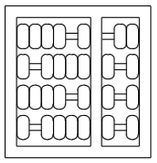

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Creating a HasCASL Library

G. M. Cabral *A. V. Moura*

Technical Report - IC-08-45 - Relatório Técnico

December - 2008 - Dezembro

Creating a HasCASL Library

Glauber Módolo Cabral* Arnaldo Vieira Moura†

Abstract

1 Introduction

In this report, we show how to specify a library in the *HasCASL* specification language. The intent it for this library to reuse previous specifications, as much as possible.

The practical use of a specification language requires that a default library exists. This could be a small library, used to guide new specifications, or, preferably, a predefined library that can be imported to construct other larger specifications.

The *HasCASL* specification language does not yet have such a library. The *CASL* specification language, of which HasCASL is an extension, already has a default library with lots of specifications covering topics from simple data types to complex algebraic structures.

Here, we show how to contruct a default library for the HasCASL language based on the *Prelude* library, from the *Haskell* programming language. This report describes the specification and the verification of our library. It also includes proofs and comments about the difficulties we faced.

This report is organized as follows. Section 2 introduces the languages involved in this work and details our proposal. Section 3 describes our specifications, including their codes. Section 4 addresses the parsing and verification of the specifications. Section 5 discusses some problems we faced during the specification of the library. Section 6 comments on some related specification languages. Section 7 lists open questions and topics for future works. Section 8 concludes the report. Appendix A lists the proof scripts used to verify the specifications.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro parcial do CNPq

†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

2 Languages

This section intended to introduce the languages involved in our work. We started with a presentation of the *CASL* specificatiton language, quickly describing its syntax and semantic. Next, we introduced the Haskell programming language, with some interesting concepts that we had to deal latter in specifications. Following, described the *HasCASL* specification language, an *CASL* extension, which we used to write our specifications. We presented some main concepts of *HasCASL* and a small example. Latter, we introduced the *HasCASL* extention to *CASL* language and its related tool, namely *Hets*, which is responsible for parsing and translating our specifications to the theorem prover. Next, we introduced the *Isabelle* theorem prover with a quick presentation of its main features. Finally, we presented our proposal to this work.

2.1 CASL

The *Common Algebraic Specification Language* (*CASL*) emerged as the product of an international initiative to create an unified language for algebraic specifications containing the largest possible set of known language constructions. This section describes the *CASL* language [1].

With few exceptions, the characteristics of *CASL* are present in some form or another in other specifications languages. However, no previous single language had all the desired purposes: some sophisticated features require specific programming paradigms; on the other hand, methods for prototyping and generation of specifications work only in the absence of certain characteristics. For example, term rewriting requires specifications with equational or conditional equational axioms.

CASL was constructed to be the kernel of a family of languages: sub-languages are obtained through syntactic or semantic restrictions, while extensions are created to support the various programming paradigms. The language definition took into account previously planned extensions, such as the support to second order functions. *CASL* is divided into several parts that can be understood and used separately, namely:

- Basic Specifications: contain declarations (of types and operations), definitions (of operations) and axioms (related operations);
- Structured Specifications: allow Basic Specifications to be combined in larger specifications;
- Architectural Specifications: define how specifications should be separated in an implementation, allowing reuse of specifications with dependence relations;
- Specification libraries: similar specifications are joined together in these libraries; their syntax has constructions that allow version control and library

distribution over the Internet.

Structured Specification language constructions are independent of the Basic Specifications, so, *CASL* sub-languages or extensions can be created by extending or restricting Basic Specification language constructions, without the need to change any of the other three language constructions. We now briefly describe the most important Basic Specification language constructions.

Basic Specification denotes a class of models which are many-sorted partial first order structures, i.e., many-sorted algebras with total and partial functions and predicates. These models are classified by signatures, which contain sort names, total and partial function names, predicate names and definitions (or profiles) for functions and predicates.

