Wolfgang Schreiner

Thinking Programs

Formal Modeling and Reasoning about Languages, Data, and Computatations

July 18, 2017

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Contents

Part I The Foundations

Chapter 6
Abstract Data Types

Der Worte sind genug gewechselt, lasst uns endlich Daten sehen. (That's enough words for the moment, now let me see some data!) — Gerhard Kocher (Vorsicht, Medizin!), after Johannn Wolfgang von Goethe (Faust, "Taten/action" rather than "Daten/data".)

Programs operate on data. It is thus natural to start our considerations of how to think about programs by a discussion of how to think about data types. For this purpose, we do not really need to know how the objects of a type are concretely represented (such representations have been discussed in Chapter 4); we may rather focus on the properties that are satisfied by the operations which have been given to us to work with these objects. This view is also in line with modern software engineering that abstracts from the implementation details of data by encapsulating them in classes that only expose a (more or less) well documented method interface to the user.

This chapter presents the core of a theory of such "abstract data types" which is a blend of universal algebra and logic; in particular we introduce a language for specifying abstract data types and give it a formal semantics. Our presentation starts in Section 6.1 with some examples of this language before we elaborate in Section 6.2 its core of "declarations"; this core give rise to the formal notions of "signatures" and "presentations" which capture the syntactic aspects of declarations. We then proceed in Section 6.3 to the mathematical concepts of (many-sorted) "algebras" and "homomorphisms" on which the notion of an "abstract data type" is based which captures the semantics of a declaration.

In Sections 6.4, 6.5, and 6.6, we give three classes of possible interpretations of declarations as abstract data types: the "loose" one (which confines itself to the logical characterization of a type), the "generated/free" one (which describes data types such as finite lists by the means of their construction), and the dual "cogenerated/cofree" one (which describes data types such as infinite streams by the ways of how they can be observed). We then extend in Section 6.7 this language of "specifying in the small" to a language of "specifying in the large": that language allows to combine specifications of individual data types to compound specifications and also to develop "generic specifications" that can be instantiated in various ways.

While the previous elaborations have given abstract data type specifications a formal semantics, it has not yet become really clear what we can practically "do" with such specifications. We therefore conclude in Section 6.8 by discussing how to reason about formally specified abstract data types, for instance, in programs that operate on such types or that implement such types.

6.1 Introduction

We start by presenting several examples of abstract data types that shall motivate the formal concepts that will be discussed in the subsequent sections. We define abstract data types by named *specifications* such as the following one:

```
% a domain with an associative operation and a neutral element
spec MONOID :=
  {
    sort Elem
    const e: Elem
    fun op: Elem × Elem → Elem
    pred isE ⊆ Elem
    axiom ∀x:Elem.  op(e,x) = x ∧ op(x,e) = x
    axiom ∀x:Elem, y:Elem, z:Elem.  op(x,op(y,z)) = op(op(x,y),z)
    axiom ∀x:Elem.  isE(x) ⇔ x=e
  }
```

This specification named MONOID introduces the following entities:

1. a *sort* Elem which denotes a non-empty set of elements;
2. a *constant operation* e which denotes one of these elements;
3. a *function operation* op which denotes a binary function on this set;
4. a *predicate operation* isE which denotes a unary predicate on this set.

We use the term *operations* to differentiate between a name and the entities denoted by this name; if clear from the context, we may also drop the appendix "operation" and just speak of constants, functions, and predicates.

Not every interpretation of these names is allowed: the specification contains various *axioms*, i.e., formulas that must be true for the denoted set, constant, function, and predicate to yield a valid *implementation* of the abstract data type. Conversely, every interpretation obeying the axioms represents a valid implementation: for instance, the specification might be implemented by

- the set of character strings for Elem, the empty string for e, string concatenation for op, and the emptiness test for isE; however, it may also be implemented by
- the set of natural numbers for Elem, the number 0 for e, the addition operation for op, and the nullness test for isE.

These two implementations differ in crucial features, e.g. the axiom

```
axiom ∀x:Elem, y:Elem.  op(x,y) = op(y,x)
```

is false for the first but true for the second one. Above specification is therefore also called *loose*, because it allows implementations with observably different behaviors.

However, a specification need not be loose. Take the specification

```
% the natural numbers
spec NAT :=
```



```
  free type Nat := 0 | +1(Nat)
  then
  {
    fun +: Nat × Nat → Nat
    axiom ∀n1:Nat, n2:Nat.
      +(0, n2) = n2 ∧
      +(+1(n1), n2) = +1(+(n1, n2))
    pred is0 ⊆ Nat, is0(n) :⇔ n=0
  }
```

Here the declaration

```
free type Nat := 0 | +1(Nat)
```

is a shortcut for the specification

```
free {
  sort Nat
  const 0: Nat
  fun +1: Nat → Nat
}
```

which introduces a sort Nat with a constant 0 and a unary *constructor* function +1. This specification is tagged as *free*, which essentially means that every term that one can build from the constructors represents a different element and that these are the only elements of the specified sort. The set denoted by Nat thus consists of the distinct elements denoted by 0, +1(0), +1(+1(0)), . . . ; consequently, this set can be identified with the set of natural numbers.

Using the keyword **then**, this specification is subsequently extended by a loose specification that introduces a binary function + on Nat which is however uniquely characterized by an axiom: for every term of form +(T_1,T_2) where T_1 and T_2 are only constructed by application of 0 and +1, the first argument T_1 "matches" one of the two universally quantified equations in the axiom: if T_1 is 0, the first equation matches and determines the value of the term to be the value of T_2; if it is of form +1(U_1), the result is the value of +1(+(U_1,T_2))). By the freedom of the specification of Nat, exactly one of the specification matches and determines unique values for the variables such that the result is uniquely determined.

Furthermore, we introduce a predicate is0 by a declaration

```
pred is0 ⊆ Nat, is0(n) :⇔ n=0
```

which is a shortcut for

```
pred is0 ⊆ Nat
axiom ∀n:Nat.  is0(n) ⇔ n=0
```

The first format, however, makes it immediately clear that the predicate is uniquely defined, because for every value of its argument, the resulting truth value is explicitly described.

