Creating a HasCASL Library

Glauber M. Cabral[1*], Arnaldo V. Moura[1†], Christian Maeder[2], Till Mossakowski[2], Lutz Schröder[2]

[1]Institute of Computing , University of Campinas, Brazil

glauber.cabral@students.ic.unicamp.br, arnaldo@ic.unicamp.br

[2]DFKI Bremen and Department of Computer Science, Universität Bremen, Germany

{Christian.Maeder, Till.Mossakowski, Lutz.Schroeder}@dfki.de

Abstract. *A prerequisite for the practical use of a specification language is the existence of a set of predefined specifications. The* HasCasl *language still lacks a library with basic data types and functions. In this paper, we describe the specification and verification of a library for the* HasCasl *language. Our library covers a subset of the Haskell Prelude library, including data types and classes representing booleans, lists, characters, strings, equality and ordering functions. We use the Heterogeneous Tool Set (*Hets*) for parsing specifications, generating theorems, and translating between the* HasCasl *and HOL languages. To verify the generated theorems, we use the Isabelle theorem prover. Because* Casl *numeric specifications use sub-typing, they cannot be translated to HOL. Although removing sub-typing allows the translation of those specification, writing proofs with them requires writing a large number of auxiliary lemmas. As writing such lemmas were out of the scope of this work, our library leaves room for a better support to functions that make use of numeric data types.*

1. Introduction

In this paper we address the modeling of a library for the specification language HasCasl [1]. The HasCasl specification language is an extension of the Casl specification language [2] which focuses on higher order properties. The contents of our library is based on the Prelude library from the Haskell programming language. In order to verify the new library we use the Hets tool [3] and the Isabelle theorem prover [4].

A prerequisite for the practical use of a specification language is the availability of a set of predefined standard specifications [5]. The Casl language has such a set of specifications in the shape of the "CASL Basic Datatypes" [6]. Currently, the HasCasl language does not have a library along the lines of the Casl library.

Although HasCasl may import the data types from the Casl library, higher order properties and data types are not available. A HasCasl library would extend the Casl library with new specifications with higher order features, such as completeness of partial orders, extended data types and parametrization change for real type dependencies [5].

*Supported by CNPq grant 132039/2007-9
†Supported by CNPq grants 472504/2007-0 and 304363/2008-1

Here, we describe the specification of a library for the HasCasl language based on the Haskell Prelude, thus making available the higher order functions and data types lacking in the Casl Basic Datatypes library while at the same time extending the support for Haskell that was the original motivation for the design of HasCasl.

Our contributions may be summarized as follows:

- Specification and verification of a library that covers a subset of the Prelude library, including the data types representing booleans, lists, characters, and strings.
- Documentation of the specification and verification process, with examples to il-

- Documentation of the specification and verification process, with examples to illustrate the use of the HASCASL framework.
- Identification of some ~~libraries~~ classes that cannot be directly imported from the CASL library when using the HASCASL language and the Isabelle theorem prover.

This paper is organized as follows. Section 2 reviews some specification frameworks ~~whose libraries~~ that exemplify the need for predefined data types and discuss some examples. Section 3 describes how we specified the library, including examples that illustrate the use ~~of the library~~. Section 4 adressess the parsing of the specifications and the generation of theorems. Section 5 addresses the use of the Isabelle theorem prover in the verification process of the library and the examples. Section 6 concludes, summarizing our contributions and discussing directions for future work.

2. Related Frameworks

There are other formal specification frameworks available. All of them include predefined libraries that can serve as a basis for new specifications.

Larch [7] and *VSE-2* [8] are two examples of specification languages based on first-order logic. *VDM* [9] and *Z* [10] are model-oriented specification languages, i.e., their specifications model a single input-output behavior. HASCASL, by contrast, allows loose specifications that can model a variety of similar behaviors in an abstract manner, allowing them to be refined later. *CafeOBJ* [11] and *Maude* [12] are specification languages that are directly executable; the price paid for this property is the reduced expressiveness of their logic in comparison with HASCASL.

Extended ML [13] creates a higher order specification language on top of the programming language ML. This approach results in a language that is hard to manage as its semantics is intermingled with the intricacies of the ML semantics. A similar approach was taken in the *Programatica* framework [14], which provides a specification logic for Haskell called *P-logic*. The similarities between HASCASL and *P-logic* include support for polymorphism and recursion based on an axiomatic treatment of complete partial orders. Because *P-logic* is built directly on top of Haskell, it is less general than HASCASL. This means that one HASCASL specification can be loosely specified with generic higher order logic in mind and later refined to the logic of Haskell programs. In contrast, *P-logic* can only specify objects in the logic of Haskell programs, including all its programming language specific features such as laziness. HASCASL also includes support for class-based overloading and constructor classes, which are needed for the specification of monads, and a Hoare logic for monad-based functional-imperative programs.