Specifications contain: declarations, which introduce components of the signature (operations or functions, and predicates), and axioms, which define properties of the structures that should be models of the specification. Operations may be declared total (by using '`->`') or partial (by using '`->?`'), and we can assign to this operations some common properties, such as associativity, avoiding the need for axiomatizing those properties for each different operation.

Partial operations are a simple way to treat errors (such as dividing by zero) and these errors are propagated to callers directly, as when an argument of an operation is not defined, the operation result is also not defined. The errors and exceptions can be treated by super-types and sub-types. The domain of a partial function can be defined as a sub-type of that function's argument type in order to make this partial function a total function over the sub-type. Functions can be declared total rather than making them total by axioms.

Predicates are similar to operations but have no return type; only parameter types are declared. Predicates may be declared and defined at the same time, instead of having their declaration and axiom in separate sections.

Axioms are written as atomic first-order formulas. Variables used in axioms may be declared in three different ways: globally, before axiom declarations; locally to a list of formulas; or individually for each formula, using explicit quantification.

Formulas are interpreted in two-valued first order logic (with values true and false). Definedness assertions are used to indicate when a term is defined or not. Assertions may be declared explicitly by a keyword or implicitly by means of an existential equation. An existential equation, declared by using '`=e=`' between two terms of the same type, is valid when both terms are defined and are equal. In contrast, strong equations, declared by using '`=`' between terms, are also valid when both terms are undefined.

Sub-sort membership, indicated by '`in`', creates a predicate asserting the membership of an element to a sort. It's a good practice to use existential equations when defining properties and strong equations when defining partial functions inductively.

CASL uses loose semantic for Basic Specifications, i.e., all structures that meet the axioms are selected as models. This semantic is interesting during requirement analysis because it creates very restrictive specifications that may be refined later by other axioms.

A data type can be declared as free, changing its loose semantic into an initial semantic. Thus, values of the same type that differ only in the order of type constructor application are treated as different elements of that type.

The third semantic allowed in *CASL* forces data types to be generated only by type constructor applications. This eliminates the confusion between terms, i.e., unless axioms force a term equality, all the terms of that type all different from each other. When needed, axioms can be used to reintroduce term equality.

Linear visibility is used to control term declaration except for type declarations, i.e., except in type declarations a term must be declared before its use.

2.2 Haskell

This section presents some general elements of the Haskell programming language. Information provided here as well as further concepts can be found online [8] or in books [22].

Haskell is a pure, strong typed functional programming language with lazy evaluation that resulted from the need of standardization in the field of functional languages. The language is functional because it implements the concepts of $\lambda - Calculus$, so the programming is done through function and computation applications. The language is strongly typed, i.e., the types of functions and values must be explicitly defined on compile time; otherwise, the compiler will try to bind those types to the broadest possible ones in the current context.

Concepts of lazy evaluation and strict evaluation relates to interpretation of the parameters of a function. Languages with strict evaluation evaluates all parameters of a function call before running its body. In the case of languages with lazy evaluation, such as Haskell, parameters of a function are evaluated only when they become necessary inside the function body.

The language is called purely functional because it does not allow a function application to change the global state of the program, only changes to variables and values local to the function execution are allowed. Changing the global state of the program is a kind of side effect which is common in imperative languages. Functional programming languages that allow side effects are called non-pure.

To allow operations that may cause side effects to be executed without causing side effects to the hole program, Haskell performs side-effect actions through a mathematical entity called a monad. Monads can sequence side-effect computations passing a copy of the actual global state implicitly to those computations. It prevents the side effects to change the real global state of the program.

Haskell functions can be declared just as in $\lambda - Calculus$ by *Lambda Abstractions* or in the *Haskell* syntax; in both styles we can name function definitions to later reuse. If a function type is not defined, the compiler will compute the broader type in the context. The *Haskell* syntax is preferred because it's easier and more practical for writing larger programs. Here we show the function `add` for summing two numbers defined with *Lambda Abstractions* and in the *Haskell* syntax, respectively; the compiler will calculate type `Integer -> Integer` to the functions as we haven't declared the type of the functions:

```
add     = \x y -> x+y
add x y = x+y
```