A free specification introduces a sort whose values are "finite", in the sense that they can be constructed by finitely many applications of constructors to a constant. For the following specification, this is not the case:

```
% infinite streams of natural numbers
spec NATSTREAM import NAT :=
  cofree cotype NatStream := head:Nat | tail:NatStream
  then
  {
    fun cons: Nat × NatStream → NatStream
    axiom ∀n:Nat, s:NatStream.
      head(cons(n,s)) = n ∧
      tail(cons(n,s)) = s
    fun counter: Nat → NatStream, counter(n) := cons(n, counter(n+1))
  }
```

Using the keyword **import**, this specification first "imports" the previously written specification NAT (whose entities thus become available to the specification). It then extends it by the declaration

```
cofree cotype NatStream := head:Nat | tail:NatStream
```

which is a shortcut for the specification

```
cofree {
  sort NatStream
  fun head: NatStream → Nat
  fun tail: NatStream → NatStream
}
```

that introduces two *observer* functions head and tail. This specification is tagged as *cofree* which essentially means that the elements of the introduced sort are "black boxes" which are only considered as different if they be distinguished by the (repeated) application of observer operations; however, every term of the new sort that one can build from the observers represents a different value. The sort NatStream can be thus identified with the set of infinite streams of natural numbers: given a stream s, head(s) denotes the first number (the "head") of the stream and tail(s) denotes the remainder of s (its "tail") which is different from s itself.

This specification is subsequently extended by a loose specification

```
{
  fun cons: Nat × NatStream → NatStream
  axiom ∀n:Nat, s:NatStream.
    head(cons(n,s)) = n ∧
    tail(cons(n,s)) = s
  ...
}
```

which introduces a function cons such that cons(n,s) denotes an infinite stream with head n and tail s. This function is constrained by two "pattern-matching" equations

that specify for every observer of NatStream the result of its application to cons(n,s). Because of the co-freedom of NatStream, these equations determine unique values for n and s such that s is a proper substream of the original stream; equations of this kind can thus not introduce any inconsistencies but indeed define a function uniquely.

In the same style, we could by a specification

```
{
  fun counter: Nat → NatStream
  axiom ∀n:Nat, s:NatStream.
    head(counter(n)) = n ∧
    tail(counter(n)) = counter(n+1)
}
```

define a function counter such that that counter(n) denotes the infinite stream $[n, n+1, n+2, \ldots]$. However, with the help of cons, this can be much more elegantly achieved: the declaration

```
fun counter: Nat → NatStream, counter(n) := cons(n, counter(n+1))
```

determines the same function, because the equations

```
head(counter(n)) = head(cons(n, counter(n+1))) = n
tail(counter(n)) = tail(cons(n, counter(n+1))) = counter(n+1)
```

follow from the axioms of the cons operation.

The specification NATSTREAM models streams of natural numbers; however, streams behave more or less the same for all kinds of elements. We can express this by creating a generic (parameterized) specification

```
% infinite streams of elements
spec STREAM[sort Elem] :=
  cofree cotype Stream := head:Elem | tail:Stream
  then
  {
    fun cons: Elem × Stream → Stream
    axiom ∀e:Elem, s:Stream.
      head(cons(e,s)) = e ∧
      tail(cons(e,s)) = s
  }
```

from which we derive NATSTREAM as a special instance:

```
spec NATSTREAM import NAT :=
  STREAM[NAT fit Elem↦Nat] with Stream↦NatStream
  then fun counter: Nat → NatStream, counter(n) := cons(n, counter(n+1))
```

The specification instantiation STREAM[NAT **fit** Elem↦Nat] generates a version of STREAM that replaces the formal parameter sort Elem by the actual argument sort Nat; by the clause **with** Stream↦NatStream the sort Stream of the resulting specification is then renamed to NatStream.

Abstract Data Types in CafeOBJ and CASL

In this chapter we introduce two software systems that support algebraic specifications of abstract data types, each in its own way:

- CafeOBJ [10, 14, 15] is an algebraic specification language in the tradition of OBJ. It is based on a many-sorted equational logic extended by subsorts, unidirectional transitions, and hidden sorts with a notion of behavioral equivalence. A subset of CafeOBJ is executable: the core of the CafeOBJ software is a term rewriting system that allows to execute initial specifications with restricted forms of conditional equations as axioms. By term rewriting, also proofs by structural induction or searches for specific reduction sequences can be performed.
- The Heterogeneous Tool Set Hets [37] is a software framework for integrating various specification languages, most prominently CASL and its various extensions such as CoCASL. Hets constructs from CASL specifications "development graphs" which structure the proofs that have to be performed to ensure various semantic constraints with which the specifications may be annotated; for proving consistency, the ideas sketched in the previous chapter have been implemented in a formal calculus [51]. Proofs of user-specified theorems are performed with the help of external automatic and interactive provers.

A comparison of the languages of CafeOBJ and CASL can be found in [40].

The specifications used in the following presentations can be downloaded from the URLs

```
https://www.risc.jku.at/people/schreine/TP/software/adt/adt.cafe
https://www.risc.jku.at/people/schreine/TP/software/adt/adt.casl
```

and loaded by executing from the command line the following commands:

```
cafeobj adt.cafe
hets adt.casl
```

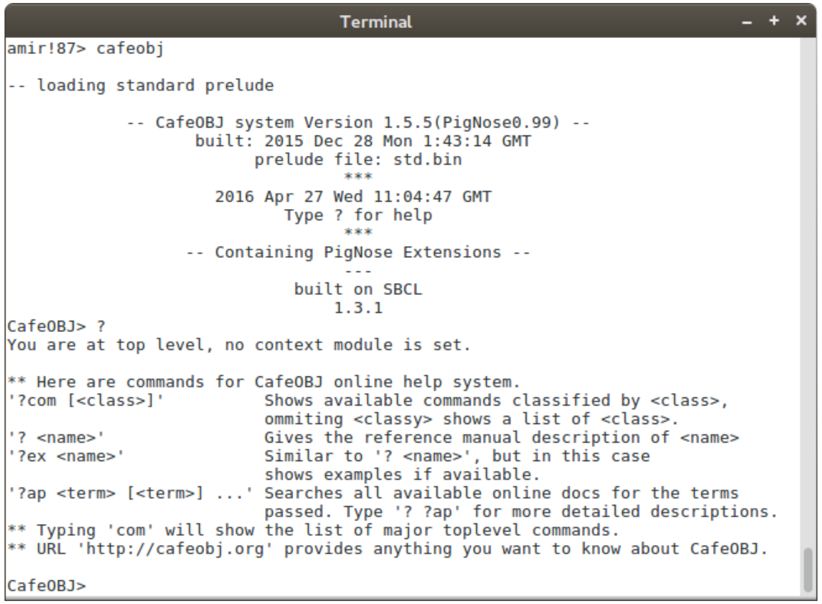

Fig. 6.16 CafeOBJ

CafeOBJ

CafeOBJ is a text-only system that is operated in a terminal; see Figure 6.16 for the startup message printed by the system. We start by writing a small specification of the abstract data type "integer numbers":

```
module! MYINTCORE {
  protecting (NAT)

  [ Int ]
  op int : Nat Nat -> Int

  vars N1 N2 : Nat
  ceq int(N1,N2) = int(p(N1),p(N2)) if N1 =/= 0 and N2 =/= 0 .
}
```

