RL-TR-95-128
Final Technical Report
July 1995

PERFORMANCE OPTIMIZATION IN ADA

Kestrel Development Corporation

Richard Jullig and Y.V. Srinivas

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-128 has been reviewed and is approved for publication.

APPROVED: Joseph Carozzoni

JOSEPH CAROZZONI
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1995	Final Sep 91 - Nov 94

4. TITLE AND SUBTITLE

PERFORMANCE OPTIMIZATION IN ADA

5. FUNDING NUMBERS

C - F30602-91-C-0080
PE - 63728F
PR - 2532
TA - 01
WU - 39

6. AUTHOR(S)

Richard Jullig and Y.V. Srinivas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Kestrel Development Corporation
3260 Hillview Avenue
Palo Alto CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory (C3CA)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-95-128

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Joseph A. Carozzoni/C3CA/(315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes the research performed to develop the first prototype of the SPECWARE system. SPECWARE supports the systematic construction of formal specifications and their stepwise refinement into programs. The formal specification language developed under the research, SLANG, is the substrate for further SPECWARE development currently in progress. The research resulted in a software development prototype offering machine-mediated production of optimized, real-time Ada code.

14. SUBJECT TERMS

Software, Automatic programming, Formal methods, Knowledge-based system

15. NUMBER OF PAGES

96

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

List of Tables

1 Introduction

This document describes the background, motivation, and results of the Performance-Optimized Ada Assistant (POAA) effort sponsored by Rome Laboratories under the KBSA program.

This effort builds upon and continues Kestrel's research into knowledge-based software synthesis technology, i.e. the development of methods and tools that exploit a formal representation of the software process to produce high-quality software via an automated, reliable, machine-supported process.

An initial goal of the POAA effort was the integration of several Kestrel-developed prototype systems embodying different software specification, synthesis, and optimization capabilities. The feasibility of the goal was strongly indicated by the emergence of a common conceptual basis. The pay-off of the integration lies in the leverage achieved through the synergy of the different capabilities.

This initial goal was achieved one year into the project. The resulting integrated system demonstrated the breadth and interplay of software synthesis capabilities on a highly stylized air traffic control problem (cf. [8]).

This integration effort also raised some serious questions about the viability of integrating the existing prototypes into a robust, usable system. Each component prototype came with its own set of engineering and usability problems. In addition, a perhaps even more serious obstacle became clear: the component systems without a doubt shared a common conceptual basis but the common conceptual primitives were not apparent in the implementation or were used under somewhat differing assumptions.

Because of the inherent complexity of software synthesis technology conceptual economy and clarity is a necessary pre-requisite for robust tools with which users can cognitively cope. Therefore the engineering of the primitive concepts and their realization within a robust system became the focus for the remainder of the project. An initial vision of how the existing capabilities would be recast using these primitives is laid out in [9].

The careful elaboration of the conceptual primitives lead to the specification language SLANG, and the development of the SPECWARE system that provides SLANG-based tools. The present SPECWARE implementation should be thought of as a system kernel: it provides basic objects and operations directly corresponding to the conceptual primitives. A more theoretical account of SLANG and SPECWARE is given in [13].

This document is divided in four parts.

Part I gives a summary description of the foundation and capabilities of the prototypes on which the effort was initially based.

Part II introduces the basic concepts of the specification language SLANG, i.e. specifications, specification morphisms, and diagrams of specifications and specification morphisms.

Part III develops from the basic building blocks in Part II a notion of specification refinement. It also gives brief description of the approach that we adopted to code generation. For simplicity of exposition, the description is given using LISP as a target language. As part of the SPECWARE prototype we developed an initial ADA backend using the same principles.

A summary of the syntax of SLANG appears in the appendix.

Part IV gives a retrospective on the effort, summarizes the achievements, and gives an outlook on future work. In particular, we briefly discuss to what extent the capabilities of the earlier prototypes have been recovered in SPECWARE and which features remain to be recreated in the future.

Part I

Description of Earlier Prototypes

KIDS, DTRE, **and** REACTO KIDS, DTRE, and REACTO constitute the knowledge-intensive components of the initial POAA. KIDS is primarily oriented toward creating algorithms given an input/output relation and toward optimizing algorithms in an applicative framework. DTRE is a data type refinement environment that supports the implementation of the Refine high-level data types used in both KIDS and REACTO. DTRE's partially implemented twin component DSS (data structure selection) supports the automated selection of date type implementations. REACTO's focus is the specification and development of reactive systems, i.e. systems that continually react to external stimuli (inputs) by producing outputs and changing their internal state. Embedded systems fall in this class, e.g. avionics or air traffic control systems.

2 KIDS: Algorithm Synthesis

2.1 Overview

The construction of a computer program is based on several kinds of knowledge: knowledge about the particular problem being solved, general knowledge about the application domain, programming knowledge peculiar to the domain, and general programming knowledge about algorithms, data structures, optimization techniques, performance analysis, etc. KIDS (Kestrel Interactive Development System [12]) is an ongoing effort to formalize and automate various sources of programming knowledge and to integrate them into a highly automated environment for developing formal specifications into correct and efficient programs (c.f. [1]).

KIDS provides tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, and other transformations. The KIDS tools serve to raise the level of language from which the programmer can obtain correct and efficient executable code through the use of automated tools.

In interaction with KIDS the user may freely select any of the design, optimization or inference functions that are meaningful within the given context. KIDS provides context-sensitive operation filtering, argument type checking and online help. When developing an algorithm with KIDS, the user typically proceeds through several conceptual phases (which may be interleaved); these are depicted in Table 1. The language used in algorithm design is currently an extended functional subset of REFINE.

