Specware 4.0 User Manual

Specware 4.0 User Manual

Table of Contents

1. Installing Specware 5
1.1. System Requirements 5
1.1.1. Hardware 5
1.1.2. Operating system 5
1.2. Installation Instructions 5
2. Getting Started 7
2.1. Starting Specware 7
2.2. Exiting Specware 7
3. Usage Model 8
3.1. Units 8
3.2. Interaction 8
4. Defining Units 10
4.1. Conceptual Model 10
4.1.1. Names 10
4.1.2. Defining Text 10
4.2. Realization via the File System 11
4.2.1. The `SWPATH` Environment Variable 11
4.2.2. Single Unit in a File 12
4.2.3. Multiple Units in a File 12
4.3. Unit Definitions Are Managed Outside of Specware 14
5. Processing Units 15
5.1. Overview 15
5.2. Resolution of Unit References 16
5.2.1. Absolute References 16
5.2.2. Relative References 18
5.3. Processing Commands 21
5.3.1. Processing a Named Unit 21
5.3.2. Processing an Unnamed Unit 21
5.3.3. Processing a Multiple-Unit File 22
5.4. Unit-Specific Processing 22

5.4.1. Typechecking Specs 22
5.4.2. Proving Properties in Specs 23
5.4.2.1. The Proof Unit 23
5.4.2.2. Proof errors 24
5.4.2.3. Proof Log Files 24
5.4.2.4. Multiple Proofs 24
5.4.2.5. Interrupting Snark 25
5.4.2.6. The Prover Base Library 25
5.4.2.7. The Experimental Nature of the Prover 25
5.5. Auxiliary Commands 26
5.5.1. Displaying Unit Values 26
5.5.2. Inspecting and Clearing the Cache 27
5.5.3. Inspecting and Setting `SWPATH` 27
6. Lisp Code Generated from Specs 29
6.1. Translation of Specware Names to Lisp Names 29
6.2. Arity and Currying Normalization 30
6.3. Representation of Other Sorts 31
7. Debugging Generated Lisp Files 33
7.1. Tracing 33
7.2. Breaking 33
7.3. Timing 34
7.4. Interrupting 35

Chapter 1. Installing Specware

1.1. System Requirements

1.1.1. Hardware

Specware has relatively modest system requirements for simple projects. Of course, as with any development tool, as your projects being developed become more complex, you may wish to work on a more powerful machine. For average use, however, the following basic hardware configuration is recommended:

- CPU: 250 Mhz
- RAM: 128 MB total, at least 64 MB free for applications
- Disk space: 15 MB for base system, 10-50 MB for user projects

1.1.2. Operating system

Specware 4.0 has been tested to work with Windows NT 4.0 and Windows 2000.

1.2. Installation Instructions

1. Before running the Specware 4.0 installer, you should have a recent version of XEmacs (21.1 or higher) installed on your machine. If you already have XEmacs, then proceed to the next step. Otherwise, open the XEmacs folder provided on the setup CD, and run `setup.exe`. Select "Install from Local Directory" as the Installation Method, and click "Next" through the remaining steps to accept the

default configuration. Note: If you choose not to complete this step of installing XEmacs, a text-only version of Specware will be installed instead.

2. After XEmacs has been installed, launch the Specware 4.0 installer `setup.exe` on the CD.
3. Follow the instructions in the installation wizard, and Specware 4.0 will be installed on your machine. A shortcut to Specware will be placed on your Desktop as well as in the Program Files folder in the Start menu.

Chapter 2. Getting Started

Specware is a development environment that runs on top of Lisp. This chapter describes the Specware environment and the basic mechanisms for running Specware.

2.1. Starting Specware

To start Specware, double-click the `Specware 4.0` shortcut on your Desktop, or select `Specware 4.0` from the `Start Menu -> Program Files -> Specware` folder. When Specware is launched, a couple things happen: XEmacs is started and a Lisp image containing Specware is started inside an XEmacs buffer. All of the user interaction (see the next chapter) with Specware occurs at the Lisp prompt.

2.2. Exiting Specware

To exit Specware, type `:exit` at the Lisp prompt. A message will appear indicating that another process exists. Type `y` to confirm that you want to exit. This will kill Specware and you may then close the XEmacs window.

Chapter 3. Usage Model

3.1. Units

Simply put, the functionality provided by Specware consists in the capability to construct specs, morphisms, diagrams, code, proofs, and other entities. All these entities are collectively called *units*.

Some of the operations made available by Specware to construct units are fairly sophisticated. Examples are colimits, extraction of proof obligations, discharging of proof obligations by means of external theorem provers, and code generation.

The Metaslang language is the vehicle to construct units. The language has syntax to express all the unit-constructing operations that Specware provides. The user defines units in Metaslang, writing the definitions in `.sw` files (this file extension comes from the first and fifth letter of “Specware”).

Currently, the only way to construct units in Specware is by writing text in Metaslang. Future versions of Specware will include the ability to construct units by other means. For instance, instead of listing the nodes and edges of a diagram in text, it will be possible to draw the diagram on the screen.

