The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers
Seattle, Washington
August 10 - 22, 2006

IJCAR’06 Workshop

Verify’06:
Verification Workshop
August 15-16, 2006
Proceedings

Editors:
S. Autexier and H. Mantel
Axiomatic Specifications in \texttt{\textcopyright{eri}Fun}

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 \texttt{\textcopyright{eri}Fun} 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 \texttt{\textcopyright{eri}Fun} system [1, 19, 20], an interactive theorem prover for the verification of programs written in the functional programming language \texttt{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

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 (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 N for natural numbers with constructors 0 and +(·, ·) for the successor function are predefined in $\mathcal{L}$, cf. Fig. 1.\(^1\) Natural numbers are compared by the predefined procedure function > (x, y:N):bool <= · · · which decides whether x is strictly greater than y. Since well-foundedness of > is assumed in $\texttt{VériFun}$, termination hypotheses for procedures function $f(x_1: \tau_1, \ldots, x_n: \tau_n):\gamma <= \ldots$ 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 $\rho$. Figure 1 shows a simple $\mathcal{L}$-program for sorting lists of

\(^1\) -(·, ·) stands for the predecessor function in $\mathcal{L}$.
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 \texttt{VeriFun}. 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 \texttt{isort sorts} and \texttt{isort permutes} specify the requirements for \texttt{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. \texttt{ordered} and \texttt{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 \texttt{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 \([\ldots]\)) extends the procedure’s signature by the symbols defined in the specification, thus making these symbols available in the procedure body. For example, procedure \texttt{ordered}, importing specification \texttt{TotalOrder}, now is defined upon lists over domain \(@O\) and uses the operator \(\le\) for comparing list elements. By this import, the signature of \texttt{ordered} is extended by the additional specification parameter \(O\) of “type” \texttt{TotalOrder}, an actual instance of which has to be provided each time procedure \texttt{ordered} is called, cf. lemma \texttt{isort sorts} in Fig. 3. In this lemma the specification is imported, too, and passed to the calls of \texttt{ordered} and \texttt{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 \(k=\emptyset\)
    then true
  else if \(tl(k)=\emptyset\)
    then true
  else if \(le(hd(k),hd(tl(k)))\) then ordered\([O](tl(k))\) else false end
end

function insert\([O::TotalOrder]\)(n:@O,k:list[@O]):list[@O] <=
  if \(k=\emptyset\)
    then \(n::\emptyset\)
  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 \(k=\emptyset\) then \(\emptyset\) 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 not new <= ∀v:@O,q:Q[@O]
    q ≠ new → \{le(min(q),v), min(ins(v,q)) = min(q), min(ins(v,q)) = v\}
  axiom dm ins not new <= ∀v:@O,q:Q[@O]
    q ≠ new → \{le(min(q),v), dm(ins(v,q)) = ins(v,dm(q)), dm(ins(v,q)) = q\}
  axiom ins not new <= ∀q:Q[@O],q ≠ new → q = ins(min(q),dm(q))
  axiom size definition <= ∀v:@O,q:Q[@O]
    \(q = new \rightarrow size(q) = 0\) \∧
    \((size(ins(v,q))) = +(size(q))) \∧
    \((q ≠ new \rightarrow size(q) ≠ 0 \land size(dm(q)) = -(size(q)))

Fig. 4. Specification of a priority queue

149
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 \textit{Monoid}

domain \mathbb{M}

operator \textit{op} : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}
operator \textit{neut} : \mathbb{M}

axiom \textit{left neut} <= \forall x : \mathbb{M} \ \textit{op}(\textit{neut}, x) = x
axiom \textit{right neut} <= \forall x : \mathbb{M} \ \textit{op}(x, \textit{neut}) = x
axiom \textit{assoc} <= \forall x, y, z : \mathbb{M} \ \textit{op}(\textit{op}(x, y), z) = \textit{op}(x, \textit{op}(y, z))

specification \textit{Group}[M: \textit{Monoid}(\mathbb{G}, \textit{op}, \textit{neut})]

operator \textit{inv} : \mathbb{G} \rightarrow \mathbb{G}

axiom \textit{inv op right} <= \forall x : \mathbb{G} \ \textit{op}(x, \textit{inv}(x)) = \textit{neut}
lemma \textit{inv op left} <= \forall x : \mathbb{G} \ \textit{op}(\textit{inv}(x), x) = \textit{neut}
lemma \textit{inv inv} <= \forall x : \mathbb{G} \ \textit{inv}(\textit{inv}(x)) = x

