Glauber M. Cabral[1] and Arnaldo V. Moura[2]

[1] Instituto de Computação, Universidade Estadual de Campinas, 13081-970, Campinas, SP. Suporte CNPq Processo: 132039/2007-9
ra069271@students.ic.unicamp.br

[2] Instituto de Computação, Universidade Estadual de Campinas, 13081-970, Campinas, SP. Suporte CNPq Processo: 305781/2005-7
arnaldo@ic.unicamp.br

Abstract. escrever

1 Introduction

escrever

2 Related Frameworks

that can

There are other formal specification frameworks available. All of them include example libraries, ~~to~~ serve as a basis for new specifications, or predefined libraries, ~~to~~ be imported by larger specifications.

that can

Larch [1] and *VSE-2* [2] are two examples of specification languages based on first-order logic. *VDM* [3] and *Z* [4] are model-oriented specification languages, i.e., their specifications model a single input-output behavior. *HasCASL*, in contrast, contains loose specifications that can model a variety of similar behaviors in an abstract manner, allowing them to be refined later. *CafeOBJ* [5] and *Maude* [6] 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 [7] creates a higher order specification language on top of the programming language *ML*. This approach resulted in a large language that is very difficult to manage. Similar approach was taken by the *Programatica* framework [8], which provides a specification logic for the *Haskell* language, called *P-logic*. The similarities between *HasCASL* and *P-logic* include~~s~~ the 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 opposite, *P-logic* can only specify objects in the logic of *Haskell* programs, including all its specialities, such as laziness. *HasCASL* also includes support for class based overloading and constructor classes, needed for the specification of monads, and the Hoare logic for imperative (monad-based) programs.

A; 's; itálico?; ion; features

Other higher order frameworks for software specification include *Spectrum* [9] and *RAISE* [10]. The first is considered a precursor of *HasCASL* and differs from it by using a three-valued logic and by limiting higher order mechanisms to continuous functions, as it doesn't have a proper higher-order specification language. The language of the *RAISE* framework differs from *HasCASL* because of the three-valued logic and the lack of support for polymorphism.

We start with some initial project decisions. Next, we use the ~~Has~~CASL 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~~ obtained.

3 Creating a HasCASL Library

Here we ~~explain our~~ discuss a chain of steps ~~into specifying~~ leading to a specification of a subset of the *Haskell Prelude* library. ~~starting with some project decisions and later explaining the needed steps to achieve a verified library in *HasCASL* specification language.~~

3.1 ~~Initial~~ First Choices

~~To start specifying~~, we had to choose between using or not using laziness and continuous functions. Laziness ~~would allow~~ would be our library to fully capture Haskell semantic, and continuous functions ~~are~~ needed to construct infinite data types. Since starting with these functionalities would require using the most advanced constructions of the *HasCASL* language and would also require deep knowledge of *Isabelle* proof scripts, we decided to start specifying the target library using strict types, letting ~~and let~~ more advanced *Haskell* features to be included within a latter refinement.

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

Differently from *Haskell*, *HasCASL* doesn't allow the same function to be used both in prefix and infix notation. This leaded to a compatibility problem between functions already defined and the curried functions we ~~defined~~. To solve this problem, we redefined functions from the *CASL* library ~~in~~ into a mixfix way and, for each mixfix definition, we created a curried version whose name was formed by enclosing the name of the mixfix function between brackets. This solution created a pattern for naming curried functions that was easy to remember and allowed all of our functions to be curried with other previously defined functions.

3.2 Using HasCASL ~~to~~ Write Specifications

As we ~~would~~ planned to later refine our library to ~~use~~ allow for laziness ~~later~~, we decided to write it entirely from scratch. We defined functions in curried form and, as explained before, when we had to define mixfix functions ~~for~~ to avoid compatibility issues, we also included a curried version of the same function. Properties over the specifications were included as theorems. The *Isabelle* theorem prover sometimes ~~needs~~ require us to rewrite ~~our~~ axioms in a different form, creating auxiliary lemmas, so the ~~lemmas~~ statements can be matched against the proof goals. To retain these lemmas in the specification source, avoiding loosing them when regenerating proof scripts via the *Hets* tool, we included the lemmas as theorems inside the specifications. All axioms and theorems ~~are~~ were named ~~to be~~ so as they could be easily referred to inside the theorem prover.