Other higher order frameworks for software specification include *Spectrum* [15] and *RAISE* [16]. The first is considered a precursor of HASCASL and differs from it in using a three-valued logic and limiting higher order mechanisms to continuous functions, i.e., it does not include a proper higher-order specification language. The language of the *RAISE* framework differs from HASCASL in the use of a three-valued logic and a lack of support for polymorphism.

In terms of the logic employed, HASCASL is related in many ways to Isabelle/HOL [17]. Indeed, Isahelle/HOL is used to provide proof support for HASCASL. Features of HASCASL not directly supported in Isabelle include higher order type constructors and constructor classes (the latter are needed e.g. for modelling side-effects via monads), subtyping, partial functions, loose generated types and advanced structured specification constructs. Similar comments apply to other higher-order theorem provers such as PVS [18].

3. Creating a HasCASL Library

We now discuss a chain of steps leading to a specification of a subset of the Haskell Prelude library. We start with some initial project decisions. Next, we use the HASCASL language to write some specifications. This is followed by a set of illustrative examples. We then use HETS to generate theorem statements and verify those using Isabelle. We close the section discussing some results that were obtained.

3.1. Initial Choices

To begin, there are design choices to be made regarding the degree to which the abstract HASCASL specification is made to match the actual Haskell types. There is a basic trade-off between having straightforward and clear higher-order specifications on the one hand and modelling all details of the Haskell behavior including laziness and an continuity of functions on the other hand. Although the latter is specifically supported in HASCASL, we have opted to start with a more abstract approach employing standard higher order function types, leaving the inclusion of programming-language specific features for later refinements of the library. The precise formal relationship between the more abstract and the more concrete levels of modelling is the subject of ongoing research.

We adopted the function and data type names from the *Prelude* library. When importing a function from the *CASL* library, we changed it to the one in the *Prelude*

the more concrete levels of modelling is the subject of ongoing research.

We adopted the function and data type names from the *Prelude* library. When importing a function from the *CASL* library, we changed it to the one in the *Prelude* library using the CASL renaming syntax. When creating new functions, we forced their names to match *Prelude* names as much as possible.

To allow an uncurried function to be passed as a parameter to functions that expect to receive a curried function, we need to create a total λ-abstraction involving the uncurried function and pass this λ-abstraction as parameter to that function. In HASCASL, the total λ-expressions are written putting the sign ! just after the period that separates λ-variables from bodies of λ-expressions.

3.2. Some Aspects of HASCASL Specification Methodology

The basic principle of property-oriented specification is to fix the required operators and predicates, the *signature*, and basic axioms governing these data. Properties implied by such a specification are also included in the specification text, but explicitly marked as theorems. Proofs in an external tool such as Isabelle often require slightly rewritten forms of the axioms and a number of lemmas. To preserve such lemmas across the development process in HETS, lemmas are also included as theorems in the specification text. All axioms and theorems are named to ease reference to them inside the theorem prover.

We started by writing the `Bool` specification, representing booleans, as follows:

```
spec Bool = %mono
free type Bool ::= True | False
fun Not__ : Bool -> Bool
fun __&&__ : Bool * Bool -> Bool
fun __||__ : Bool * Bool -> Bool
fun otherwiseH: Bool
vars x,y: Bool
. Not(False) = True %(NotFalse)%
. Not(True) = False %(NotTrue)%
. False && x = False %(AndFalse)%
. True && x = x %(AndTrue)%
. x && y = y && x %(AndSym)%
. x || y = Not(Not(x) && Not(y)) %(OrDef)%
. otherwiseH = True %(OtherwiseDef)%
. Not x = True <=> x = False %(NotFalse1)% %implied
. Not x = False <=> x = True %(NotTrue1)% %implied
. not (x = True) <=> Not x = True %(notNot1)% %implied
. not (x = False) <=> Not x = False %(notNot2)% %implied
end
```

Classes in HASCASL are similar to the ones in Haskell. A class declaration includes functions and axioms over type variables from that class. Type instances declare a type do be an instance of a class. This means that the type must obey the function declarations in the class interface, as well as the associated axioms of the class.