Development Step	*Description*	*Examples*
Develop domain theory	Define types and operations Derive laws	Costas array theory k–queens theory
Create specification	Define problem in terms of underlying theory	Find one Costas array Find all Costas arrays
Apply design tactic	Select tactic from menu Apply to chosen problem spec	Global search tactic Divide&Conquer tactic
Perform Optimizations	Select optimiz'n from menu Apply to chosen expression	Simplification Finite Differencing
Select data type refinements	Annotate variables with implementation directives	Cf. section 3 sets to bitvectors
Compile	Choose target language Invoke compiler	Common Lisp Ada

Table 1: KIDS algorithm design phases

2.2 Domain Theories and Problem Specifications

KIDS has a basic capability for creating and managing domain theories in KIDS. In KIDS' theory development mode users can enter definitions of new functions or create new definitions by abstraction on existing expressions. The inference system can be used to verify common properties such as associativity, commutativity, or idempotence. More interestingly, we have used RAINBOW II (cf. below) to automatically derive distributive and monotonicity laws. A theory is comprised of a list of imported theories, a set of introduced types, new operations and their definitions, laws, and rules. A hierarchic library of theories is maintained with importation as the principle link. Figure 1 shows the definition of four concepts from the domain theory for the k–queens problem.

Figure 2 shows the formalization (specification) of the k–queens problem.

2.3 Directed Inference

Deductive inference is necessary for applying general knowledge to particular problems. The RAINBOW II system performs a form of deduction called *directed inference*. In directed inference, a *source* term (or formula) is transformed into a *target* term (or formula) bearing a specified relationship to the first [11]. As special cases it can perform first-order theorem-proving and formula simplification. It also allows the inference of sufficient conditions (antecedents) or necessary conditions (consequents) of a formula. This flexibility allows us to formulate a variety of design and optimiza-

$injective(M\colon seq(integer), S\colon set(integer))\colon boolean$
$= range(M) \subseteq S$
and $\forall(i,j)(i \in domain(M)$ **and** $j \in domain(M)$ **and** $i \neq j \implies M(i) \neq M(j))$

$bijective(M\colon seq(integer), S\colon set(integer))\colon boolean$
$= injective(M,S)$ **and** $range(M) = S$

$no\_two\_queens\_per\_up\_diagonal(S\colon seq(integer))\colon boolean$
$= \forall(i,j)(i \in domain(S)$ **and** $j \in domain(S)$ **and** $i \neq j \implies (S(i) - i \neq S(j) - j))$

$no\_two\_queens\_per\_down\_diagonal(S\colon seq(integer))\colon boolean$
$= \forall(i,j)(i \in domain(S)$ **and** $j \in domain(S)$ **and** $i \neq j \implies (S(i) + i \neq S(j) + j))$

Figure 1: Fragment of the k–queens theory

function :: $Queens$ :: $(k\colon\colon integer)\colon\colon set(seq(integer))$
where :: $1 \leq k$
returns :: $\{assign :: | :: bijective(assign, :: \{1..k\})$
and:: $no\_two\_queens\_per\_up\_diagonal(assign)$
and:: $no\_two\_queens\_per\_down\_diagonal(assign)\}$.

Figure 2: Specification of k–queens problem

tion problems as inference tasks. Directed inference can play a *constructive* role in design rather than simply *verifying* work done by the user or by some system.

The conceptual coherence of KIDS derivations depends partly on the large "grain-size" of the KIDS operations and their high level of automation (effectiveness). Directed inference provides a technical unifying foundation. Term simplification is naturally performed as the search for a minimal complexity equivalent term. Finite differencing can be decomposed into an abstraction operation followed by simplification of some subterms. Partial evaluation and specialization are both decomposed into an unfold step followed by simplification. Algorithm design tactics make repeated use of directed inference – for example, the global search tactic requires the derivation of a necessary condition in order to obtain a search tree pruning mechanism. Data type refinement uses inference to check applicability conditions by deriving properties such as upper and lower bounds of sets. The coherence of this view of the various development steps stems from the common underlying set of rules (axioms) used by the inference system. Furthermore, all of these development operations mainly depend on the existence of distributive, monotonicity, and other laws concerning the preservation of structure under change.

2.4 Algorithm Design

KIDS supports algorithm design as a process of interpreting, intuitively and technically, a given problem as an instance of a particular class of algorithms. Associated with each algorithm class are program schemes (proved correct), that under the given interpretation yield programs that solve the specified problem.

Figure 3 shows the *global search theory*; i.e. the operations for creating and splitting search spaces and for extracting solutions, and axioms that govern the interaction of the operations.

Figure 4 shows an abstract program whose primitive operations are the ones introduced by the global search theory.

Finally, Figure 5 shows an interpretation of the abstract global search theory into the k–queens theory. This interpretation formally specifies how the k–queens problem can be viewed as a global search problem. By applying this interpretation to the abstract global search program of Figure 4 we obtain a program that solves the k–queens problem.

2.5 Algorithm Optimization

The initially derived program is correct, but usually very inefficient. KIDS provides a suite of simplification and optimization functions that the user can interactively apply.