We started by writing the `Bool` specification, representing booleans. See Specification 3.1, on page 3.

Classes in *HasCASL* are similar to the ones in *Haskell*. A class declaration includes functions and axioms over variables from that class. Type instances declares a type do be an instance of a class. This means that the type must obey the function declarations from the class. Here there is a difference ~~from the~~ when compared to the

Specification 3.1 Boolean Specification

```
spec Bool = %mono
free type Bool ::= True | False
fun Not__ : Bool -> Bool
fun __&&__ : Bool * Bool -> Bool
fun <&&> : Bool -> 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)%
. <&&> x y = x && y                 %(AndPrefixDef)%
. <||> x y = x || y                 %(OrPrefixDef)%
. 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
```

Haskell classes: *HasCASL* type instances do not include a default declaration. Thus, we We ~~must~~ had to explicitly define how functions from a class operate over were to the type whose instance is been declared.

The next specification we created was for Equality, containing equality and inequality functions. We created a class named `Eq` and included axioms for Symmetry, Reflexivity and Transitivity of the equality function. We mapped the equality from *HasCASL* to our equality function. Our function was not mapped to the *HasCASL* one because it would be too restrictive. We declared the `Bool` and the `Unit` types as instances of the class `Eq`. The axiom `%(IBE3)%` defines the equality for the `Bool` type and, because the type `Unit` has only one constructor, the axiom `%(EqualTDef)%` already defines equality over this type. See Specification 3.2, on page 5.

Our next specification was `Ord`, for Ordeding r relations. Although there is an `Ord` specification in the *HasCASL* repository, this version uses laziness and earlier was incompatible with ~~our project~~ decisions. Our specification includes the data type `Ordering`, which maps to a data type the condition of been greater then a, equal to and lesser then. To declare the type as an instance of the class `Eq` we defined that the equality between the constructors was `False`. The class `Ord` was defined as a subclass of the class `Eq`, as in *Haskell*. We defined the function included `__<__` with axioms for i Irreflexivity, t Transitivity and t Totality and a theorem for a Asymmetry. The other ordering functions were defined using combinations of ~~between~~ the functions `__<__`, `__==__` and `Not__`. We declared the types `Ord`, `Bool` and `Unit` as instances of this class through axioms that defined how the function `__<__` operates over constructors of each one of those types. The type `Nat` was also declared an instance of the class `Ord`. Theorems were used to verify that the other ordering functions operate as expected over those types.

The monadic types `Monad a` and `Either a b`, where `a` and `b` are types, were included in two phases. First, we created one specification for each one of those data types. These specifications included the data type declaration itself, an associated function responsible for applying a function to an element of that data type, and instance declarations for the classes `Eq` and `Ord`. Next, we created specifications to include monadic properties of those data types. The type `Maybe a` was declared an instance of the classes `Functor` and `Monad` and the type `Either` was declared an instance of the class `Functor`. We declared the monadic properties in separated specifications ~~because~~ used by the monadic properties still cannot be translated into ~~to~~ the language HOL of the `Isabelle` theorem prover. This separation allowed us to prove theorems for type instance declarations from both types and also specify their monadic properties.

We created two specifications in order to define functions. The specification `Composition` contains the declaration of the function composition operator. *HasCASL* already contains a composition specification that uses λ-expressions. To facilitate the proof inside *Isabelle*, we created another specification using function application in place of the λ-expression. The specification `Function` extends the `Composition` ~~one~~ specification by defining some functions like `id` and `curry`.