It is necessary to differentiate functions, as the previously defined `add`, from operators, as the operator `+`. A function in Haskell is always defined in a prefix way and an operator has an infix definition. Besides these differences, it's possible to simulate a function with an operator and vice versa. Operators can be used as functions if enclosed by parenthesis; a function can be used as an operator if enclosed by backquotes. We can use the operator `+` as a function like this: `(+) x y` and the function `add` as an operator like this: `x `add` y`

Just as in other functional languages, the main data representation in Haskell are lists. There can be lists of primitive types, lists of tuples, lists of lists, lists of functions, etc.; the only requirement is that all elements of the list have the same type. The order and the quantity of elements within a list are taken into account when comparing them for equality.

Two basic operators to manipulate lists are the operator ":" (list construction) and the operator "++" (list concatenation). A list is always constructed from an empty list and some element by the list construction operator, and two lists can be concatenated only if their elements have the same type.

Another feature largely explored in Haskell is pattern matching. Functions can be defined by pattern matching their parameters, as follows:

```
fat :: Int -> Int
fat 1 = 1
fat x = x * fat(x-1)
```

Each call to the function `fat` will pattern match against each line of its definition, from the first one to the last one, until the parameters of the function call match parameters from one of the definitions. Thus, the more specific definitions must come before the generic one. In Haskell Source Code 2.2.1, on page 6, we can see pattern matching applied in case expressions, list constructors and let expressions.

A fundamental tool in Haskell is the data type construction. A data type must have at least one constructor that may be empty or may have type variables. Type variables are used to construct polymorphic data types; the constructor and its type

variables may be enclosed by parenthesis in order to avoid ambiguity. In Haskell Source Code 2.2.1, on page 6, we defined the polimorphic type `Split a b` with one constructor (`Split b [[a]]`).

We can collect functions and data types from similar contexts in libraries. Haskell libraries are called modules and can control which functions and data types from that module should be exposed to users. We've created a module in Haskell Source Code 2.2.1, on page 6, and let all the functions to be exposed to the users. There is a standard library, called *Prelude*, which defines basic functions that operate on primitive types, such as: *Bool*, *Char*, *List*, *String*, numeric types and tuples involving those types. All Haskell compilers must implement the Prelude library, as this implementation is part for the language definition.

Haskell Source Code 2.2.1 Haskell source code for GenSort sorting program – Part 1

```
module GenSort where
import Data.List
data Split a b = Split b [[a]]
genSort :: Ord a => ([a] -> Split a b) -> (Split a b -> [a]) -> [a] -> [a]
genSort split join l = case l of
  _ : _ : _ -> let Split c ls = split l in
               join $ Split c $ map (genSort split join) ls
  _ -> l
splitInsertionSort :: [a] -> Split a a
splitInsertionSort (a : l) = Split a [l]
joinInsertionSort :: Ord a => Split a a -> [a]
joinInsertionSort (Split a [l]) = insert a l
insertionSort :: Ord a => [a] -> [a]
insertionSort = genSort splitInsertionSort joinInsertionSort
splitQuickSort :: Ord a => [a] -> Split a a
splitQuickSort (a : l) =
  let (ls, gs) = partition (< a) l in Split a [ls, gs]
joinQuickSort :: Split a a -> [a]
joinQuickSort (Split a [ls, gs]) = ls ++ (a : gs)
quickSort :: Ord a => [a] -> [a]
quickSort = genSort splitQuickSort joinQuickSort
splitMergeSort :: [a] -> Split a ()
splitMergeSort l =
  let (l1, l2) = splitAt (div (length l) 2) l in Split () [l1, l2]
joinMergeSort :: Ord a => Split a () -> [a]
joinMergeSort (Split _ [l1, l2]) = merge l1 l2
```