Theory $\mathcal{G}$
Sorts $D, R, \hat{R}$
Operations

I: $D \rightarrow$ *boolean*
O: $D \times R \rightarrow$ *boolean*
$\hat{I}$: $D \times \hat{R} \rightarrow$ *boolean*
$\hat{r}_0$: $D \rightarrow \hat{R}$
Satisfies: $R \times \hat{R} \rightarrow$ *boolean*
Split: $D \times \hat{R} \times \hat{R} \rightarrow$ *boolean*
Extract: $R \times \hat{R} \rightarrow$ *boolean*
$\succ$: $\hat{R} \times \hat{R} \rightarrow$ *boolean*

Axioms

GS0. $I(x) \implies \hat{I}(x, \hat{r}_0(x))$
GS1. $I(x)$ **and** $\hat{I}(x, \hat{r})$ **and** *Split*$(x, \hat{r}, \hat{s}) \implies \hat{I}(x, \hat{s})$ **and** $\hat{r} \succ \hat{s}$
Gs2. $I(x)$ **and** $O(x, z) \implies$ *Satisfies*$(z, \hat{r}_0(x))$
GS3. $I(x)$ **and** $\hat{I}(x, \hat{r}) \implies$
(*Satisfies*$(z, \hat{r})$ = $\exists(\hat{s})$ (*Split*$^*(x, \hat{r}, \hat{s})$ **and** *Extract*$(z, \hat{s})$))
GS4. Well-foundedness of $\succ$

Figure 3: Abstract Global Search theory

function $F(x\colon D)\colon set(R)$
where $I(x)$
returns $\{z \mid O(x, z)\}$
= **if** $\Phi(x, \hat{r}_0(x))$
then $F\_gs(x, \hat{r}_0(x))$
else $\{::\}$

function $F\_gs(x\colon D, \hat{r}\colon \hat{R})\colon set(R)$
where $I(x)$ **and** $\hat{I}(x, \hat{r})$ **and** $\Phi(x, \hat{r})$
returns $\{z \mid$ *Satisfies*$(z, \hat{r})$ **and** $O(x, z)\}$
= $\{z \mid$ *Extract*$(z, \hat{r})$ **and** $O(x, z)\}$
$\cup$ *reduce*$(\cup, \{ F\_gs(x, \hat{s}) \mid$ *Split*$(x, \hat{r}, \hat{s})$ **and** $\Phi(x, \hat{s})\})$.

Figure 4: Abstract Global Search program

$$
\begin{array}{rcl}
F & \mapsto & queens \\
D & \mapsto & integer \\
I & \mapsto & 1 \leq k \\
R & \mapsto & set(seq(integer)) \\
O & \mapsto & \lambda k, assign.\ bijective(assign, :: \{1..k\}) \\
 & & \textbf{and}:: no\_two\_queens\_per\_up\_diagonal(assign) \\
 & & \textbf{and}:: no\_two\_queens\_per\_down\_diagonal(assign)\} \\
\hat{R} & \mapsto & seq(integer) \\
\hat{I} & \mapsto & \lambda k, part\_sol.\ length(part\_sol) \leq k \textbf{ and } range(part\_sol) \subseteq \{1..k\} \\
Satisfies & \mapsto & \lambda assign, part\_sol.\ \exists(r)(assign, concat(part\_sol, r)) \\
\hat{r}_0 & \mapsto & [::] \\
Split & \mapsto & \lambda k, part\_sol, part\_sol'.\ length(part\_sol) < k \\
 & & \quad \textbf{and } \exists(i{:}\, integer)\ (i \in \{1..k\} \textbf{ and } part\_sol' = append(part\_sol, i)) \\
Extract & \mapsto & \lambda assign, part\_sol.\ assign = part\_sol
\end{array}
$$

Figure 5: Global search theory for k–queens

However, there are several opportunities for automating the selection and application of KIDS operations. The steps of the queens derivation are typical of almost all the global search algorithms that we have derived. After algorithm design the program bodies are fully simplified, partial evaluation is applied, followed by finite differencing, and data type refinement.

2.6 Derivation History

KIDS' history mechanism support the recording, browsing, and saving of derivation histories; the reloading and re-enacting of a saved histories, and the replay of (portions of) a derivation thread in a different context [6]. Facilities are available for producing pretty-printed hardcopies of derivation histories, that highlight the program portions affected by a derivation step.

3 DTRE: Data Type Refinement

3.1 DTRE

We have extended the REFINE language with a language for describing implementations. Correspondingly, we extended the underlying transformation system to compile specifications annotated with implementation directives. The resulting environment is called DTRE (Data Type Refinement Environment) [4].

Set	Seq	Tuple	Map
List	List	Pair	aList
BitVector	Array1	List	Array1
iBitVector	String-Seq		Code
Stack	BitVector		Lambda

Table 2: Selection of data type implementations in DTRE

As an example of an implementation directive, assume that V is a set of sets of integers. To achieve an implementation of V as a list of bitvectors we annotate V as follows:

V: set(set(integer)) impl-by set-to-list(set-to-bitvector(std-integer))

During compilation, DTRE interprets the implementation directives when refining operations on V. In our example, it would refine set operations on V into list operations and operations on elements of V into operations on bitvectors.

3.2 Data Type Theories, Interpretations, and Implementations

In DTRE knowledge is expressed and captured at a very high level: at the level of theories of abstract data types, (e.g. sets), provably correct implementations of abstract data types, qualitative approximations to quantitative methods, program analysis methods, and data structure selection rules [3].

DTRE's current refinement knowledge is based on REFINE's atomic types (boolean, character, symbol, integer, etc.) and set-theoretic compound types (set(α), set(α), map(α, β), tuple(α, β), where α and β range over types.

The properties (behavior) of types is defined by type theories; thus the knowledge base contains theories about sets, sequences, etc, as well as theories about implementation types including lists, stacks, queues. An implementation (e.g. *Set-to-List* is expressed as a *theory interpretation*, i.e. a map between the theory of sets and the theory of lists such that the translation of the set axioms follow from the list axioms. This ensures that the interpretation (translation) preserves the semantics of the set operations. Figure 6 shows a fragment of a theory interpretation that trannslates set operations into Common Lisp integer bitvectors.