Specification 3.2 Equality specification

```
spec Eq = Bool then
class Eq {
var a: Eq
fun __==__ : a * a -> Bool
fun <==> : a -> a -> Bool
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 = x == y                                             %(EqualPrefixDef)%
. </=> x y = x /= y                                             %(DiffPrefixDef)%
. (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
```

The list specification was the largest one and it still doesn't aggregate all the functions that the *Haskell Prelude* defines, specially those involving numeric types. Once again, we had to redefine our specification to remove laziness. We divided this specification in six parts in order to bring related functions together, in almost the same way as the *Haskell Prelude* does. In the first part, we defined ~~the~~ `free type List a`, depending on a type `a`, and some general functions like: `foldr`, `foldl`, `map`, `filter`, and others. Two of them, `head` and `tail`, had to made be partial, as they are not defined when applied to ~~on~~ an empty list. paragraph The second part of the specification declares ~~type~~ `List a` as instance of the classes `Eq` and `Ord` by defining how functions `__==__` and `__<__` work over lists. The third part contains eight theorems involving some functions of the first part of the specification. These theorems are needed in order to specify how those functions interact. They should not be defined as axioms because they must follow from the function definitions. The fo u rth part contains five functions that are listed in the *Haskell Prelude* as List operat~~ions~~ ors from the first part. Again, some of these functions had to be partial as they are not defined on empty lists. The fifth part aggregates some special folding functions or functions that create sublists. The last part of this specification brings in functions related to Lists and that are not defined in the *Haskell Prelude*, but are implemented library on every compiler and are necessary even to write basic programs.

Our `Char` specification imports the `Char` specification from *CASL 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 instances of the classes `Eq` and `Ord`. Because the types `Char` and `List` are also instances of those classes, we didn't need to include new axioms ~~to~~ in the type instance declarations. ~~To~~ In order to verify that ~~this works~~ that the specifications worked as expected, we proved some theorems involving operators from those classes.

3.3 Specification Examples

To exemplify the use of our library, we created two example specifications involving ordering algorithms. See Specification 3.3, on page 9. 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 expressions as parameters for the `filter` functions. The λ-expressions were made total by using the sign `!` just after the final ~~point~~ period that separates variables from expressions. In order to prove the correctness of the specification, we created four theorems involving the sorting functions.

The second specification use~~s~~ d a new data type (`Split a b`) as an internal ~~representation for the sorting~~ in intermediate and final functions. We used the idea that we can split a list and then rejoin their elements, following each algorithm. We defined a general sorting function, `GenSort`, which is responsible for applying Then the splitting and the joining functions over a list.

The Insertion Sort algorithm ~~is~~ was implemented ~~by~~ with the aid of a joining function that uses the `insert` function to insert split elements into the list. The Quick Sort algo-

rithm uses a splitting function that separates the list in two new lists: a ~~the~~ first one
containing elements smaller than the first element of the original list and ~~the~~ a
one ~~second~~ with the other elements. The joining function inserts an element in the
middle of two lists.

The Selection Sort algorithm uses a splitting function that relies on the `minimum` function to extract the smaller element from the rest of the list. The joining function just joins two lists. The Merge Sort algorithm is implemented by splitting the initial list in the middle, using the splitting function, and then merging the elements using a joining function. The latter takes the smaller head of both lists and then merges the other list and the remaining elements of the list from which the head was taken.

We specified two predicates found in the *CASL* library repository (but not in the *CASL* Library itself). First, `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. Finally, 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.

3.4 Parsing Specifications and Generating Theorems with Hets

Although *Hets* can deal with specifications in different files, we decided to write all the specifications from the previous section in a single file ~~to easy~~ for the case of management. We used the *Hets* plugin for the *emacs* text editor to parse and generate the *Isabelle* files. This plugin allowed us to ~~simply~~ parse our specification, to check for mistakes, and also to parse and generate the graph of theories based on a ~~the~~ syntactic analysis.

Parsing ~~our~~ the specifications generated the graph shown in Figure 1, on page 10. In our case.

