Code Generation in Specware 2.0: Release Notes

Y. V. Srinivas

June 21, 1996

Contents

1 Overview

Code generation in Specware is represented as a logic morphism from the logic of Slang to the logic of a target language such as Lisp or C++. A logic morphism translates the syntax of one logic into the syntax of another, such that the semantics is preserved.

In general, there may be several logic morphisms between any two logics or languages. On a given language fragment, the different logic morphisms typically yield translations with different performance characteristics. It is desirable to translate one part of a language via one logic morphism and another part via another morphism. Sheaves and diagram refinements provide a systematic way of combining logic morphisms.

Code generation is just another kind of refinement; so, the process is similar to that of refinement within Slang.

2 General Concepts

The concepts of "logic" and "logic morphism" that are used in Specware are described in detail in Meseguer's paper "General Logics". A logic consists of an entailment system (the syntax) and an institution (the semantics) suitably related. We will not need institutions for code generation in Specware.

2.1 Entailment Systems

Definition 2.1: *Entailment System.* An entailment system is a 3-tuple $\langle \mathbf{Sig}, sen, \vdash \rangle$ consisting of

1. a category **Sig** of signatures and signature morphisms,

2. a functor $sen : \mathbf{Sig} \to \mathbf{Set}$ (where **Set** is the category of sets and functions) which assigns to each signature Σ the set of Σ-sentences, and to each signature morphism $\sigma : \Sigma \to \Sigma'$, the function which translates Σ-sentences to Σ'-sentences (this function will also be denoted by σ), and

3. a function $\vdash$ which associates to each signature Σ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$, called Σ-entailment,

such that the following properties are satisfied:

1. *reflexivity:* for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$;

2. *monotonicity:* if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash_\Sigma \varphi$

3. *transitivity:* if $\Gamma \vdash_\Sigma \varphi_i$, for $i \in I$, and $\Gamma \cup \{\, \varphi_i \mid i \in I \,\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$;

4. $\vdash$-*translation:* if $\Gamma \vdash_\Sigma \varphi$, then for any signature morphism $\sigma : \Sigma \to \Sigma'$, $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(\varphi)$.

$\square$

2.2 Morphisms of Entailment Systems

To map one entailment system into another, we map the syntax while preserving entailment. A simple way to map syntax is to map signatures to signatures, and sentences over a signature to sentences over the translated signature. If the former is a functor, the latter becomes a natural transformation.

DEFINITION 2.2: *Entailment system morphism (plain version).* A morphism between entailment systems $\langle \Phi, \alpha \rangle : \langle \mathbf{Sig}, sen, \vdash \rangle \to \langle \mathbf{Sig}', sen', \vdash' \rangle$ is a pair consisting of a functor $\Phi : \mathbf{Sig} \to \mathbf{Sig}'$ which maps signatures to signatures and a natural transformation $\alpha : sen \stackrel{.}{\to} sen' \circ \Phi$ which maps sentences to sentences such that entailment is preserved:

$$\Gamma \vdash_\Sigma \varphi \Rightarrow \alpha_\Sigma(\Gamma) \vdash'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi).$$

□

Morphisms which map signatures to signatures are not flexible enough, especially for code generation. In general, it may be necessary to map built-in elements of one logic into defined elements of another, and vice versa. This can be realized by mapping signatures to specifications, and vice versa, or, in general, specifications to specifications. However, morphisms which map specifications to specifications are too unconstrained, so we impose the restriction that there be an underlying map of signatures. This gives the right amount of flexibility in mapping sentences. We will omit the general definition of entailment system morphism.

2.3 Localizing Logic Morphisms

Logic morphisms (and entailment system morphisms) map entire categories of specifications are once. In the Specware context, we want to be able to translate different specifications via different morphisms. In other words, we want to build a logic morphism from pieces of other logic morphisms. This is achieved in Specware by defining a new kind of arrow, inter-logic spec morphism, which localizes the action of a logic morphism to a single specification. These arrows then participate in the normal Specware modularization mechanisms of sequential and parallel composition.