Table 2 shows a selection of implementations for set-theoretic types currently provided by DTRE.

```
Theory-Interpretation    SET-TO-CL-IBITVECTOR

 Source-Theory  Set-Theory
 Target-Theory  CL-iBitVector-Theory

 Type-Parameters alpha
 Impl-Parameters beta, beta0, beta1, beta2

 Interpretation-Vars
       x : alpha impl-by beta,
       S : set(alpha) impl-by set-to-cl-ibitvector(beta0),
      S1 : set(alpha) impl-by set-to-cl-ibitvector(beta1),
      S2 : set(alpha) impl-by set-to-cl-ibitvector(beta2)

 Interpretation-Specs

  EmptySet:       {}           tr==>   0,
  Empty:          Empty(S)     tr==>  (Zerop S),
  Size:           Size(S)      tr==>  (LogCount S),
  Membership:     x in S       tr==>  (LogBitP x S),
  With:           S With x     tr==>  (LogIor  S (ASH 1 x)),
  Equality:       S1 = S2      tr==>  (eql S1 S2),
  Intersect: S1 Intersect S2   tr==>  (LogAnd S1 S2),
  ....
end-interpretation
```

Figure 6: Interpretation from sets to integer bitvectors

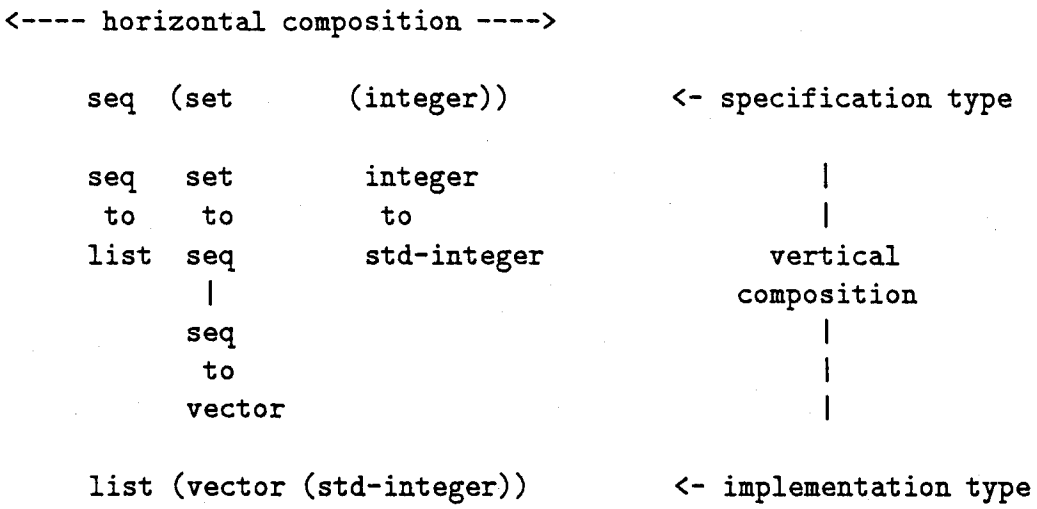


Figure 7: Horizontal and vertical composition of implementations

Since implementations are simply maps between theories they are composable, horizontally and vertically, as illustrated in in Figure 7. Composability of interpretations makes the DTRE approach to refinement by far more flexible than typical object-oriented approaches, e.g. libraries of C++ object classes or Ada package libraries. Nevertheless, DTRE can exploit existing libraries as convenient refinement targets.

3.3 DSS

While DTRE provides a language and tools for stating and realizing data type implementation decisions, DSS (data structure selection aids in the decision making process itself [2]. In its current implementation DSS supports data collection needed for making implementation choices but is not very knowledgeable about making appropriate choices. However, we have developed qualitative performance characteristics (continuing our work on the KBSA Peformance Assistant) that will allow DSS to make implementation selection decisions. In a fully implemented DSS we expect the user to make certain key decisions and DSS to appropriately "fill in the rest".

In the current DSS prototype, the primary mode of user interaction is to iteratively select data type implementations, run a test suite, and analyze performance data. The primary operations available to the user are shown in Table 3.

Operation	*Explanation*
Focus	Select specification to work on
DSS	Select implementation from context-sensitive menu
Instrument	Instrument spec for *metering* or *animation*
Compile	Compile (annotated/instrumented) specification
Animate	Run test with animation instrumentation
Test	Run testing suite and collect execution time data

Table 3: DSS Prototype operations

Operation	{}		**empty**		**size**		**in**		**with**	
Implement'n	*asympt.*	*rank*	*asympt.*	*rank*	*asympt.*	*rank*	*asympt.*	*rank*	*asympt.*	*ra*
Bitvector	O(M)	3	O(M)	3	O(M)	3	O(1)-	0	O(1)-	(
List	O(1)-	0	O(1)-	0	O(M)	3	O(M)	3	O(M)	3

Table 4: Performance characterization of set implementations

3.4 Quantitative and Qualitative Performance Measures

Running a program on a test suite provides accurate performance of limited generality. Ideally, we would like to estimate the performance of implementations by calculating the true expected cost of the operations for the actual distribution of inputs to the program. Unfortunately, the state of the art of this sort of quantitative performance estimation is still far from being practicable [16]. Experimentation with different implementations is equally costly in conventional approaches but is made feasible by DSS/DTRE since the effort of creating a new implementation is reduced to a menu selection and a compiler invocation. To alleviate the analysis and decision burden on the user, we have (as yet unimplemented) qualitative methods that approximate the ideal quantitative analysis, trading sharpness of the estimates for reduced computation cost. We explain our approach by example.

We arranged eleven asymptotic measures on a scale from 0 to 10, 0 meaning "a few instructions" and 10 meaning superexponential. For each high-level data type implementation, we associate one of these asymptotic estimates with the operations of the type. Table 4 shows this for some set operations implemented as Bitvectors or Lists.

To make an implementation choice for a particular set-valued variable we need to determine which set operations are actually performed on that variable. If the only the operations {}, **empty**, and **size** are performed, then we should select the Bitvector implementations over the List implementations since, relative to these operations,

the Bitvector implementation of sets *strongly dominates* the List implementation, i.e. performs as well or better for each operation. The converse choice is indicated if only the operations **size**, **in**, and **with** occur (on the variable under consideration). If all five operations occur then the choice is not clear. It then depends on the relative frequency of each operations. We approximate the relative frequency by qualitative weights computed as a function of the loop nesting level at which an operation occurs.