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

We started our proofs by using the automatic proof mode of *Hets* (menu: Edit / Proofs / Automatic). This method analyzed the theories and directives (`%mono`, `%implies, etc`) and then revealed the nodes from sub-specifications that created theorems, for example, by the `%implied` directive.

The next step was to ~~prove~~ verify each red node. To do so, we ~~did a~~ right clicked on a node and chose the option *Prove* from the *node menu.* This opened the Emacs text editor. After *Isabelle* had parsed the full theory file (and proved it or not, according to ~~*Isabelle*~~ its rules), we closed the Emacs window. ~~and thus~~ By doing so the proof status for that theory was reported back to *Hets* by *Isabelle.* If the node was proved, its color changed to green; otherwise, it kept the red color. If sub-nodes were proved, they were omitted again by *Hets.*

paragraph

At this point, we could not yet prove all the theorems we had created. Most of the unproved nodes had yet one

Specification 3.3 Specification Examples: ExamplePrograms and SortingPrograms - Part 1

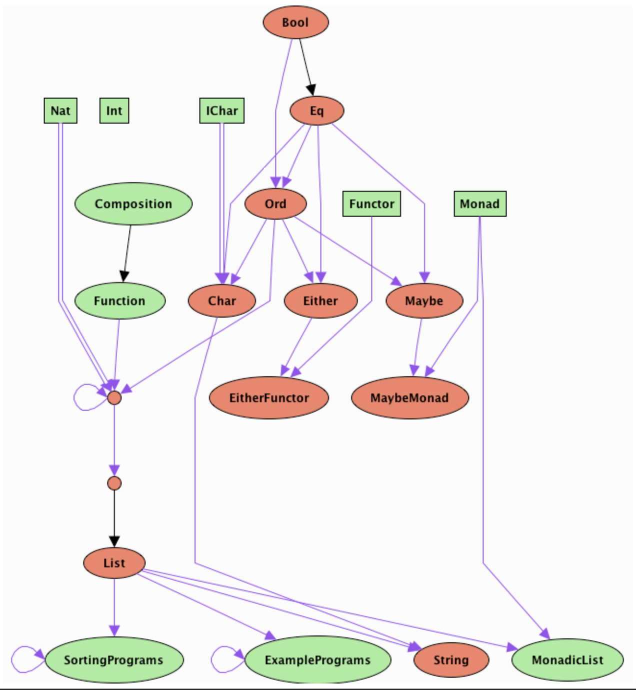
```
spec ExamplePrograms = List then
var a: Ord; x,y: a; xs,ys: List a
fun quickSort: List a -> List a
fun insertionSort: List a -> List a
. quickSort (Nil: List a) = Nil                         %(QuickSortNil)%
. quickSort (Cons x xs)
     = ((quickSort (filter (\ y:a .! y < x) xs))
        ++ (Cons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

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)%
```

Specification 3.3 Specification Examples: ExamplePrograms and SortingPrograms - Part 2

```
. splitQuickSort (Cons x xs)
     = let (ys, zs) = partition (<<> x) 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)%
     => unzip (zip xs ys) = (xs, ys)                       %(ZipSpec)%
. 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 isOrdered: List a;
      permutation: List a * List a
. 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) = True                                 %(PermutationCons)%
. permutation (Cons x xs, Cons y ys) <=>
     ((x==y) = True /\ permutation (xs, 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
```

Figure 1 Initial state of the proof graph.



or two theorems to be proved. The actual status of our proofs can be seen in Figure 2, on page 11.

3.5 Verifying Specifications with Isabelle

The task of proving the theorems ~~from our~~ 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 ~~our~~ proofs using excerpts from interesting ones.

The four theorems from the specification `Bool` were translated by *Hets* to *Isabelle* theorems, like the one shown by Isabelle Proof Script Excerpt 3.1, on page 11.

