Creating a HASCASL Library

Glauber M. Cabral[1], **Arnaldo V. Moura**[1], **Christian Maeder**[2], **Till Mossakowski**[23], **Lutz Schröder**[23]

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

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

[2]DFKI Bremen

[3]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 the Heterogeneous Tool Set (*HETS*) for parsing specifications, generating proof obligations, and translating between the* HASCASL *and HOL languages. To discharge the arising proof obligations, we use the Isabelle/HOL interactive theorem prover.*

1. Introduction

In the present work, we address the modeling of a library for the specification language HASCASL [18]. The HASCASL specification language is an extension of the CASL specification language [1] by higher order language features. The structure of our library follows the Prelude library of the Haskell programming language. In order to verify the new library we use the HETS tool [13] and the Isabelle theorem prover [14].

A prerequisite for the practical use of a specification language is the availability of a set of predefined standard specifications. The CASL language has such a set of specifications in the shape of the "CASL Basic Datatypes" library [17]. 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 import data types from CASL library. Translating the imported specifications into the HOL language does not provide all the necessary lemmas needed by Isabelle theorem prover to write proofs involving those imported specifications. We have therefore implemented isomorphisms between data types from the CASL library and native data types from Isabelle/HOL 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.

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. An electronical appendix containing the sources of our library can be found at `http://sites.google.com/site/glaubersp/publications/sblp2010`.

Our contributions [4; 3] 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 process, with examples to illustrate the use of the HASCASL framework.
- Specification and partial verification of a refined library to include support for lazy evaluation.

The paper is organized as follows. Section 2 reviews some specification frameworks that exemplify the need for predefined data types, and discusses 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 proof obligations. 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* [9] are two examples of specification languages based on first-order logic. *VDM* [10] and *Z* [19] 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* [6] and *Maude* [5] 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 [11] 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 [8], 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. I.e., while HASCASL can serve as a generic higher order logic, and in particular allows for loose specification and development by refinement as already mentioned, *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* [20]. 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 the HOL language employed in the Isabelle theorem prover. [14]. Indeed, Isabelle 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), partial functions, loose generated types, and advanced structured specification constructs. Similar comments apply to other higher-order theorem provers such as *PVS* [15].

3. Creating a HASCASL Library

The basic principle of property-oriented specifications as supported by general-purpose specification languages such as HASCASL is to fix the required operators and predicates, the specification *signature*, and basic axioms governing the data semantics. Properties implied by such a specification are also included in the specification, but explicitly marked as theorems. Proofs in an external tool, such as Isabelle, often require slight adjustments to the 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. All axioms and theorems in the specification 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 in order 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. Later, we refined the library to include support for lazy evaluation of functions, as described in Section 3.3.

3.1. Specifying a library with strict evaluation

We start the specification of our library with the `Bool` specification, representing Boolean values. Note that the lowercase negation `not` as well as strong equality `=` and equivalence `<=>` are meta concepts of CASL and HASCASL. As importing `Bool` from the CASL library would not allow for the inclusion of laziness, we wrote the specification from scratch, as follows:

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

A main concept in both HASCASL and Haskell is type class polymorphism, allowing types and operations to depend on type variables. A type class declaration in the HASCASL language includes a declaration of a type variable of the new class followed by function and axiom declarations depending on this type variable, serving as an interface to the declared type class. A type instance declaration then defines a type as being 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 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 strong HASCASL builtin equality (=), because this is considered too restrictive. The definition of the difference function is given in terms of the equality function. The types `Bool` and `Unit` are then declared to be instances of the class `Eq`. From the definition of a free type, some implicit axioms are automatically created stating that all constructors are pairwise disjoint. Since user-defined equality may however be coarser than strong equality, we nevertheless need to state explicitly that the constructors of a type we have defined are not equal with respect to the user-defined equality. In case of the `Bool` type, such a statement 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
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) = notH (x == y)                               %(DiffDef)%
. (x /= y) = (y /= x)                                    %(DiffSymDef)% %implied
. (x /= y) = True <=> notH (x == y) = True               %(DiffTDef)% %implied
. (x /= y) = False <=> (x == y) = True                   %(DiffFDef)% %implied
. (x == y) = False => not (x = y)                        %(TE1)% %implied
. notH (x == y) = True <=> (x == y) = False              %(TE2)% %implied
. notH (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
. notH (True == False) = True         %(IBE7)% %implied
. notH notH (True == False) = False   %(IBE8)% %implied
type instance Unit : Eq
. (() == ()) = True    %(IUE1)% %implied
. (() /= ()) = False   %(IUE2)% %implied
```

Similar use of the class polymorphism is made in the specification `Ord` of total order relation. The specification includes a data type `Ordering`, which just enumerates the conditions of an element being greater than, equal to, or less than another. 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 `notH`__. 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 constructor classes are added. Specifically, the constructor class `Functor` has, e.g., the type constructors `Maybe` and `Either a` as instances, the latter being obtained by partial evaluation of the binary type constructor `Either`, and the constructor class `Monad` has the type constructor `Maybe` as an 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 type depends on numeric data types. As numeric data types cannot be fully translated to HOL (more on this in Section 5) we separate numeric functions and classes in two specifications so as to allow for partial verification in Isabelle/HOL. Functions that express counting properties over lists, making them dependent on numeric data types, are aggregated in a separate specification named `ListWithNumbers`. The main 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 `List a` is defined as a free type, 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 the Haskell Prelude, some of them partial, being 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 given 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 an instance 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 λ-abstraction as parameter for the `filter` function. (A λ-dot followed by an exclamation mark constructs a *total* function.) These λ-abstractions are needed to accommodate curried functions when uncurried ones are expected. As minor correctness indication of the specification, we created two small example theorems for sorting, as shown below:

