Specware Transformation Manual

Release 4.2

Kestrel Institute

June 25, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Experimental nature

The "Transformation Shell" is an experimental addition to the Specware system, currently under active development. This manual records a snapshot, and is probably already outdated as you read it.

No guarantees should be expected concerning the correct operation of the Transformation Shell. Assurance, as can be provided by the "correct by construction paradigm", depends on properly discharging the proof obligations engendered by specs and refinements, which is independent of the operation of the Transformation Shell.

1.2 Transformations

Specware 4.2 has a new construct in its "spec calculus" fragment of Metaslang, not yet documented in the language manual. To the productions of `spec_term`, add a new construct `spec_transformation`, with grammar rule:

```
spec_transformation  ::=  transform spec_term by transformation_list
transformation_list  ::=  "{" [ transformation_step
                          { , transformation_step }* ] "}"
```

Transformations are refinements that transform specs by means of rewriting techniques, possibly combining automated strategies with ad hoc rewrite steps, based on higher-order pattern matching so as to apply generic and domain-specific theorems and equational logic. The Transformation Shell can be viewed as an interactive tool for constructing `transformation_lists` for `spec_transformations`.

1.3 Interaction

The Transformation Shell is started from within the Specware Shell, and operates likewise in a read-command – perform-command – report-back cycle. The commands issued by the user correspond to `transformation_steps` as occuring in `transformation_lists`, but have a simplified syntax, reducing the number of keystrokes required for entering them. At any time, the Transformation Shell can be made to produce a `transformation_list` in proper Metaslang syntax that can be used as is in a `.sw` file.

Most Specware Shell commands are also available from the Transformation Shell and can be invoked directly without need to leave the Transformation Shell. The `proc` command for processing a unit is available, but has additional effects as described in the next chapter. The abbreviation `p` for `proc` is not available; it has been shadowed by the abbreviation `p` for the new Transformation Shell command `move previous`.

1.4 A (very) simple example

The following example has been chosen for extreme simplicity, rather than for being realistic or illustrating the potential of transformations. Consider the following spec:

```
spec
  theorem commutative_+ is
    fa (i: Integer, j : Integer) i + j = j + i

  theorem neutral_+_0 is
    fa (i: Integer) i + 0 = i

  op double : Integer -> Integer
  def double i = 0 + 2 * i
endspec
```

The two theorems are proof obligations; under the assumption that they have been or will be discharged, it is safe to apply them in simplifying the definition of op `double`. Assuming that the spec goes by the name `Example`, the user can enter the Transformation Shell by issuing the (Specware Shell) command

```
* transform Example
```

The Transformation Shell responds with

```
Entering Transformation Construction Shell
**
```

Note the slightly different prompt: two asterisks instead of a single one. We give the rest of the dialogue, followed by an explanation:

```
** at double
(fn i -> 0 + 2 * i)
** lr commutative_+
(fn i -> 2 * i + 0)
** lr neutral_+_0
(fn i -> 2 * i)
** done
 {at double,
    lr commutative_+,
    lr neutral_+_0}
*
```

and the user is back in the Specware Shell, as indicated by the prompt.

The `at op-name` command puts the focus of the Transformation Shell on an `op_definition`; the effect of most transformations is limited to the current focus. By way of feedback, the contents of the focus is printed whenever there is a change. As can be seen by the fact that we have a `lambda_form`, the `expression` the Transformation Shell is operating on may be different from the `expressions` recorded in a `spec_form`. The `lr claim-name` command applies the axiom or theorem, the essence of whose `expression` must be an equality, as a left-to-right rewrite rule. At the `done` command, the list of transformations is given in Metaslang syntax; the elaboration of

```
transform Example by
 {at double,
    lr commutative_+,
    lr neutral_+_0}
```

results in a spec that is identical to `Example` except that the definition of op `double` is now

```
def double i = 2 * i
```

CHAPTER

TWO

THE TRANSFORMATION SHELL

2.1 Entering and leaving the Transformation Shell

The Transformation Shell is entered from within the Specware Shell by the command

```
transform <spec>
```

The `<spec>` argument is optional; the zero-argument version means: use the last argument of the kind "unit" last used for a Specware Shell command. It must, specifically, be a unit defining a spec.