3.2. Interaction

All the interaction between the user and Specware takes place through the Lisp shell. The `.sw` files that define units are edited outside of Specware, i.e. using XEmacs, Notepad or any other text editor of choice. These files are processed by Specware by giving suitable commands at the Lisp shell.

When `.sw` files are processed by Specware, progress and error messages are displayed in the XEmacs buffer containing the Lisp shell. In addition, the results of processing are saved into an internal cache that Specware maintains. Lastly, processing of certain kinds of units result in new files being created. For example, when Lisp code is

generated from a spec, the code is deposited into a `.lisp` file.

Specware also features auxiliary commands to display information about units, inspect and clear the internal cache, and inspect and change an environment variable that determines how unit names are resolved to `.sw` files.

Chapter 4. Defining Units

4.1. Conceptual Model

A unit definition consists of a name and some defining text. The name identifies the unit and the text describes how the unit is constructed.

An application developed with Specware consists of a set of unit definitions, some of which may come from libraries. Units have unique names within the application.

4.1.1. Names

A unit name is a finite, non-empty sequence of identifiers (identifiers are defined in the Metaslang grammar). The sequence of identifiers is essentially a “path” in a tree: the units comprising an application are organized in a tree.

This provides a convenient and mundane way to organize the name space of the units comprising an application. Libraries are subtrees of the whole tree. Parallel development of different parts of an application can be carried out in different subtrees that can be later put together without any risk of name clashes.

4.1.2. Defining Text

The defining text of a unit is written in the Metaslang language. Metaslang features various operations to construct specs, morphisms, and all the other kinds of units. For instance, it is possible to construct a spec by explicitly listing its sorts, ops, and axioms. It is also possible to construct a spec by applying the colimit operation to a diagram of specs and morphisms.

The defining text of a unit may contain references to other units. For instance, a spec constructed by extending another one contains a references to the spec being extended. References can be absolute or relative, and they are resolved to units in the tree. The former consist in complete paths in the tree; the latter consist in partial paths that are

combined with the path of the referencing unit (according to simple rules explained later) to form a complete path in the tree.

4.2. Realization via the File System

The conceptual model just described is realized by means of the file system of the underlying operating system. The file system has essentially a tree structure. The tree of units comprising a Specware application is mapped to subtrees of the file system; the identifiers comprising a path are mapped to file and directory names.

Future versions of Specware will have a more sophisticated UI that will realize the conceptual model directly. Users will graphically see the units organized in a tree and they will be able to add, remove, move, and edit them. The mapping to the file system may even be made totally transparent to the user.

4.2.1. The `SWPATH` Environment Variable

The mapping of the conceptual unit tree to the file system is defined by the environment variable `SWPATH`. Similarly to the "path" environment variable in operating systems, `SWPATH` is a string consisting of a colon-separated list of absolute directory paths in the file system.

The value of `SWPATH` can be inspected by means of the following command (at the Lisp shell):

```
:swpath
```

The value of `SWPATH` can be changed by means of the following command:

```
:swpath <string>
```

The `<string>` must be a colon-separated list of absolute directory paths, surrounded by double quotes.

Changes to `SWPATH` only apply to the currently running Specware session. If Specware is quit and then restarted, `SWPATH` loses the value assigned to it during the previous session, reverting to its default value.

Roughly speaking, the unit tree consist of all the units defined in `.sw` files under the directories listed in `SWPATH`. The name of each unit is its path from the directory in `SWPATH` under which the file defining the unit is: i.e., if the unit is under directory `dir`, its name is its absolute path in the file system "minus" the `dir` prefix. This approximate statement is made precise below and illustrated by examples.

4.2.2. Single Unit in a File

The simplest way to define a unit is to write its defining text into a `.sw` file under one of the directories in `SWPATH`. The name of the unit is the name of the file, without `.sw`, prefixed by the path from the directory in `SWPATH` to the file.

For example, suppose that `SWPATH` includes the directory `\users\me\specware` (assuming the Windows operating system; currently, since `SWPATH` is colon-separated, it is not possible to use drive identifiers in directories; Specware automatically prefixes directories with `c:`; this restriction will be removed in future versions of Specware). The user creates a file named `A.sw` immediately under the directory `c:\users\me\specware\one\two`, containing the following text:

```
spec
  sort X
endspec
```

The absolute path of the file in the file system is `c:\users\me\specware\one\two\A.sw`. The unit is a spec containing just a sort `X`. The name of the unit is `one/two/A`. Note that the path components are separated by forward slashes "/", even though the underlying file system uses backward slashes "\". Unit names are sequences of identifiers separated by forward slashes, regardless of the underlying operating system.

4.2.3. Multiple Units in a File

It is also possible to put multiple units inside a `.sw` file. The file must be under one of the directories in `SWPATH`. Instead of just containing the defining text of a unit, the file contains one or more unit definitions. A unit definition consists of an identifier, an equal sign "=", and defining text.