```
spec ExamplePrograms = ListNoNumbers then
var a : Ord; x : a; xs : List a
fun quickSort : 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 .! notH (y < x)) xs)    %(QuickSortCons)%
fun insertionSort : List a -> List a
. insertionSort (Nil : List a) = Nil                       %(InsertionSortNil)%
. insertionSort (Cons x xs) = insert x (insertionSort xs)   %(InsertionSortConsCons)%
then %implies
. quickSort (Cons True (Cons False Nil)) = Cons False (Cons True Nil)     %(Program02)%
. insertionSort (Cons True (Cons False Nil)) = Cons False (Cons True Nil) %(Program03)%
```

The second specification uses a new data type (`Split a b`) for general sorting. The idea is to split a list into parts with at least one shorter list (given as `List (List a)`) plus a possible fixed part (of type `b`) and then rejoin the recursively sorted parts (using `map`) plus the unchanged fixed part. Splitting and joining is specific for each sorting algorithm. The general sorting function is `genSort` using function parameters `split` and `join`.

Below we present the generic sorting functions for the merge sort algorithm only, with `splitMergeSort` and `joinMergeSort` being the arguments for `genSort`. It splits the initial list in the middle and then *merges* the recursively sorted sublists. Approriate splitting and joining functions have been also defined for quick sort, insertion sort, and selection sort but are omitted here. (Selection sort splits off the `minimum` from a list and joining is done by `Cons`.)

Furthermore, you see two predicates to assert properties about our function definitions. First, `isOrdered` guarantees that a list is correctly ordered after sorting; then `permutation`, based on `elem`, `delete` and equality, guarantees that sorting does not change the list's elements. We created theorems to verify that all four sorting algorithms produce equal results and that a result list is ordered and a permutation of the input list (as given below for merge sort only).

```
spec SortingPrograms = ListWithNumbers then
var a, b : Ord;
free type Split a b ::= Split b (List (List a))
var x, y : a;
    xs, ys, zs : List a;
    r : 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
. length xs < 2 = True => genSort split join xs = xs                    %(GenSortF)%
. split xs = Split r xxs => genSort split join xs
        = join (Split r (map (genSort split join) xxs))                 %(GenSortT)%
fun splitMergeSort : List b -> Split b Unit
. splitMergeSort xs =
        let (ys, zs) = splitAt (length xs div 2) xs
        in Split () (Cons ys (Cons zs Nil))                       %(SplitMergeSort)%
fun merge : List a -> List a -> List a
. merge Nil ys = ys                                                     %(MergeNil)%
. merge xs Nil = xs                                                 %(MergeConsNil)%
. x < y = True =>
    merge (Cons x xs) (Cons y ys) = Cons x (merge xs (Cons y ys))  %(MergeConsConsT)%
. x < y = False =>
    merge (Cons x xs) (Cons y ys) = Cons y (merge (Cons x xs) ys)  %(MergeConsConsF)%
fun joinMergeSort : Split a Unit -> List a
. joinMergeSort (Split () (Cons ys (Cons zs Nil))) = merge ys zs    %(JoinMergeSort)%
fun mergeSort : List a -> List a
. mergeSort xs = genSort splitMergeSort joinMergeSort xs               %(MergeSort)%
pred isOrdered : List a
. length xs < 2 = True => isOrdered xs                              %(IsOrderedNil)%
. isOrdered (Cons x (Cons y ys)) <=>
    y < x = False /\ isOrdered (Cons y ys)                     %(IsOrderedConsCons)%
pred permutation : List a * List a
. permutation (Nil : List a, Nil)                                 %(PermutationNil)%
. 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 : Ord; xs : List a;
. isOrdered (mergeSort xs)         %(Theorem09)%
. permutation (xs, mergeSort xs)   %(Theorem13)%
```

3.3. Refining the library to support lazy evaluation

In HASCASL, as in the partial λ-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 replacing a (strict) argument type s with a partial function type $\mathit{Unit} \rightarrow ?s$. For the latter type, HASCASL provides the syntactic shorthand $?s$. Thus, non-strict function types such as $?s \rightarrow ?t$ are obtained [18].

In order to convert to lazy function types, we change each type s in variable declarations and arguments of corresponding function types to $?s$. This applies in particular to constructors of datatypes. For example, the `List` type, originally 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)
```