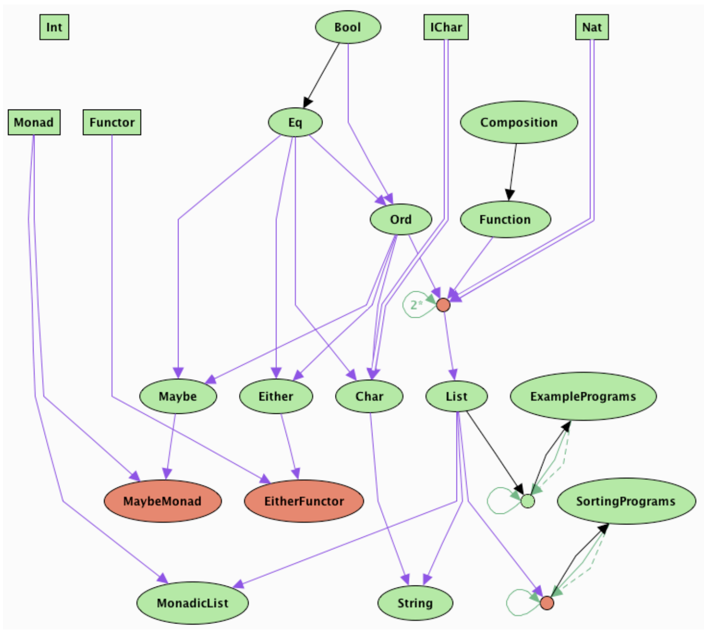
All the proofs for the theorems of the `Bool` specification followed this pattern:

– *apply (auto)*:
  This command ~~tries to~~ simplif~~y~~ies the actual goal automatically, and goes as deep as it can. In this case, the command could only eliminate the universal quantifier, ~~getting~~ and produced the result:

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

– *apply (case_tac x)*:

Figure 2 Actual state of the proof graph.



Isabelle Proof Script Excerpt 3.1 Proof for theorem NotFalse1 from Bool specification

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

the case_tac method executes a case distinction over all constructors of the data type of 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 ~~could finalize~~ was able to terminate all the proof automatically.

```
goal:
No subgoals!
```

One example of a proof for an `Eq` theorem is shown in the Isabelle Proof Script Excerpt 3.2, on page 12. In this proof, we used a new command~~,~~, manually! `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.

Isabelle Proof Script Excerpt 3.2 Equality proof

```
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
```

Almost all Ord theorem proofs used the same commands and tactics ~~from~~ as in the previous proofs. One interesting ~~proof~~ case was the ~~one~~ proof for the axiom `%(LeTAsymmetry)%`, presented in the Isabelle Proof Script Excerpt 3.3, on page 13. Sometimes, *Isabelle* ~~expected~~ required us to rewrite axioms to match goals because it cannot change the axioms ~~to~~ into all ~~their~~ 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 ~~is~~ was interpreted internally by *Isabelle* as:

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

From here,

~~Hence~~, 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.

Isabelle Proof Script Excerpt 3.3 Proof for the axiom LeTAsymmetry from specification Ord.

```
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
```

We started most of our proofs by applying the command `apply(auto)`, as we wanted *Isabelle* to act automatically as much as possible. Sometimes this command could do some reductions. Sometimes it could only remove *HOL* universal quantifiers. Sometimes it got 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. Two theorems from the `Maybe` specification exemplify the use of the previous commands, as shown in the Isabelle Proof Script Excerpt 3.4, on page 14. namelly

The `List` specification still had unproved theorems, {`FoldlDecomp` and `ZipSpec`} inside one of its sub-nodes. The other nodes had ~~could have~~ all its theorems proved.

parag.

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

Isabelle Proof Script Excerpt 3.4 Proof for theorems IMO05 and IMO08 from specification Maybe.

```
theorem IMO05 : "ALL x. Just(x) <' Nothing = False'"
apply(rule allI)
apply(case_tac "Just(x) <' Nothing")
apply(auto)
done

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

induction. In the Isabelle Proof Script Excerpt 3.5, on page 14, we can see one example of a proof by induction for a `List` theorem.

Isabelle Proof Script Excerpt 3.5 Proof for theorem FilterProm from specification List.