This case works almost exactly as if the file were replaced by a directory with the same name (without `.sw`) containing one `.sw` file for every unit defined therein. Each such file has the identifier of the unit as name (plus `.sw`) and its defining text as content.

The only difference between the case of multiple units per file and the almost equivalent case where the file is replaced by a directory containing single-unit files, is that in the former case the last separator is not forward slash but the sharp sign "#". (This is reminiscent of the URI syntax, where subparts of a document are identified using the sharp sign.)

Suppose, as in the previous example, that `SWPATH` includes the directory `\users\me\specware`. The user creates a file named `three.sw` immediately under the directory `c:\users\me\specware\one\two`, containing the following text:

```
B = spec
  sort Y
endspec

C = spec
  sort Z
endspec
```

This file defines two specs, one containing just a sort `Y`, the other containing just a sort `Z`. The name of the first spec is `one/two/three#B`, the name of the second spec is `one/two/three#C`.

As a particular instance of the case of multiple units per file, it is possible to have just one unit definition in the file. This is different from just having defining text of a unit in a file. If the file contains a unit definition, then the identifier at the left of the equal sign is part of the unit's name, together with the sharp sign and the file path (relative to the directory in `SWPATH`). If instead the file contains defining text for a unit, then the name

of the unit is the file path (relative to the directory in `SWPATH`), without any sharp sign and additional identifier.

Despite the possibility of having one unit definition in a file, in this manual we use the term “multiple-unit file” to denote a file that contains one or more unit definitions. The term “single-unit file” is instead used to denote a file that only contains the defining text of a unit.

4.3. Unit Definitions Are Managed Outside of Specware

The `.sw` files are created, deleted, moved, and renamed by directly interacting with the file system of the underlying operating system.

The content of the `.sw` files can be edited with any desired text editor. A possibility is to use XEmacs, which is started when Specware is started and is used to interact with Specware. The XEmacs-Specware combo can be thought as a (rather limited) Integrated Development Environment (IDE).

Note that unit definitions can be managed without starting Specware at all. As described in the next chapter, Specware is used to process unit definitions. Future versions of Specware will provide true IDE functionality: unit definitions will be also managed within Specware, and the mapping to the file system could be even made transparent to the user.

Chapter 5. Processing Units

5.1. Overview

Unit definitions are processed by Specware. The user instructs Specware to process units by supplying certain commands. Specware has access, via the Lisp run time environment, to the underlying file system, so it can access the unit definitions contained in `.sw` files. The environment variable `SWPATH` determines which `.sw` files are accessed by Specware to find unit definitions.

Processing a unit causes processing of the units referenced in the defining text of the unit, recursively. For instance, if a spec `A` extends a spec `B` which in turns extends a spec `C`, when `A` is processed also `B` and `C` are processed. There must be no circularities in the chain of unit dependencies.

Processing causes progress and/or error messages to be displayed on the screen, in the XEmacs buffer containing the Lisp shell (where the user also supplies commands to Specware). Progress messages inform the user that units are processed without error. Error messages provide information on the cause of errors, so that the user can fix them by editing the unit definitions. When an error occurs in the definition of some unit, Specware displays the `.sw` file containing the definition in an XEmacs buffer, with the cursor at the point where the erroneous text is.

The processing of certain kinds of units also results in the creation of new files, as an additional side effect. For instance, Lisp programs are a kind of unit, constructed by the `generate` operator of Metaslang. A side effect of processing one such unit is that the resulting code is written into a `.lisp` file.

When Specware processes a unit, it saves the processing results into an internal cache, associating the results to the unit's name. By using this cache, Specware avoids unnecessary re-computations: it only re-processes the units whose files have changed since they were processed last time. From the point of view of the final result, this caching mechanism is completely transparent to the user. However, it improves the performance and response time of the system.

5.2. Resolution of Unit References

References to units may occur in the defining text of units and, as described later, in Specware commands. A unit reference is resolved to the defining text of the unit, which is contained in a `.sw` file. Reference are either absolute or relative; these two kinds are syntactically distinguished from each other and they are resolved in slightly different ways.

5.2.1. Absolute References

An absolute reference starts with "/" (forward slash), followed by a "/"-separated sequence of one or more identifiers, where the last separator can be "#" (sharp). Examples are `/a/b/c`, `/d`, and `/e#f`.

Specware resolves an absolute reference in the following steps:

1. If the reference contains "#", the "#" itself and the identifier following it are removed, obtaining a "/"-separated sequence of one or more identifiers, preceded by "/". Otherwise, no removal takes place. Either way, the result of this first step is a "/"-separated sequence of identifiers preceded by "/".
2. If the underlying operating system is Windows, the "/" signs of the "/"-separated sequence of identifiers preceded by "/", resulting from the previous step, are turned into "\" (backward slash); in addition, `.sw` is appended at the end. Otherwise, the "/" signs are left unchanged and `.sw` is appended at the end. Either way, the result of this second step is a (partial) path in the file system.
3. The path resulting from the previous step is appended after the first directory of `SWPATH`. If the resulting absolute path identifies an existing file, that is the result of this third step. Otherwise, the same attempt is made with the second directory of `SWPATH` (if any). Attempts continue until a directory is found in `SWPATH` such that the absolute path obtained by concatenating the directory with the result of the previous step identifies an existing file; such a file is the result of this step. If no such directory is found, the unit reference cannot be resolved and an error is signaled by Specware.

