)$) is isomorphic to $\{0\} \cup \mathbb{Z}^+$ (represented by terms over *zero* and *succ*) as well as to $\{0\} \cup \mathbb{Z}^-$ (represented by terms over *zero* and *pred*), where term i represents an integer in $\mathbb{Z}^+$ if $sign(i) = pos$ and an integer in $\mathbb{Z}^-$ if $sign(i) = neg$. Terms i with $sign(i) = neut$ represent $\{0\}$.

Functions *sign* and *abs* also provide a termination argument for algorithms over integers, which is used to prove termination of procedures operating on *Integer*: Procedures `function` $f[I{:}Integer](x{:}@I, \ldots){:}\tau$ <= … are usually defined recursively, typically by case analysis over the sign of at least one of its arguments.[3] Termination is verified by proving a termination hypothesis like $C \to abs(x) > abs(t)$ for recursive calls $f(t, \ldots)$ under condition C.

Before verifying properties of procedures importing *Integer*, an instance of specification *Integer* has to be provided to ensure consistency of the specification: By encoding integers as natural numbers (viz. even numbers to represent the non-negative integers and odd numbers to represent the negative ones) a model for the axioms of *Integer* is obtained, thus proving consistency of this specification.

Alternatively, integers can be straightforwardly represented by pairs of sign and natural number, by pairs of natural numbers, by a *free* data type with three constructors *Zero*, *Succ*, and *Pred*, etc. Having a concrete representation of integers, in principle there is no need to use an abstract presentation. But each of the above concrete representations introduce formal peculiarities which makes the formulation of procedures and statements about them (as well as proofs of these statements) awkward. E.g., $(-, 0)$ and $(+, 0)$ always need a special treatment when using pairs of sign and natural number. Similarly, a user-defined equality has to be provided when using pairs of natural numbers, where (n, m) denotes the integer $n - m$. Defining a freely generated data type requires restriction to normal forms of integers to be able to use the built-in equality. Thus, all procedures have to be defined to return results in normal form, and formulation of properties is restricted to normal form integers. Encoding integers by even and

[3] Since integers do not have a unique representation using *zero*, *succ*, and *pred*, a case analysis on the leading "constructor" neither makes sense nor is syntactically possible.

```
structure int_sign <= pos, neut, neg

specification Integer
  domain @I
  operator zero : @I
  operator succ : @I → @I
  operator pred : @I → @I
  operator sign : @I → int_sign
  operator abs : @I → ℕ
  axiom pred succ <= ∀i:@I pred(succ(i)) = i
  axiom succ pred <= ∀i:@I succ(pred(i)) = i
  axiom succ not id <= ∀i:@I succ(i) ≠ i
  axiom pred not id <= ∀i:@I pred(i) ≠ i
  axiom succ injective <= ∀i1, i2:@I succ(i1) = succ(i2) → i1 = i2
  axiom pred injective <= ∀i1, i2:@I pred(i1) = pred(i2) → i1 = i2
  axiom sign definition <= ∀i:@I (i = zero ↔ sign(i) = neut)∧
    (sign(pred(zero)) = neg) ∧ (sign(succ(zero)) = pos)∧
    (sign(i) = pos → sign(succ(i)) = pos ∧ sign(pred(i)) ≠ neg)∧
    (sign(i) = neg → sign(pred(i)) = neg ∧ sign(succ(i)) ≠ pos)
  axiom abs definition <= ∀i:@I
    case {sign(i);
          pos : abs(i) = ⁻(abs(succ(i))) ∧ abs(i) = ⁺(abs(pred(i))),
          neut : abs(i) = 0,
          neg : abs(i) = ⁺(abs(succ(i))) ∧ abs(i) = ⁻(abs(pred(i)))}
```

Fig. 8. Specification of the integers

odd natural numbers avoids all pitfalls of the other representations, but with the high price of unreadable definitions and counterintuitive proof obligations.

All these problems are circumvented when using an axiomatic specification as the built-in equality then can be used like for freely generated data types. Hence a *naive implementation* is used for proving *consistency* of specification *Integer*, but then the *abstract presentation* coming with *Integer* is used for *working* with integers, i.e. to formulate procedures operating on integers and stating lemmas about them, as this eases definitions and proofs significantly.