```
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 specification `Char` was another case where we had to use the `rule` command to remove universal quantification by hand in order to avoid loops. ~~Besides~~ Except this problem, all theorems needed only one or two applications of the command `simp add:` to be proved. An example can be seen in the Isabelle Proof Script Excerpt 3.6, on page 15.

The specification for `String` also used few commands in order to have its theorems proved. Almost all proofs were done ~~with~~ combin~~ations~~ing ~~of~~ the `auto` and ~~the~~ `simp add:` commands. ~~the~~ In the Isabelle Proof Script Excerpt 3.7, on page 15, we show the largest proof in the `String` theory.

Isabelle Proof Script Excerpt 3.6 Proof for theorem ICO07 from specification Char.

```
theorem ICO07 : "ALL x. ALL y. ord'(x) >='' ord'(y) = x >='' y"
apply(rule allI)+
apply(simp only: GeqDef)
apply(simp add: GeDef)
done
```

Isabelle Proof Script Excerpt 3.7 Proof for theorem StringT2 from specification String.

```
theorem StringT2 :
"ALL x.
 ALL xs.
 ALL y.
 ALL ys. xs /= ys = True' --> X_Cons x ys ==' X_Cons y xs = False'"
apply(auto)
apply(simp add: ILE02)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: DiffDef)
apply(simp add: NotFalse1)
done
```

Proofs of the `ExamplePrograms` theorems were very long. They were done using basically 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 theorem ~~used previously~~ as a parameter ~~failed when using~~ the `simp add:` ~~command~~. In the Isabelle Proof Script Excerpt 3.8, on page 16, we show the proof for an `insertionSort` function application.

command failed when using

previous

Isabelle Proof Script Excerpt 3.8 Proof for theorem Program03, an example of insertionSort function application from specification ExamplePrograms.

```
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`, 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 in the Isabelle Proof Script Excerpt 3.9, on page 17, with some comments inserted. The command `prefer` is used to choose which goal to prove ~~in~~ *Isabelle* interactive mode, ~~and~~ the command `oops` indicates that we could not prove the theorem, and that we gave up the proof.

when operating in the

3.6 Results

a result of

difficult

As ~~products of~~ our work, we specified a subset of the *Prelude* library and documented how to write specifications in *HasCASL*, identifying points ~~where the~~ process ~~is not yet easy~~. The specified subset can already be used to write ~~bigger~~ specifications ~~and~~ can also serve as example ~~to~~ the specification of other libraries, ~~as~~ *HasCASL* still lacks a manual, ~~as the *CASL* one~~.

in this

larger

for

complete reference

given that

The latter

inspiring

Isabelle Proof Script Excerpt 3.9 Actual status of the proof for theorem Theorem07 of specification SortingPrograms.

```
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))")
(* Proof for xs= Cons a Nil *)
prefer 2
apply(simp add: GenSortF)
(* Proof for xs=Cons a as*)
apply(case_tac Lista)
apply(auto)
prefer 2
(* Proof for xs = Cons a (Cons b Nil)*)
oops
```

4 Conclusions and Future Work

$\longrightarrow$ escribir

5 Acknowledgments

$\longrightarrow$ escribir

References

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

[2] Hutter, D., Mantel, H., Rock, G., Stephan, W., Wolpers, A., Balser, M., Reif, W., Schellhorn, G., Stenzel8, 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; http://www.springer.de (1998) 351–358

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

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

[5] 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)

[6] Clavel, M., Durn, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Talcott, C.: Predefined data modules. In Clavel, M., Durn, F., Eker, S., Lincoln, P., Mart-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; http://www.springer.de (July 2007) 231–305

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

[8] 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)

[9] 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)

[10] Group, T.R.L.: The RAISE Specification Language. The BCS Practitioners Series. Prentice-Hall, Inc. (January 1993)