This specification can be written either on the command-line or into a text file, e.g. `adt.cafe`; then the command

```
input adt .
```

reads and processes the file. The specification introduces a module `MYINTCORE` which defines the core of the abstract data type; the exclamation mark in the keyword module! indicates that for the initial interpretation of the specification is desired (this is essentially just a hint for the human user, the system

treats all modules alike). The module imports the abstract data type `NAT` which is subsequently extended by the specification; this data type is part of the system library and provides an efficient implementation of the natural numbers (based on machine integers). The keyword `protecting` indicates that the interpretation of that type shall be preserved, i.e., not modified by the extension (again this is just a hint for the user). The module then introduces a new sort `Int` with a constructor `int` that maps pairs of natural numbers to integers; the idea is that the term `int(`N_1`,`N_2`)` denotes the integer $N_1 - N_2$.

The `vars` clause introduces universally quantified variables which may be used in subsequent axioms. The keyword `ceq` indicates that the given axiom is a conditional equation; i.e., the equation on the left hand side is true, provided that the condition on the right hand side holds. The right hand side may be a propositional combination of equations T_1 `==` T_2 where T_1 `=/=` T_2 is a shortcut for `not` T_1 `==` T_2. The CafeOBJ system treats axiomatic equations as left-to-right rewrite rules; thus the given axiom says that any occurrence of a term of form `int(`N_1`,`N_2`)` may be rewritten to the term `int(p(`N_1`),p(`N_2`))` provided that the stated condition holds. The operation `p` imported from `NAT` represents the predecessor function $\lambda x.\ x-1$; thus the conditional equation all in all states that in an application of `int(`N_1`,`N_2`)` to non-zero values N_1 and N_2 both N_1 and N_2 may be replaced by their predecessors. The predicates `==` respectively `=/=` actually represent "reduction (in)equality"; for determining their truth value the system reduces both argument terms as much as possible (until no more rewriting rule can be applied); the predicates are then considered as true if the resulting terms are identical respectively different.

If we would have not used the builtin representation of the natural numbers but provided our own definition in a specification `MYNAT`, we could have written the axiom simply as

```
eq int(s(N1),s(N2)) = int(N1,N2) .
```

Here the keyword `==` indicates that the axiom is an unconditional equality. The constructor `s` imported from `MYNAT` represents the successor function $\lambda x.\ x+1$; the constraint that the reduction rule can be only applied to non-zero values could be then expressed by pattern-matching. In any case, the definition is executable; by executing

```
open MYINTCORE .
```

we enter the name space of the module such that we can execute

```
reduce int(5,3) .
```

which shows by the output

```
-- reduce in %MYINTCORE : (int(5,3)):Int
(int(2,0)):Int
(0.0000 sec for parse, 0.0040 sec for 26 rewrites + 36 matches)
```

that 26 rewrite rules have been applied to reduce the given term to its canonical form `int(2,0)`. By setting the option

```
set trace on .
```

the application of all rewrite rules can be indeed monitored (we omit the verbose output). By executing

```
close .
```

we leave the name space of the module again.

We continue by extending the data type by a couple of operations:

```
module* MYINT {
  protecting (MYINTCORE)

  op 0 : -> Int
  op _ +  _ : Int Int -> Int
  op _ <= _ : Int Int -> Bool

  vars N1 N2 M1 M2 : Nat
  eq 0 = int(0,0) .
  eq int(N1,N2) + int(M1,M2) = int(N1 + M1,N2 + M2) .
  eq int(N1,N2) <= int(M1,M2) = N1 + M2 <= M1 + N2 .
}
```

Here an integer constant `0` is introduced (constants are in CafeOBJ just operations without arguments), a binary integer function + and a binary integer predicate <= (predicates are just operations into the predefined sort `Bool` with constants `true` and `false`); CafeOBJ allows to use infix notation for the binary operations. All three operations are uniquely defined by axiomatic equations; thus we indicate by the asterisk in the keyword `module*` that a loose interpretation of the extension suffices (again this is just a hint to the user). We may also compute with this specification, e.g. if we execute

```
open MYINT .
reduce int(5,3) + int(2,7) .
close .
```

we get the result

```
-- reduce in %MYINT : (int(5,3) + int(2,7)):Int
(int(0,3)):Int
(0.0000 sec for parse, 0.0040 sec for 67 rewrites + 93 matches)
```

Next we are defining the core of a generic type "list of elements":

```
module* ELEM { [ Elem ] }

module! LISTCORE[ E :: ELEM ] {
  [ List ]
  op empty : -> List
  op cons : Elem List -> List
}
```


The loosely interpreted specification `ELEM` introduces a sort `Elem`; this specification is used for the parameter of the initially interpreted generic specification `LISTCORE` which introduces a sort `List` with constructors `empty` and `cons`. A generic module may in CafeOBJ have multiple parameters whose identities can be distinguished by the given name (`E` in above example); if there should be two `ELEM` parameters with name `E1` and `E2`, we could distinguish by the notation `Elem.E1` and `Elem.E2` their respective sorts.

