# 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. Although the Common Algebraic Specification Language (*CASL*) has a library with such predefined specifications, its higher order extension, named* HASCASL*, still lacks a library with basic higher order 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 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.*

## 1. Introduction

In this paper we address the modeling of a library for the specification language HASCASL [16]. The HASCASL specification language is an extension of the CASL specification language [1] 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 [10] and the Isabelle theorem prover [12].

A prerequisite for the practical use of a specification language is the availability of a set of predefined standard specifications [15]. The CASL language has such a set of specifications in the shape of the "CASL Basic Datatypes" library [14]. 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. Another question involves proof support for HASCASL specifications that imports data types from CASL library. Translating the imported specifications to HOL language don't provide all the necessary lemmas needed by Isabelle theorem prover to write proofs involving those imported specifications. Homomorphisms are been created between data types from CASL library and native data types from HOL language to improve proof support.

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 [15].

Here, we describe the specification of a library for the HASCASL language based on the Haskell Prelude library, 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 HASCASL library that covers a subset of the Prelude library, including the data types representing booleans, lists, characters and strings, with related functions.
- Documentation of the specification and verification processes, with examples to illustrate the use of the HASCASL framework.
- Specification and partial verification of a refined library to include lazy evaluation support.

This paper is organized as follows. Section 2 reviews some specification frameworks that exemplify the need for predefined data types and discuss some examples. Section 3 describes how we specified the library, including examples that illustrate its use. Section 4 addresses 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* [5] and *VSE-2* [7] are two examples of specification languages based on first-order logic. *VDM* [8] and *Z* [17] 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* [4] and *Maude* [3] 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* [9] 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 [6], 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* [2] and *RAISE* [18]. 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 [11]. Indeed, Isabelle/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 modeling 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* [13].

## 3. Creating a HASCASL Library

The basic principle of property-oriented specifications, such as HASCASL specifications, 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 auxiliary lemmas. To preserve such lemmas across the development process in HETS, some of them are also included as theorems in the specification text. All axioms and theorems in the specifications are named to ease reference to them inside the theorem prover.

There is a basic trade-off between having straightforward and clear higher-order specifications on the one hand and modeling all details of the Haskell behavior including laziness and continuity of functions on the other hand. We chose to design our library in two steps to better understand the HASCASL language and tools, before getting into more advanced features needed to model the Haskell specificities. We started with a more abstract approach employing standard higher order function types and strict evaluation of types, described in Sections 3.1 and 3.2. Latter, we refined the library to include support for lazy evaluation of types, as described in Section 3.3.

### 3.1. Specifying a library with strict evaluation

We start our library with the `Bool` specification, representing booleans. As importing it from the CASL library would not allow the inclusion of laziness, we wrote it from scratch, 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
```

A main concept in both HASCASL and Haskell languages in type class polymorphism, allowing types and operations to depend on type variables. A type class declaration in the HASCASL language includes type variables declarations followed by function and axiom declarations depending on those type variables, serving as an interface to the declared type class. Type instances declarations defines a type as been an instance of a type

class, meaning that the type must obey the function declarations in the class interface, as well as the associated axioms of the class.

The specification `Eq` of user-declared equality makes use of the type class polymorphism. The class `Eq` includes equality and difference functions and axioms for symmetry, reflexivity and transitivity. Our equality function is not mapped to the HASCASL equality because it would be too restrictive. The definition of the difference function is made in terms of the equality function previously written. The types `Bool` and `Unit` are declared to be instances of the class `Eq`. When defining a free type, some hidden axioms are automatically created to emphasize that all the constructors, in pairs, are not equal with respect to the equality function of the language. Thus, we need to manually state that the constructors of a type we've defined are not equal with respect to our equality function. Such statement, in case of the `Bool` type, is made by the axiom `%(IBE3)%` below. Because the type `Unit` has only one constructor, the axiom `%(EqualTDef)%` already defines equality over this type. The specification can be written as follows:

```
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       %(EqualTransT)%
. (x /= y) = Not (x == y)                                     %(DiffDef)%
. (x /= y) = (y /= x)                                         %(DiffSymDef)% %implied
. (x /= y) = True <=> Not (x == y) = True                     %(DiffTDef)% %implied
. (x /= y) = False <=> (x == y) = True                        %(DiffFDef)% %implied
. (x == y) = False => not (x = y)                             %(TE1)% %implied
. Not (x == y) = True <=> (x == y) = False                    %(TE2)% %implied
. Not (x == y) = False <=> (x == y) = True                    %(TE3)% %implied
. not ((x == y) = True) <=> (x == y) = False                  %(TE4)% %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 polymorphism is made in the specification `Ord` of total order relation. 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 of each other. As in Haskell, the class `Ord` is made 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 in terms of the functions `__<__`, `__==__` 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.

