Specware to Isabelle Interface Manual

Release 4.2

Kestrel Institute

November 26, 2012

CONTENTS

CHAPTER

ONE

CONCEPTS

This document describes a Specware interface that allows the use of Isabelle-HOL to discharge proof obligations that arise in developing Specware specifications. The interface is essentially just a Specware Shell command and an Emacs command that converts a Specware spec to an Isabelle theory, along with extensions in the Specware syntax to allow Isabelle proof scripts to be embedded in Specware specs, and to allow the user to specify translation of Specware ops and types to existing Isabelle constants and types. The translation translates Specware declarations, definitions, axioms and theorems to the corresponding Isabelle versions. The logics are similar so it is usually straightforward to compare the source and target of the translations. In addition, Specware has implicit type obligations, particularly sub-type obligations, that are explicated in the Isabelle target.

We assume the user is familiar with Isabelle-HOL. See the tutorial at http://isabelle.in.tum.de/documentation.html. The current version of the Isabelle translator works with Isabelle2009-1. An example Specware spec with Isabelle proofs is given in `Examples/IsabelleInterface/BoolEx.sw`. This spec corresponds to the Isabelle theory in section 2.2.4 of the Isabelle-HOL tutorial.

To see examples of how to specify translation of Specware types and ops to existing Isabelle types and constants, see the bottom of the Specware Base library specs such as `Library/Base/Integer.sw` or `Library/Base/List.sw`.

A Specware definition may translate into one of four different kinds of Isabelle definitions: `defs`, `recdefs` and the newer `funs` and `functions`. Simple recursion on coproduct constructors translates to `fun`, but more complicated recursion is usually translated to `fun`. Some recursion still translates to `recdef` because the `fun` and `function` support is new, but the user can force translation to `function`. Non-recursive functions are translated to `defs`, except in some cases they are translated to `fun` which allows more pattern matching.

The main difference in the logics of Specware and Isabelle-HOL is that Specware has predicate sub-types. In most cases a sub-type is translated to its super-type and translations of quantifications over a sub-type introduce an explicit application of the sub-type predicate. A subtlety is that we need to consider the case that polymorphic type variables may be instantiated with subtypes. When necessary, e.g., for a predicate like `injective?`, a single op is translated to two Isabelle ops, the ordinary one and one with an extra argument for a predicate or predicates corresponding to subtype predicates for type variables. Another subtlety is with respect to equality of functions with subtype domains. These are translated to Isabelle functions with expanded domains, but to preserve equality these are regularized to have a single value outside the restricted domain. This regularization is not needed if the function is applied to an argument, because it may only be applied to an argument for which the predicate holds, so in some cases we do the regularization lazily, i.e., give the function its unregularized definition, but regularize it in contexts where it may be used in an equality.

There is a capability for translating a sub-type differently from its super-type. This is used for the type `Nat` which is translated to `nat` rather than `int`. In general, this may lead to coercions between `nat` and `int` being inserted.

This initial translator has a few limitations. It should translate all Specware specs but not all translated definitions and constructs will be accepted by Isabelle-HOL. In particular, only case expressions that involve a single level of pattern-matching on constructors are accepted. An exception, is that nesting is allowed in top-level case expressions that are converted into definition cases.

CHAPTER

TWO

USAGE

2.1 Starting Up

Specware and Isabelle can both be started up normally, each running under their own XEmacs job, but it convenient to run them under the same XEmacs. To do this run `Isabelle_Specware`.

Currently Isabelle does not run under Windows except with cygwin so this script is not available there. However, the translator can run from Specware even if Isabelle is not running.

2.2 Using The Translator

The translator is called using the emacs command `Generate Isabelle Obligation Theory` from the Specware menu (with keyboard shortcuts `c-c c-i` or `c-c TAB`). The translation is written to a file in the `Isa` sub-directory of the current directory and the file is visited in a buffer. The user may then process the Isabelle theory providing proof steps as necessary. These proofs may then be copied back to the Specware spec so that the next time it is translated, the translation will include the proofs.

The translator may also be called using the Specware Shell command `gen-obligations` (or the abbreviation `gen-obligs`) applied to a unit id.

2.3 Proof Scripts in Specs

An embedded Isabelle proof script in a Specware spec consists of an introductory line beginning with `proof Isa`, the actual Isabelle script on subsequent lines terminated by the string `end- proof`. For example, the simple proof script `apply(auto)` can be embedded as follows:

```
proof Isa
  apply(auto)
end-proof
```

If the last command before end-proof is not `done`, `sorry`, `qed` or `by`, the command `done` is inserted.