4. There are two alternative steps here, depending on whether the original unit reference contains “#” or not.
    a. This is the case that the original unit reference does not contain “#”. If the file resulting from the previous step is a single-unit file, i.e., it contains the defining text for a unit, the defining text is the final result of resolution. Otherwise, an error is signaled by Specware.
    b. This is the case that the original unit reference contains “#”. The file resulting from the previous step must be a multiple-unit file, i.e., it must contain a sequence of one or more unit definitions. If this is not the case, the unit reference cannot be resolved and an error is signaled by Specware. If that is the case, a unit definition is searched in the file, whose identifier (at the left of “=”) is the same as the identifier that follows “#” in the original unit reference. If no such unit definition is found, the unit reference cannot be resolved and an error is signaled by Specware. If instead such a unit definition is found, its defining text (at the right of “=”) is the final result of resolution.

For example, consider a reference `/a/b/c`. Since it does not contain “#”, the first step does not do anything. Assuming that the underlying operating system is Windows, the result of the second step is `\a\b\c.sw`. Suppose that `SWPATH` is `\users\me\specware:\tmp`, that `c:\users\me\specware` does not contain any `a` subdirectory, and that `c:\tmp\a\b\c.sw` exists. The result of the third step is the file `c:\tmp\a\b\c.sw`. If such a file is a single-unit file, its content is the result of the fourth step.

As another example, consider a reference `/e#f`. The result of the first step is `/e`. The result of the second step (still assuming Windows) is `\e.sw`. Assuming `SWPATH` as before and that `c:\users\me\specware` contains a file `e.sw`, the file `c:\users\me\specware\e.sw` is the result of the third step. The file must be a multiple-unit file. Assuming that this is the case and that it contains a unit definition with identifier `f`, its defining text is the result of the fourth step.

The directories in `SWPATH` are searched in order in the third step of resolution. So, in the last example if the directory `c:\tmp` also contains a file `e.sw`, such a file is ignored. This features can be used, for example, to shadow selected library units that

populate certain file system directories in `SWPATH`.

For example, suppose that the first directory in `SWPATH` is `\specware\libs` and that the directory `c:\specware\libs\data-structures` contains files `Sets.sw`, `Bags.sw`, `Lists.sw`, etc. defining specs of sets, bags, lists, etc. The unit reference `/data-structures/Sets` resolves to the content of the file `c:\specware\libs\data-structures\Sets.sw`. If the user wanted to experiment with a slightly version of the spec of sets, it is sufficient to prepend another directory to `SWPATH`, e.g., `shadow-lib` and to create the slightly different version of the spec of sets in `c:\shadow-lib\data-structures\Sets.sw`. The same unit reference `/data-structures/Sets` will now resolve to the new version.

5.2.2. Relative References

A relative reference is a "/"-separated sequence of one or more identifiers, where the last separator can be "#". Examples are `a/b/c`, `d`, and `e#f`. So, absolute and relative references can be distinguished by the presence or absence of "/" at their beginning.

The resolution of relative references does not depend on `SWPATH`, but on the location of the unit-defining text where the reference occurs. There are two cases two consider: the reference occurring in a single-unit file; and the reference occurring in a multiple-unit file.

Suppose that the relative reference occurs in a single-unit file. Then Specware attempts to resolve the reference in the following steps:

1. If the reference contains "#", the "#" itself and the identifier following it are removed, obtaining a "/"-separated sequence of one or more identifiers. Otherwise, no removal takes place. Either way, the result of this first step is a "/"-separated sequence of identifiers.
2. If the underlying operating system is Windows, the "/" signs of the "/"-separated sequence of identifiers, resulting from the previous step, are turned into "\"; in addition, `.sw` is appended at the end. Otherwise, the "/" signs are left unchanged and `.sw` is appended at the end. Either way, the result of this second step is a

(partial) path in the file system.

3. The path resulting from the previous step is appended after the absolute path of the directory of the file containing the relative reference. If the resulting absolute path identifies an existing file, that is the result of this third step. Otherwise, the unit reference cannot be resolved and an error is signaled by Specware.
4. There are two alternative steps here, depending on whether the original unit reference contains "#" or not.
    a. This is the case that the original unit reference does not contain "#". If the file resulting from the previous step is a single-unit file, i.e., it contains the defining text for a unit, the defining text is the final result of resolution. Otherwise, an error is signaled by Specware.
    b. This is the case that the original unit reference contains "#". The file resulting from the previous step must be a multiple-unit file, i.e., it must contain a sequence of one or more unit definitions. If this is not the case, the unit reference cannot be resolved and an error is signaled by Specware. If that is the case, a unit definition is searched in the file, whose identifier (at the left of "=") is the same as the identifier that follows "#" in the original unit reference. If no such unit definition is found, the unit reference cannot be resolved and an error is signaled by Specware. If instead such a unit definition is found, its defining text (at the right of "=") is the final result of resolution.