Furthermore, we extend the core type by the usual operations:

```
module* LIST[ E :: ELEM ] {
  protecting (LISTCORE(E))
  protecting (NAT)

  op head : List -> Elem
  op tail : List -> List
  op append : List List -> List
  op length : List -> Nat

  var E : Elem
  vars L L1 L2 : List

  eq head(cons(E, L)) = E .
  eq tail(cons(E, L)) = L .

  eq append(empty, L2) = L2 .
  eq append(cons(E,L1), L2) = cons(E, append(L1, L2)) .

  eq length(empty) = 0 .
  eq length(cons(E,L)) = 1 + length(L) .
}
```

Now we instantiate the generic type `LIST` with above type `MYINT`:

```
view INT->ELEM from ELEM to MYINT { sort Elem -> Int }
module* INTLIST { protecting (LIST(INT->ELEM)) }
```

The `view` declaration introduces a morphism `INT->ELEM` that maps the signature of `ELEM` to the signature of `MYINT`. We then define the module `INTLIST` by the application of `LIST` to this view and thus derive the type "list of integer numbers". By the commands

```
open INTLIST .
let L =
  append(cons(int(3,1),cons(int(5,8),empty)),cons(int(12,7),empty)) .
reduce L .
reduce length(L) .
close .
```

we locally define a list `L`; first we compute its canonical form, second its length. The resulting output is

```
-- setting let variable "L" to : append(...) : List
```

```
-- reduce in %INTLIST : (append(...)):List
(cons(int(2,0),cons(int(0,3),cons(int(5,0),empty)))):List
(0.0000 sec for parse, 0.0040 sec for 123 rewrites + 175 matches)

-- reduce in %INTLIST : (length(append(...)):Nat
(3):NzNat
(0.0000 sec for parse, 0.0000 sec for 148 rewrites + 215 matches)
```

While above examples have demonstrated the suitability of CafeOBJ for executing specifications of a certain form, the capability of the underlying term rewriting engine may be also applied to certain forms of reasoning. As an example, we demonstrate the proof of

```
length(append(L1,L2)) == length(L1)+length(L2)
```

for arbitrary integer lists `L1` and `L2`. Since sort `List` is generated with constructors `empty` and `cons`, we may perform this proof by structural induction over `L1`.

First we start the proof of the base case by executing

```
open INTLIST .
op L2 : -> List .
```

Here we enter the name space of `INTLIST` which we extend by a new list constant `L2`. We then show that the equality holds for `empty` and `L2`:

```
reduce length(append(empty,L2)) == length(empty) + length(L2) .
```

which is indeed confirmed:

```
-- reduce in %INTLIST : (... == ...):Bool
(true):Bool
(0.0000 sec for parse, 0.0000 sec for 4 rewrites + 13 matches)
```

Next we introduce a new list constant `L1` which allows us to state the induction assumption (namely that the property holds for `L1` and `L2`) by an additional rewrite rule:

```
op L1 : -> List .
eq length(append(L1,L2)) = length(L1) + length(L2) .
```

Finally we introduce a new integer constant `I` which allows us to formulate the induction step (namely the claim that the property holds for `cons(I,L1)` and `L2`):

```
op I : -> Int .
reduce length(append(cons(I,L1),L2)) == length(cons(I,L1)) + length(L2) .
```

Indeed the output

```
-- reduce in %INTLIST : (... == ...):Bool
(true):Bool
(0.0000 sec for parse, 0.0000 sec for 5 rewrites + 84 matches)
```

also confirms this claim. CafeOBJ may thus help to perform those kinds of proofs which can be reduced to equality reasoning (or also to a search for reduction/transition sequences, which we will not discuss further).

CASL and Hets

The heterogeneous toolset Hets can be started from the command line with a list of CASL specification files as arguments; it then analyzes the correctness of the syntax and of the static semantics of the specifications. For instance, for the input file `adt.casl` whose content will be explained below, the tool produces the following output:

```
> hets adt.casl
Analyzing library adt
Downloading Basic/Numbers ...
Analyzing library Basic/Numbers version 1.0
Analyzing spec Basic/Numbers#Nat
Analyzing spec Basic/Numbers#Int
Analyzing spec Basic/Numbers#Rat
Analyzing spec Basic/Numbers#DecimalFraction
... loaded Basic/Numbers
Analyzing spec adt#MyIntCore
Analyzing spec adt#MyInt
Analyzing spec adt#Elem
Analyzing spec adt#ListCore
Analyzing spec adt#List
Analyzing spec adt#IntList
Analyzing spec adt#ListProof
```

The content of file `adt.casl` represents the CASL counterpart to the CafeOBJ specifications given in the previous section. It starts with a header

```
library adt
from Basic/Numbers get Nat
```

which ensures that the data type `Nat` from the standard library can be subsequently used. It then continues with the specification

```
spec MyIntCore = Nat then %mono
  free {
    type Int ::= int(p:Nat;m:Nat)
    forall n1,n2:Nat
    . int(suc(n1),suc(n2)) = int(n1,n2)
  }
end
```

which defines the core of the type “integer numbers” as an extension of the given type `Nat`: the `type` declaration introduces a sort `Int` with a binary constructor `int` from `Nat` to `Int` and two corresponding selectors `p` and `m`, i.e., for any `Int` value i, we have i = `int(p(`i`),m(`i`))`. CASL is built upon full first-order logic, thus the specification contains a quantified formula as an axiom. The free interpretation of the extension constrained by this axiom ensures that every integer has a canonical representation. The annotation `%mono` asserts that the extension is *monomorphic*, i.e., that every algebra N of `Nat` is extended to at least one algebra I, and that any two extensions I, I' of N are isomorphic.

We continue by extending the core type by some operations:

```
spec MyInt = MyIntCore then %def
  op 0: Int = int(0,0)
  op  __+__(i1,i2:Int): Int = int(p(i1)+p(i2),m(i1)+m(i2))
  pred __<=__(i1,i2:Int) <=> p(i1)+m(i2) <= p(i2)+m(i1)
end
```

A constant is just a zero-ary operation, but predicates are in CASL different from operations. Above specification introduces these entities by definitions but the function and the predicate could have also been introduced in an axiomatic form:

```
op __+__: Int * Int -> Int
pred __<=__: Int * Int
forall p1, m1, p2, m2: Nat
. int(p1,m1) +  int(p2,m2) =   int(p1+p2,m1+m2)
. int(p1,m1) <= int(p2,m2) <=> p1+m2 <= p2+m1
```

The annotation `%def` asserts that the extension is *definitional*, i.e., that every algebra I of `MyIntCore` is extended to exactly one algebra I'. The annotations `%mono` and `%def` are special cases of the annotation `%cons` which just states that an extension is *conservative*, i.e., that every algebra I of the original type is extended to at least one algebra I'; as we will see below, it is easier to show that an extension is just conservative than to show that it is also monomorphic or definitional.