The proof script should occur immediately after the theorem or definition that it applies to. If the script applies to a proof obligation that is not explicit in the spec, then the name of the obligation should appear after `proof Isa`, on the same line. There are rare cases where an obligation is inserted between a definition and an immediately following proof script, which causes the proof script to be ignored. If this happen, then the name of the op should be explicitly given.

If the user does not supply a proof script for a theorem then the translator will supply the script `by auto` which may be all that is required to prove simple theorems.

Annotations for theorems may be included on the `proof Isa` line. For example,

```
theorem Simplify_valif_normif is
  fa(b,env,t,e) valif (normif b t e) env = valif (IF(b, t, e)) env
  proof Isa [simp]
    apply(induct_tac b)
    apply(auto)
  end-proof
```

translates to

```
theorem Simplify_valif_normif [simp]:
  "valif (normif b t e) env = valif (IF b t e) env"
    apply(induct_tac b)
    apply(auto)
  done
```

In this example we see that universal quantification in Specware becomes, by default, implicit quantification in Isabelle. This is normally what the user wants, but not always. The user may specify the variables that should be explicitly quantified by adding a clause like `fa t e.` to the `proof Isa` line. For example,

```
theorem Simplify_valif_normif is
  fa(b,env,t,e) valif (normif b t e) env = valif (IF(b, t, e)) env
  proof Isa [simp] fa t e.
    apply(induct_tac b)
    apply(auto)
  end-proof
```

translates to

```
theorem Simplify_valif_normif [simp]:
  "\<forall> t e. valif (normif b t e) env = valif (IF b t e) env"
    apply(induct_tac b)
    apply(auto)
  done
```

The `\<forall>` will be displayed as a universal quantification symbol using X-Symbol mode in Isabelle. Note that instead of `fa` in the `proof Isa` line the user may use the X-Symbol for universal quantification.

Recursive functions that are translated to `recdefs` can have a measure function specified on the `proof Isa` line, by including it between double-quotes. For example:

```
proof Isa "measure (\<lambda>(wrd,sym). length wrd)" end-proof
```

2.4 Translation Tables

A translation table for Specware types and ops is introduced by a line beginning `proof Isa Thy_Morphism` followed optionally by an Isabelle theory (which will be imported into the translated spec), and terminated by the string `end-proof`. Each line gives the translation of a type or op. For example, for the Specware Integer theory we have:

```
proof Isa Thy_Morphism Presburger
  type Integer.Int -> int
  type Integer.Integer -> int
  type Nat.Nat     -> nat (int,nat) [+,*,div,rem,mod,<=,<,>=,>,abs,min,max]
  Integer.zero     -> 0
  Integer.one      -> 1
  Integer.ipred    -> pred
```

```
  Integer.isucc     -> succ
  IntegerAux.-      -> -
  Integer.+         -> +      Left 65
  Integer.-         -> -      Left 65
  Integer.*         -> *      Left 70
  Integer.<=        -> \<le>  Left 50
  Integer.<         -> <      Left 50
  Integer.>=        -> \<ge>  Left 50
  Integer.>         -> >      Left 50
  Integer.sign      -> sign
  Integer./         -> div    Left 70
  Integer.div       -> div    Left 70
  Integer.mod       -> mod    Left 70
  Integer.min       -> min            curried
  Integer.max       -> max            curried
end-proof
```

A type translation begins with the word `type` followed by the fully-qualified Specware name, `->` and the Isabelle name. If the Specware type is a sub-type, you can specify coercion functions to and from the super-type in parentheses separated by commas. Note that by default, sub-types are represented by their super-type, so you would only specify a translation if you wanted them to be different, in which case coercion functions are necessary. Following the coercions functions can appear a list of overloaded functions within square brackets. These are used to minimize coercions back and forth between the two types.

An op translation begins with the fully-qualified Specware name, followed by `->` and the Isabelle constant name. If the Isabelle constant is an infix operator, then it should be followed by `Left` or `Right` depending on whether it is left or right associative and a precedence number. Note that the precedence number is relative to Isabelle's precedence ranking, not Specware's. Also, an uncurried Specware op can be mapped to a curried Isabelle constant by putting `curried` after the Isabelle name, and a binary op can be mapped with the arguments reversed by appending `reversed` to the line.