3.2 Algorithms

Figure 9 gives some definitions of arithmetic operations over integers. Addition of integers, e.g., is defined recursively similarly to the recursive definition of addition of natural numbers. Termination of procedure *plus* is shown using procedure $>$ with termination function $\lambda x, y.abs(x)$. This yields two termination hypotheses for procedure *plus*, viz. $sign(x) = pos \rightarrow abs(x) > abs(pred(x))$ and $sign(x) = neg \rightarrow abs(x) > abs(succ(x))$. Both termination hypotheses are easily proved using the axioms of *Integer*. Subtraction, multiplication, and negation are defined in a similar way, based on the case analysis of the sign of one argument, and termination is shown in the same way as for addition. Procedures

```
function plus[I:Integer](x, y:@I):@I <=
case sign(x) of
  pos : succ(plus[I](pred(x), y)),
  neut : y,
  neg : pred(plus[I](succ(x), y))
end

function minus[I:Integer](x, y:@I):@I <=
case sign(y) of
  pos : minus[I](pred(x), pred(y)),
  neut : x,
  neg : minus[I](succ(x), succ(y))
end

function times[I:Integer](x, y : @I):@I <=
case sign(x) of
  pos : plus[I](times[I](pred(x), y), y),
  neut : zero,
  neg : minus[I](times[I](succ(x), y), y)
end

function uminus[I:Integer](x:@I):@I <=
case sign(x) of
  pos : pred(uminus[I](pred(x))),
  neut : x,
  neg : succ(uminus[I](succ(x)))
end

function quotient[I:Integer](x, y:@I):@I <=
case sign(y) of
  pos :
    if abs(y) > abs(x)
      then zero
      else case sign(x) of
        pos : succ(quotient[I](minus[I](x, y), y))
        neut : zero
        neg : pred(quotient[I](plus[I](x, y), y))
    end end
  neut : ⋆4
  neg :
    if abs(y) > abs(x)
      then zero
      else case sign(x) of
        pos : pred(quotient[I](plus[I](x, y), y))
        neut : zero
        neg : succ(quotient[I](minus[I](x, y), y))
    end end
end

function remainder[I:Integer](x, y:@I):@I <=
case sign(y) of
  pos :
    if abs(y) > abs(x)
      then x
      else case sign(x) of
        pos : remainder[I](minus[I](x, y), y)
        neut : zero
        neg : remainder[I](plus[I](x, y), y)
    end end
  neut : ⋆4
  neg :
    if abs(y) > abs(x)
      then x
      else case sign(x) of
        pos : remainder[I](plus[I](x, y), y)
        neut : zero
        neg : remainder[I](minus[I](x, y), y)
    end end
end
```

Fig. 9. Arithmetic operations over integers

quotient and *remainder* have more complicated termination hypotheses. Using $\lambda x, y.abs(x)$ as a termination function again, termination hypotheses are obtained which require lemmas relating *abs*, *sign*, *plus*, *minus*, and $>$.

3.3 Verification

In general, lemmas about recursively defined procedures are proved by induction. Sound induction schemas can be obtained uniformly from the recursion structure of *terminating* procedures. For example, procedure *plus* yields following induction schema for proving a formula $\phi[x]$ with a free integer variable x:

$$\frac{\begin{array}{l}\forall x : @I \;\; sign(x) = neut \rightarrow \phi[x] \\ \forall x : @I \;\; sign(x) = pos \wedge \phi[pred(x)] \rightarrow \phi[x] \\ \forall x : @I \;\; sign(x) = neg \wedge \phi[succ(x)] \rightarrow \phi[x])\end{array}}{\forall x : @I \;\; \phi[x]}$$

[4] The value $\star$ denotes an indetermined result, i.e. *quotient* and *remainder* are only incompletely defined, cf. [21].

```
lemma plus associative[I:Integer] <= ∀x, y, z:@I
  plus[I](x, plus[I](y, z)) = plus[I](plus[I](x, y), z)

lemma plus commutative[I:Integer] <= ∀x, y:@I
  plus[I](x, y) = plus[I](y, x)

lemma sign plus pos neg[I:Integer] <= ∀x, y:@I
  sign(x) = pos ∧ sign(y) = neg →
    if{abs(x) > abs(y),
      sign(plus[I](x, y)) = pos,
      if{abs(y) > abs(x), sign(plus[I](x, y)) = neg, sign(plus[I](x, y)) = neut}}

lemma plus uminus same is zero[I:Integer] <= ∀x:@I
  plus[I](x, uminus[I](x)) = zero

lemma quotient remainder[I:Integer] <= ∀x, y:@I
  sign(y) ≠ neut → x = plus[I](times[I](quotient[I](x, y), y), remainder[I](x, y)))

specification AbelianGroup[G:Group]
  axiom op comm <= ∀x, y : @G op(x, y) = op(y, x)

instance IntegersGroup[I:Integer] <=
  AbelianGroup(@I, zero, plus[I], uminus[I])
```