2.4 Inter-Logic Spec Morphism

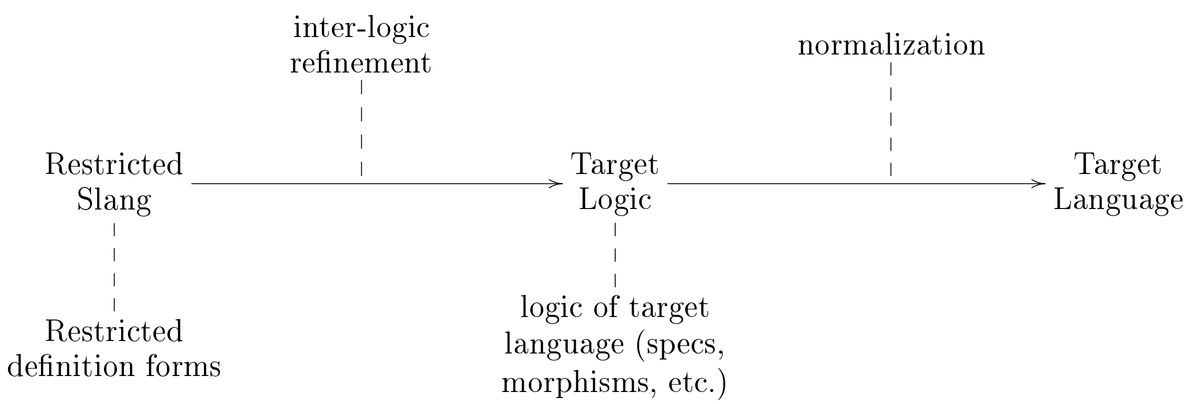
An inter-logic spec morphism connects specs in different logics. It is similar to a spec morphism in that it maps the source signature to the target signature such that source axioms translate to target theorems. The difference is that the structure of the source and target signatures may be different (e.g., Lisp specs do not have sorts), and built-in (primeval) entities in the source may map to defined entities in the target (e.g., the Slang built-in "implies" is not part of Lisp). Similarly the structure of source and target axioms (sentences) may be different (e.g., Slang operations always take a single argument and return a single result, whereas Lisp and C++ operations are n-ary and generally return only one result). Polymorphic operations in one logic may translate to families in another logic, and vice versa. For example, Slang equality (which is polymorphic) translates to a family of equalities in Lisp, while families of List operations translate to the "polymorphic" List operations of Lisp.

2.5 Inter-Logic Interpretation

An inter-logic interpretation is similar to an interpretation except that the source and target specs are in different logics. An inter-logic interpretation is a pair consisting of a an inter-logic spec morphism and a definitional extension in the target spec category, such that the codomains of the two arrows match.

3 Code Generation for Slang

Code generation for specs in Slang is summarized below. Only a subset of Slang is translatable because the target logic is generally weaker.



3.1 Translatable Definitions

A translatable definition for an operation `f` is either an explicit definition of the form `f(x) = e(x)` or a collection of conditional equations of the form `p(x) => f(x) = e(x)`. Here `p` is a predicate and `e` is any expression; neither should contain quantifiers. The arguments should be the only universally quantified variables, i.e., no implicit bindings may be introduced. Argument patterns involving `relax`, `quotient`, `product`, and `embed` are allowed. Similarly, the defined term `f(x)` may be wrapped using `relax` or `embed`.

3.2 Parameterized Specs and Instantiations

Normally, colimit specs are refined (within Slang or from Slang to code) using diagram refinement. However, diagram refinements are not sufficient for generating code for parameterized specs such as lists and arrays; these have to be refined as instantiations.

For code generation via instantiation, the morphism from the parameter spec to the body spec has to be declared as a parameterized spec. The syntax for parameterized specs is similar to that of spec morphisms, except for a change in keyword:

⟨pspec-term⟩ ⟶
`parameterized-spec` ⟨symbol⟩ : ⟨spec-term⟩ `->` ⟨spec-term⟩
(`is` | `=`) `{` [⟨sm-rules⟩] `}`

4 User Interface Changes

4.1 Refinement of Instantiations

When generating code via instantiation, a partial diagram refinement is displayed because only the parameterized spec (i.e., the morphism from parameter to body) and the actual spec need be refined. This partial diagram refinement is then completed internally by Specware before parallel composition.

Note that, when the defining diagram of the resulting inter-logic interpretation is viewed, the completed (rather than the original) diagram refinement will be shown.

4.2 Writing Code to Files

The code corresponding to an inter-logic interpretation can be saved into a file via two options available by middle-clicking the mouse: "Write code to file" and "Write code to file named". The former writes the code to a file in the current directory; the name of the file is constructed from the source spec. The latter queries the user for a filename.

To check the current directory, type "`:pwd`" in the Lisp listener. To change the current directory, type "`:cd <new-directory-string>`".