specification \textit{GroupHomomorphism}[G1: \textit{Group}(\mathbb{G1}, \textit{op1}, \textit{neut1}, \textit{inv1}),
G2: \textit{Group}(\mathbb{G2}, \textit{op2}, \textit{neut2}, \textit{inv2})]

operator \textit{h} : \mathbb{G1} \rightarrow \mathbb{G2}

axiom \textit{homomorphism} <= \forall x, y : \mathbb{G1} \ \textit{op2}(\textit{h}(x), \textit{h}(y)) = \textit{h}(\textit{op1}(x, y))
lemma \textit{h keeps neut} <= \textit{h}(\textit{neut1}) = \textit{neut2}
lemma \textit{h keeps inv} <= \forall x : \mathbb{G1} \ \textit{h}(\textit{inv1}(x)) = \textit{inv2}(\textit{h}(x))

specification \textit{RingUnit}[G: \textit{Group}(\mathbb{R}, \textit{plus}, \textit{zero}, \textit{minus}),
M: \textit{Monoid}(\mathbb{R}, \textit{mult}, \textit{one})]

axiom \textit{plus commutativity} <= \forall x, y : \mathbb{R} \ \textit{plus}(x, y) = \textit{plus}(y, x)
axiom \textit{left distributivity} <= \forall x, y, z : \mathbb{R} \ \textit{mult}(x, \textit{plus}(y, z)) = \textit{plus}(\textit{mult}(x, y), \textit{mult}(x, z))
axiom \textit{right distributivity} <= \forall x, y, z : \mathbb{R} \ \textit{mult}(\textit{plus}(y, z), x) = \textit{plus}(\textit{mult}(y, x), \textit{mult}(z, x))

Fig. 6. Axiomatic specifications of algebraic structures

$O(P)$, by which specification \textit{PriorityQueue} is parameterized. Hence, procedure \textit{ordered} of Fig. 3 is called with $O(P)$ in the lemma \textit{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 \textit{inheritance}. When inheriting from other specifications, these specifications are \textit{refined} by adding new operators on the imported domains and/or restricting the imported operators by additional axioms.\footnote{There is no syntactical or semantical difference between importing, parameterizing, and inheriting. The different namings only indicate different purposes of utilizing specifications.} Inheritance of specifications is related to inheritance in object oriented languages (OO) where a
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 $\oplus G$ 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 $\oplus R$, 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 $list$.

152
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, ::, 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 |·|. 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$(i) = \text{pos}$ and an integer in $\mathbb{Z}^-$ if sign$(i) = \text{neg}$. Terms $i$ with sign$(i) = \text{neut}$ 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 $f[:I\to\text{Integer}](x:@I, \ldots) : \tau <= \ldots$ are usually defined recursively, typically by case analysis over the sign of at least one of its arguments. Termination is verified by proving a termination hypothesis like $C \to \text{abs}(x) > \text{abs}(t)$ 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 -> \mathbb{N}
axiom pred succ <= \forall i : \@I pred(succ(i)) = i
axiom succ pred <= \forall i : \@I succ(pred(i)) = i
axiom succ not id <= \forall i : \@I succ(i) \neq i
axiom pred not id <= \forall i : \@I pred(i) \neq i
axiom succ injective <= \forall i1, i2 : \@I succ(i1) = succ(i2) \rightarrow i1 = i2
axiom pred injective <= \forall i1, i2 : \@I pred(i1) = pred(i2) \rightarrow i1 = i2
axiom sign definition <= \forall i : \@I (i = zero \rightarrow sign(i) = neut) \land
       (sign(pred(zero)) = neg) \land (sign(succ(zero)) = pos) \land
       (sign(i) = pos \rightarrow sign(succ(i)) = pos \land sign(pred(i)) \neq neg) \land
       (sign(i) = neg \rightarrow sign(pred(i)) = neg \land sign(succ(i)) \neq pos)
axiom abs definition <= \forall i : \@I
case \{ sign(i) ;
    pos : abs(i) = - (abs(succ(i))) \land abs(i) = + (abs(pred(i))),
    neut : abs(i) = 0,
    neg : abs(i) = + (abs(succ(i))) \land 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
Sound induction schemas can be obtained uniformly from the recursion structure of \textit{terminating} procedures. For example, procedure \texttt{plus} yields following induction schema for proving a formula $\phi[x]$ with a free integer variable $x$:

$$\forall x : \#I \ sign(x) = \text{neut} \rightarrow \phi[x]$$
$$\forall x : \#I \ sign(x) = \text{pos} \land \phi[\text{pred}(x)] \rightarrow \phi[x]$$
$$\forall x : \#I \ sign(x) = \text{neg} \land \phi[\text{succ}(x)] \rightarrow \phi[x]$$
$$\forall x : \#I \ \phi[x]$$