Isabelle/HOL fails to support constructor classes and specifications that use them, such as `Functor` and `Monad`, cannot be translated to HOL. To allow verification using the Isabelle tool, the data types `Maybe a` and `Either a b`, which are instances of the classes `Functor` and `Monad`, are developed in two phases. 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 class `Functor` has both `Maybe a` and `Either a b` types as instances and the class `Monad` has the type `Maybe a` as instance.

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 `NumericClasses` specification consists of numeric data types and numeric classes, similar to the Prelude library. Numeric functions that are not part of any class in the Prelude library are condensed in the specification `NumericFunctions`.

We create the `NumericClasses` specification by importing the types `Nat` (for natural numbers), `Int` (for integer numbers) and `Rat`(for rational numbers) from the CASL library. We declare all of them to be instances of the classes `Eq` and `Ord`. We create the class `Num`, subclass of the class `Eq`, as the generic numeric class, comprising all the numeric types as its instances and generic axioms for defining numeric data types. Two subclasses of the class `Num`, named `Integral` and `Fractional`, are specified with their corresponding axioms and type instances. The class `Integral` has types `Nat`, `Int` and `Rat` as instances and the class `Fractional` has types `Int` and `Rat` as instances.

The list data types depends on numeric data types. As numeric data types cannot be fully translated to HOL (more on this in Section 4) we separate numeric functions and classes in two specifications to allow its verification with the Isabelle tool. Functions that express counting properties over lists, thus depending on numeric data types, are aggregated in a separate specification named `ListWithNumbers`. The main numeric specification, named `ListNoNumbers`, is organized in 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 and are 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 part of the specification which clarify how the functions specified there interact. The fourth part contains five more list operations functions from Prelude, some of them, been partial, are left undefined on empty lists.

The fifth part aggregates some special folding functions or functions that create sub-lists. 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 type `Char` from the CASL Basic Datatypes library and declares this type as instances of the classes `Eq` and `Ord`. The `String` specification imports the specifications `Char` and `ListNoNumbers`, 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.

### 3.2. Specification Examples

To exemplify the use of our library, we create two example specifications involving ordering algorithms. In the first specification we use two sorting algorithms: *Quick Sort* and *Insertion Sort*. They are defined using functions from our library (`filter`, `__++__` and `insert`) and total lambda abstractions as parameters for the `filter` functions. These total abstractions are needed to accommodate curried functions when uncurried ones are expected. In order to prove the correctness of the specification, we create three 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)%
. 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.

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 $\lambda$-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 definitions. 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. To illustrate this specification, bellow we present the generic sorting functions and functions, axioms, theorems and predicates relating to the merge sort algorithm:

```
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 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)%
. 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 = mergeSort xs %(Theorem02)% .quickSort xs = mergeSort xs %(Theorem04)%
.mergeSort xs = selectionSort xs %(Theorem06)% .isOrdered(mergeSort xs) %(Theorem09)%
.permutation(xs, mergeSort xs) %(Theorem13)%
end
```

### 3.3. Refining the library to support lazy evaluation

In HASCASL, as in the partial $\lambda$-calculus, function application is *strict*. In contrast, function application in Haskell is *lazy*, allowing function arguments to be unevaluated and, thus, yielding defined results on undefined arguments. Non-strict functions may be emulated in a strict setting by moving to functions types $Unit \to ?a$ as argument types. For a type $s$, the non-strict function type $Unit \to ?s$ may be inserted by the syntactic sugar $?s$. Thus, non-strict function types such as $?s \to ?t$ may be obtained. [16]

To support lazy function types, we change each type $s$, inside variable and function definitions, to $?s$. When declaring new types, type constructors depending on type variables must change their variables as described before and, also, must be defined as partial, including the sign `?` after the constructor. For example, the `List` type, declared as `free type List a ::= Nil | Cons a (List a)` when using strict types, should be modified to `free type List a ::= Nil | Cons (?a) (?List a)`, changing the type variables to lazy ones and defining the constructor `Cons` as partial.