4.3 Executing Code

To execute Lisp code, load or compile and load the code file written by Specware, change the package to `"CODE"`, and type any Lisp expression using the vocabulary in the code file.

To execute C++ code, it is necessary to write a "main" function (in which one may evaluate and print expressions using the vocabulary in the code file) and then compile, link, load, and execute.

5 Lisp Code Generation

5.1 Lisp Specs

Lisp specs consist of constants, operations and definitions. In addition, a set of files containing Lisp code may be associated with a Lisp spec. The constants, operations and definitions in these files are considered to be part of the Lisp spec although not explicitly represented.

Constants and operations do not have signatures. The only modularity mechanism that is provided in the syntax is import.

5.2 Lisp Spec Morphisms

Lisp spec morphisms are similar to Slang spec morphisms in that they map constants to constants, operations to operations and definitions to definitions. We distinguish three kinds of morphisms:

import morphisms, which include one spec into another,

parameter morphisms, which exhibit the parameter to a parameterized spec, and

instantiation morphisms, which bind a parameter spec to an actual spec.

All morphisms, except instantiation morphisms, are injective.

5.3 Implicit Sharing

Import morphisms are construed as inclusions in Lisp. Thus, when Lisp specs are combined via colimits, the common imports are automatically identified.

5.4 Names in Lisp Specs

The constant/operation names in a Lisp spec are required to be unique. When generating Lisp code, the system tries to preserve operations names used in the source (Slang) spec, unless there is a name clash, in which case the system chooses a unique name.

5.5 Slang to Lisp Morphisms and Interpretations

A Slang-to-Lisp (inter-logic) morphism consists of two maps: one which maps Slang operations to Lisp operations, and one which maps each equality used in the source spec to a Lisp operation (i.e., the Slang polymorphic equality is represented as a family of Lisp equality operations). Sorts are ignored.

A Slang-to-Lisp (inter-logic) interpretation is a pair consisting of a Slang-to-Lisp morphism and a (Lisp) definitional extension.

5.6 Slang to Lisp Interpretation Morphisms

These are similar to interpretation morphisms in Slang. They are primarily used for translating parameterized specs.

5.7 Translation of Slang Primevals

Not all Slang primevals are primeval in Lisp. Thus, there is a Lisp spec `SLANG-BASE`, which contains Lisp operations corresponding to Slang primevals. The Slang-to-Lisp morphism `EMPTY-TO-SLANG-BASE` translates Slang primevals. Currently, this morphism is not user-definable, because it refers to some internal operations in Slang.

When translating to Lisp, subsorts and quotient sorts are ignored, product sorts are represented as lists and coproduct sorts are represented as tagged cons-pairs.

5.8 Adding a New Basic Translation

The system provides syntax for adding primitive Slang-to-Lisp interpretations. The user has to provide a target spec which imports `SLANG-BASE` (so that primevals have images), and two maps, one which translates operations and one which translates equalities.

Note that since subsorts and quotient sorts are ignored, the arities of Slang operations may change.

To add a new primitive translation for a parameterized spec (e.g., two-dimensional arrays), one has to define Slang-to-Lisp interpretations for the parameter and body specs, and also a Slang-to-Lisp interpretation morphism between the two. Currently, there is a restriction that the domain-to-mediator morphism of the parameter refinement must be surjective.

Note that, to properly translate a parameterized spec, every operation in the body spec which depends on some operation in the parameter spec must be explicitly defined in the body spec (and not in an attached code file). This ensures that instantiation creates a new definition with the proper substitution for the parameter operation.

5.9 Ecl-grammar: Grammar for Lisp specs, morphisms, etc.

Slang specs and morphisms

Slang specs and morphisms can only be referred to by name in Ecl-grammar.

⟨slang-spec-ref⟩ ⟶ ⟨symbol⟩

⟨slang-morphism-ref⟩ ⟶ ⟨symbol⟩

Lisp specs

⟨ecls-ref-term⟩ ⟶ ⟨symbol⟩

⟨ecls-primitive-term⟩ ⟶
`lisp-spec` [⟨symbol⟩ `is`]
[`import` ⟨ecls-term⟩ (`,` ⟨ecls-term⟩)*]
[(`file` | `files`) ⟨string⟩ (`,` ⟨string⟩)*]
[(`const` | `consts`) ⟨symbol⟩ (`,` ⟨symbol⟩)*]
[(`op` | `ops`) ⟨symbol⟩ (`,` ⟨symbol⟩)*]
[⟨ecls-definition-term⟩ (⟨ecls-definition-term⟩)*]
`end-lisp-spec`