Fig. 10. Properties of integers

Figure 10 displays some properties of *Integer* operations which have been verified in ✓eriFun by induction, some of which were subsequently used to verify other proof obligations. For example, lemma *sign plus pos neg* has been used for proving termination of procedure *quotient*.

The integers together with addition and unary minus form an abelian group, defined by specification *AbelianGroup* of Fig. 10. The instance property follows immediately from some of the properties displayed in Fig. 10. Like procedures, lemmas, and specifications, an instance can import specifications, too. So the parameterized instantiation of *AbelianGroup* given in Fig. 10 is easily proved correct. This instantiation makes, e.g., the instance $\forall x : @I \ uminus(uminus(x)) = x$ of lemma *inv inv* of specification *Group* of Fig. 6 available for integers, such that it can be used in subsequent proofs of lemmas about integers.

4 Semantics

The operational semantics for $\mathcal{L}$-programs P is defined in [17, 18] by an interpreter $eval_P : \bigcup_\tau \mathcal{T}(\Sigma(P))_\tau \mapsto \bigcup_\tau \mathcal{T}(\Sigma(P)^c)_\tau$ which maps ground terms of *arbitrary* monomorphic data types τ to constructor ground terms of the respective monomorphic data types using the definition of the procedures and data types in P. In [3] the language is extended to $\mathcal{L}^*$ including higher-order functions.

To be able to define the semantics of programs with specifications and lemmas about them, we first define the components of specifications and instances. Furthermore, environments (written as $[\ldots]$) and conversions—i.e. renamings or instantiations—of specifications are defined. Last, environment terms which are used, e.g., to address substructures are defined.

Definition 1 (Specifications, Environments, and Instances). *A* specification *s is a tuple $s = (S, \mathcal{E}, \mathcal{V}_T, \mathcal{V}_O, \mathcal{AX})$, where S is an identifier, $\mathcal{E}$ is the environment of the specification, $\mathcal{V}_T$ is a list of type variables and type operator variables denoting the domain of the specification, $\mathcal{V}_O$ is a list of first-order function variables denoting the operators of the specification, and $\mathcal{AX}$ is a finite set of second-order formulas defining the axioms of the specification.*

A specification conversion *is a pair $\gamma := \langle \xi, \sigma \rangle$ where ξ is a type substitution and σ is a term substitution. γ is an s-*conversion for specification *s if $dom(\xi) \subseteq \mathcal{V}_T$ and $dom(\sigma) \subseteq \mathcal{V}_O$.*

An environment *$\mathcal{E}$ is a list of named specification conversions $\mathcal{E} = \langle e_1 : (s_1, \gamma_1), \ldots, e_n : (s_n, \gamma_n) \rangle$ where each γ_i is an s_i-conversion. The domain of $\mathcal{E}$ is defined by $dom(\mathcal{E}) := \langle e_1, \ldots, e_n \rangle$, and $\mathcal{E}(e_i) := (s_i, \gamma_i)$ for each $e_i \in dom(\mathcal{E})$.*

An instance *is a tuple $(I, \mathcal{E}, s, \gamma)$ where I is the name of the instantiation, $\mathcal{E}$ is an environment, s is a specification, and γ is an s-conversion.*

Specification conversions provide a renaming or a specialization of a specification. Environments are used to make some (renamed) specifications available in different contexts, like in specification, instance, procedure, or lemma definitions.