Although not a requisite for supporting lazy function types, we decided to move our specifications to classical logic, in which the formula `p \/ not p` is provable, in contrast to the default intuitionistic logic used by HASCASL, in which that formula is not provable. Classical logic inherits mechanized proof support from Isabelle/HOL and may thus simplify the verification process of our library. To use classical logic, the `Bool` type was redefined to `type Bool := ?Unit`. As partial functions into `Unit` can be regarded as predicates, with definedness corresponding to satisfaction [18], all functions over `Bool` could be regarded as predicates and we could remove from our axioms and theorems comparisons with `Bool` type constructors. The new specification for the `Bool` type, using classical logic and lazy function types now reads:

```
spec Bool = %mono
type Bool := ?Unit
fun notH__ : ?Bool -> ?Bool
fun __&&__, __||__ : ?Bool * ?Bool -> ?Bool
vars x, y : ?Bool
. notH false        %(NotFalse)%     . not (notH true) %(NotTrue)%
. not (false && x)  %(AndFalse)%     . (true && x) = x %(AndTrue)%
. (x || y) = notH (notH x && notH y)  %(OrDef)%
. notH x <=> not x              %(NotFalse1)% %implied
. not (notH x) <=> x            %(NotTrue1)% %implied
. not x <=> notH x              %(notNot1)% %implied
. not (not x) <=> not (notH x)   %(notNot2)% %implied
```

Besides supporting non-strict functions, HASCASL has a notion of executable specification that includes general recursion and, hence, the possibility of non-termination, in the style of a strict functional programming language. This is achieved by changing the domain semantics of the specifications[18].

In practical terms, one must import the specification named `RECURSION` and use types that are instances of the classes `Cpo` and `Cppo`, standing for *complete partial order* and *complete partial pointed order* types, respectively. As an example, the lazy list type (including finite lazy lists and infinite lists) can be specified as:

```
        var a: Cpo
        free domain LList a ::= Nil | Cons a ?(LList a)
```

General recursive function definitions using this domain semantics may be written in HASCASL as recursive equations in the standard functional programming style, by means of syntactic shorthands provided by HASCASL [18]. One thus obtains a strict functional programming language, in which non-strict functions can be emulated as previously detailed.

4. Parsing Specifications and Generating Theorems with HETS

Both the HETS tool and the Isabelle theorem prover are easy to use as plugins to the *emacs* text editor. Using this integration, we can write specifications with syntax highlighting 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 manage the proof status for all the specifications. Parsing our specifications resulted in the graph shown in Figure 1.

As can be seen, all the dark gray nodes indicate specifications that have one or more unproved theorems. The light gray ones either do not have theorems or all proofs are already done. The rectangular nodes indicate imported specifications, and the elliptical