⟨ecls-definition-term⟩ ⟶
`definition of` ⟨symbol⟩ `is` ⟨spec-op-term⟩

Lisp spec morphisms

⟨eclm-ref-term⟩ ⟶ ⟨symbol⟩

⟨eclm-identity-term⟩ ⟶ `identity-morphism`

⟨eclm-import-term⟩ ⟶ `import-morphism`

⟨eclm-rules-term⟩ ⟶
`{` [⟨eclm-rule-term⟩ (`,` ⟨eclm-rule-term⟩)*] `}`

⟨eclm-primitive-term⟩ ⟶
`lisp-morphism` [⟨symbol⟩ `:`] ⟨ecls-term⟩ `->` ⟨ecls-term⟩ `is`
⟨eclm-rules-term⟩

⟨eclm-rule-term⟩ ⟶ ⟨symbol⟩ `->` ⟨symbol⟩

Slang-to-Lisp morphisms

⟨s2eclm-ref-term⟩ ⟶ ⟨symbol⟩

⟨s2eclm-rules-term⟩ ⟶
`op-rules`
`{` [⟨s2eclm-op-rule-term⟩ (`,` ⟨s2eclm-op-rule-term⟩)*] `}`
`poly-op-rules`
`{` [⟨s2eclm-poly-op-rule-term⟩ (`,` ⟨s2eclm-poly-op-rule-term⟩)*] `}`

⟨s2eclm-primitive-term⟩ ⟶
`spec-to-lisp-morphism` [⟨symbol⟩ `:`] ⟨slang-spec-ref⟩ `->` ⟨ecls-term⟩ `is`
⟨s2eclm-rules-term⟩

⟨s2eclm-op-rule-term⟩ ⟶
(`(` ⟨qualified-name⟩ `:` ⟨spec-sort-term⟩ `)` | ⟨qualified-name⟩) `->` ⟨symbol⟩

⟨s2eclm-poly-op-rule-term⟩ ⟶
`(` ⟨qualified-name⟩ `:` ⟨spec-sort-term⟩ `)` `->` ⟨symbol⟩

Slang-to-Lisp interpretations

⟨s2ecli-ref-term⟩ ⟶ ⟨symbol⟩

⟨s2ecli-primitive-term⟩ ⟶
(`spec-to-lisp-interpretation` | `spec-to-lisp-ip-scheme`)
[⟨symbol⟩ `:`] ⟨slang-spec-ref⟩ `=>` ⟨ecls-term⟩ `is`
`mediator` ⟨ecls-term⟩
(`dom-to-med` | `domain-to-mediator`) ⟨s2eclm-term⟩
(`cod-to-med` | `codomain-to-mediator`) ⟨eclm-term⟩

Slang-to-Lisp interpretation morphisms

⟨s2eclim-ref-term⟩ ⟶ ⟨symbol⟩

⟨s2eclim-primitive-term⟩ ⟶
`spec-to-lisp-ip-scheme-morphism`
[⟨symbol⟩ `:`] ⟨s2ecli-term⟩ `->` ⟨s2ecli-term⟩ `is`
`domain-sm` ⟨slang-morphism-ref⟩
`mediator-sm` ⟨eclm-term⟩
`codomain-sm` ⟨eclm-term⟩

6 C++ Code Generation

C++ code generation is similar to Lisp code generation except for a few differences:

- sorts are retained,
- subsorts and quotient sorts are not ignored,
- equality translates to an overloaded equality in C++(rather than a family),
- names can be overloaded in C++.

6.1 Ec-grammar: Grammar for C++ specs, morphisms, etc.

Polymorphic sorts

⟨spec-sort-operation⟩ ⟶
`apply-sort` `(` ⟨spec-sort-term⟩ `,` ⟨spec-sort-term⟩ `)`

C++ specs

⟨ecs-ref-term⟩ ⟶ ⟨symbol⟩

⟨ecs-primitive-term⟩ ⟶
`c-spec` [⟨symbol⟩ `is`]
[`import` ⟨ecs-term⟩ (`,` ⟨ecs-term⟩)*]
[(`file` | `files`) ⟨string⟩ (`,` ⟨string⟩)*]
[(`sort` | `sorts`) ⟨symbol⟩ (`,` ⟨symbol⟩)*]
[⟨ecs-const-decl-term⟩ (⟨ecs-const-decl-term⟩)*]
[⟨ecs-op-decl-term⟩ (⟨ecs-op-decl-term⟩)*]
[⟨ecs-sort-definition-term⟩ (⟨ecs-sort-definition-term⟩)*]
[⟨ecs-const-definition-term⟩ (⟨ecs-const-definition-term⟩)*]
[⟨ecs-op-definition-term⟩ (⟨ecs-op-definition-term⟩)*]
`end-c-spec`