Definition 2 (Environment Terms). *Given an environment $\mathcal{E}$, the set $\mathcal{T}_\mathcal{E}$ of* specification typed environment terms *is defined as the smallest set satisfying (1)–(3). To each environment term $e \in \mathcal{T}_\mathcal{E}$, a corresponding specification conversion $\gamma_\mathcal{E}$ is assigned.*

1. *$a_s \in \mathcal{T}_\mathcal{E}$ for each $a \in dom(\mathcal{E})$ with $\mathcal{E}(a) = (s, \gamma)$; and $\gamma_\mathcal{E}(a_s) := \gamma$,*
2. *$b(a_s)_{s'} \in \mathcal{T}_\mathcal{E}$ for each $a_s \in \mathcal{T}_\mathcal{E}$ with $s = (S, \mathcal{E}_s, \mathcal{V}_T, \mathcal{V}_O, \mathcal{AX})$ and for each $b \in dom(\mathcal{E}_s)$ with $\mathcal{E}_s(b) = (s', \gamma')$; and $\gamma_\mathcal{E}(b(a_s)_{s'}) := \gamma_\mathcal{E}(a_s) \circ \gamma'$, and*
3. *$I[a^1_{s_1}, \ldots, a^n_{s_n}]_s \in \mathcal{T}_\mathcal{E}$ for each instance $(I, \mathcal{E}_I, s, \gamma)$ with $\mathcal{E}_I = \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n) \rangle$ and $a^1_{s_1}, \ldots, a^n_{s_n} \in \mathcal{T}_\mathcal{E}$, if $\gamma_{I[a^1_{s_1}, \ldots, a^n_{s_n}]} = \bigcup_{i=1}^n \gamma_{(s_i,\gamma_i), a^i_{s_i}}$ with $\gamma_{(s_i,\gamma_i), a^i_{s_i}} = \langle \{\gamma_i(\tau)/\gamma_\mathcal{E}(a^i_{s_i})(\tau) | \tau \in \mathcal{V}_T(s_i)\},\ \{\gamma_i(x)/\gamma_\mathcal{E}(a^i_{s_i})(x) | x \in \mathcal{V}_O(s_i)\} \rangle$ is well-defined; and $\gamma_\mathcal{E}(I[a^1_{s_1}, \ldots,\ a^n_{s_n}]) := \gamma_{I[a^1_{s_1}, \ldots, a^n_{s_n}]} \circ \gamma$.*

Sometimes an environment term e_s is denoted as $e{:}s$.

With (1) all specifications used directly in the environment can be addressed by their name and the induced conversion is defined by the environment. If some specification uses other specifications in its environment, they can be referenced by using the name, given in the environment with (2) and the conversion is the concatenation of the conversion induced by a_s and the conversion which is used in the environment of s to import s'. Furthermore, existing instances with an environment can be specialized using properly typed environment terms (3), i.e.

the environment terms need to address the specifications used in the environment of the instance. The corresponding conversion is the concatenation of the conversions induced by the subterms and the conversion defined in the actual instance. The conversions induced by the subterms are slightly modified such that they do not replace the variables of the specifications, but the variables already renamed by the instance conversion. So, each environment term e_s defines an s-conversion which provides a renaming of the original specification so that it matches its usage in the environment.

The semantics of a procedure `function` $\mathit{func}[\mathcal{E}](x_1{:}\tau_1, \ldots, x_m{:}\tau_m){:}\tau \texttt{ <= } \ldots$ with $\mathcal{E} := \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n)\rangle$ is defined as the semantics of the *signature extended* $\mathcal{L}^*$-procedure `function` $\mathit{func}^*(\gamma_1(\mathcal{V}_O(s_1)), \ldots, \gamma_n(\mathcal{V}_O(s_n)), x_1{:}\tau_1, \ldots, x_m{:}\tau_m){:}\tau \texttt{ <= } \ldots$ where $\mathcal{V}_O(s)$ denotes the list of operators of specification s. I.e., the semantics of a procedure with an environment is defined as the semantics of the corresponding signature extended procedure without an environment, but additional renamed first-order parameters corresponding to the operators of the used specifications. Calls of procedure *func* are translated to calls of the signature extended procedure func^* by computing the specification conversions $\gamma_\mathcal{E}$ for the environment terms and applying them to the actual environment (given by environment terms) of the procedure call, yielding the properly renamed function variables of the addressed specifications. Hence, a procedure call $\mathit{func}[a_1{:}s_1, \ldots, a_n{:}s_n](t_1, \ldots, t_m)$ in an environment $\mathcal{E}$ with $a_i{:}s_i \in \mathcal{T}_\mathcal{E}$ is translated into call $\mathit{func}^*(\gamma_\mathcal{E}(a_1)(\mathcal{V}_O(s_1)), \ldots, \gamma_\mathcal{E}(a_n)(\mathcal{V}_O(s_n)), t_1, \ldots, t_m)$.