Haskell Source Code 2.2.1 Haskell source code for GenSort sorting program – Part 2

```
merge :: Ord a => [a] -> [a] -> [a]
merge l1 l2 = case l1 of
  [] -> l2
  x1 : r1 -> case l2 of
     [] -> l1
     x2 : r2 -> if x1 < x2
       then x1 : merge r1 l2
       else x2 : merge l1 r2
mergeSort :: Ord a => [a] -> [a]
mergeSort = genSort splitMergeSort joinMergeSort
splitSelectionSort :: Ord a => [a] -> Split a a
splitSelectionSort l =
  let m = minimum l in Split m [delete m l]
joinSelectionSort :: Split a a -> [a]
joinSelectionSort (Split a [l]) = a : l
selectionSort :: Ord a => [a] -> [a]
selectionSort = genSort splitSelectionSort joinSelectionSort
```

2.3 HasCASL

This section presents the language *HasCASL*[18]. The formal language definition can be found in another document [19].

The language *HasCASL* is an extension of *CASL* with concepts of higher-order logic such as high order types and functions, polymorphism and type constructors. *HasCASL* was planned to have Haskell as its subset; this should make it possible to transform a *HasCASL* specification in a Haskell program in a simple way.

Standard higher-order logic does not allow recursive types and functions widely used in functional languages. *HasCASL* ~~tries to~~ solve this problem without using denotational semantic, by creating an internal logic to λ-abstractions which is not a primitive concept, but that emerges from the constructions. Thus, although higher-order properties can be obtained, *HasCASL* remains close to the *CASL* language.

The sentences in *HasCASL* differ from those in *CASL* in two respects:

- Quantifiers (universal, existential and unique existential) can be applied on type variables and have restrictions related to sub-types;
- *CASL* predicates are replaced by terms of the type `Unit`.

Unlike in functional programming languages, polymorphic operators must be explicitly instantiated, since it is not yet clear, theoretically, how they relate to resolution of sub-type overloads and implicit instantiation.

As *HasCASL* tries to keep as close as possible to *CASL*, its semantic is also based on set theory. Intentional Henkin models are chosen to model higher-order signatures in the *HasCASL* semantic. In this model, the types of functions are interpreted by arbitrary sets equipped with an application function of the appropriate type (opposed to a partial type `s ->? t` ~~been~~ interpreted by the complete set of all partial functions from `s` to `t`). The interpretation of the λ-terms is part of the model structure rather than just being an existential axiom.

The intensional Henkin model has some advantages, including: it eliminates the completeness problem; allows initial models of signatures containing partial functions; and allows the operational semantics of functional programming languages to be applied, instead of directly using higher-order logic operational semantic.

Unlike Haskell, in which function evaluation is lazy, the evaluation of functions in *HasCASL* is strict, i.e., undefined arguments always result in undefined values. One way to emulate the lazy evaluation is to move a parameter with type `a` to the unit type `Unit ->? a`.

To illustrate the language syntax, we'll take a look into the Specification 2.3.1, on page 9. Types are defined by the reserved word `type`, which may be preceded by the qualifiers `free` and `generated`, as in *CASL*. Defining types which contain function types as constructor parameters and recursion only on the right side of the arrow should be done with the reserved word `cofree`; when recursion is present in both sides of the arrow, the types must be defined with the reserved word `free`. Type `Bool` was defined as `free` type with two constructors (`True` and `False`).

Functions may be defined by the word `fun`, which differs from the command `op` ~~with~~ relation to their behaviour over sub-typing [20]. A lazy type differs from a strict one by a question mark in front of the type, as in `?Bool`. Functions in mixfix notation have their parameters indicated by the placeholder `__` and the parameter types must be defined as tuples. Thus, the function `__&&__` expects two elements of type `Bool` (indicated by: `?Bool * ?Bool`) and returns one element of that type (indicated by: `-> ?Bool`). Curried functions are defined applying them to the parameters in opposite to using the placeholder and the types of the parameters should be separated by `->` ~~in place~~ of `*`.

Variables are introduced by the word `var` followed by a list of one or more variables ~~and~~ the type of these variables, separated from the list of variables by a colon.

Axioms and theorems are introduced by a final point. Annotations are included in front of axioms and theorems to make it easier to reference them and to allow their use by tools; the annotation should be a name between `%(` and `)%`.