The normal way to leave the Transformation Shell is the command

```
done
```

which returns processing to the interactive Specware Shell loop. Before handing back control to the Specware Shell, the Transformation Shell reports the `transformation_list` corresponding to the transformations performed, or "`No transformations`" if none were performed. The command `exit` and its alias `quit` – actually a Specware Shell commands – terminates the whole Specware session immediately, without reporting any transformations performed.

2.2 Focus and navigation

Most transformations are only applied to a restricted part of the spec, called the (transformation) "focus". The focussing command

```
at <op>
```

puts the focus on the defining `expression` for an op.

It is possible to navigate by moving the focus around by issuing a move command

```
move <m1> <m2> <m3> ...
```

in which each navigation directive `<m1>`, `<m2>`, ..., is one of `first`, `last`, `previous`, `next`, `widen`, `search <token>`, `reverse-search <token>` and `all`. The keyword `move` is optional, and each navigation directive may be abbreviated by its first letter; for example, the command `p` is equivalent to `move previous`.

Assuming the focus has been set `at x`, where op `x` is defined by

```
op x : Nat = (1 + 2) * (if 3 = 4 then 5 else 6)
```

the subsequent effect of these navigation commands is as follows:

```
(1 + 2) * (if 3 = 4 then 5 else 6)
** first
1 + 2
** last
2
** previous
1
** next
2
** widen
1 + 2
** search if
if 3 = 4 then 5 else 6
** reverse-search +
1 + 2
** all
(1 + 2) * (if 3 = 4 then 5 else 6)
```

So `first` focusses on the first child of the *current* focus that is an `expression`, `last` focusses on the last child, `previous` and `next` on the previous and next sibling, while `widen` widens the focus to the encompassing `expression`. The effect of `search` and `reverse-search` should be obvious. Finally, `all` widens the focus to the original one.

2.3 Rewrite, unfold, fold

In the following two Transformation Shell commands, `<claim>` is the name of an axiom or theorem occurring in the spec, including any imported specs, whose `expression` is a possibly universally quantified equation. For example, the `expression` can be

```
[a] fa (x : List a) x ++ [] = x
```

In particular, all theorems in the Base library can be used having such a form.

The left-to-right rewrite command

```
lr <claim>
```

applies the equation, viewed as a rewrite rule, in the left-to-right direction. More precisely, the first subexpression of the focus is found that matches the left-hand side of the equation. The substitution that made the left-hand side match is applied to the right-hand side of the equation,and the result replaces the matched subexpression. The matching algorithm uses higher-order matching; for example, `1 + 1` matches `f(i, i)` by the substitution

```
(f, i) := (fn x -> x + x, 1)
```

The matching algorithm takes account of the types, which should also match.

The right-to-left rewrite command

```
rl <claim>
```

applies the equation as a rewrite rule in the right-to-left direction: the first subexpression of the focus is found that matches the right- hand side of the equation, which then is replaced by the left-hand side after applying the matching substitution.

In the following two Transformation Shell commands, `<op>` is the name of an op that has a definition in the spec, including any imported specs. The definition can occur as an `op_definition`, as in

```
op [a] twice : (a -> a) -> (a -> a)
def twice f x = f(f x)
```

or in the form of an `op_declaration` containing a defining `expression`, as in

```
op [a] twice (f : a -> a) : a -> a = fn x -> f(f( x))
```

For the purpose of using this in (un)folding transformations, these are equivalent.