E.g., the specification `Eq` of user-declared equality makes use of the type class mechanism. The class `Eq` includes an equality predicate and axioms for symmetry, reflexivity and transitivity. The types `Bool` and `Unit` are declared to be instances of the class `Eq`. The axiom `%(IBE3)%` defines equality on `Bool` and, because the type `Unit` has only one constructor, the axiom `%(EqualTDef)%` already defines equality over this type.

```
spec Eq = Bool then
class Eq {
var a: Eq
fun __==__ : a * a -> Bool ; fun __/=__ : a * a -> Bool
vars x,y,z: a
. x = y => (x == y) = True %(EqualTDef)%    . x == y = y == x  %(EqualSymDef)%
. (x == x) = True                                              %(EqualReflex)%
. (x == y) = True /\ (y == z) = True => (x == z) = True        %(EqualTransI)%
. (x /= y) = Not (x == y)                                      %(DiffDef)%
. <==> x y = x == y                                            %(EqualPrefixDef)%
. (x /= y) = True <=> Not (x == y) = True           %(DiffTDef)% %implied
. (x /= y) = False <=> (x == y) = True              %(DiffFDef)% %implied
. (x == y) = False => not (x = y)                   %(IE1)% %implied
. Not (x == y) = True <=> (x == y) = False          %(IE2)% %implied
. Not (x == y) = False <=> (x == y) = True          %(IE3)% %implied
. not ((x == y) = True) <=> (x == y) = False        %(IE4)% %implied
}
type instance Bool: Eq
. (True == True) = True              %(IBE1)% %implied
. (False == False) = True            %(IBE2)% %implied
. (False == True) = False            %(IBE3)%
. (True == False) = False            %(IBE4)% %implied
. (True /= False) = True             %(IBE5)% %implied
. (False /= True) = True             %(IBE6)% %implied
. Not (True == False) = True         %(IBE7)% %implied
. Not (Not (True == False)) = False  %(IBE8)% %implied
type instance Unit: Eq
. (() == ()) = True  %(IUE1)% %implied       . (() /= ()) = False %(IUE2)% %implied
end
```

Similar use of the class mechanism is made in the specification `Ord` of total order relations. The specification includes a data type `Ordering`, which maps the conditions of being greater than, equal to, or less than to a data type. This type is made into an instance of the class `Eq` by declaring all its constructors to be distinct. As in Haskell, the class `Ord` is a subclass of the class `Eq`. Its interface includes a predicate `__<__` with axioms for irreflexivity, transitivity, and totality and a theorem for asymmetry. The other ordering functions are defined using `__<__`, `__==__` and `Not__`. The types `Ord`, `Bool`, `Unit`, and `Nat` are declared to be instances of `Ord` and equipped with axioms defining the predicate `__<__` in each case. Several theorems capture the fact that the ordering functions operate as expected over those types.

The types `Maybe a` and `Either a b`, where `a` and `b` are types, are developed in two phases necessitated by the nature of the proof support for HASCASL in Isabelle/HOL. In a first specification step, the data types themselves are declared, along with associated map functions and instance declarations for the classes `Eq` and `Ord`. In a second step, suitable instance declarations for classes are added. Specifically, the type `Maybe a` is declared to be an instance of the classes `Functor` and `Monad` and the type `Either` is declared to be an instance of the class `Functor`. This separation of a basic type definition from aspects relating to the type class mechanism salvages the possibility to use Isabelle/HOL, which fails to support constructor classes such as `Functor` and `Monad`, for proofs about the basic definitions.

Two more specifications deal with generalities about functions. The specification `Composition` contains the declaration of the function composition operator. The specification `Function` extends `Composition` by defining some standard functions including `id` and `curry`.

The largest prelude specification is the specification of lists; in fact, some of the functions from the Haskell prelude are not yet covered by our specifications, notably those involving numeric types. The specification is organized into six parts in order to collect related functions in common blocks, largely following the structure of the Haskell prelude. In the first part, the polymorphic type `free type List a` is defined, along with some basic functions including `foldr`, `foldl`, `map`, and `filter`. Two of these functions, `head` and `tail`, are inherently partial, being undefined on the empty list.

The second part of the specification declares `List a` as an instance of the classes `Eq` and `Ord` and defines how the functions `__==__` and `__<__` operate on lists. The third part contains a number of theorems over the first first part of the specification which clarify how the functions specified there interact. The fourth part contains five more list operations functions from the prelude, some of them partial, being undefined on empty lists.

The fifth part aggregates some special folding functions or functions that create sublists. The last part of this specification brings in some list functions which are not defined in the *Haskell Prelude* library, but feature in the standard module *Data.List* from the Haskell hierarchical libraries.

The specification `Char` imports the `Char` specification from the CASL Basic Datatypes library and extends it by declaring this type as an instance of the classes `Eq` and `Ord`. The `String` specification imports the specifications `Char` and `List`, defines the type `String` as `List Char`, and declares this type as an instance of the classes `Eq` and `Ord`, with the definitions of the relevant operations determined by the corresponding definitions for `Char` and `List`. A number of theorems proved over the specification confirm that the definitions have the expected behavior.