So, resolution of a relative reference occurring in a single-unit file is like resolution of an absolute reference, except that the directory where the reference occurs is used instead of `SWPATH`.

Suppose, instead, that the relative reference occurs in a multiple-unit file. Then Specware attempts to resolve the reference in the following steps:

1. If the relative reference is a single identifier, Specware attempts to find a unit definition with that identifier inside the file where the reference occurs. If such a unit definition is found, its defining text is the final result of resolution. Otherwise, the following steps are carried out.

2. If the reference contains “#”, the “#” itself and the identifier following it are removed, obtaining a “/”-separated sequence of one or more identifiers. Otherwise, no removal takes place. Either way, the result of this first step is a “/”-separated sequence of identifiers.
3. If the underlying operating system is Windows, the “/” signs of the “/”-separated sequence of identifiers, resulting from the previous step, are turned into “\”; in addition, `.sw` is appended at the end. Otherwise, the “/” signs are left unchanged and `.sw` is appended at the end. Either way, the result of this second step is a (partial) path in the file system.
4. The path resulting from the previous step is appended after the absolute path of the directory of the file containing the relative reference. If the resulting absolute path identifies an existing file, that is the result of this third step. Otherwise, the unit reference cannot be resolved and an error is signaled by Specware.
5. There are two alternative steps here, depending on whether the original unit reference contains “#” or not.
    a. This is the case that the original unit reference does not contain “#”. If the file resulting from the previous step is a single-unit file, i.e., it contains the defining text for a unit, the defining text is the final result of resolution. Otherwise, an error is signaled by Specware.
    b. This is the case that the original unit reference contains “#”. The file resulting from the previous step must be a multiple-unit file, i.e., it must contain a sequence of one or more unit definitions. If this is not the case, the unit reference cannot be resolved and an error is signaled by Specware. If that is the case, a unit definition is searched in the file, whose identifier (at the left of “=”) is the same as the identifier that follows “#” in the original unit reference. If no such unit definition is found, the unit reference cannot be resolved and an error is signaled by Specware. If instead such a unit definition is found, its defining text (at the right of “=”) is the final result of resolution.

So, resolution of a relative reference occurring in a multiple-unit file is like resolution of a relative reference occurring in a single-unit file, preceded by an attempt to locate the unit in the file where the reference occurs, only in case such a reference is a single

identifier.

5.3. Processing Commands

The Specware command to process units is `:sw`. The user supplies this command in the XEmacs buffer of the Lisp shell, followed by an argument. The argument is a unit reference, a piece of text defining an unnamed unit, or the name of a multiple-unit file. The whole command (including the argument) must fit in one line.

5.3.1. Processing a Named Unit

The command to process a named unit is:

```
:sw <unit-reference>
```

The unit reference can be absolute or relative. If it is absolute, Specware attempts to resolve it as explained in Section 5.2.1. If it is relative, Specware attempts to resolve it as explained in Section 5.2.2, as if the reference occurred in a single-unit file of the current Lisp directory. Either way, if resolution fails an error is signaled by Specware.

If instead resolution succeeds, Specware parses and evaluates the unit-defining text that results from resolution. Parsing and evaluation carry out the computations to construct the unit; they are Specware's "core" functionality. Parsing and evaluation implement the semantics of the Metaslang language.

If the defining text of the unit includes references to other units. Specware recursively resolves the references and parses and evaluates their defining text.

5.3.2. Processing an Unnamed Unit

The command to process an unnamed unit is:

```
:sw <unit-defining-text>
```

The unit defining text is written in the Metaslang language. It can be anything that can appear in a single-unit file or at the right of "=" in a unit definition is a multiple-unit file. The only restriction is that it must fit in one line. Specware parses and evaluates the text. If it contains references to other units, Specware recursively resolves the references and parses and evaluates their defining text.

5.3.3. Processing a Multiple-Unit File

The command to process a multiple-unit file is:

```
:sw <file-reference>
```

The file reference is syntactically exactly like a unit reference that does not contain "#". Specware attempts to resolve the reference as if it were a unit reference. If it is a relative reference, it is resolved as if it occurred inside a single-unit file in the Lisp current directory. However, the file obtained at the third step must be a multi-unit file, and not a single-unit file. If it is indeed a multi-unit file, Specware parses and evaluates all the unit definitions inside the file.

5.4. Unit-Specific Processing

This section describes what happens when specific kinds of units are processed.

5.4.1. Typechecking Specs

The user can define specs by explicitly listing the sorts, ops, and axioms comprising the spec, possibly after importing one or more spec. When these spec definitions are processed, Specware typechecks all the expressions that appear in the spec. Typechecking means checking that the expressions are type-correct, according to the rules of the Metaslang language.

In general, only some of the ops and variables that appear in an expression have explicit type (i.e., sort) information. Typechecking also involves reconstructing the types of

those ops and variables that lack explicit type information.