One special case is the `Bool` type, which is redefined to `type Bool := ?Unit` from `free type Bool ::= True | False`. This modification associates our

boolean type with the one already defined by the HASCASL language, allowing us to remove from our axioms and theorems the comparison to the constructors of the `Bool` type, that was needed in the strict version of the library.

Although tuples are strict in HASCASL, tuples with lazy types could be used as function parameters in our library. However, this approach leads to problems with composing types with tuples, such as in the type for lists of tuples (`?List (?a,?b)`). One solution is to create a lazy type `free type Pair a b ::= Pair (?a) (?b)?` and use this type in place of tuples, as in `?List (?Pair a b)`.

## 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.

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 we have written. Some nodes, such as `ExamplePrograms` and `SortingPrograms`, do have theorems but are marked green because the theorems are inserted in sub-specifications.

Verification in HETS typically start with the automatic proof method 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*, as described in Section 5, 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 are hidden away again. The yellow color indicates that the node is proved but it lacks consistence verification inserted in the specification by the `%mono` annotation. Consistence verification was not explored in this work.

In Figure 2, the nodes highlighted with red rectangles illustrates the specifications we verified completely.

## 5. Verifying Specifications with ISABELLE

The task of proving the theorems generated by our specification was a major undertaking. We started by verifying the strict version on the library and, from a total of 17 specifications, 9 of them were completely verified and 8 of them were partially verified. Opened proofs are usually related to the lack of support for translating constructor classes

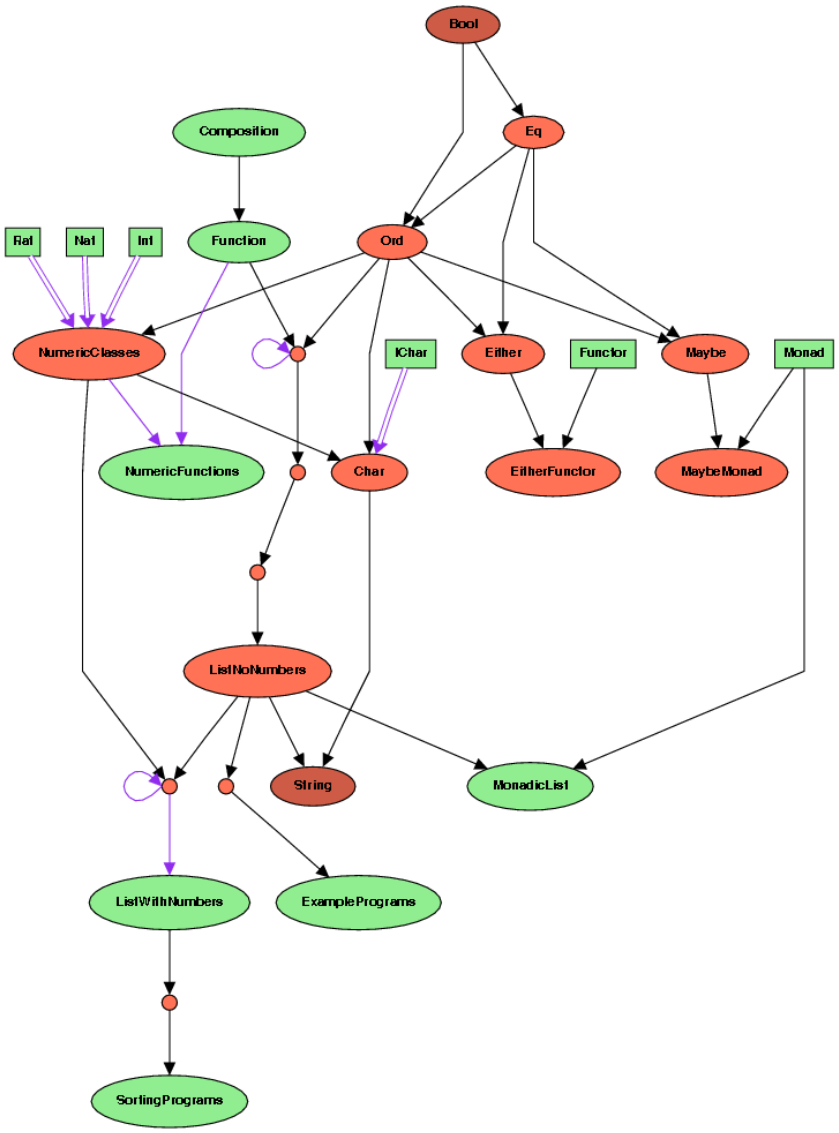


**Figure 1. Initial state of the development graph.**

to HOL language or to the lack of homomorphisms between types from the the CASL library and types from the HOL language. These homomorphisms are needed to write proofs for specifications that import types from the CASL library, such as the numeric ones. Workarounds for this problem are under research investigation.