For specifying the type "list of elements", we start with the specification

```
spec ListCore[sort Elem] = %mono
  free type List[Elem] ::= empty | cons(Elem,List[Elem])
end
```

where the generic specification `ListCore` extends by a free type declaration every argument type with a sort `Elem` in a monomorphic way. The specification introduces a sort with the compound name `List[Elem]` with constructors `empty` and `cons`; the sorts resulting from specific instantiations of the generic specification will thus receive correspondingly instantiated names.

We could have also written the type declaration as

```
free type List[Elem] ::= empty | cons(head:?Elem,tail:?List[Elem])
```

which additionally introduces two partial selectors `head` and `tail`; these operations are only defined on values constructed by application of `cons`. We could also introduce them in an axiomatic way

```
op head: List[Elem] ->? Elem
op tail: List[Elem] ->? List[Elem]

forall l:List[Elem]; e:Elem
. def head(l) <=> not l = empty
. head(cons(e,l)) = e
. def tail(l) <=> not l = empty
. tail(cons(e,l)) = l
```

where the arrows `->?` indicate that the operations are partial and the corresponding `def` predicates denote by preconditions the domains of these operations. However, since the selectors are subsequently not used (and adding the additional axioms prevents a quick automatic proof given below), we do without them.

Now we equip the data type with additional operations:

```
spec List[sort Elem] given Nat = ListCore[sort Elem] then %def
  op append: List[Elem] * List[Elem] -> List[Elem]
  forall l1,l2:List[Elem]; e:Elem
  . append(empty,l2) = l2
  . append(cons(e,l1),l2) = cons(e,append(l1,l2))

  op length: List[Elem] -> Nat
  forall l:List[Elem]; e:Elem
  . length(empty) = 0
  . length(cons(e,l)) = 1+length(l)
end
```

The `given` clause imports the specification `Nat` in such a way that it also can appear as (a part of) an argument in an instantiation of the specification (the previous chapter used the keyword import for this purpose). For instance, we may now define the type "list of integers" as

```
spec IntList = List[MyInt fit Elem |-> Int]
```

Finally, we introduce by an extension

```
spec ListProof[sort Elem] = List[sort Elem] then %implies
  forall l1,l2:List[Elem]
  . length(append(l1,l2)) = length(l1)+length(l2)
end
```

an additional axiom; the annotation `%implies` indicates that the extension is *implied*, i.e., that the original type is identical to the extended type.

The annotations given in the specifications represent claims that have to be proved; the remainder of this section demonstrates how Hets supports these proofs. By typing

```
hets -g adt.casl
```

Hets is started in a graphical mode where the window illustrated in Figure 6.17 is displayed. This window shows the "development graph" of the included specifications; in this graph the named nodes represent specifications and the arrows represent dependencies among specifications. The black arrows represent "definition links" that indicate that a specification is used in the definition of another specification; the colored arrows represent "theorem links" that postulate relations between the theories; these links thus represent proof obligations that have to be handled.

We start by selecting in menu `Edit` the entry `Proofs` and from the submenu the `Auto-DG-Prover` which applies the rules of the proof calculus for

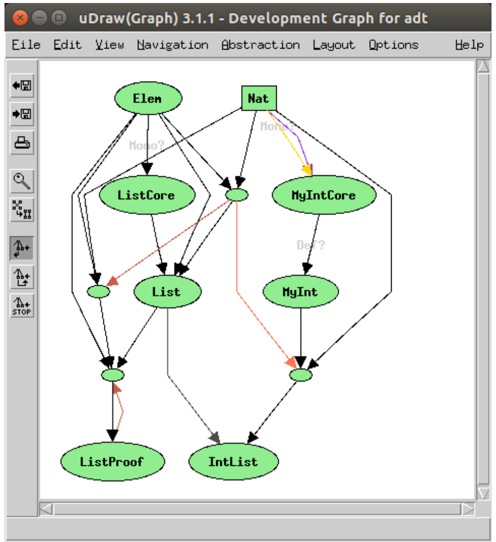

Fig. 6.17 Hets Development Graph

development graphs. This reduces the original proof obligations to the core obligations that we have to deal with; the results are shown in the left diagram of Figure 6.18. The grey labels `Mono?` and `Def?` represent the obligations to prove that the corresponding extensions are monomorphic respectively definitional; the red node indicates the obligation to prove the additional axiom in the implied extension.

By selecting the link labeled `Mono?` between `Elem` and `ListCore` and right-clicking the mouse, a menu pops up from which we may select the entry `Check conservativity`. Indeed the builtin prover is able to deduce that the extension is monomorphic and the question mark in the label disappears. However, for the other two links labeled `Mono?` and `Def?` the resulting window shows that the prover can only deduce that the extensions are conservative, not that they are monomorphic respectively definitional. Since the first one should be actually easy to establish (only an equational axiom is provided), further investigations demonstrate that using the general kind of `free { }` specifications lets the proof always fail (while a corresponding proof with a `free type` declaration works); we thus suspect a limitation of the prover. However, that the second one could not be established, is not surprising: it demands convoluted reasoning that equations over free types with axioms (representing quotient term algebras) are indeed definitional. We thus replace the corresponding annotations by the simpler annotation `%cons` for which the checks succeed: the edges are subsequently labeled `Cons`.

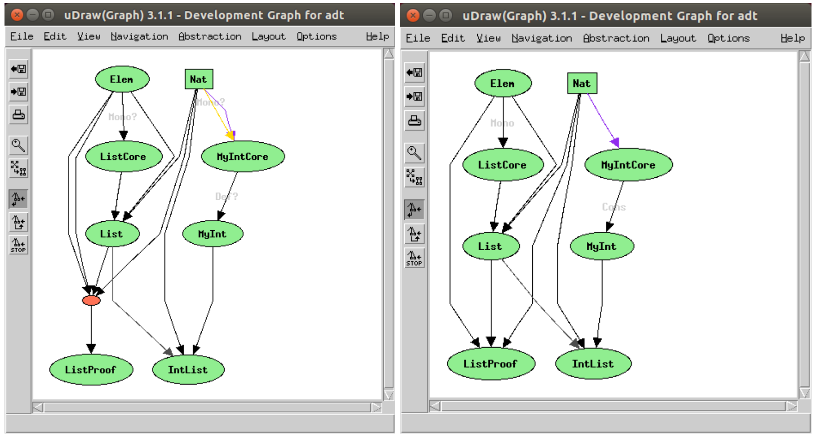

Fig. 6.18 Hets Development Graph (before and after proof)

It then remains to prove the formula introduced by the `%implies` clause. After selecting with the mouse the red node, a right-click shows a menu from which we select the `Prove` entry; this lets the proof management GUI pop up that is displayed as the left window in Figure 6.19. Here we see in the list `Goals` the formula `Ax1` to be proved; by selecting this formula and pressing the button `Display`, the window shown at the bottom of Figure 6.19 pops up and displays the formula. Furthermore, we may select in the list `Pick theorem prover` from a choice of automatic and interactive provers the one we wish to apply for the given task.