3.5 Automated Implementation Selection

Automated implementation selection takes place in two steps: (1) Identify plausible implementations. For instance, the implementation of sets as stacks depends on certain data flow conditions being true. (2) From the set of plausible implementations, identify the strongly dominating ones, if any, or the maximally good ones. The final choice is made after experimentation and testing.

3.6 Assertions, Analyses, and Bounds

Determination of plausible implementations requires substantial program analysis. Data and value flow analysis, and bounds analysis, for instance, are needed to determine whether compound structures are safely accessed, can be updated in place, can share structure, can be statically allocated, or fulfil certain special-case conditions. The DSS/DTREprototype provides intra-procedural data flow analysis and bounds analysis.

4 REACTO: Reactive System Development

4.1 Overview

REACTO is a system that supports the acquisition and correct implementation of software specifications for reactive systems [5]. The system utilizes a finite state machine formalism derived from the work of Harel ([7]), set-theoretic data structures, and relies on both classic verification techniques and consistency-preserving transformational implementation of specifications.

Formal reasoning and manipulation of programs is greatly simplified by *referential transparency*, which insures that the meaning of a program fragment is not dependent on context or state. The attractiveness of functional and logic programming derive from their maintenance of referential transparency. Although suppression of the notion of state makes manipulation of programs easier, it seriously detracts from

Development Phase	*Description*	*Tools*
Specification acquisition	Specify a reactive problem Edit and browse REACTO specs Save/restore specs to/from library	Graphics editor Hierarchy editor Spec library management
Specification compilation	Consistency check, compile, and verify REACTO specs	Consistency checker Compiler, Verifier
Specification simulation	Simulate the execution of REACTO specs	Simulator, Variable trace Graphics display

Table 5: REACTO development phases

their expressiveness. In particular, specification of a reactive system is extremely awkward without the notion of state. The challenge addressed in the REACTOeffort is to provide a notion of state and state change in a way that supports analysis and manipulation similar to that possible in functional and logic languages. The idea is to isolate state changes to those specified using a finite state machine formalism and to provide a functional language that specifies changes to abstract data structures when finite state machine transitions are executed.

Reliability requirements are addressed in two ways. First vthe correctness of the transformations implementing the compiler will be proved. Second the user may associate assertions with states. These assertions are redundant in the sense that the execution of the machine is not determined by the assertions. Their purpose is to specify invariants that the specifier believes must be true whenever the associated state is entered. Traditional verification technology is used to discharge (verify) these assertions at compile time. Any remaining assertions are checked (by computation) upon entry to the associated state at run-time.

When developing a reactive specification with REACTO, the user typically proceeds through several development phases (which may be interleaved); the system provides various tools to ease the development tasks in each phase (see Table 5).

4.2 The FSM Formalism and Specification Acquisition

REACTO uses graphically presented finite state machines (FSM) as the underlying framework to model reactive systems. A hierarchical notion of state is employed by REACTO to model the hierarchical, modularized structure of a complex reactive system, and to support stepwise refinement of the design. States are composed of substates in such a way that properties that are associated with a state apply to all of its substates. Thus a transition leaving a state is equivalent to a set of transitions each of which exits from a substate of that state.

```
state voter_select
    own_vars (selection: map(tuple(integer, integer), integer))
    assertion ∀(x)(X ∈ domain(M)
                    ⟹ reduce(+, {tally(< x, y >) | (y) y ∈ [1..M(x)]})
                    ≤ size(voters_who_voted) − 1)
    consists_of (select_init, cast_loop, validate_entry)
    initial_state (select_init)


"done"
transition cast_loop_transition_2
    from cast_loop to new_voter
    predicate examine_interface_var(*keyboard_input*) = done
    action tally ← {| < I, J >→ tally(< i, j >) + selection(< i, j >)
                    | (i, j) i ∈ domain(M) and j ∈ [1..M(i)] |};
           number_of_votes_cast ← size({< i, j > | (i, j)
                                        i ∈ domain(M) and j ∈ [1..M(i)]
                                        and selection(< i, j >) = 1});
           clear_interface_var(*keyboard_input*)
```

Figure 8: Fragment of the Voting–Machine Specification

The FSM formalism provides the top-level control structure allowing the specification of the state hierarchy of a reactive system. Associated with each state are variables, scoped with respect to the state hierarchy. Associated with each transition is a predicate over visible variables which guards the execution of the transition and an action which consists of assignments of values to visible variables. The values assigned to variables are specified as an expression in a functional language. Figure 8 shows the definition of a state and a transition from a REACTO specification *Voting-Machine.*

Specification acquisition is done via an interface consisting of a nested icon graphic display and hierarchy browser. The graphic display presents the finite state machine hierarchy of states and transitions. Text, such as variables associated with a state, and actions associated with a transitions are presented using the hierarchy browser. Within the browser there is mouse sensitivity to the syntax of program text.

4.3 The Functional Language and the Transformation System

The functional language used in REACTO specifications is a functional subset of extended REFINE. This language provides set-theoretic type constructors, such as finite sets, sequences, maps, tuples, and relations, which substantially relieve the programmer from having to choose data representations. A rich set of operations are defined on these types including reduction, bounded quantification, set and sequence formation, lambda definitions, etc. allowing concise algorithmic description. REACTO uses the DTRE compiler as its transformation system.