As the support for verification of lazy specifications with the Isabelle tool is an in-progress work, some specifications from the lazy version of the library, specially those ones that make use of partiality, cannot be translated to the HOL language. Thus, from the existing 17 specifications, it was possible to fully verify the specifications `Bool`, `Eq`, `Ord`, `Either` and `Maybe` from the lazy version of the library.

Next, we indicate how we constructed proofs for theorems of the strict version of our library using excerpts from interesting ones.

Most of our proofs was initiated by applying the command `apply(auto)`, as

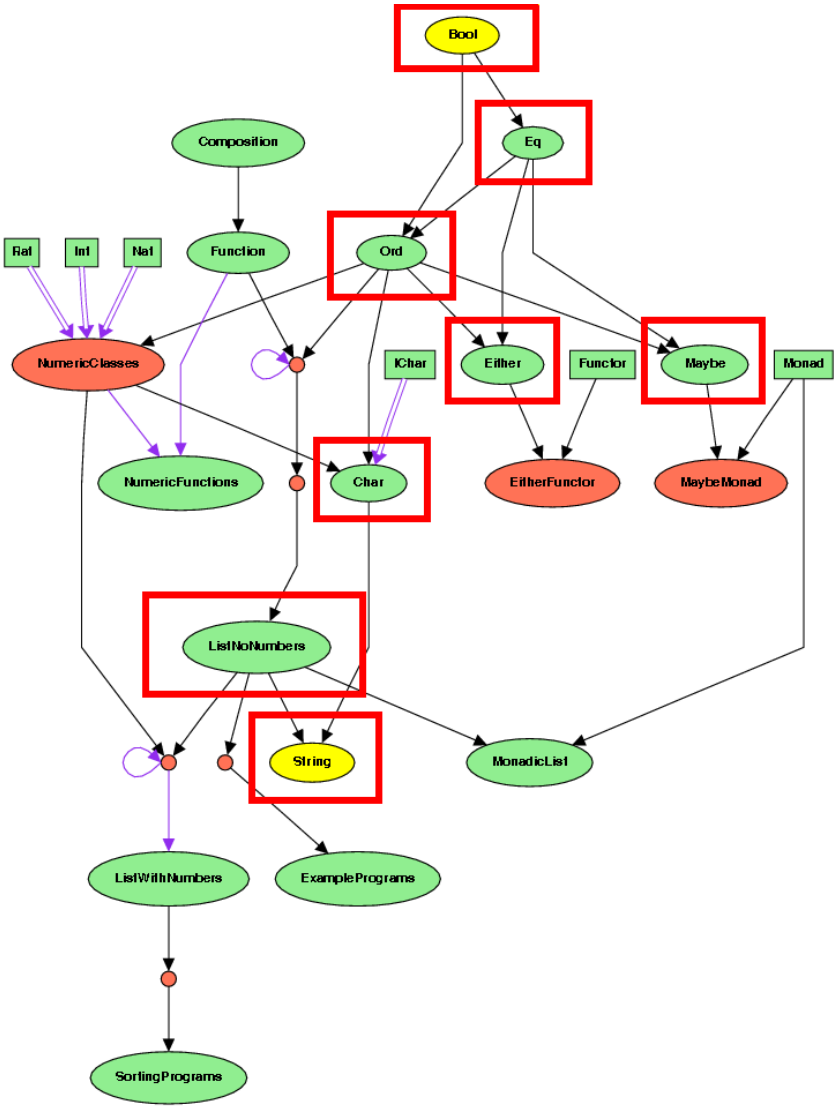


**Figure 2. Later state of the development graph.**

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)*:
  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 DiffTDef :
"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
```

Sometimes, Isabelle required us to rewrite axioms to match goals because it cannot change the axioms into all its equivalent forms. Such a case occurred with the theorem %(LeTAsymmetry)%. To prove this theorem, we applied the command rule ccontr. The command rule uses the specified rule to simplify the goal. The rule ccontr starts a 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 LeTTransitive)
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). 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 `ListNoNumber` specification has lots of recursive axioms and, thus, needs induction rules to prove the theorems. 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 `ListNoNumbers` 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 and their proofs are not described 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. 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: GeFLeTEqTRel)
apply(simp add: LeqTLeTEqTRel)
apply(simp only: InsertCons2)
apply(simp only: InsertNil)
done
```