Since the stated axiom crucially depends on equational reasoning, we choose by the entry `eprover` the theorem prover E which is a powerful automatic prover for first-order logic with equality. Furthermore, since the proof is based on the principle of structural induction, we select in the list `Selected comorphism path` a sequence of logic translations from CASL to E which ends with the translation `CASL2SoftFOLInduction2` that replaces goals with induction premises. We then press the button `Prove` which lets the interface to the E prover pop us that is displayed in the right window in Figure 6.19. Pressing the button `Prove` in that window lets the proof almost immediately succeed (however, if we would not have removed the partial selectors `head` and `tail` from the specification, the proof would even after a minute not have terminated yet). Thus also the red node in the development graph disappears; the resulting view is depicted to the right of Figure 6.18.

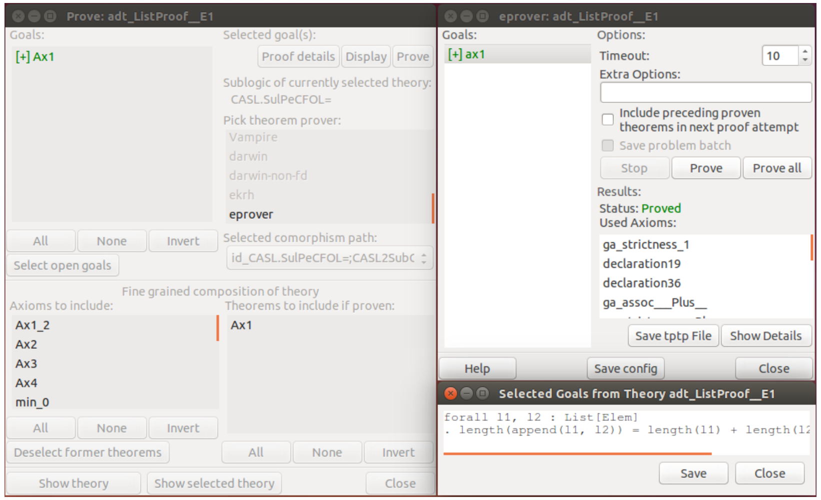

Fig. 6.19 Hets Proof Management GUI

Summary

The main benefit of CafeOBJ is that it allows to validate certain specifications by executing them and investigating the outcomes. This allows to rapidly prototype an abstract data type by first modeling and analyzing it in CafeOBJ; once its properties are thoroughly understood, it may be implemented in a more efficient form in a real programming language. However, this is only possible for specifications with initial semantics whose axioms are expressed in a restricted form of conditional equational logic, which resembles very much functional programming; the data type specifications thus look more like concrete programs than abstract theories.

The characteristic feature of CASL is its expressiveness which allows to write specifications on a very high-level of abstraction by leveraging the full power of first-order logic without being restricted by considerations of executability. Certain important aspects (such as the conservativity of extensions) may be fully automatically checked, albeit only for restricted forms of specifications. Also the specifier may state general theorems which can be proved with computer assistance. Here fully automatic proving, however, is only rarely successful; typically (at least partially) interactive proofs are required. CASL/Hets thus represents a framework for building and analyzing libraries of high-level data type theories; the comprehensive CASL standard library may serve as a starting point for own developments.

Chapter 7
Programming Languages

Alles ist eine Frage der Sprache. (Everything is a question of language.) — Ingeborg Bachmann (Alles)

In daily life, virtually all of human communication is expressed in one of the thousands of natural languages that are spoken world-wide; these languages are rich in their expressive capabilities, flexible in their applications, subtle in their nuances, and beautiful in their form. However, they are also full of gaps and ambiguities; while most of these can be usually overcome by intelligent beings that are able to deduce the intended interpretation from the context of the communication, they are from time to time are also the source of misunderstandings and disagreements, minor mishaps as well as major disasters. Thus, when communicating with ignorant partners such as computers, software developers use artificial languages that are designed in order to unambiguously express their intentions of how a computer program shall operate to solve a specific computational problem. However, even if millions of software developers use such programming languages every day, it is probably fair to say that only a minor fraction understands these languages in a sufficient depth to be able to answer subtle and critical questions about the behavior of the resulting programs. Ultimately, such an in-depth understanding requires a formal basis.

The goal of this chapter is to provide such a basis by showing how the semantics of programming languages can be precisely described in the language of logic, using the same kinds of techniques that have been introduced in the previous chapters for modeling "mathematical" languages. For this purpose, building upon the language of data types introduced in Chapter 6, Section 7.1 introduces an imperative programming language, i.e., a language whose core elements are commands that operate by reading from and writing to a common store. For this language we will give a formal type system; only well-typed programs will subsequently receive a semantics. Then Section 7.2 gives this language a "denotational" semantics that interprets commands as functions on stores; these functions are partial, i.e., may not return a result, which indicates that a program aborts or loops forever. Because partial functions are comparably inconvenient to deal with, we subsequently switch from a functional semantics to a relational one that allows arbitrarily many outcomes, which will also become useful in later chapters. Based on these results, we are able to prove the correctness of program transformations such as loop unrolling.