4.4 The Verifier

The REACTO verifier is designed to prove the consistency of the assertions associated with the state of a REACTO specification with its operational behavior. It is based on an extension of Floyd's inductive assertions method. Since the underlying structure of a REACTO specification is a variant of a flowchart program, the use of Floyd's inductive assertion method is natural and convenient. Formally the approach is to prove by induction over execution sequences that each time a state s is entered the associated assertion is satisfied. One can use as an induction hypothesis the claim that assertions associated with other states (in particular those with transitions into s) are true. Thus the task is to verify that for each transition t into s if the assertions in its originating state are true, and it is enabled, then the assertion associated with s is true. Although verification technology has been limited in its success there are two important advantages that this approach has that limits the burden placed on the theorem prover. First, the problem is factored into small pieces, namely of verifying the correctness of each transition. Second, verification occurs at the specification level where operations are still suitably abstract and the theorem prover not overwhelmed by implementation detail.

Verification is done in two steps. The first step is to deduce the verification conditions. This is done mechanically by a *verification condition generator*. The second step is proving the truth of the verification conditions. This is done with a mechanical theorem prover. The theorem prover is designed to support verification activities. It is based on a goal-oriented proof procedure hierarchical deduction ([14], [15]) incorporated with term-rewriting, partial-evaluation, and forward-inference procedures. The prover can be used as an automated system, or as an interactive proof checker.

The verifier also provides a proof management facility, which helps extract the unproven verification conditions, and permits the user to make off-line development of proofs for them. A knowledge-base manager is designed to support a flexible use of a large set of axioms and rules derived from the domain theory of the specification language. A dependency maintenance procedure is incorporated which permits the user

to trace the history of a derivation, and supports efficient addition and/or retraction of assumptions.

4.5 The Simulator

REACTO provides an execution simulator that supports rapid prototyping of REACTO specifications. With a graphics-based environment, it allows the user to quickly execute a specification to see that its behavior is that which is intended. While a specification is simulated, the simulator will display the dynamic state changes graphically and print the current values of the state variables being traced by the user.

Part II

Core Slang

In this part, we describe the core concepts of SLANG:

specifications, which are theories in a higher order logic, i.e., typed lambda calculus[1] extended with products, coproducts, quotients and subsorts,

morphisms, which are symbol to symbol translations between specifications such that the axioms of the source specification are translated into theorems of the target specification, and

diagrams, which are directed multigraphs with the nodes labeled by specifications and the arcs labeled by specification morphisms.

The interconnection of specifications via diagrams is the primary way of putting systems together out of smaller pieces. In particular, diagrams are used to express parameterization, instantiation, importation, and the refinement of specifications. The power of the notation arises from the explicit semantics for specifications and morphisms, and the ability of diagrams to express exactly the structure of specifications and their refinement.

Convention. In Part III, we will encounter other kinds of morphisms and diagrams, e.g., interpretation morphisms and interpretation diagrams. Hence, we have the convention that when "morphism" (or "diagram") is used without a qualifier, it means "specification morphism" (resp., "specification diagram"). Other uses are qualified with the kind of objects involved.

[1]We assume the standard inference rules for typed lambda calculus; the additional rules are in the section on implicit axioms for built-in operations (see section6.5.9)

5 Names in SLANG[2]

Specifications, morphisms and diagrams can each be named, as can many of their components, such as nodes and arcs of diagrams. There is a consistent syntax for introducing names as is illustrated below.[3]

NAMED	NOT NAMED

`spec <name> is`
`  <development-element>*`
`end-spec`
	`spec`
	`  <development-element>*`
	`end-spec`

`morphism m : <name> -> <name>` `  is  { <sm-rules> }`	`{ <sm-rules> }`

`diagram <name> is`
`  <nodes-and-arcs>`
`end-diagram`
	`diagram`
	`  <nodes-and-arcs>`
	`end-diagram`

The keyword `is` may be replaced with the symbol =. Names are used in the usual way to denote the objects to which they are bound. Thus, for example, in any syntactic context in which a specification is required, a name of a specification may be substituted.

5.1 Naming and Scoping Rules

Specifications, diagrams, and morphisms each have their own individual, global namespace. Thus the same name may be used to denote, say, a specification and a morphism. Because these namespaces are global two different specifications (respectively, morphisms, diagrams) must have different names—there is no context that can disambiguate which specification a name refers to. If a specification, diagram or morphism appears as a top-level expression, i.e., it is not a subexpression of a diagram-, morphism- or specification-returning expression, it must be named. Otherwise, there is no way to refer to such an object. Conversely, a subexpression, if it is not a name denoting a specification, morphism, or diagram, cannot be named. For example,

[2]The discussion in this section refers to concepts introduced in later sections. So, the reader may wish to re-read this section after the rest of the manual. The reason for this forward reference is so that all the information about names is in one place.

[3]In this dicussion and in others throughout the manual, a BNF syntax description language is used. See section A.1 in the appendix containing the BNF for SLANG. Note that, as in the syntax for diagrams in the example, some small liberties are taken in the interest of brevity.

```
diagram FOO is
  nodes X: spec Y is ... end-spec
end-diagram
```

is illegal since the specification at node `X` cannot introduce the name `Y`.

The names of nodes and arcs are local to a diagram and have their own namespaces. This means that two diagrams may use the same name as a node name, and within the same diagram a node and an arc may have the same name.

Similarly the names used in a specification are local to the specification. Sorts, operations, definitions, and theorems all have distinct namespaces. However, because of type inference, it is generally not an error for two operations to have the same name as long as context can be used to disambiguate references.

5.2 Lexical Conventions

Valid names start with either an upper or lower case letter or an asterisk (*), and are followed by any letter, digit, an asterisk (*), exclamation point (!), hyphen (-), or question mark (?). (Also, see section A.2.14 in the BNF appendix.) Names are not case sensitive: all names are converted to uppercase internally.