3.3. Specification Examples

To exemplify the use of our library, we created two example specifications involving ordering algorithms. In the first specification we used two sorting algorithms: *Quick Sort* and *Insertion Sort*. They were defined using functions from our library (`filter`, `__++__` and `insert`) and total lambda abstractions as parameters for the `filter` functions. In order to prove the correctness of the specification, we created four theorems involving the sorting functions, as shown below:

```
spec ExamplePrograms = List then
var a: Ord; x,y: a; xs,ys: List a
fun quickSort: List a -> List a
fun insertionSort: List a -> List a
. quickSort (Nil: List a) = Nil                                    %(QuickSortNil)%
. quickSort (Cons x xs)
    = ((quickSort (filter (\ y:a .! y < x) xs)) ++ (Cons x Nil))
        ++ (quickSort (filter (\ y:a .! y >= x) xs))               %(QuickSortCons)%
```

```
fun quickSort: List a -> List a
fun insertionSort: List a -> List a
. quickSort (Nil: List a) = Nil                                    %(QuickSortNil)%
. quickSort (Cons x xs)
    = ((quickSort (filter (\ y:a .! y < x) xs)) ++ (Cons x Nil))
        ++ (quickSort (filter (\ y:a .! y >= x) xs))               %(QuickSortCons)%
. insertionSort (Nil: List a) = Nil                                %(InsertionSortNil)%
. insertionSort (Cons x xs) = insert x (insertionSort xs)          %(InsertionSortConsCons)%
then %implies
var a: Ord; x,y: a; xs,ys: List a
. andL (Cons True (Cons True (Cons True Nil))) = True              %(Program01)%
. quickSort (Cons True (Cons False (Nil: List Bool)))
    = Cons False (Cons True Nil)                                   %(Program02)%
. insertionSort (Cons True (Cons False (Nil: List Bool)))
    = Cons False (Cons True Nil)                                   %(Program03)%
end
```

The second specification used a new data type (`Split a b`) as an internal representation for the sorting functions. We used the idea that we can split a list and then rejoin their elements according to each sorting algorithm. We then defined a general sorting function, `GenSort`, which is responsible for applying the splitting and the joining functions over a list. Below we present the specification followed by a detailed explanation.

```
spec SortingPrograms = List then
var a,b : Ord;
free type Split a b ::= Split b (List (List a))
var x,y,z,v,w: a; r,t: b; n: Nat; xs,ys,zs,vs,ws: List a; rs,ts: List b; xxs: List (List a);
    split: List a -> Split a b; join: Split a b -> List a
fun genSort: (List a -> Split a b) -> (Split a b -> List a) -> List a -> List a
fun splitInsertionSort: List b -> Split b b
fun joinInsertionSort: Split a a -> List a
fun insertionSort: List a -> List a
fun splitQuickSort: List a -> Split a a
fun joinQuickSort: Split b b -> List b
fun quickSort: List a -> List a
fun splitSelectionSort: List a -> Split a a
fun joinSelectionSort: Split b b -> List b
fun selectionSort: List a -> List a
fun splitMergeSort: List b -> Split b Unit
fun joinMergeSort: Split a Unit -> List a
fun merge: List a -> List a -> List a
fun mergeSort: List a -> List a
. xs = (Cons x (Cons y ys)) /\ split xs = Split r xxs => genSort split join xs
         = join (Split r (map (genSort split join) xxs))           %(GenSortT1)%
. xs = (Cons x (Cons y Nil)) /\ split xs = Split r xxs => genSort split join xs
         = join (Split r (map (genSort split join) xxs))           %(GenSortT2)%
. xs = (Cons x Nil) \/ xs = Nil => genSort split join xs = xs      %(GenSortF)%
```

```
. splitInsertionSort (Cons x xs)
    = Split x (Cons xs (Nil: List (List a)))                       %(SplitInsertionSort)%
. joinInsertionSort (Split x (Cons xs (Nil: List (List a))))
    = insert x xs                                                  %(JoinInsertionSort)%
. insertionSort xs = genSort splitInsertionSort joinInsertionSort xs  %(InsertionSort)%
. splitQuickSort (Cons x xs) = let (ys, zs) = partition (\t:a .! x < t) xs
      in Split x (Cons ys (Cons zs Nil))                           %(SplitQuickSort)%
. joinQuickSort (Split x (Cons ys (Cons zs Nil))) = ys ++ (Cons x zs) %(JoinQuickSort)%
. quickSort xs = genSort splitQuickSort joinQuickSort xs           %(QuickSort)%
. splitSelectionSort xs = let x = minimum xs
     in Split x (Cons (delete x xs) (Nil: List(List a)))           %(SplitSelectionSort)%
. joinSelectionSort (Split x (Cons xs Nil)) = (Cons x xs)          %(JoinSelectionSort)%
. selectionSort xs = genSort splitSelectionSort joinSelectionSort xs  %(SelectionSort)%
. def((length xs) div 2) /\ n = ((length xs) div 2)
    => splitMergeSort xs = let (ys,zs) = splitAt n xs
       in Split () (Cons ys (Cons zs Nil))                         %(SplitMergeSort)%
. xs = (Nil: List a) => merge xs ys = ys                           %(MergeNil)%
. xs = (Cons v vs) /\ ys = (Nil: List a) => merge xs ys = xs       %(MergeConsNil)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w) = True
    => merge xs ys = Cons v (merge vs ys)                          %(MergeConsConsT)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w) = False
    => merge xs ys = Cons w (merge xs ws)                          %(MergeConsConsF)%
. joinMergeSort (Split () (Cons ys (Cons zs Nil))) = merge ys zs   %(JoinMergeSort)%
. mergeSort xs = genSort splitMergeSort joinMergeSort xs           %(MergeSort)%
then
vars a: Ord; x,y: a; xs,ys: List a
preds __elem__ : a * List a;   isOrdered: List a;   permutation: List a * List a
. not x elem (Nil: List a)                                         %(ElemNil)%
. x elem (Cons y ys) <=> x = y \/ x elem ys                        %(ElemCons)%
. isOrdered (Nil: List a)                                          %(IsOrderedNil)%
. isOrdered (Cons x (Nil: List a))                                 %(IsOrderedCons)%
. isOrdered (Cons x (Cons y ys)) <=>
    (x <= y) = True /\ isOrdered(Cons y ys)                        %(IsOrderedConsCons)%
. permutation ((Nil: List a), Nil)                                 %(PermutationNil)%
. permutation (Cons x (Nil: List a), Cons y (Nil: List a)) <=> x=y  %(PermutationCons)%
. permutation (Cons x xs, Cons y ys) <=> (x=y /\ permutation (xs, ys))
    \/ (x elem ys /\ permutation(xs, Cons y (delete x ys)))        %(PermutationConsCons)%
then %implies
var a,b : Ord;   xs, ys : List a;
.insertionSort xs = quickSort xs                                   %(Theorem01)%
.insertionSort xs = mergeSort xs                                   %(Theorem02)%
.insertionSort xs = selectionSort xs                               %(Theorem03)%
.quickSort xs = mergeSort xs                                       %(Theorem04)%
.quickSort xs = selectionSort xs                                   %(Theorem05)%
.mergeSort xs = selectionSort xs                                   %(Theorem06)%
.isOrdered(insertionSort xs)                                       %(Theorem07)%
.isOrdered(quickSort xs)                                           %(Theorem08)%
.isOrdered(mergeSort xs)                                           %(Theorem09)%
.isOrdered(selectionSort xs)                                       %(Theorem10)%
.permutation(xs, insertionSort xs)                                 %(Theorem11)%
.permutation(xs, quickSort xs)                                     %(Theorem12)%
.permutation(xs, mergeSort xs)                                     %(Theorem13)%
.permutation(xs, selectionSort xs)                                 %(Theorem14)%
end
```