References

1. V.S. Alagar and K. Periyasamy. *Specification of Software Systems*. Texts in Computer Science. Springer, London, UK, 2nd edition, 2011. https://doi.org/10.1007/978-0-85729-277-3.
2. Clark Barrett and Cesare Tinelli. CVC3. Department of Computer Science, New York University, NY, USA, 2015. https://www.cs.nyu.edu/acsys/cvc3.
3. Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer, London, UK, 3rd edition, 2012. https://doi.org/10.1007/978-1-4471-4129-7.
4. Michael Bidoit and Peter D. Mosses. *CASL User Manual — Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *Lecture Notes in Computer Science*. Springer, Berlin, 2004. https://doi.org/10.1007/b11968 and http://www.informatik.uni-bremen.de/cofi/CASL-UM.pdf.
5. Dines Bjørner and Martin Henson, editors. *Logics of Specification Languages*. Springer, Berlin, Germany, 2008. https://doi.org/10.1007/978-3-540-74107-7.
6. Aaron R. Bradley and Zohar Manna. *The Calculus of Computation — Decision Procedures with Applications to Verification*. Springer, Berlin, Germany, 2007. https://doi.org/10.1007/978-3-540-74113-8.
7. Manfred Broy and Ralf Steinbrüggen, editors. *Calculational System Design*. NATO Science Series. IOS Press, Amsterdam, The Netherlands, 2000. http://www.iospress.nl/book/calculational-system-design.
8. Bruno Buchberger and Franz Lichtenberger. *Mathematik für Informatiker*. Springer, Berlin, Germany, 2nd edition, 1981. In German, http://www.risc.jku.at/publications/download/risc_2230/mathematik_informatiker_bookmarks.pdf.
9. Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985. https://doi.org/10.1145/6041.6042 and http://lucacardelli.name/papers/onunderstanding.a4.pdf.
10. Razvan Diaconescu. A Methodological Guide to the CafeOBJ Logic. In *In [5]*, pages 153–240. Springer, Berlin, Germany, 2008.
11. Gilles Dowek and Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Undergraduate Topics in Computer Science. Springer, London, UK, 2011. https://doi.org/10.1007/978-0-85729-076-2.
12. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, New York, NY, USA, 1992. https://doi.org/10.1007/978-1-4757-2355-7.
13. Maribel Fernández. *Programming Languages and Operational Semantics — A Concise Overview*. Undergraduate Topics in Computer Science. Springer, London, UK, 2014. https://doi.org/10.1007/978-1-4471-6368-8.
14. K. Futatsugi, A.T. Nakagawa, and T. Tamai, editors. *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, Amsterdam, The Netherlands, 2000.

https://www.elsevier.com/books/cafe-an-industrial-strength-algebraic-formal-method/futatsugi/978-0-444-50556-9.
15. Kokichi Futatsugi et al. Cafeobj. Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan, 2015. https://cafeobj.org.
16. Jean H. Gallier. *Logic for Computer Science — Foundations of Automatic Theorem Proving*. Harper and Row, New York, NY, USA, 1986. http://www.cis.upenn.edu/~jean/gbooks/logic.html and http://www.researchgate.net/publication/31634432_Logic_for_computer_science__foundations_of_automatic_theorem_proving__J.H._Gallier.
17. Joseph A. Goguen and Grant Malcom, editors. *Software Engineering with OBJ — Algebraic Specification in Action*, volume 2 of *Advances in Formal Methods*. Springer US, New York, NY, USA, 2000. https://doi.org/10.1007/978-1-4757-6541-0.
18. Theorema Working Group. The Theorema System. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2014. https://www.risc.jku.at/research/theorema/software.
19. Carl A. Gunter. *Semantics of Programming Languages — Structures and Techniques*. MIT Press, Cambridge, MA, USA, 1992. https://mitpress.mit.edu/books/semantics-programming-languages.
20. John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge, UK, 2009. http://www.cambridge.org/az/academic/subjects/computer-science/programming-languages-and-applied-logic/handbook-practical-logic-and-automated-reasoning.
21. C.A.R. Hoare. Programs are Predicates. *Philosophical Transactions of the Royal Society of London*, 312(1522):475–489, October 1984. https://www.jstor.org/stable/37446.
22. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International, Upper Saddle River, NJ, USA, 1998. http://www.unifyingtheories.org/.
23. Michael Huth and Mark Ryan. *Logic in Computer Science — Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK, 2nd edition, 2004. http://www.cambridge.org/az/academic/subjects/computer-science/programming-languages-and-applied-logic/logic-computer-science-modelling-and-reasoning-about-systems-2nd-edition.
24. Bart Jacobs and Jan Rutten. An Introduction to (Co)algebra and (Co)induction. In Davide Sangiori and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, number 52 in Cambridge Tracts in Theoretical Computer Science, chapter 2, pages 38–99. Cambridge University Press, Cambridge, UK, November 2011. https://doi.org/10.1017/CBO9780511792588.003 and http://www.cwi.nl/~janr/papers/files-of-papers/2011_Jacobs_Rutten_new.pdf and http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.1418 (1997 version).
25. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1990. http://homepages.cs.ncl.ac.uk/cliff.jones/publications/Jones1990.pdf.
26. Dexter Kozen and Alexandra Silva. Practical Coinduction. Technical report, Computing and Information Science, Cornell University, Ithaca, NY, USA, November 2012. http://hdl.handle.net/1813/30510.
27. Steven G. Krantz. *Handbook of Logic and Proof Techniques for Computer Science*. Birkhäuser, Boston, MA, USA, 2002. https://doi.org/10.1007/978-1-4612-0115-1.
28. Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. Springer, Berlin, Germany, 2008. https://doi.org/10.1007/978-3-540-68635-4.
29. Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer, Berlin, Germany, 1999. https://doi.org/10.1007/978-3-662-03809-3.
30. Leslie Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, Boston, MA, USA, 2002. http://research.microsoft.com/en-us/um/people/lamport/tla/book.html.