The keywords in SLANG are:

```
arcs                    embed                   mediator-sm
axiom                   end-definition          morphism
body-ip                 end-diagram             nodes
by                      end-spec                of
cocone-morphism         ex                      op
cod-to-med              fa                      project
codomain-sm             from                    quotient
codomain-to-mediator    identity-morphism       relax
colimit                 import                  sort
const                   import-morphism         sort-axiom
construct               instantiate             sorts
constructors            interpretation          spec
definition              ip-scheme               spec-interpretation
diagram                 ip-scheme-morphism      theorem
dom-to-med              is                      translate
domain-sm               lambda                  translation-morphism
domain-to-mediator      mediator
```

The following characters have special meaning depending upon the context:

( ) -> , | : = . < > { } [] / +

6 Specifications

A specification is a *theory presentation*, a finite description of a formal theory. A theory consists of a *signature* (a set of sorts and and a set of operations whose domains and ranges are constructed from the given sorts) and a set of closed formulas (over the signature) that is closed under logical entailment. A theory presentation consists of a finite signature and a finite set of closed formulas. Such a presentation generates a theory consisting of the given signature and all closed formulas entailed by the given formulas.

A SLANG specification consists of a set (possibly empty) of *specification elements.*[4] Each specification element is a declaration which introduces one or more primitive sorts, an operation, an axiom, a theorem, a definition, or a constructor set. The order of the declarations is not relevant. We discuss each of these specification elements below.

Specifications are either given as *basic specifications* or constructed via *specification operations.* This section introduces basic specifications, i.e., primitive specification expressions in which all specification elements are explicitly given. Section 10 describes specification-building operations. Figure 9 shows an example of a basic specification.

6.1 Sorts

The primitive sorts of a specification are introduced via sort declarations. For instance, in the NAT-SPEC example below,

```
sorts NAT, NZ-NAT
```

introduces the sorts NAT and NZ-NAT. Note that each sort declaration consists of the keyword sort or sorts followed by a list of one or more sort identifiers separated by comma.[5]

[4] In the BNF description of SLANG these are called *development-elements*

[5] In general, we will give examples in the running text, perhaps accompanied by some description of the syntax. Again, the precise syntax can always be found in the BNF grammar in the appendix.

```
%% This is an example to illustrate the elements of a specification.
   %% This specification is NOT INTENDED to
   %%   completely or correctly characterize the natural numbers.

   spec NAT-SPEC is
     sorts NAT, NZ-NAT
     sort-axiom NZ-NAT = NAT | non-zero?

     const zero    : NAT
     const one     : NAT
     op non-zero? : NAT          -> Boolean
     op plus       : NAT, NAT    -> NAT
     op times      : NAT, NAT    -> NAT
     op div        : NAT, NZ-NAT -> NAT

     constructors {zero, one, plus} construct NAT

     axiom (equal (plus zero x) x)
     axiom commutativity-of-plus is
       (fa (x y) (equal (plus x y) (plus y x)))
     axiom (equal (plus x (plus y z)) ((plus x y) z))
     axiom (fa (x : NZ-NAT) (ex (y : NAT) (equal (times x y) one)))

     theorem (fa x (equal (plus x zero) x))

     definition of times is
        axiom (equal (times x zero) zero)
        axiom (equal (times x one) x)
        axiom (equal (times x (plus y z)) (plus (times x y) (times x z)))
     end-definition

   end-spec
```

Figure 9: Example of a basic specification

6.2 Sort Constructors

Sort constructors are functions which operate on sorts. They are used to generate compound sorts from primitive sorts or other compound sorts. SLANG has five sort constructors:

product Product sorts are denoted by a sequence of two or more sorts separated by commas. The *empty product*, i.e., the product of zero components, containing the empty tuple as its unique member, is denoted by `()`; when the empty product occurs as the domain of a function sort, its syntax may be omitted (see the examples below).

coproduct Coproduct sorts are denoted by a sequence of two or more sorts separated by "`+`". The *empty coproduct*, i.e., the coproduct of zero components is denoted by `[]`.

function A function sort is denoted by giving its domain sort and range sort separated by "`->`".

quotient Quotient sorts are denoted by a sort and an equivalence relation separated by "`/`".

subsort Subsorts are denoted by a sort and a predicate separated by "`|`".

A sort term is either the name of a primitive sort or a term constructed from other sort terms via the operators "`->`", "`,`", "`+`", "`/`", and "`|`". Given primitive sorts `A`, `B`, and `C`, here are some examples of sort terms denoting constructed sorts.

```
(1) A, B, C          product sort of A, B, and C
(2) ()               empty product sort
(3) A + B + C        coproduct sort of A, B, and C
(4) []               empty coproduct sort
(5) A -> B, C        function sort with domain A
                     and range the product sort of B and C
(6)   -> A           function sort with empty product as domain
(7) A|p              subsort of A consisting of those elements
                     which satisfy the predicate p : A -> Boolean
(8) A/e              quotient of A consisting of equivalence
                     classes of elements of A generated by the
                     equivalence relation e : A, A -> Boolean
```

For an explanation of the semantics of constructed sorts, see Section 6.5.

Precedence. The sort constructors "," and "+" have equal precedence which is higher than that of "->". Similarly, "|" and "/" have equal precedence which is higher than that of "," or "+". This precedence may be overridden by using parentheses, "(" and ")". The function sort constructor "->" is right associative. The product and coproduct sort constructors are *not* associative; hence, insertion of parentheses corresponds to grouping, and will generate different products/coproducts from the ungrouped version.