Specification 2.3.1 Initial Bool Specification from scratch

```
spec Bool = %mono
     free type Bool ::= True | False
     fun Not__: ?Bool -> ?Bool
     fun __&&__: ?Bool * ?Bool -> ?Bool
     fun __||__: ?Bool * ?Bool -> ?Bool
     fun otherwise: ?Bool
     vars x,y: ?Bool
     . Not(False) = True                  %(Not_False)%
     . Not(True) = False                  %(Not_True)%
     . False && False = False             %(And_def1)%
     . False && True = False              %(And_def2)%
     . True  && False = False             %(And_def3)%
     . True  && True = True               %(And_def4)%
     . x || y = Not(Not(x) && Not(y))     %(Or_def)%
     . otherwise = True                   %(Otherwise_def)%
end
```

2.4 Heterogeneous Specifications: HetCASL and Hets

Nowadays, in the formal method area, different logics and methods are used to specify large systems because there isn't a single best solution to achieve all the desired functionalities. These heterogeneous specifications must have a formal interoperability between the languages involved in such a way that each language may have its own proof method and all formal proofs must be consistent when viewed in terms of the heterogeneous specification.

The various sub-languages and extensions of *CASL* may be linked by the language *Heterogeneous CASL* (*HetCASL*) [14], which has the structural constructions of *CASL*. *HetCASL* extends the semantic properties of *CASL* by being institution independent with constructions that forcing the relationship between translations of specifications that occur in conjunction with translations of the logics. It is worth emphasizing that *HetCASL* preserves the fact that the logic used by individual specifications are orthogonal to *CASL*.

The *Heterogeneous Tool Set* (*Hets*) [14] is a syntactic analyzer and a proof manager for *HetCASL* specifications, implemented in Haskell, which combines the various proof tools for each individual logic used in various sub-languages and extensions of *CASL*. *Hets* is based on a graph of logics and languages, providing a clear semantic and a proof calculus for heterogeneous specifications.

Each logic of the graph is represented by a set of types and functions in Haskell. The syntax and semantics of the heterogeneous specifications in *HetCASL* and their

implementations are parametrized by an arbitrary graph of logics inside *Hets*. This allows easily management of each *Hets* module implementation using software engineering techniques.

HasCASL specifications are translated to the Isar language, which is the language used by the *Isabelle* theorem prover [15], a semi-automatic theorem prover for higher-order logics. Hets supports other first-order theorem provers for proving *CASL* specifications. Other *CASL* sub-languages or extensions maybe proved by translating them to *CASL* or *HasCASL*.

The structure of proofs in *Hets* is based on the formalism of development graphs [13], widely used for specifications of industrial systems. The graph structure allows for a direct visualization of the specification structure and facilitates the management of specifications with many sub-specifications.

A development graph consists of a number of nodes (corresponding to complete specifications or parts of specifications) and a set of edges called definition links that indicate dependency between the various specifications and their sub-specifications. Each node is associated with a signature and a local set of axioms; these axioms are inherited by other nodes which depend on this node through definition links. Different types of edges are used to indicate when the logic is changed between two nodes.

A second type of edge, a theorem link, is used to indicate relations between different theories, serving to represent proof needs that arise during the specification development. Theorem links can be global or local (represented by edges with different shapes in the graph): global links indicate that all valid axioms in the source node are valid in the target node; local links indicate that only axioms defined in the source node are valid in target node.

Global theory links are broken down into simpler links (global or local) through proof calculus for development graphs. Local links may be proved by transforming them into local proof goals; this transformation will mark the node corresponding to that goal to be proved using the theorem prover for the logic represented on this node.

2.5 Isabelle

This section describes the theorem prover *Isabelle* [10]; a full description can be found on the manual [15].

Isabelle is a generic theorem prover that allows the use of several logics as formal calculus to assist in theorem proofs. *Hets* uses *Isabelle* to prove theorems in higher-order logic, but the prover allows, for example, the use of axiomatized set theory, among other logics. Support for multiple logic is one of the prominent features of the tool.