For example, the semantics of procedure *plus* of Fig. 9 is defined as the semantics of the transformed procedure

$$
\begin{array}{l}
\texttt{function}\ \mathit{plus}^*(\mathit{zero}{:}@I, \mathit{succ}, \mathit{pred}{:}@I \to @I, \mathit{sign}{:}@I \to \mathit{int\_sign}, \\
\qquad\qquad\qquad \mathit{abs}{:}@I \to \mathbb{N}, x, y{:}@I){:}@I \texttt{ <= } \\
\mathit{case}\ \mathit{sign}(x)\ \mathit{of} \\
\quad \mathit{pos} : \mathit{succ}(\mathit{plus}^*(\mathit{zero}, \mathit{succ}, \mathit{pred}, \mathit{sign}, \mathit{abs}, \mathit{pred}(x), y)), \\
\quad \mathit{neut} : y, \\
\quad \mathit{neg} : \mathit{pred}(\mathit{plus}^*(\mathit{zero}, \mathit{succ}, \mathit{pred}, \mathit{sign}, \mathit{abs}, \mathit{succ}(x), y)) \\
\mathit{end}
\end{array}
$$

where the signature is extended with first-order function variables and the recursive calls are translated to calls of the signature extended procedure.

A lemma `lemma` $\mathit{lem}[\mathcal{E}] \texttt{ <= } \forall x_1{:}\tau_1, \ldots, x_m{:}\tau_m\ b$ with environment $\mathcal{E} := \langle e_1{:}\ (s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n)\rangle$ is *true* in a program P iff for all substitutions σ_O with $\mathit{dom}(\sigma_O) = \bigcup_{i=1}^n \gamma_i(\mathcal{V}_O(s_i))$ assigning the first-order function variables *arbitrary* terminating $\mathcal{L}$-procedures holds: $\mathit{eval}_P(\sigma_\xi(\sigma_O(b))) = \mathit{true}$ for all constructor ground substitutions σ_ξ with $\mathit{dom}(\sigma_\xi) = \{x_1{:}\tau_1, \ldots, x_m{:}\tau_m\}$ if $\mathit{eval}_P(\sigma'_{\xi'}(\sigma_O(ax))) = \mathit{true}$ for all axioms $\forall z_1{:}\upsilon_1, \ldots, z_k{:}\upsilon_k\ ax \in \bigcup_{i=1}^n \gamma_i(\mathcal{AX}(s_i))$ and for all constructor ground substitutions $\sigma'_{\xi'}$ with $\mathit{dom}(\sigma'_{\xi'}) = \{z_1{:}\upsilon_1, \ldots, z_k{:}\upsilon_k\}$, where $\mathcal{AX}(s)$ denotes the set of axioms of specification s.[5]

An instance $(I, \langle e_1{:}(s_1, \gamma_1), \ldots, e_n{:}(s_n, \gamma_n)\rangle, s, \gamma)$ is *true* in a program P iff for all substitutions σ_O with $\mathit{dom}(\sigma_O) = \bigcup_{i=1}^n \gamma_i(\mathcal{V}_O(s_i))$ assigning the first-order

[5] σ_ξ denotes a pair of a term substitution σ and a type substitution ξ.

function variables *arbitrary* terminating $\mathcal{L}$-procedures holds: $eval_P(\sigma_\xi(\sigma_O(b))) = true$ for all axioms $\forall x_1{:}\tau_1, \ldots, x_m{:}\tau_m\ b \in \gamma(\mathcal{AX}(s))$ and all constructor ground substitutions σ_ξ with $dom(\sigma_\xi) = \{x_1{:}\tau_1, \ldots, x_m{:}\tau_m\}$ if $eval_P(\sigma'_{\xi'}(\sigma_O(ax))) = true$ for all axioms $\forall z_1{:}\upsilon_1, \ldots, z_k{:}\upsilon_k\ ax \in \bigcup_{i=1}^{n} \gamma_i(\mathcal{AX}(s_i))$ available in the environment and for all constructor ground substitutions $\sigma'_{\xi'}$ with $dom(\sigma'_{\xi'}) = \{z_1{:}\upsilon_1, \ldots, z_k{:}\upsilon_k\}$.