Typechecking is an integrated process that checks the type correctness of expressions while reconstructing the missing type information. This is done by deriving and solving type constraints from the expression. For instance, if it is known that an op `f` has sort `A -> B` then the type of the variable `x` in the expression `f(x)` must be `A`, and the type of the whole expression must be `B`.

If the missing type information cannot be uniquely reconstructed and/or if certain constraints are violated, Specware signals an error, indicating the textual position of the problematic expression.

Since the Metaslang type system features subsorts defined by arbitrary predicates, it is in general undecidable whether an expression involving subsorts is type-correct or not. When Specware processes a spec, it assumes that the type constraints arising from subsorts are satisfied, thus making typechecking decidable.

The proof obligations associated with a spec, which are extracted via the Metaslang `obligations` operator, include conjectures derived from the type constraints arising from subsorts. If all of these conjectures are discharged (using a theorem prover) then all the expressions in the spec are type-correct.

5.4.2. Proving Properties in Specs

Specware provides a mechanism for verifying the correctness of properties either specified in Metaslang specs or automatically generated as proof obligations arising from refinements or typechecking. Currently Specware comes packaged with the Snark first-order theorem prover. Interaction with Snark is through the proof unit described below.

5.4.2.1. The Proof Unit

The user invokes the Snark theorem prover by constructing a proof term. A typical proof term is of the form:

```
prove property in spec_term
```

```
using hypothesis1 hypothesis2 ...
options "(use-paramodulation t)
         (use-resolution nil)
         (use-hyperresolution t)"
```

If this term is in file `proof.sw` then issuing the command `:sw proof` will result in translating `hypothesis1`, `hypothesis2` and `property` to Snark and then invoking the Snark prover to try to prove `property` from `hypothesis1` and `hypothesis2` using the options in the `options` list. Note that `property`, `hypothesis1`, and `hypothesis2` are all properties (i.e. axioms, conjectures, or theorems) that appear in `spec_term`. Note also that `hypothesis1` and `hypothesis2` are required to appear earlier in `spec_term` than `property`. For most users the `options` list will be omitted. Additionally the `using` part can be omitted as well. In this case all the properties that appear in `spec_term` prior to `property` will be used to prove `property`.

After Snark completes Specware will report on the success or failure of the Snark proof.

5.4.2.2. Proof errors

Specware will report an error if `property` does not occur in `spec_term` or if one of the axioms do not occur in `spec_term` prior to `property`.

Snark will likely break into Lisp if then user inputs an incorrect option.

5.4.2.3. Proof Log Files

In the course of its execution Snark typically outputs a lot of information as well as a proof when it finds one. All this output can be overwhelming to the user, yet invaluable in understanding why his proofs succeeded or failed. To deal with all this output Specware redirects all the Snark output to log files. In our example above, which executed a proof in the file `proof.sw`, Specware will create a subdirectory called `snark>" at the same level as <"proof.sw`. In that directory log file, `proof.log` will be created that contains all the Snark output.

5.4.2.4. Multiple Proofs

Just as there can be multiple units per file, there can be multiple proofs in single file. For example, in file `proof.sw` we could include more than one proof as follows:

```
p1  = prove prop1 using ax1 ax2
p1a = prove prop1 using ax3
p2  = prove prop2
```

In this case Snark will be invoked three separate times, writing three different log files. In this case an additional subdirectory will be created under `snark`, called `proof`. The three log files will then be: `snark/proof/p1.log`, `snark/proof/p1a.log`, and `snark/proof/p2.log`.

5.4.2.5. Interrupting Snark

As is the case with any first-order prover, Snark is likely to either loop for ever or run for a longer time than the user would like. The user can provide a time limit for Snark by using an appropriate option. However there are likely to be times when the user wants to stop Snark in the middle of execution. He can do this by typing Cntrl-C, Cntrl-C in the *common-lisp* buffer. This will then interrupt Snark and place the user in the Lisp debugger. He can exit the debugger by issuing the `:pop` command. A log file will still be written that he can look at if he needs to.

5.4.2.6. The Prover Base Library

Specware has a base library that is implicitly imported by every spec. Unfortunately, the axioms in this library are not necessarily written to be useful by Snark. Instead of having Snark use these libraries we have created a separate base library for Snark. This library is located at `/Library/Base/ProverBase.sw`. The axioms in this spec are automatically sent to Snark as part of any proof.

5.4.2.7. The Experimental Nature of the Prover

Our experience with the current prover interface is very new and as such we are still very much experimenting with it and don't expect it to be perfect at this point in time. Many simple theorems will be provable. Some that the user thinks should be might not, and the user will be required to add further hypothesis and lemmas that may seem unnecessary. We are currently working on making this interface as robust and predictable as possible, and welcome any feedback the user can offer.

One are where the user can directly experiment is with the axioms that make up the `ProverBase`. The axioms that make up an effective prover library are best determined by an experimental evolutionary process. The user is welcome to play with the axioms in the `ProverBase`, by adding new ones or changing or deleting old ones. Keep in mind the goal is to have a single library that is useful for a wide range of proofs. Axioms that are specific to different proofs should be created in separate specs and imported where needed.