The Insertion Sort algorithm was implemented with the aid of a joining function

that uses the `insert` function to insert split elements into the list. The Quick Sort algorithm uses a splitting function that partitions the list in two new lists according to the function __<__, passed as parameter inside a total λ-abstraction.

The Selection Sort algorithm uses a splitting function that relies on the `minimum` function to extract the smallest element from the rest of the list. The Merge Sort algorithm splits the initial list in the middle and then joins the recursively sorted sublists by merging them into one final sorted list.

We specified three predicates to verify properties about our function denfitions.

First, `__elem__` verifies that an element is contained in a list; next, `isOrdered` guarantees that a list is correctly ordered; and then, `permutation` guarantees that one list is a permutation of the other, i.e., both lists have the same elements. Although the axiom `%(PermutationCons)%` was unnecessary to define the predicate, we included it because it was needed when writing proofs in the theorem prover.

We created theorems to verify that the application of the algorithms, in pairs, resulted in the same list; to verify that applying each algorithm to a list results in an ordered list; and to verify that a list is a permutation of the list returned by the application of each algorithm.

4. Parsing Specifications and Generating Theorems with Hets

Both the HETS tool and the Isabelle theorem prover are easily used as plugins to the *emacs* text editor. Using this integration, we can write the specifications with syntax highlight inside emacs and, then, call the HETS tool to parse the specifications and generate the so-called development graph (showing the specification structure). From the development graph we are able to start the Isabelle theorem prover to verify a specific node and visually see the proof status for all the specifications. Parsing our specifications resulted in the graph seen in Figure 1, on page 9.

As can be seen, all the red (dark gray) nodes indicate specifications that have one or more unproved theorems. The green (light gray) ones either do not have theorems or all proofs are already done. The rectangular nodes indicate imported specifications, and the elliptical ones indicate specifications from the present library. Some nodes, such as `ExamplePrograms` and `SortingPrograms`, do have theorems but are marked green because the theorems are inserted in sub-specifications.