5 Related Work

ACL2 provides mechanisms to build structured theories [13]. In a so-called *encapsulation* the user can define function symbols and constraints (axioms) on them. Theorems derived from axioms introduced by encapsulations may be *functionally instantiated* [7], i.e. a functional substitution maps the symbols of the encapsulation to concrete functions. If it can be shown that these functions satisfy the axioms of the encapsulation, the theorem can be instantiated with these functions. This corresponds to instantiation of specifications like in Fig. 7, where *Monoid* is instantiated. ACL2 uses a global namespace, and thus prohibits inheritance or multi-usage, cf. *GroupHomomorphism* in Fig. 6, of specifications, since no renaming is possible. Furthermore every instantiation of a specification has to be concrete in the sense that all domain and operator variables have to be instantiated. Instantiations like *IntegersGroup*, cf. Fig. 10, are not possible.

IMPS supports axiomatic specifications by so-called *little theories* [11]. The user can build theories based on different axioms and instantiate derived theorems by *translation* into other theories, i.e. giving a mapping between specifications like with parameterized instantiations, cf. Fig. 10. IMPS allows no multi-usage and only a very restricted version of inheritance, where the user has to give a translation between independent specifications manually to prove the inheritance relation between these specifications. I.e., inheritance cannot be accomplished by parameterization, cf. specification *Group* of Fig. 6, but has to be stated explicitly after defining specifications. Hence, common domains and operators first have to be defined twice and then are shown to be equivalent.

PVS implements direct support for theory interpretations [16]. A *theory* can use uninterpreted types and functions and stipulate axioms, like a specification in √eriFun. A theory can be imported into another theory by *mapping* some or all uninterpreted types and functions to concrete types and functions. This realizes inheritance and instantiation and allows multiple usage of the same theory upon import. The completely instantiated axioms are proof obligations (*type checking constraints)* presented to the user. However, it is not possible to link imported theories, i.e. it is not possible to use two different theories sharing some operators or domains, cf. specification *RingUnit* of Fig. 6.

Isabelle/HOL [15] uses *axiomatic type classes* to provide axiomatic specifications, inspired by type classes [12] in Haskell. Axiomatic type classes specify the constraints on some previously defined function symbols. Type classes can include other type classes and be instantiated with either concrete implementations or other type classes, i.e. type classes support inheritance and (possibly parameterized) instantiations. Isabelle's axiomatic type classes are restricted to only one domain per type class as well as only one instance of each type for a given class and do not allow the usage of type operator variables like Q in specification *PriorityQueue* of Fig. 4.

A more recent approach in Isabelle is the concept of *locales* [5]. Locales define a layer on top of the underlying logic of Isabelle which is used to manage certain *contexts*. Such a context is usually a set of function symbols and axioms about them, i.e. an axiomatic specification. Locales can be combined using so-called *locale expressions* allowing inheritance, sharing, and multi-usage, like specifications *RingUnit* and *GroupHomomorphism* in Fig. 6. Since locales are constructed on top of Isabelle's logic, usage of type operator variables is prohibited, too.

The MAYA-System [4] is not a reasoning system, but manages *development graphs* which are a representation of axiomatic specifications and links between them. These links represent imports and instantiations of specifications. MAYA reads specification languages like CASL [2] and transforms them into a development graph. The proof obligations resulting from the development graph are passed to some external proof system. MAYA allows inheritance, instantiation, and multi-usage of specifications, but supports neither type operator variables nor referencing of substructures, which is used, e.g., in lemma *sort sorts* of Fig. 5.

Nuprl allows axiomatic specifications by definition of *classes* [8]. A class is a collection of sets and operators on them, where the signature of an operator is a type in Nuprl's constructive type theory. Axioms are further types and also part of classes. Inheritance is modeled by intersection of classes (types), and multi-usage and sharing is possible by applying renamings to classes. But the mapping of classes to renamed instances always has to be given explicitly *after* defining composed classes. Hence, e.g. modeling of algebraic structures like in Fig. 6 is possible, but linking the *Monoid* specifications used in the definition of *RingUnit* requires two explicit renamings of *Monoid* after the definition of *RingUnit*. Parameterization of classes is formulated in terms of dependent types. Instantiation and referencing of substructures is not possible.

Coq supports *modules* to structure theories [6]. A module consists of definitions of parameters (domains and operators) and axioms. Modules can be extended and parameterized as well as instantiated, cf. sorting with priority queues of Figs. 3 and 7. Modules may be used multiple, but sharing between different instances of one module is not supported. Thus, definition of *RingUnit* in Fig. 6 is not possible in Coq whereas *GroupHomomorphism* can be defined since both imported groups do not share any elements.