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

### 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 \textit{terminating} procedures. For example, procedure \texttt{plus} yields following induction schema for proving a formula $\phi[x]$ with a free integer variable $x$:

$$\forall x : \#I \ sign(x) = \text{neut} \rightarrow \phi[x]$$
$$\forall x : \#I \ sign(x) = \text{pos} \land \phi[\text{pred}(x)] \rightarrow \phi[x]$$
$$\forall x : \#I \ sign(x) = \text{neg} \land \phi[\text{succ}(x)] \rightarrow \phi[x]$$
$$\forall x : \#I \ \phi[x]$$

The value $\ast$ denotes an indetermined result, i.e. \texttt{quotient} and \texttt{remainder} are only incompletely defined, cf. [21].
lemma plus associative: Integer <= ∀ x, y, z: @
      plus[I](x, plus[I](y, z)) = plus[I](plus[I](x, y), z)

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

lemma sign plus pos neg: Integer <= ∀ x, y: @
      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: Integer <= ∀ x: @
      plus[I](x, uminus[I](x)) = zero

lemma quotient remainder: Integer <= ∀ x, y: @
      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 VeriFun 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 ∀ 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 L-programs P is defined in [17, 18] by an interpreter evalP : ∪τ T(Σ(P))τ → ∪τ T(Σ(P)c)τ, 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 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 \(s = (S, E, V_T, V_O, AX)\), where \(S\) is an identifier; \(E\) is the environment of the specification, \(V_T\) is a list of type variables and type operator variables denoting the domain of the specification, \(V_O\) is a list of first-order function variables denoting the operators of the specification, and \(AX\) is a finite set of second-order formulas defining the axioms of the specification.

A specification conversion is a pair \(\gamma := (\xi, \sigma)\) where \(\xi\) is a type substitution \(\sigma\) is a term substitution. \(\gamma\) is an \(s\)-conversion for specification \(s\) if \(\text{dom}(\xi) \subseteq V_T\) and \(\text{dom}(\sigma) \subseteq V_O\).

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

An instance is a tuple \((I, E, s, \gamma)\) where \(I\) is the name of the instantiation, \(E\) is an environment, \(s\) is a specification, and \(\gamma\) 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 \(E\), the set \(T_E\) of specification typed environment terms is defined as the smallest set satisfying (1)–(3). To each environment term \(e \in T_E\), a corresponding specification conversion \(\gamma_E\) is assigned.

1. \(a_s \in T_E\) for each \(a \in \text{dom}(E)\) with \(E(a) = (s, \gamma)\); and \(\gamma_E(a_s) := \gamma\),
2. \((b(a_s))_{\sigma'} \in T_E\) for each \(a_s \in T_E\) with \(s = (S, E_s, V_T, V_O, AX)\) and for each \(b \in \text{dom}(E_s)\) with \(E_s(b) = (s', \gamma')\); and \(\gamma_E((b(a_s))_{\sigma'}) := \gamma_E(a_s) \circ \gamma'\), and
3. \(I[a_{s_1}^1, \ldots, a_{s_n}^n]_s \in T_E\) for each instance \((I, E_I, s, \gamma)\) with \(E_I = \langle e_1 : (s_1, \gamma_1), \ldots, e_n : (s_n, \gamma_n) \rangle\) and \(a_{s_1}^1, \ldots, a_{s_n}^n \in T_E\), if \(\gamma_I[a_{s_1}^1, \ldots, a_{s_n}^n] = \bigcup_{i=1}^n \gamma_{(s_i, \gamma_i)}(a_{s_i}^i)\) with \(\gamma_{(s_i, \gamma_i)}(a_{s_i}^i) = \{r_{s_i}(\tau)/\gamma_{e_i}(a_{s_i}^i)_s(\tau) | \tau \in V_T(s_i)\}, \{r_{s_i}(x)/\gamma_{e_i}(a_{s_i}^i)_s(x) | x \in V_O(s_i)\}\) is well-defined; and \(\gamma_E(I[a_{s_1}^1, \ldots, a_{s_n}^n]) := \gamma_I[a_{s_1}^1, \ldots, a_{s_n}^n] \circ \gamma\).

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 \( \text{function } \text{func} \langle E \rangle [x_1: \tau_1, \ldots, x_m: \tau_m]: \tau <= \ldots \) with \( E := \langle e_1: (s_1, \gamma_1), \ldots, e_n: (s_n, \gamma_n) \rangle \) is defined as the semantics of the signature extended \( \mathcal{L}^* \)-procedure \( \text{func}^* \langle \gamma \rangle E \langle \gamma \rangle [x_1: \tau_1, \ldots, x_m: \tau_m]: \tau <= \ldots \) where \( \gamma \) denotes the list of operators of specification \( s \).