Proofs in HETS typically start with automatic proofs over the specification structure. This method analyzes the theories and directives (`%mono`, `%implies`, `etc`), calculating dependencies between proof nodes and then revealing the hidden nodes from sub-specifications that contain theorems.

The next step is to verify each red node. To do so, we select a red node and choose the option *Prove* from the HETS node menu. This allows us to select the theorem prover to be used. At the moment, Isabelle is the only prover option for HASCASL specifications. After executing the proof inside the Isabelle interface for *emacs*, the status of the proof is sent back to the HETS tool. If the node is proved, its color changes to green; otherwise, it keeps the red color. When all the sub-nodes of a common node are proved, they become hidden again.

At present, some open proof obligations still remain, as illustrated in Figure 2, on page 10. Most of these are related to the lack of support for constructor classes in Isabelle/HOL; workarounds for this problem are under investigation.

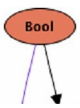

Figure 1. Initial state of the development graph.

5. Verifying Specifications with Isabelle

The task of proving the theorems generated by our specification was a major undertaking. Although some theorems remained unproved, we verified almost all of them. Next, we indicate how we constructed proofs using excerpts from interesting ones.

We started most of our proofs by applying the command `apply(auto)`, as we wanted Isabelle to act automatically as much as possible. Below, we show the proof for a theorem from the specification `Bool`:

```
theorem NotFalse1 : "ALL x.
 Not' x = True' = (x = False')"
apply(auto)
apply(case_tac x)
apply(auto)
done
```

Next, we explain the proof script commands:

- *apply (auto)*:

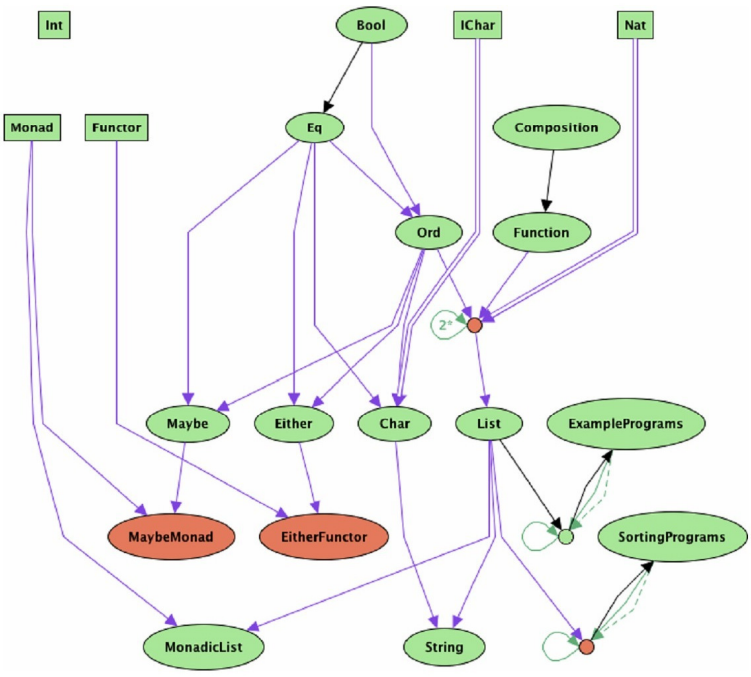

Figure 2. Later state of the development graph

Figure 2. Later state of the development graph.

This command simplifies the actual goal automatically, and goes as deep as it can in reductions. In this case, the command could only eliminate the universal quantifier, and produced the result:

```
goal (1 subgoal):
 1. !!x. Not' x = True' ==> x = False'
```

- *apply (case_tac x)*:
  The case_tac method executes a case distinction over all constructors of the data type of the variable `x`. In this case, because the type of `x` is `Bool`, `x` was instantiated to `True` and `False`:

```
goal (2 subgoals):
 1. !!x. [| Not' x = True'; x = False' |] ==> x = False'
 2. !!x. [| Not' x = True'; x = True' |] ==> x = False'
```

- *apply (auto)*:
  At this time, this command was able to terminate all the proof automatically.

```
goal:
No subgoals!
```

One example of a proof for an `Eq` theorem follows. In this proof, we used the Isabelle command `simp add:`. This command expects a list of axioms and previously proved theorems as parameters to be used in an automatic tentative of proving the actual goal. This command uses other axioms from the theory, together with the theorems passed as parameters, when trying to simplify the goal. If the goal cannot be reduced, the command produces an error; otherwise, a new goal is received.

```
theorem DiffIDef :
"ALL x. ALL y. x /= y = True'
= (Not' (x ==' y) = True')"
apply(auto)
apply(simp add: DiffDef)
```

```
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: DiffDef)
done
```