The prover has an excellent support for mathematical notation: new symbols may be included using common mathematical syntax and proofs can be described in

a structured way or as a sequence of proof commands. Proofs may include TEX codes so that formatted documents can be generated directly from the proof source. The major limitation of theorem provers is the usual need for an extensive previous experience from the users. In order to facilitate the process of proof construction, *Isabelle* has some tools that automate some proof contents, such as equations, basic arithmetic and mathematical formulas.

The higher-order language (HOL) is used to write theories. Its syntax is very similar to those of functional programming languages because it is based on the typed λ-calculus. This language allows construction of data types, types with functions as parameters and other constructions common in functional languages. Translation of *HasCASL* specifications to HOL theories are automatically made by the *Hets* tool.

Isabelle has an extension, called Isar, which allows one to describe proofs that can be read by humans and can be easily interpreted by computers. It has an extensive library of mathematical theories already proved (for example, in topics like algebra and set theory), and also many examples of proofs carried out in a formal verification context. In this work, proofs were written using proof commands, although they are less powerful than the notation used in Isar.

2.6 Proposal

A prerequisite for the practical use of a specification language is the availability of a set of standard specifications previously defined [17]. *CASL* language has such set of specifications defined in "CASL Basic Datatypes" [16]; instead of providing common blocks for reuse as programming languages usually do, this document provides complete specification examples that illustrate the use of *CASL* both in terms of Basic Specifications and Structured Specification. There are two groups of examples: one with basic data types and one with specifications that express properties of complex structures. In the first case, we can find simple data types, such as numbers and characters, as well as structured data types, such as lists, vectors and matrices; the second group contains algebraic structures such as rings and monoids and mathematical properties such as equivalence relation and partial order.

Currently, *HasCASL* language does not have a library along the lines of *CASL* library. According Scröder, data types described in "CASL Basic Datatypes" can serve as a basis for building a standard library to each *CASL* extensions. In case of *HasCASL*, it is suggested the inclusion of new specifications that involve higher order features such as completeness of partial orders as well as extending data types and changing parameterization for real type dependence. As example, higher order functions operating on lists, such as *map*, *filter* and *fold*, can be specified after importing already defined functions on List data type from *CASL* library, to improve reuse.

Based on these suggestions, we propose to build a library for *HasCASL* based on *CASL* library and Haskell Prelude library. Creating such a library can be very useful

to increase *HasCASL* usage in real projects by providing predefined specifications for reuse. As Prelude library must be implemented by all Haskell compilers, having its data types already specified in *HasCASL* can contribute to automatic code generation in the future as, once these data types are already specified, verified and refined to Haskell code, larger specifications using them can be created and translated to Haskell in an easier way.

Creation of such a library required studing how Haskell functions and types operate and finding solutions to include these elements on our library with maximum reuse of *CASL* library data types. Learning *CASL*, *HasCASL* and *Isabelle* and dealing with their peculiarities were the center of the project difficulties.

All generated specifications were verified by *Hets* tool and most of them were proved using *Isabelle* to ensure their correctness.

3 Specifying the library

In this section we intended to explain what we've been working on. We started by discussing the choices we'd to do when we started the project. Later, for each specification we'd done, we listed its source and explained some problems we've faced and some choices we've made when writing that specification.

3.1 Initial choices

To fully capture Haskell features, our library should use laziness, be refined to use continuous, this allowing infinite data types. Since starting with all these functionalities would require using the most advanced constructions of the *HasCASL* language and would require deep know-how of *Isabelle* proof scripts, it would not be the best first approach to use as an algebraic specification methodology. Thus, we decided that the library should be specified using strict types and more advanced Haskell features should be left for a latter refinement.

Different from Haskell, *HasCASL* doesn't allow the same function to be used both in prefix and infix notation. Thus, all functions from the *CASL* library which were defined in a mixfix way (and thus expected tuples as parameters) wouldn't be compatible with Haskell curried functions. To solve this problem, we redefined functions from the *CASL* library in a mixfix way and, for each mixfix definition, we created a curried version whose name would be 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 functions.