All the theorems from our last proof, `SortingPrograms`, were left unproved as they depends on numeric types. Although for all of them we could prove some goals, the goal representing the general case is yet unproved in all the theorems. We present a proof example indicating the cases that were verified. The command `prefer` is used to choose which goal to prove when operating in the Isabelle interactive mode. The command `oops` indicates that the proof is to be left opened and Isabelle should try the next theorem.

```
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: GenSortF)
(* 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: GenSortF)
(* case xs=Cons a as*)
apply(case_tac Lista)
apply(auto)
oops
```

## 6. Conclusions and Future Work

In this paper, we discussed how to specify a HASCASL library based on the Prelude library and described 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.

We wrote a library which covers a subset of the Prelude library and is presented in two versions: one version supporting strict types and another one supporting lazy types. Each version is composed by 17 specifications, including data types – such as booleans, lists, characters, string –, classes – covering almost all classes presented in Prelude library – and functions related to these data types and classes. From the library supporting strict types we verified 9 specifications completely and 8 of them partially. As support for translating lazy types from HASCASL to HOL is limited at the moment, we were able to fully verify 5 specifications from the library supporting lazy types.

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 serve as an example for the specification of other libraries and, also, as documentation about the process of writing specifications with the HASCASL framework.

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 verification of numeric functions 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 under grants 132039/2007-9, for the first author, and 472504/2007-0 and 304363/2008-1, for the second author. The other authors acknowledge support from the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe).

## References

[1] E. Astesiano, M. Bidoit, H. Kirchner, B. K. Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 2002. URL http://citeseer.ist.psu.edu/astesiano01casl.html.

[2] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. 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. URL http://www4.informatik.tu-muenchen.de/proj/korso/papers/v10.html.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*, chapter 9. Predefined Data Modules, pages 231–305. Springer, July 2007. ISBN 978-3-540-71940-3. doi: 10.1007/978-3-540-71999-1_9.

[4] R. Diaconescu and K. Futatsugi. *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.

[5] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specifications*. Springer, 1993. ISBN 0-387-94006-5. URL http://nms.lcs.mit.edu/Larch/pub/larchBook.ps.

[6] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An Overview of the Programatica ToolSet. In *High Confidence Software and Systems Conference (HCSS04)*, 2004. URL http://www.cse.ogi.edu/~hallgren/Programatica/HCSS04.

[7] D. Hutter, H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. VSE: Controlling the Complexity in Formal Software Developments. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98 – International Workshop on Current Trends in Applied Formal Method*, volume 1641 of *Lecture Notes in Computer Science*, pages 351–358. Springer, 1998. doi: 10.1007/3-540-48257-1_26.

[8] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall, Inc., 2 edition, 1990. ISBN 0-13-880733-7.

[9] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of extended ML: a gentle introduction. *Theor. Comput. Sci.*, 173(2):445–484, 1997. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(96)00163-6. URL http://homepages.inf.ed.ac.uk/dts/eml/gentle-tcs.ps.

[10] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In B. Beckert, editor, *VERIFY 2007, 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 119–135. 2007. URL http://CEUR-WS.org/Vol-259.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 978-3-540-43376-7. doi: 10.1007/3-540-45949-9.

[13] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. SRI International, Menlo Park, 2001. URL http://pvs.csl.sri.com.

[14] M. Roggenbach, T. Mossakowski, and L. Schröder. CASL *Libraries*, volume 2960 of *Lecture Notes in Computer Science*, chapter V, pages 163–171. Springer, 2004. ISBN 978-3-540-21301-7. doi: 10.1007/b96103.

[15] L. Schröder. *Higher Order and Reactive Algebraic Specification and Development*. PhD thesis, February 2006. URL http://www.informatik.uni-bremen.de/~lschrode/papers/Summary.ps.

[16] L. Schröder and T. Mossakowski. HasCASL: Integrated Higher-Order Specification and Program Developments. *Theoretical Computer Science*, 410(12-13):1217–1260, 2009. URL http://www.informatik.uni-bremen.de/~lschrode/papers/HasCASL.pdf.

[17] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, 2 edition, 1992. ISBN 0-13-983768-X. URL http://spivey.oriel.ox.ac.uk/mike/zrm/.

[18] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, Inc., January 1993. ISBN 0-13-752833-7.