Almost all proofs of Ord theorem used the same commands and tactics as in the previous proofs. One interesting case occurred in the proof for the axiom `%(LeTAsymmetry)%`. Sometimes, Isabelle required us to rewrite axioms to match goals because it cannot change the axioms into all its equivalent forms. We applied the command `rule ccontr` to start a proof by contradiction. After some simplification, Isabelle was not able to use the axiom `%(LeIrreflexivity)%` to simplify the goal, and produced:

```
goal (1 subgoal):
 1. !!x y. [| x <' y = True'; y <' x = True' |] ==> False
```

We needed to define an auxiliary lemma, `LeIrreflContra`, which Isabelle automatically proved. This theorem was interpreted internally by Isabelle as:

```
?x <' ?x = True' ==> False
```

From here, we could tell Isabelle to use this lemma, thus forcing it to attribute the variable `x` to each `?x` variable in the lemma using the command `rule_tac x="x" in LeIrreflContra`. The same tactic was used to force the use of the axiom `%(LeTTransitive)%`. The command `by auto` was used to finalize the proof.

```
lemma LeIrreflContra :
  " x <' x = True' ==> False"
by auto

theorem LeTAsymmetry :
"ALL x. ALL y. x <' y = True'
  --> y <' x = False'"
```

```
apply(auto)
apply(rule ccontr)
apply(simp add: notNot2 NotTrue1)
apply(rule_tac x="x" in LeIrreflContra)
apply(rule_tac y="y" in LeITransitive)
by auto
```

Some applications of the command `apply(auto)` may get into a loop. An example of a loop occurred when proving theorems from the `Maybe` and `Either` specifications. To avoid the loop, we applied the universal quantifier rule directly, using the command `apply(rule allI)`. The command `rule` applies the specified theorem directly. When there were more than one quantified variable, we could use the `+` sign after the rule in order to tell Isabelle to apply the command as many times as it could.

After we removed the quantifiers, we could use the command `simp only:` to do some simplification. Differently from `simp add:`, the command `simp only:` rewrites only the rules passed to it as parameters when simplifying the actual goal. Most of the time they could be used interchangeably. Sometimes, however, `simp add:` got into a loop and `simp only:` had to be used with other proof commands. One theorem from the `Maybe` specification exemplify the use of the previous commands:

```
theorem IMO08 :
"ALL x. compare Nothing (Just(x))
==' GT = Nothing >' Just(x)"
apply(rule allI)+
apply(simp add: GeDef)
done
```

The `List` specification still had unproved theorems, namely, `FoldlDecomp` and `ZipSpec`, inside one of its sub-nodes. The other nodes had all its theorems proved.

Almost all theorems in this specification needed induction to be proved. Isabelle executes induction over a specified variable using the command `induct_tac`. It expects as parameter an expression or a variable over which to execute the induction. Below, we can see one example of a proof by induction for a `List` theorem.

```
theorem FilterProm :
"ALL f. ALL p. ALL xs.
 X_filter p (X_map f xs) =
   X_map f (X_filter
     (X__o__X (p, f)) xs)"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(case_tac "p(f a)")
apply(auto)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
done
```

The specifications `Char` and `String` used combinations of the previous commands without any great difficulty. For this reason, their proofs are not exemplified here.

Proofs of the `ExamplePrograms` theorems were very long. They were done basically using three commands: `simp only:`, `case_tac` and `simp add:`. The latter was used as the last command to allow Isabelle finish the proofs with fewer commands. Before the `simp only:` applications, we tried the `simp add:` command without success. We then used the `simp only:` command directly when the `simp add:` command failed when using the previous theorem as a parameter. Next, we show the proof for an `insertionSort` function application.

```
theorem Program03 :
"insertionSort(X_Cons True'
  (X_Cons False' Nil')) =
   X_Cons False' (X_Cons True' Nil')"
apply(simp only: InsertionSortConsCons)
apply(simp only: InsertionSortNil)
apply(simp only: InsertNil)
apply(case_tac "True' >'' False'")
apply(simp only: GeTLeTEqTRel)
apply(simp add: LeqTLeTEqTRel)
apply(simp only: InsertCons2)
apply(simp only: InsertNil)
done
```