To write our library, we used names from Prelude functions and types. When importing, we changed the imported name to the one used by the Prelude version

using *CASL* renaming syntax. When there was any function in Prelude that had no equivalent ~~in an existent~~ *CASL* specification, we included that function in our *HasCASL* type to match Prelude types and functions as much as possible.

3.2 Our first specification: Bool

We started our library by importing type Boolean from the *CASL* library, like shown in Specification 3.2.1, on page 13.

Specification 3.2.1 Initial Bool Specification importing *CASL* type

```
from Basic/SimpleDatatypes get Boolean
spec Bool = {Boolean with
         Boolean |-> Bool,
         Not__ |-> not__,
         __And__ |-> __&&__,
         __Or__ |-> __||__
        }
     then
     op otherwise: Bool
    . otherwise = True
```

As we were still pondering about using laziness, we decided that it should be better to specify Boolean from scratch, since the one imported from *CASL* had only total functions. This tentative is shown in Specification 2.3.1, on page 9.

Next, we ~~we~~ decided to use only stric types, as we could, later, refine our specifications to use laziness. We have also included curried versions for both boolean operations that are mixfix in the *CASL* version, as well as some axioms that ~~whoul~~ would be needed later in *Isabelle* proofs that couldn't be ~~done~~ concluded automatically. As "*otherwise*" is an *Isabelle* reserved word, we appended an *H*, from *H*askell, to its name. We thus achieved Specification 3.2.3, on page 15.

3.3 The Specification for Equality

After defining the `Bool` type, the next step was to specify equality functions. As we were working over `Bool`, we could not use *HasCASL* predicates and their related operations. We thus had to redefine all functions and operations related to element comparison to use our `Bool` type. As in the Haskell Prelude, equality functions were grouped in a class named `Eq`, giving us Specification 3.3.1, on page 16.

Equality was defined including axioms for symmetry, reflexivity and transitivity. An axiom mapping *HasCASL* equality to our equality was created, namelly,

Specification 3.2.2 Initial Bool Specification from scratch

```
spec Bool = %mono
     free type Bool ::= True | False
     fun Not__: ?Bool ->? ?Bool
     fun __&&__: ?Bool * ?Bool ->? ?Bool
     fun __||__: ?Bool * ?Bool ->? ?Bool
     fun otherwise: ?Bool
     vars x,y: ?Bool
     . Not(False) = True                  %(Not_False)%
     . Not(True) = False                  %(Not_True)%
     . False && False = False             %(And_def1)%
     . False && True = False              %(And_def2)%
     . True  && False = False             %(And_def3)%
     . True  && True = True               %(And_def4)%
     . x || y = Not(Not(x) && Not(y))     %(Or_def)%
     . otherwise = True                   %(Otherwise_def)%
end
```

`%(EqualTDef)%`, since the opposite map cannot be created because it would be too restrictive. Negation was defined by negating equality, as any equation involving negation could be translated to a negated equality and thus proved using the equality axioms. Curried versions for both functions were also defined. Seven auxiliary theorems were created and proved to be used by *Isabelle*, if needed.

Type instances were declared, as it's done in Prelude, for `Bool` and `Unit` data types. In the first case, although `Bool` is a free data type and, hence, `True` is different from `False`, this difference had to be axiomatized by the axiom `%(IBE3)%` because our equality is not mapped to the *HasCASL* equality. All the other theorems for `Bool` instance declarations should follow from `%(IBE3)%` and the other `Eq` axioms. In the second case, as `()` is the only element from type `Unit`, instance definitions should be theorems as they follow from the `Eq` axioms.

3.4 The Specification for Ordering

The next specification we defined was `Ord`, for Ordering relations. Our first approach was to import the partial order defined by the Ord specification inside the library *HasCASL/Metatheory/Ord*. As importing this library would cause problems to our strict library, because the imported one uses lazy types, we decided to specify our own version.

To create the Ord specification we defined the `Ordering` data type and declared this type as an instance of the Eq class. Three axioms relate the three constructors and