The unfold command

```
unfold <op>
```

"unfolds" one or more occurrences of `op_name <op>` in the focus, replacing them by the `expression` defining `<op>`. So the definition is used very much as if it was an axiom used by an `lr` rewrite command. For example, in the context of a definition for op `twice` as above, `unfold twice` applied to the focus `posNat? (twice pred n)` results in `posNat? (pred (pred n))`.

The fold command

```
fold <op>
```

"folds" the first occurrence matching the defining expression for <op>, replacing it by <op>.

Note. Folding may introduce circularity in definitions, and the result may therefore be an ill-formed spec. Formally, this means that the proof obligation cannot be discharged for the requirement that the defining equation have a unique solution.

2.4 Simplification

The simplify command

```
simplify <r1> <r2> <r3> ...
```

applies a rewriting simplifier with the supplied rules `<r1>` `<r2>`, etcetera, which must be given in the form of rewrite commands or (un)fold commands.

For example, instead of giving a sequence of rewrite commands

```
lr commutative_+
lr neutral_+_0
```

a user can issue a single simplify command

```
simplify lr commutative_+ lr neutral_+_0
```

If any of the rules is found to apply, the simplify command will try to reapply all rules on the whole resulting new contents of the focus, as well as its repertoire of some standard simplification rules.

The simplify-standard command

```
simp-standard
```

applies a standard simplifier, without additional rules. The keyword `simp-standard` may be abbreviated to `ss`.

The partial-evaluation command

```
partial-eval
```

evaluates the closed subexpressions of the focus – that is, expressions not containing unbound variables. The keyword `partial-eval` may be abbreviated to `pe`.

The abstract-common-subexpressions command

```
abstract-cse
```

abstract common (repeated) subexpressions in the focus expression. For example, applying it to

```
("object " ++ obj, "object " ++ obj ++ newline))
```

results in

```
let cse1 = "object " ++ obj in
(cse1, cse1 ++ newline)
```

The keyword `abstract-cse` may be abbreviated to `cse`.

2.5 Diverse

The undo command

```
undo <n>
```

undoes the last `<n>` commands performed by the Transformation Shell The `<n>` parameter is optional, with default 1.

The print-current-focus command

```
pc
```

print the current focus expression.

In the course of interactively applying transformations using the Transformation Shell, a user may need to modify the spec being processed in order to proceed, for example by adding a theorem needed for rewriting. The process command

```
proc <unit-term>
```

elaborates the `<unit-term>` as possibly modified by the user, and restarts the Transformation Shell on the processed spec, re-applying any earlier effectful transformation commands. The `<unit-term>` is optional; the zero-argument version means: use the same spec as before.

The trace-rewrites command

```
trace-rewrites
```

starts a print trace for individual rewrites. The keyword `trace-rewrites` may be abbreviated to `trr`.

The untrace-rewrites command

```
untrace-rewrites
```

turns off printing a trace for individual rewrites. The keyword `untrace-rewrites` may be abbreviated to `untrr`.

CHAPTER

THREE

ISOMORPHIC TYPE REFINEMENT

The following transformation is not available through the Transformation Shell but only through the "spec calculus" of Metaslang. One of the alternatives for a `transformation_step` is an `isomorphic_type_refinement`, which takes the form

```
isomorphism(<f>, <g>)
```

It operates on a whole spec. The parameters `<f>` and `<g>` must be ops that constitute the witnesses of an isomorphism between two types, say `T` and `T'`, so that

```
<f> : T  -> T'
```

and

```
<g> : T' -> T
```

are each other's inverse (and therefore bijections). The effect is that for each `op_declaration` and `op_definition` in the spec for an op having some type `F(T)` involving `T`, a modified copy is added for an op having type `F(T')`.