All the theorems from our last proof, `SortingPrograms`, still couldn't be proved. Although for all of them we could prove some goals, the last one, representing the general case, is yet unproved. To show our progress in the proofs, we present an example with some comments inserted. The command `prefer` is used to choose which goal to prove when operating in the Isabelle interactive mode. The command `oops` indicates that we could not prove the theorem, and that we gave up the proof.

```
theorem Theorem07 : "ALL xs.
isOrdered(insertionSort(xs))"
apply(auto)
apply(case_tac xs)
(* Proof for xs=Nil *)
prefer 2
apply(simp only: InsertionSort)
apply(simp add: GenSortT)
(* Proof for general case *)
apply(simp only: InsertionSort)
apply(case_tac List)
apply(auto)
apply(case_tac
"X_splitInsertionSort
  (X_Cons a (X_Cons aa Lista))")
(* case xs= Cons a Nil *)
prefer 2
apply(simp add: GenSortT)
(* case xs=Cons a as*)
apply(case_tac Lista)
apply(auto)
oops
```

6. Conclusions and Future Work

In this paper, we specified a HASCASL library based on the Prelude library and discussed some application examples. We also verified the library using the HETS tool as the parser and the Isabelle theorem prover as the verification tool. We commented on some of the more difficult aspects of the process.

Our library covers a subset of the Prelude library, including the data types representing booleans, lists, characters and string. A few theorems were left unproved due to difficulties in using the Isabelle theorem prover. We left out of the specification functions

involving numbers because the numeric libraries from the CASL Basic Datatypes library couldn't be imported, as previously planned, without a major recoding effort.

The specified subset can already be used to write larger specifications. We included some example specifications involving lists and booleans to illustrate the library application. Our specification can also serve as an example for the specification of other libraries.

Our library can be extended in several ways. One can write new maps between CASL data types and their equivalent versions in the HOL language, allowing the specification and verification of numeric functions and data types from the Prelude library. Another extension could be the specification of other data types accepted by some Haskell compilers and that are not specified in the Prelude library, such as more sophisticated data structures. With more data types specified, more realistic examples could be created to serve as examples of more practical verifications.

7. Acknowledgments

The first and second authors wish to thank Rachid Rebira for his support and opinion, and CNPQ for financial support. The other authors acknowledge support from the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe).

References

[1] Schröder, L., Mossakowski, T.: Hascasl: Integrated higher-order specification and program development. Theoretical Computer Science **410**(12-13) (2009) 1217–1260

[2] Astesiano, E., Bidoit, M., Kirchner, H., Brückner, B.K., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: The Common Algebraic Specification Language. Theoretical Computer Science (2002)

[3] Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In Beckert, B., ed.: VERIFY 2007, 4th International Verification Workshop. Volume 259 of CEUR Workshop Proceedings. (2007) 119–135

[4] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)

[5] Schröder, L.: Higher order and reactive algebraic specification and development. Habilitation thesis, Universität Bremen (2005)

[6] Roggenbach, M., Mossakowski, T., Schröder, L.: CASL libraries. In: CASL Reference Manual. LNCS Vol. 2960 (IFIP Series). Springer (2004)

[7] Guttag, J.V., Horning, J.J.: Larch: languages and tools for formal specifications. Springer-Verlag New York, Inc., New York, NY, USA (1993)

[8] Hutter, D., Mantel, H., Rock, G., Stephan, W., Wolpers, A., Balser, M., Reif, W., Schellhorn, G., Stenzelß, K.: VSE: Controlling the complexity in formal software developments. In Hutter, D., Stephan, W., Traverso, P., Ullmann, M., eds.: Applied Formal Methods - FM-Trends 98 – International Workshop on Current Trends in Applied Formal Method. Volume 1641 of Lecture Notes in Computer Science., Springer; Berlin (1998) 351–358

[9] Jones, C.B.: Systematic software development using VDM. 2nd edn. Prentice-Hall, Inc. (1990)

[10] Spivey, J.M.: The Z Notation: a Reference Manual. 2nd edn. Prentice-Hall, Inc. (June 1992)

[11] Diaconescu, R., Futatsugi, K.: CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. Volume 6 of AMAST Series in Computing. World Scientific Publishing Co., Singapure (July 1998)

[12] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Predefined data modules. In Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., eds.: All About Maude - A High-Performance Logical Framework. Volume 4350 of Lecture Notes in Computer Science., Springer; Berlin (July 2007) 231–305

[13] Kahrs, S., Sannella, D., Tarlecki, A.: The definition of extended ml: a gentle introduction. Theor. Comput. Sci. **173**(2) (1997) 445–484

[14] Hallgren, T., Hook, J., Jones, M.P., Kieburtz, R.B.: An overview of the programatica toolset. In: High Confidence Software and Systems Conference (HCSS04). (2004)

[15] Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Regensburger, F., Slotosch, O., Stølen, K.: The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik (May 1993)

[16] George, C., Haff, P., Havelund, K., Haxthausen, A.E., Milne, R., Nielson, C.B., Prehn, S., Wagner, K.R.: The Raise Specification Language. Prentice Hall, New York (1992)

[17] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283. (2002)

[18] Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference, Version 2.4. SRI International, Menlo Park (2001)