For example, the semantics of procedure \( \text{plus} \) of Fig. 9 is defined as the semantics of the transformed procedure

\[
\begin{align*}
\text{function } \text{plus}^* & : (\text{zero}:@I, \text{succ}, \text{pred}:@I \rightarrow @I, \text{sign}:@I \rightarrow \text{int}_\text{sign}, \\
& \quad \text{abs}:@I \rightarrow \mathbb{N}, x, y:@I) ; @I <= \\
\text{case sign(x) of} & \\
\text{pos} : & \text{succ}^* (\text{zero, succ, pred, sign, abs, pred}(x), y), \\
\text{neut} : & y, \\
\text{neg} : & \text{pred}^* (\text{zero, succ, pred, sign, abs, succ}(x), y) \\
\text{end}
\end{align*}
\]

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 \( \text{lemma lem} [E] \) is true in a program \( P \) iff for all substitutions \( \sigma \) with \( \text{dom}(\sigma) = \bigcup_{i=1}^{n} \gamma_i(\gamma(\sigma(s_i))) \) assigning the first-order function variables arbitrary terminating \( \mathcal{L} \)-procedures holds: \( \text{eval}(\sigma^* (\text{abs}(b))) \) is true for all constructor ground substitutions \( \sigma \) with \( \text{dom}(\sigma) = \{x_1: \tau_1, x_m: \tau_m\} \) if \( \text{eval}(\sigma^* (\text{abs}(\text{add}))) \) is true for all axioms \( \forall x_1: \gamma_1, \ldots, x_k: \gamma_k \in \bigcup_{i=1}^{n} \gamma_i(\gamma(\sigma(s_i))) \) and for all constructor ground substitutions \( \sigma^* \) with \( \text{dom}(\sigma^*) = \{x_1: \gamma_1, \ldots, x_k: \gamma_k\} \), where \( \gamma(\sigma) \) denotes the set of axioms of specification \( s \).

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 \( \sigma \) with \( \text{dom}(\sigma) = \bigcup_{i=1}^{n} \gamma_i(\gamma(\sigma(s_i))) \) assigning the first-order

\( ^5 \sigma \) denotes a pair of a term substitution \( \sigma \) and a type substitution \( \xi \).
function variables arbitrary terminating \( \mathcal{L} \)-procedures holds: 
\[
\text{eval}_P(\sigma_\xi(\sigma_O(b))) = \text{true}
\]
for all axioms \( \forall x_1: \tau_1, \ldots, x_m: \tau_m \ b \in \gamma(AX(s)) \) and all constructor ground substitutions \( \sigma_\xi \) with \( \text{dom}(\sigma_\xi) = \{x_1: \tau_1, \ldots, x_m: \tau_m\} \) if 
\[
\text{eval}_P(\sigma'_\xi(\sigma_O(ax))) = \text{true}
\]
for all axioms \( \forall z_1: \upsilon_1, \ldots, z_k: \upsilon_k \ ax \in \bigcup_{i=1}^n \gamma_i(AX(s_i)) \) available in the environment and for all constructor ground substitutions \( \sigma'_\xi \) with \( \text{dom}(\sigma'_\xi) = \{z_1: \upsilon_1, \ldots, z_k: \upsilon_k\} \).

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 \texttt{VērīFun}. 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 \texttt{PriorityQueue} of Fig. 4.

A more recent approach in Isabelle is the concept of \textit{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 \texttt{RingUnit} and \texttt{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 \texttt{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 \texttt{Monoid} specifications used in the definition of \texttt{RingUnit} requires two explicit renamings of \texttt{Monoid} after the definition of \texttt{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 \texttt{RingUnit} in Fig. 6 is not possible in Coq whereas \texttt{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 \texttt{Verifun} 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 \texttt{Verifun}. This is because \texttt{Verifun}'s heuristics were designed
Table 1. Comparison with other systems

<table>
<thead>
<tr>
<th>Feature</th>
<th>Verifun</th>
<th>ACL2</th>
<th>IMPS</th>
<th>PVS</th>
<th>Isabelle/HOL</th>
<th>Locales</th>
<th>MAYAN</th>
<th>Nuprl</th>
<th>Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parameterization</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>type operator vars</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>referencing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>multi-usage</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>sharing</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>instantiation</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

✓: 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 Verifun, 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