CHAPTER

FOUR

DEFINING TRANSFORMATIONS

4.1 Transforming Expressions

To add an expression transformation that can be invoked from the transformation shell, add a new definition to the *MetaRules* spec found in *Languages/MetaSlang/Transformations/MetaRules*:

```
op testTransform (spc:Spec) (tm : MSTerm ) : Option MSTerm =
  let _ = writeLine "Pop quiz bub."
  in (Some tm)
```

The transformation *op* takes a spec *spc* which contains the term *tm* that is to be transformed. The user will use the transformation navigation commands (e.g. *at*, *move*, etc.) to focus a term, and then apply the given transformation. The transformation metaprogram interpreter will then apply the given transform to each subterm (starting with and including the top-most term under focus) until the *op* defining the transformation returns *Some t*, with the *MSTerm t* being the new term [1]. This transformed term will be substituted in the position of the input term in the focused term. In the case that the transformation returns *None* for a given subterm, the transformation script will continue to apply the transformation to other subterms of the currently focused term.

After rebuilding Specware, this transform can be invoked from the transformation shell using the *apply* command:

```
at (someDef) { apply testTransform }
```

4.2 Transforming Specifications

There are some example transformations defined in *Languages/MetaSlang/Transformations/Simple.sw*. This section uses those as examples for defining specification transformations.

Defining a new transformation is a relatively straightforward process. Create a new specware spec, preferably in a file under the *Languages/MetaSlang/Transformations/* directory:

```
SimpleTransform = spec
```

Import supporting code as necessary. As the transformation be manipulating MetaSlang ASTs, the specs imported below should be useful:

```
import ../Specs/AnnSpec
import ../Specs/MSTerm
import ../Specs/Position
import ../Specs/QualifierMap
import ../AbstractSyntax/QualifiedId
```

[1] If the transformation op returns *Some t*, but the returned term *t* is the same as the original *tm*, then the transformation script will issue a warning.

```
import /Languages/SpecCalculus/Semantics/Evaluate/UnitId/Utilities
import /Languages/SpecCalculus/AbstractSyntax/UnitId
```

Next, define an *op* that takes a *Spec* as input, and produces a *Spec*. It is crucial that the *op* be qualified with the *SpecTransform* qualifier. This qualifier is necessary to allow the transformation to be identified when building Specware itself, which will in turn allow the transformation to be used in a Specware metaprogram using the unqualified name of the *op*. The following example transformation simply does a shallow copy of the input specification:

```
op SpecTransform.copySpec (spc : Spec) : Spec =
  {types=spc.types,
   ops=spc.ops,
   elements=spc.elements,
   qualifier=spc.qualifier}
```

If the transformation that you are defining is more sophisticated, and needs typing or other contextual information, it may be useful to define the transformation *op* within the *Env* [2] monad, which provides access to *op* type and definition information from the context in which the input specification is defined.

```
import /Languages/SpecCalculus/Semantics/Monad
import /Languages/SpecCalculus/Semantics/Environment
op SpecTransform.copySpecM (spc : Spec) : Env Spec =
   return {types=spc.types,
           ops=spc.ops,
           elements=spc.elements,
           qualifier=spc.qualifier}
```

Specification transformations can also be defined to take arguments in addition to the spec to transform. To define such a transformation, simply add the extra arguments to the transformation in curried form:

```
op SpecTransform.exArgs(spc: Spec) (tm: MSTerm): Env Spec =
  let _ = writeLine "Using transform on: " ^ printTerm tm
  in return spc

endspec
```

4.2.1 Integrating Transformations

To integrate new transformations into specware, you must import the module defining the transformation into the *Languages/SpecCalculus/Semantics/Evaluate/Transform* spec. Open that spec and add the following to the list of imports:

```
import SimpleTransform
```

Then rebuild specware:

```
$SPECWARE4> Applications/Specware/bin/linux/bootstrap
```

After the Specware build finishes, the new transform is available for manipulating specs:

```
S0 = spec
  op inc : Nat -> Nat
endspec

S1 = transform S0 by { copySpec }
```

[2] Found in */Languages/SpecCalculus/Semantics/Monad* and */Languages/SpecCalculus/Semantics/Environment*.

```
S2 = transform S1 by { copySpecM }

S3 = transform S2 by { exArgs 1 }
```