⟨ecs-const-decl-term⟩ ⟶
`const` ⟨symbol⟩ `:` ⟨spec-sort-term⟩

⟨ecs-op-decl-term⟩ ⟶
`op` ⟨symbol⟩ `:` ⟨spec-sort-term⟩

⟨ecs-sort-definition-term⟩ ⟶
`sort-definition` ⟨symbol⟩ `=` ⟨spec-sort-term⟩

⟨ecs-const-definition-term⟩ ⟶
`const-definition` ⟨symbol⟩ `=` ⟨spec-op-term⟩

⟨ecs-op-definition-term⟩ ⟶
`op-definition` ⟨symbol⟩ `=` ⟨spec-op-term⟩

C++ spec morphisms

⟨ecm-ref-term⟩ ⟶ ⟨symbol⟩

⟨ecm-identity-term⟩ ⟶ `identity-morphism`

⟨ecm-import-term⟩ ⟶ `import-morphism`

⟨ecm-rules-term⟩ ⟶
`sort-rules`
`{` [⟨ecm-sort-rule-term⟩ (`,` ⟨ecm-sort-rule-term⟩)*] `}`
`op-rules`
`{` [⟨ecm-op-rule-term⟩ (`,` ⟨ecm-op-rule-term⟩)*] `}`

⟨ecm-primitive-term⟩ ⟶
`c-morphism` [⟨symbol⟩ :] ⟨ecs-term⟩ `->` ⟨ecs-term⟩ `is`
⟨ecm-rules-term⟩

⟨ecm-sort-rule-term⟩ ⟶ ⟨symbol⟩ `->` ⟨symbol⟩

⟨ecm-op-rule-term⟩ ⟶ ⟨symbol⟩ `->` ⟨symbol⟩

Slang-to-C++ morphisms

⟨s2ecm-ref-term⟩ ⟶ ⟨symbol⟩

⟨s2ecm-rules-term⟩ ⟶
`sort-rules`
`{` [⟨s2ecm-sort-rule-term⟩ (`,` ⟨s2ecm-sort-rule-term⟩)*] `}`
`op-rules`
`{` [⟨s2ecm-op-rule-term⟩ (`,` ⟨s2ecm-op-rule-term⟩)*] `}`

⟨s2ecm-primitive-term⟩ ⟶
`spec-to-c-morphism` [⟨symbol⟩ :]
⟨slang-spec-ref⟩ `->` ⟨ecs-term⟩ `is` ⟨s2ecm-rules-term⟩

⟨s2ecm-sort-rule-term⟩ ⟶ ⟨qualified-name⟩ `->` ⟨symbol⟩

⟨s2ecm-op-rule-term⟩ ⟶
(`(` ⟨qualified-name⟩ `:` ⟨spec-sort-term⟩ `)` | ⟨qualified-name⟩) `->` ⟨symbol⟩

Slang-to-C++ interpretations

⟨s2eci-ref-term⟩ ⟶ ⟨symbol⟩

⟨s2eci-primitive-term⟩ ⟶
(`spec-to-c-ip-scheme` | `spec-to-c-interpretation`)
[⟨symbol⟩ :] ⟨slang-spec-ref⟩ `=>` ⟨ecs-term⟩ `is`
`mediator` ⟨ecs-term⟩
(`dom-to-med` | `domain-to-mediator`) ⟨s2ecm-term⟩
(`cod-to-med` | `codomain-to-mediator`) ⟨ecm-term⟩

Slang-to-C++ interpretation morphisms

⟨s2ecim-ref-term⟩ ⟶ ⟨symbol⟩

⟨s2ecim-primitive-term⟩ ⟶
`spec-to-c-ip-scheme-morphism`
[⟨symbol⟩ :] ⟨s2eci-term⟩ `->` ⟨s2eci-term⟩ `is`
`domain-sm` ⟨slang-morphism-ref⟩
`mediator-sm` ⟨ecm-term⟩
`codomain-sm` ⟨ecm-term⟩