5.5. Auxiliary Commands

5.5.1. Displaying Unit Values

When a unit definition is processed, a unit value is produced. For example, a spec is essentially a set of sorts, ops, and axioms. A spec can be constructed by means of various operators in the Metaslang language, but the final result is always a spec, i.e., a set of sorts, ops, and axioms.

The `:show` command is used to display the value of a unit. Like `:sw`, it is also supplied in the XEmacs buffer of the Lisp shell and it is followed by a one-line argument. The argument is either a unit reference or a unnamed unit.

The command to display a named unit is:

```
:show <unit-reference>
```

This command first processes the unit, exactly as if it were `:sw <unit-reference>`. In addition, if processing does not yield errors, the value of the unit is displayed on screen. Of course, if the referenced unit has been already processed via a `:sw` command and its defining text has not been changed after that, the `:show` command will access the results saved in Specware's internal cache.

The command to display an unnamed unit is:

```
:show <unit-defining-text>
```

This command first processes the unit, exactly as if it were `:sw <unit-defining-text>`. In addition, if processing does not yield errors, the value of the unit is displayed on screen.

The `:show` command serves to inspect the value of units constructed via the Metaslang operators. This is especially useful for beginning users as an aid to clarify the semantics of such operators.

5.5.2. Inspecting and Clearing the Cache

As already mentioned, Specware maintains an internal cache that associates processing results to unit references.

A list of the units currently present in the cache is displayed on screen via the following command:

```
:list
```

The cache is cleared (i.e., re-initialized) via the following command:

```
:sw-init
```

Normally, there is no need to use these commands.

5.5.3. Inspecting and Setting `SWPATH`

The value of the `SWPATH` environment variable is displayed on screen via the following command:

```
:swpath
```

The value of the `SWPATH` environment variable is changed via the following command:

```
:swpath <string>
```

As already explained, the string must be a colon-separated list of absolute directory paths of the underlying operating system, surrounded by double quotes.

Chapter 6. Lisp Code Generated from Specs

The translation of executable specs to Lisp code is straightforward for the most part as Lisp is a higher-order functional language. Functional expressions go to lambda expressions and most Specware sorts are implemented as lisp lists and vectors apart from the strings, numbers, characters and booleans which are implemented by the corresponding lisp datatypes. This guide is meant primarily to help the user in calling and debugging the functions generated from a spec, so we concentrate on the translation of op names to Lisp names and the implementation of sorts. The implementation details of procedural constructs such as pattern-matching are omitted. The interested user is free to examine the lisp code itself, which is simple but verbose for pattern-matching constructs.

6.1. Translation of Specware Names to Lisp Names

Specware ops are implemented using lisp defuns if they are functions, defparameter otherwise. Their names are uppercased and put in the package with the same name as the qualifier or `SW-SPEC` if unqualified. However, if the name is that of a built-in Lisp symbol, the name is prepended with the character "!" and not uppercased. If the qualifier of the op is the same as a built-in lisp package then "`-SPEC`" is appended to the spec name to get the package name. For example, the lisp code for the spec:

```
A qualifying spec
  def two: Nat = 2
  def add1(x:Nat): Nat = x + 1
endspec
```

is

```
(DEFPACKAGE "A")
(IN-PACKAGE "A")
```

```
(DEFPARAMETER TWO 2)
(DEFUN ADD1 (X) (NAT-SPEC::|!+| X 1))
```

6.2. Arity and Currying Normalization

All Specware functions are unary. Multiple argument functions are modeled using either functions with product domains, or curried functions. For efficiency we wish to exploit Common Lisp's support of nary functions. Arity normalization aims to minimize unpacking and packing of products when passing arguments to functions with product domains, and currying normalization aims to minimize closure creation when calling curried functions. The saving is particularly important for recursive functions where there is saving at each recursive call, and in addition, currying normalization may permit the Common Lisp compiler to do tail recursion optimization. The naming scheme does not require knowledge of the definition of a function when generating calls to the function.

For each function whose argument is a product, two entry points are created: an nary function whose name is derived from the op as described above, and a unary function whose name has "`-1`" appended. E.g. for

```
op min : Integer * Integer -> Integer
```

there are two lisp functions `#'MIN` and `#'MIN-1`. A call with an explicit product is translated to the nary version, otherwise the unary version is used. For example, `min(1,2)` translates to `(MIN 1 2)`, and `foldr min inf l` translates to `(FOLDR-1-1-1 #'MIN-1 INF L)`. When generating lisp for a definition, the form is examined to see whether the definition is naturally nary. If it is, then the primary definition is nary and the unary function is defined in terms of the nary function, otherwise the situation is reversed. For example, given the definition

```
def min(x,y) = if x <= y then x else y
```

we get the two Common Lisp definitions:

```
(DEFUN MIN (X Y) (if (<= x y) x y))
(DEFIN MIN-1 (X) (MIN (CAR X) (CDR X)))
```

and given the definition

```
def multFG(x: Nat * Nat) = (F x) * (G x)
```

we get the two Common Lisp definitions:

```
(DEFUN MULTFG (X Y) (PAIR1-1 (CONS X Y)))
(DEFUN MULTFG-1 (X) (* (F-1 X) (G-1 X)))
```

For each curried function (i.e. for each function whose codomain is a function) there is an additional uncurried version of the function with "-1" added n times to the name where n is the number of curried arguments. E.g. for

```
op foldr: fa(key,a,b) (a * b -> b) -> b -> map(key,a) -> b
```

there are two lisp functions `#FOLDR` and `#FOLDR-1-1-1`.