31. Wei Li. *Mathematical Logic — Foundations for Information Science*, volume 25 of *Progress in Computer Science and Applied Logic*. Birkhäuser, Basel, Switzerland, 2nd edition, 2014. https://doi.org/10.1007/978-3-0348-0862-0.
32. Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Wiley & Teubner, Chichester, UK & Stuttgart, Germany, 1996. https://books.google.at/books/about/Specification_of_Abstract_Data_Types.html?id=L7NQAAAAMAAJ&redir_esc=y.
33. Robert Lover. *Elementary Logic — For Software Development*. Springer, London, UK, 2008. https://doi.org/10.1007/978-1-84800-082-7.
34. David Makinson. *Sets, Logic and Maths for Computing*. Undergraduate Topics in Computer Science. Springer, London, UK, 2008. https://doi.org/10.1007/978-1-4471-2500-6.
35. Zohar Manna, Stephen Ness, and Jean Vuillemin. Inductive Methods for Proving Properties of Programs. *Communications of the ACM*, 16(8), August 1973. https://doi.org/10.1145/355609.362336.
36. John C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing series. MIT Press, Cambridge, MA, USA, 1996. https://mitpress.mit.edu/books/foundations-programming-languages.
37. Till Mossakowski et al. Hets — the Heterogeneous Tool Set. Research Group Theoretical Computer Science, Otto von Guericke Universität Magdeburg, Germany, 2015. http://theo.cs.uni-magdeburg.de/Research/Hets.html.
38. Till Mossakowski, Anne E. Haxthausen, Donald Sannella, and Andrzej Tarlecki. CASL — the Common Algebraic Specification Language. In *In [5]*, pages 241–298. Springer, Berlin, Germany, 2008.
39. Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst-Reichl. Algebraic-Coalgebraic Specification in CoCASL. *The Journal of Logic and Algebraic Programming*, 67(1–2):146–197, 2005. https://doi.org/10.1016/j.jlap.2005.09.006.
40. Peter D. Mosses. CASL for CafeOBJ Users. In *In [14]*, chapter 6, pages 121–144. Elsevier, Amsterdam, The Netherlands, 2000. https://doi.org/10.1016/B978-044450556-9/50066-6 and http://www.brics.dk/RS/00/51.
41. Peter D. Mosses, editor. *CASL Reference Manual — The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer, Berlin, 2004. https://doi.org/10.1007/b96103 and http://www.informatik.uni-bremen.de/cofi/CASL-RM.pdf.
42. Markus Nebel. *Formale Grundlagen der Programmierung*. Vieweg+Teubner, Wiesbaden, Germany, 2012. https://doi.org/10.1007/978-3-8348-2296-3.
43. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Berlin, Germany, 1999. https://doi.org/10.1007/978-3-662-03811-6.
44. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, London, UK, 2007. https://doi.org/10.1007/978-1-84628-692-6 and http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.ps.gz.
45. Tibas Nipkow, Lawrence Paulson, et al. Isabelle. University of Cambridge, UK, and Technische Universität München, Germany, 2015. https://isabelle.in.tum.de.
46. Tobias Nipkow and Gerwin Klein. *Concrete Semantics — With Isabelle/HOL*. Springer, Heidelberg, Germany, 2014. https://doi.org/10.1007/978-3-319-10542-0.
47. The ocaml.org Team. OCaml, 2015. https://ocaml.org.
48. University of Illinois and University of Iasi. K Framework, 2017. http://www.kframework.org.
49. Peter Padawitz. Swinging Types = Functions + Relations + Transition Systems. *Theoretical Computer Science*, 243:93–165, July 2000. https://doi.org/10.1016/S0304-3975(00)00171-7 and http://fldit-www.cs.uni-dortmund.de/%7Epeter/Rome.ps.gz.
50. Lawrence C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions. In Gordon Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction — Essays in Honour of Robin Milner*, pages 187–211. MIT Press, Cambridge, MA, USA, 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.146.1835.

51. Markus Roggenbach and Lutz Schröder. Towards Trustworthy Specification I: Consistency Checks. In Maura Cerioli and Gianna Reggio, editors, *Recent Trends in Algebraic Development Techniques: 15th International Workshop, WADT 2001 Joint with the CoFI WG Meeting Genova, Italy, April 1–3, 2001, Selected Papers*, volume 2267 of *Lecture Notes in Computer Science*, pages 305–327. Springer, Berlin, Germany, 2002. https://doi.org/10.1007/3-540-45645-7_15.
52. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA, 1997. http://www.cs.ox.ac.uk/bill.roscoe/publications/68b.pdf.
53. Kenneth Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Education, Columbus, OH, USA, 7th edition, 2012. http://highered.mheducation.com/sites/0073383090/information_center_view0/index.html.
54. Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. https://doi.org/10.1016/j.jlap.2010.03.012.
55. David Sangiorgi. On the Origins of Bisimulation and Coinduction. *ACM Transactions on Programming Languages and Systems*, 31(4):15:1–15:41, May 2009. https://doi.org/10.1145/1516507.1516510.
56. Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Springer, Berlin, Germany, 2012. https://doi.org/10.1007/978-3-642-17336-3.
57. Michael Schenke. *Logikkalküle in der Informatik — Wie wird Logik vom Rechner genutzt?* Studienbücher Informatik. Springer Vieweg, Wiesbaden, Germany, 2013. https://doi.org/10.1007/978-3-8348-2295-6.
58. David A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Allyn and Bacon, Boston, MA, USA, 1986. http://people.cis.ksu.edu/~schmidt/text/densem.html.
59. David A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, MA, USA, 1994. https://mitpress.mit.edu/books/structure-typed-programming-languages.
60. Klaus Schneider. *Verification of Reactive Systems — Formal Methods and Algorithms*. Texts in Theoretical Computer Science. Springer, Berlin, Germany, 2004. https://doi.org/10.1007/978-3-662-10778-2.
61. Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 21(3):277–291, May 2009. https://doi.org/10.1007/s00165-008-0069-4 and https://www.risc.jku.at/people/schreine/papers/fac2008.pdf.
62. Wolfgang Schreiner. The RISC ProofNavigator. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2011. https://www.risc.jku.at/research/formal/software/ProofNavigator.
63. Wolfgang Schreiner. Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 79:124–142, February 2012. Pedro Quaresma and Ralph-Johan Back (eds), Proceedings of the First Workshop on CTP Components for Educational Software (THedu'11), Wrocław, Poland, July 31, 2011, https://doi.org/10.4204/EPTCS.79.8 and https://www.risc.jku.at/research/formal/software/ProgramExplorer/papers/THeduPaper-2011.pdf.
64. Wolfgang Schreiner. The RISC ProgramExplorer. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2015. https://www.risc.jku.at/research/formal/software/ProgramExplorer.
65. Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, Amsterdam, The Netherlands, and Cambridge, MA, USA, 1990. https://mitpress.mit.edu/books/handbook-theoretical-computer-science-0.
66. Daniel J. Velleman. *How To Prove It — A Structured Approach*. Cambridge University Press, Cambridge, UK, 2nd edition, 2006. http://www.cambridge.org/at/academic/subjects/mathematics/logic-categories-and-sets/how-prove-it-structured-approach-2nd-edition.
67. Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, USA, 1994. https://mitpress.mit.edu/books/formal-semantics-programming-languages.
68. Martin Wirsing. Algebraic Specification. In *In [65]*, chapter 13, pages 675–788. Elsevier and MIT Press, Amsterdam, The Netherlands, and Cambridge, MA, USA, 1990.
69. Traian Florin Şerbănuţă. The K Primer (version 3.3). *Electronic Notes in Theoretical Computer Science*, 304:57–80, June 2014. https://doi.org/10.1016/j.entcs.2014.05.003.

© 2017 Wolfgang Schreiner.

Index