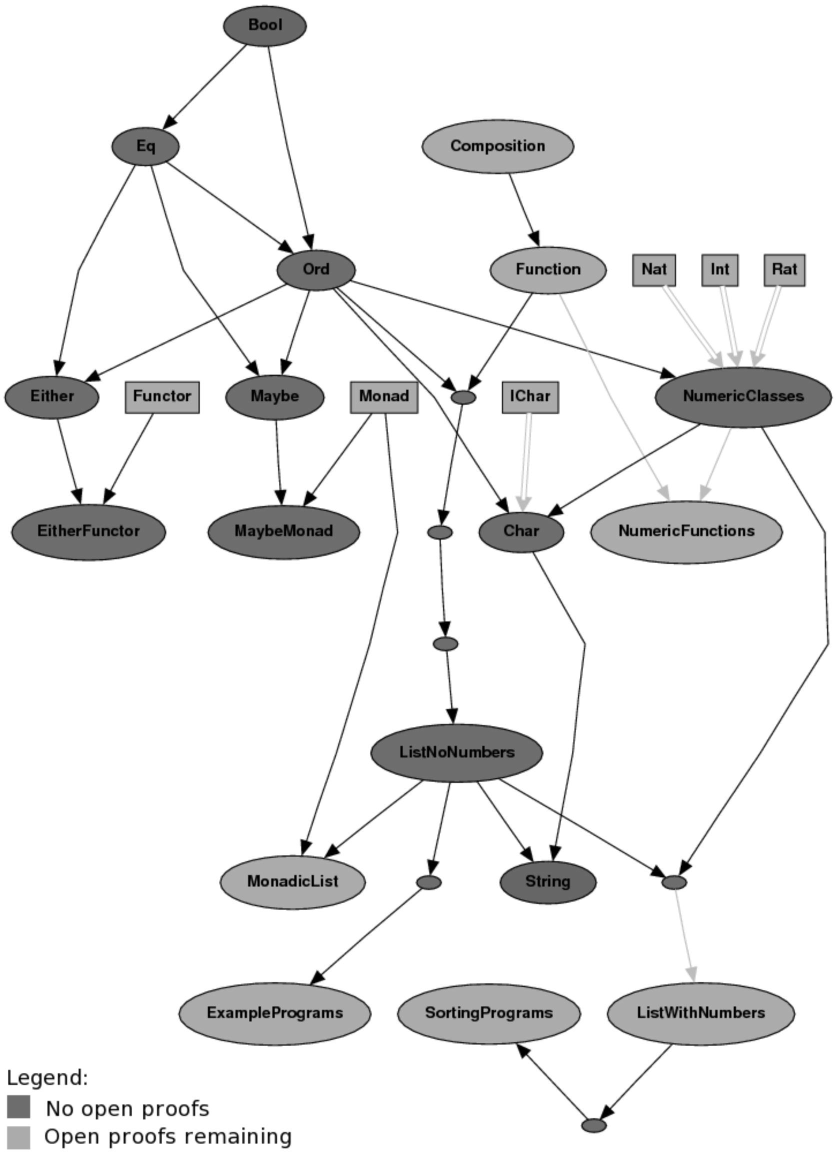

Figure 1. Initial state of the development graph.

ones indicate specifications we have written. Some nodes, such as `ExamplePrograms` and `SortingPrograms`, are marked light gray because their proof obligations have been moved into the preceding small circular dark gray nodes.

Verification in HETS typically starts 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 proof obligations.

The next step is to verify each dark gray node. To do so, we select a dark gray 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 HAS-CASL 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 a node