As with arity normalization, the definition of a curried function is examined to see whether it should be used to generate the curried or the uncurried version, with the other being defined in terms of this primary version.

As well as producing more efficient code the currying normalization makes code easier to debug using the Common Lisp trace facility. For example if a function has a call of the form `foldr x y z`, this call is implemented as `(FOLDR-1-1-1 x y z)`, so you can trace `FOLDR-1-1-1` to find out how it is being called and what it is returning.

6.3. Representation of Other Sorts

`Character` and `String` sorts are represented as Lisp characters and strings, `Nat` and `Integer` as Lisp integers, lists are represented using Lisp lists, and `Boolean` `true` and `false` by the symbols `T` and `NIL`.

Sums are represented as the cons of the constructor name in keyword package and the fields of the constructor.

Binary products are implemented as cons cells (except for function arguments which are described in the previous section): `CONS` to construct and `CAR` and `CDR` to access the first and second fields. Non-binary products are implemented as vectors: constructed using `VECTOR` and the ith element accessed by `(SVREF x i-1)`.

Records are implemented the same as products with the order of the fields being alphabetic in the field names.

Restrictions and comprehensions are implemented using their supersort.

A quotient is represented as as a vector of three elements: the quotient tag (which is the value of the lisp variable `SLANG-BUILT-IN::QUOTIENT-TAG`), the representation of the quotient relation, and the actual value in the underlying sort.

Chapter 7. Debugging Generated Lisp Files

7.1. Tracing

If you need to debug your application, there a number of useful lisp facilities you should be aware of. The simplest trick is to trace some functions you care about to see what they are doing.

```
(trace foo)
  This will display the arguments to foo each time it is
  called, and will display the results each time it returns.

(untrace foo)
  This will turn off any tracing on foo.
```

7.2. Breaking

If you need a more detail view of runtime behavior, you might want to BREAK some functions you care about.

```
(trace (foo :break-all t))

  This will invoke the debugger each time foo is called,
  and upon each exit from foo.
```

Once you arrive in the debugger, the following commands are most useful:

```
 :down <n>    Move to a deeper frame  <n> is optional
 :up <n>      Move to a higher frame  <n> is optional
 :zoom <n>    Display n frames        <n> is optional

 :pri         Enter a dialog that lets you set printer
```

```
control variables.  For example, setting depth
to 5 and length to 10 will let you see the
top level structure of expressions, while
suppressing deep expressions and the tails of
long expressions.  Note that you'll need to
specify values for many contexts, but just
hitting return leaves a value unchanged.  You
will likely want to modify the trace, debugger,
and current values.

 (pprint *)   Pretty print the expression for the current
frame.  Note that this only works immediately
after arriving at a frame, e.g. via :down 0
if necessary.

 :cont        Continue as if nothing happened.
 :restart     Resume execution at this frame.
 :reset       Return to lisp top level.  (E.g., bail out
to try again.)
 :exit        Exit from lisp to operating system - ends
session.

 :help        Online documentation.

 (misc ...)   The debugger is is a read/eval/print loop, so
              arbitrary lisp forms will be evaluated (in the
              current dynamic context).

 (untrace foo)  Stop entering debugger when foo is called.
                Note that you may still enter the debugger
  for each exit from calls to foo already
  recursively in progress.
```

7.3. Timing

If you are curious about the overall performance of your application, the TIME macro will provide some quick information:

```
(time (foo nil))
  This will report the time and space used by foo, e.g.:

 USER(1): (time (list 1 2 3))
 ; cpu time (non-gc) 0 msec user, 0 msec system
 ; cpu time (gc)     0 msec user, 0 msec system
 ; cpu time (total)  0 msec user, 0 msec system
 ; real time  231 msec
 ; space allocation:
 ;  6 cons cells, 0 symbols, 0 other bytes, 0 static bytes
 (1 2 3)
```

Note that TIME it transparent, i.e., it returns whatever its argument would return, including multiple values, etc., so it is safe to intersperse it nearly anywhere.

Common Lisp has more facilities for rolling your own timers: see the generic Common Lisp documentation, or contact Kestrel Technologies.

7.4. Interrupting

Finally, note that a useful trick in lisp is to start your application, e.g. (foo nil), then at an appropriate time hit control-C. This will interrupt your application and put your into the debugger. From there you can enter the command :zoom to see the top of the stack. That can often be quite revealing.