6 Conclusion

Axiomatic specifications as presented here have been integrated into an experimental version of the √`eriFun` system [14] and further developed based on the experiences gained by using specifications in several case studies. However, proving properties about axiomatically specified entities necessitates a high degree of user interaction in √`eriFun`. This is because √`eriFun`'s heuristics were designed

Table 1. Comparison with other systems

	✓eriFun	ACL2	IMPS	PVS	Isabelle/ HOL	Locales	MAYA	Nuprl	Coq
inheritance	✓	✗	•	✓	•	✓	✓	✓	✓
parameterization	✓	✗	✗	✓	•	✓	✓	✓	✓
type operator vars	✓	✗	✗	✗	✗	✗	✗	✗	✗
referencing	✓	✗	✗	✗	✗	✗	✗	✗	✗
multi-usage	✓	✗	✗	✓	✗	✓	✓	•	✓
sharing	✓	✗	✗	✗	✗	✓	✓	•	✗
instantiation	✓	•	✓	✓	✓	✓	✓	✗	✓

✓: full support •: partial support ✗: no support

for controlling the verification of properties of programs rather than the proof of theorems in pure first order logic. Hence, a pure first-order reasoner is needed when working with axiomatic specifications. We therefore presently investigate how to integrate a first-order theorem prover into the system.

We presented the essential features of axiomatic specifications in ✓eriFun, illustrated by several examples. Our proposal offers a certain flexibility, as many kinds of combinations between specifications are supported: Specifications can be inherited, parameterized, referenced, multi-used, linked, and instantiated. Using specifications it is even possible to perform induction proofs over non-freely generated data types like integers without introducing formal clutter by using normal forms, defining congruence relations explicitly, etc. A comparison with related work, summarized in Tab. 1, reveals that our approach might extend proposals known from the literature in a useful way, especially wrt. the possibilities of developing *structured hierarchies* of specifications.

References

1. http://www.verifun.org.
2. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
3. J. Auer. Erweiterung des ✓eriFun-Systems um Funktionen höherer Ordnung. Diploma thesis, Programmiermethodik, Technische Universität Darmstadt, Germany, July 2005.
4. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager MAYA. In H. Kirchner and C. Ringeissen, editors, *9th International Conference on Algebraic Methodology And Software Technologies (AMAST)*, volume 2422 of *Lecture Notes in Computer Science*, pages 495–501. Springer-Verlag, 2002.
5. C. Ballarin. Locales and Locale Expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer-Verlag, 2004.

6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.
7. R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J. S. Moore. Functional Instantiation in First-Order Logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press Professional, Inc., 1991.
8. R. L. Constable and J. Hickey. Nuprl's Class Theory and Its Applications. In F. L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–115. IOS Press, 2000.
9. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *Monographs in Theoretical Compter Science*. Springer-Verlag, 1985.
10. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *Monographs in Theoretical Compter Science*. Springer-Verlag, 1990.
11. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little Theories. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 567–581, Saratoga, NY, June 1992. Springer-Verlag.
12. C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.
13. M. Kaufmann and J. S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, Feb. 2001.
14. A. Krauss. Programmverifikation mit Modulen und Axiomatischen Spezifikationen in √eriFun. Diploma thesis, Programmiermethodik, Technische Universität Darmstadt, Germany, June 2005.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
16. S. Owre and N. Shankar. Theory Interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, Apr. 2001.
17. S. Schweitzer. *Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme*. Doctoral dissertation, Programmiermethodik, Technische Universität Darmstadt, Germany, to appear.
18. C. Walther, M. Aderhold, and A. Schlosser. The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, Programmiermethodik, Technische Universität Darmstadt, Germany, Apr. 2006.
19. C. Walther and S. Schweitzer. About √eriFun. In F. Baader, editor, *19th International Conference on Automated Deduction (CADE)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 322–327. Springer-Verlag, 2003.
20. C. Walther and S. Schweitzer. Verification in the Classroom. *Journal of Automated Reasoning - Special Issue on Automated Reasoning and Theorem Proving in Education*, 32(1):35–73, 2004.
21. C. Walther and S. Schweitzer. Reasoning about Incompletely Defined Programs. In G. Sutcliffe and A. Voronkov, editors, *12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 427–442. Springer-Verlag, 2005.