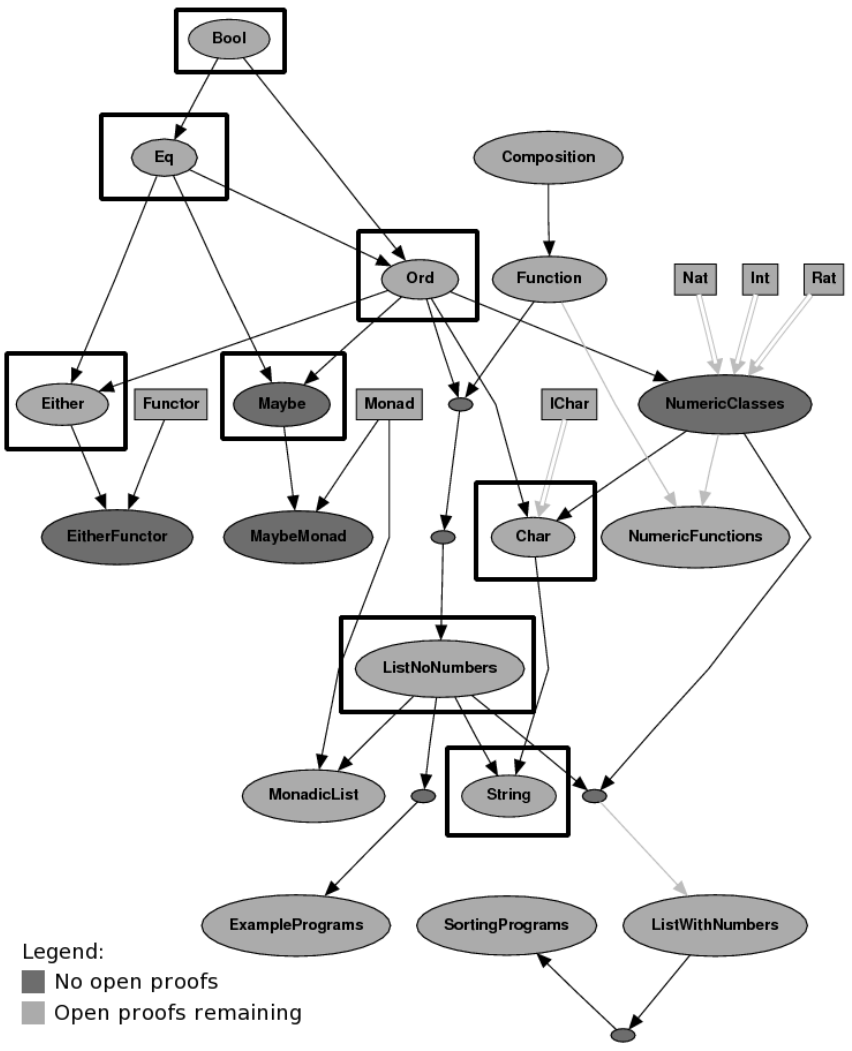

Figure 2. Later state of the development graph.

is fully proven, its color changes to light gray (and small nodes may be hidden); otherwise, it keeps the dark gray color. There can be nodes that are proved but they lack the consistency verification [16; 12] called for in the specification by `%cons`, `%mono`, or `%def` annotations that we do not elaborate here.

In Figure 2, the nodes highlighted with black rectangles represent 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 of the library and, from a total of 17 specifications, 9 of them were completely verified and 8 of them were partially verified. Unfinished proofs are usually related to the lack of support for translating constructor classes to the HOL language or to the lack of implemented isomorphisms between types from the CASL library and types from the HOL language. Such isomorphisms are needed to efficiently write proofs for specifications that import types from the CASL library, such as the numeric ones, using the extensive specific proof mechanisms already implemented for these types in the theorem prover. The implementation of a set of type isomorphisms is currently under way as part of the development of the HasCasl proof support.

As the support for verification of lazy specifications with the Isabelle tool is work in progress, some specifications from the lazy version of the library, especially those ones that make use of partiality, cannot yet be translated to the HOL language. But from the existing 17 specifications, it was possible to fully verify the specifications for `Bool`, `Eq`, `Ord`, `Either` and `Maybe` in 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 the most interesting ones. Below, we show the proof for a theorem from the specification `Bool`:

```
theorem NotFalse1 : "ALL (x :: Bool).         apply(case_tac x)
 notH x = True' = (x = False')"               apply(auto)
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. notH 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. [| notH x = True'; x = False' |] ==> x = False'
   2. !!x. [| notH x = True'; x = True' |] ==> x = False'
  ```
- *apply (auto)*:
  This time, this command was able to complete 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` with the modifier `add`, which expects a list of axioms and previously proven theorems as parameters to be used in an attempt to simplify the current goal, in addition to a default set of axioms from the theory that the prover collects automatically. If the goal cannot be reduced, the command produces an error; otherwise, a new goal is obtained. In the case at hand, we need to add the definition of an operator; definitions are usually not included in the default simplification set to maintain encapsulation of definitions.

```
theorem DiffTDef :                             apply(case_tac "x ==' y")
"ALL (x :: 'a). ALL (y :: 'a).                 apply(auto)
 x /= y = True' = (notH (x ==' y) = True')"   apply(simp add: DiffDef)
apply(auto)                                    done
apply(simp add: DiffDef)
```

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 proof by contradiction. After some simplification, Isabelle stopped at a point where application of the axiom `%(LeIrreflexivity)%` was needed but the original form of the axiom did not quite fit; specifically, we arrived at:

```
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, namely:

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

We applied this lemma manually, instantiating the unification variable `?x` with `x` using the command `rule_tac x="x" in LeIrreflContra`. The same tactic was used to force the application of the axiom `%(LeTTransitive)%`. The command `by auto` was used to finalize the proof.

```
lemma LeIrreflContra :                 apply(auto)
  " x <' x = True' ==> False"          apply(rule ccontr)
by auto                                apply(simp add: notNot2 NotTrue1)
                                       apply(rule_tac x="x" in LeIrreflContra)
theorem LeTAsymmetry :                 apply(rule_tac y="y" in LeTTransitive)
"ALL x. ALL y. x <' y = True'          by auto
  --> y <' x = False'"
```

Application of the command `apply(auto)` may sometimes 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)`. If 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 removing the quantifiers, we manualy applied simplification again, this time – in order to avoid the loop – using the modifier `only` to limit the set of axioms applied in the reduction.

Here is a theorem from the `Maybe` specification exemplifying the use of some of the mentioned commands:

```
theorem IMO03 : "ALL (x :: 'o).            apply(case_tac "Just(x) <' Nothing")
  Nothing >=' Just(x) = False'"            apply(auto)
apply(rule allI)                           done
apply(simp only: GeqDef GeDef OrDef)
```

The `ListNoNumber` specification has many 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 show an example of a proof by induction for a theorem in `ListNoNumbers`.

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

The theorems from the specification `SortingPrograms` were left unproven although we could prove some subgoals. 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 open and Isabelle should try the next theorem.

```
theorem Theorem06 : "ALL xs.
 mergeSort(xs) = selectionSort(xs)"
apply(auto)
apply(case_tac xs)
apply(case_tac List)
apply(auto)
(* Proof for goal 3 *)
prefer 3
apply(simp add: MergeSort SelectionSort)
apply(simp add: GenSortF)
```

```
(* Proof for goal 2 *)
prefer 2
apply(simp add: MergeSort SelectionSort)
apply(simp add: GenSortF)
apply(simp add: MergeSort SelectionSort)
apply(case_tac "X_splitSelectionSort
 (X_Cons a (X_Cons aa Lista))")
apply(case_tac "X_splitMergeSort
 (X_Cons a (X_Cons aa Lista))")
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 involved aspects of the process.

We wrote two versions of a library which covers a subset of the Prelude library: one version supporting strict types and another one supporting lazy types. Each version is composed of 17 specifications, including data types – such as Booleans, lists, characters, string –, classes – covering almost all classes present in the 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 in 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 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 FormalSafe, FKZ 01IW07002).

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. *Theoret. Comput. Sci.*, 286:153–196, 2002.

[2] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The requirement and design specification language SPECTRUM, an informal introduction, version 1.0. Technical report, Department of Informatics, Technical University of Munich, 1993.

[3] G. M. Cabral. Criação de uma biblioteca padrão para a linguagem HASCASL. Master's thesis, Institute of Computing, University of Campinas, May 2010.

[4] G. M. Cabral and A. V. Moura. Creating a HASCASL library. Technical Report IC-09-03, Institute of Computing, University of Campinas, January 2009. 68 pages.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, vol. 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[6] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, vol. 6 of *AMAST Series in Computing*. World Scientific, Singapore, July 1998.

[7] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specifications*. Texts and Monographs in Computer Science. Springer, 1993.

[8] 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.

[9] 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, eds., *Applied Formal Methods - FM-Trends 98 – International Workshop on Current Trends in Applied Formal Methods*, vol. 1641 of *LCNS*, pp. 351–358. Springer, 1998.

[10] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall, 2nd edition, 1990.

[11] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoret. Comput. Sci.*, 173(2):445 – 484, 1997.

[12] C. Lüth, M. Roggenbach, and L. Schröder. CCC - the CASL consistency checker. In J. Fiadeiro, ed., *Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004)*, vol. 3423 of *LNCS*, pp. 94–105. Springer, 2005.

[13] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In B. Beckert, ed., *VERIFY 2007, 4th International Verification Workshop*, vol. 259 of *CEUR Workshop Proceedings*, pp. 119–135. 2007.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[15] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. SRI International, Menlo Park, 2001.

[16] M. Roggenbach and L. Schröder. Towards trustworthy specifications I: Consistency checks. In M. Cerioli and G. Reggio, eds., *Recent Trends in Algebraic Specification Techniques, 15th International Workshop, WADT 2001*, vol. 2267 of *LNCS*. Springer, 2001.

[17] M. Roggenbach, L. Schröder, and T. Mossakowski. Libraries. In P. Mosses, ed., CASL *reference manual*, vol. 2960 of *LNCS*, chapter V, pp. 163–171. Springer, 2004.

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

[19] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, 2 edition, 1992.

[20] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, Inc., January 1993.