The following examples of constructed sorts illustrate precedence and associativity:

```
(1) A, B -> B, C        parses as the function sort (A,B) -> (B,C)
(2) A, (B -> B), C      parses as the product sort of A, (B -> B) and C
(3) A -> B -> C         parses as A -> (B -> C)
(4) A, (B, C)           is different from both A,B,C and (A,B),C
(5) A|p -> B + C        parses as the function sort (A|p) -> (B + C)
(6) A|p/e               parses as the quotient sort (A|p)/e
(7) A|p|q               parses as the subsort (A|p)|q
```

6.3 Sort Axioms

Sort axioms can be used to equivalence (already introduced) sorts to constructed sorts. Here are some examples; note that the sorts and ops used in a sort-axiom must be declared separately.

```
sorts NAT, NZ-NAT, COST
sort-axiom NZ-NAT = NAT | non-zero?
sort-axiom COST = NAT
op non-zero? : NAT -> Boolean

sorts ARROW, CPOA % composable pair of arrows
sort-axiom CPOA = (ARROW, ARROW) | composable?
op composable? : ARROW, ARROW -> Boolean
```

The left hand side of a sort axiom must be a primitive sort; the right hand side can be either a primitive sort or a constructed sort.

There is a semantic restriction. The sort algebra is a *free algebra*: two sorts are equal iff they are structurally equivalent. The semantic restriction is that sort axioms cannot be used to equivalence constructed sorts that are not structurally equivalent. Simple uses of sort axioms will not violate this restriction; however, the equivalencing of sorts (see Section 10.2) that occurs during colimit constructions could possibly violate the restriction. If such a violation is detected during the colimit construction, the construction will be stopped and an error message generated.

6.4 Operations

Specifications introduce operations as named constants of a specified sort. For example, NAT-SPEC contains the following operation declarations:

```
const zero    : NAT
const one     : NAT
op non-zero?  : NAT          -> Boolean
op plus       : NAT, NAT     -> NAT
```

Each operation declaration consists of the keyword `op` or `const` followed by the name of the operation, followed by a colon and a sort term which specifies the rank or signature of the operation. Typically, the rank is a function sort.

Although either the keyword `op` or the keyword `const` can be used for parsing an operation, the system chooses a specific keyword while printing: `op` if the signature of the operation is a function sort, and `const` otherwise.

Constants vs. Nullary Operations. Note that the two declarations

```
op f : s
op f : -> s
```

introduce two different operations; the former denotes a constant while the latter denotes a nullary function. The difference between them becomes apparent when they are used in a term or formula (see Section 6.6): the former appears as `f` while the latter appears as `(f)`.

6.5 Built-In Sorts and Operations

6.5.1 Boolean.

The sort `Boolean` (the sort of truth-values) and the normal operations on it are built-in, i.e., they are implicitly present in every specification. The names and signatures of these operations are given below for reference:

```
sort Boolean
const true  : Boolean
const false : Boolean
op not      : Boolean          -> Boolean
```

```
op and      : Boolean, Boolean -> Boolean
op or       : Boolean, Boolean -> Boolean
op implies  : Boolean, Boolean -> Boolean
op iff      : Boolean, Boolean -> Boolean
```

6.5.2 Quantifiers.

The quantifiers `fa` and `ex` are also built-in. The syntax for these quantifiers is similar to that of terms (see Section 6.6), except that bound variables are allowed. For example:

```
(fa (x y) (equal (plus x y) (plus y x)))

(fa (x : NZ-NAT) (ex (y : NAT) (equal (times x y) one)))
```

A quantified formula consists of three elements enclosed in parentheses: a quantifier name (`fa` or `ex`), a sequence of bound variables enclosed in parentheses, and a term of sort `Boolean`. Each bound variable consists of a name and an optional data type, which is either a primitive sort or a constructed sort such as product or function sort. See the paragraph **Implicit Axioms** for further examples.

6.5.3 Equality.

Each sort comes equipped with a predefined equality. Operations defined in a specification are congruences with respect to such equalities.

The operation symbol for this equality is always `equal` (type-inference will resolve the overloading). It is not necessary to declare the equality for a sort, although one may add axioms which constrain its meaning.[6]

6.5.4 Function Sorts.

There are two built-in operations on function sorts: a quantifier `lambda` which builds elements of function sorts, and an implicit `apply` operation that applies a function to its argument. The syntax for `apply` is Lisp-like: `(F a)`. (See section 6.6.) The syntax for `lambda` is analogous to that of the boolean quantifiers:

```
(lambda (x y) (plus (times x x) y))
```

[6]Note that if an operation such as `op equal : S, S -> Boolean` is declared, it will be treated as an operation distinct from the built-in equality for the sort `S`.

Note, however, that the order of bound variables in a lambda-expression is important (it is irrelevant for boolean quantifiers).

6.5.5 Product Sorts.

There are two kinds of built-in operations on every product sort: an n-ary tuple constructor that constructs elements of the product sort and projections that select components of tuples. The syntax of the tuple constructor is "`<a1 ... an>`", with the following judgement[7] indicating the typing:

$$\frac{\texttt{a1: S1, ..., an: Sn}}{\texttt{< a1 ... an >: S1, ..., Sn}}$$

There is also a family of projection functions, one for each element of a tuple:

```
(project i): S1,...,Sn -> Si     one each for i = 1,...,n
```

Note that, since `project` is a higher order function, the application of a projection function is written as `((project i) <a1 ... an>)`. Also, `project` is polymorphic. Hence, it is implicitly indexed by the product sort, as is ($\texttt{project}_{\texttt{S1,...,Sn}}$ `i`).

Here are some more examples of tuples:

```
<>          empty tuple (the unique element in the empty product sort)
<a b>       if a is of sort A and b is of sort B,
            then <a b> is an element of the product sort A,B
<a <b c>>   tuples can be nested
```

6.5.6 Coproduct Sorts.

The coproduct of a set of sorts is intuitively their disjoint union.[8] For every coproduct sort, there is a family of embedding operations, one for each component sort. The embeddings map elements of the component sorts into the coproduct sort. They are duals to the projections associated with product sorts.

```
(embed i): Si -> S1+...+Sn     one each for i = 1,...,n
```

`embed` is a higher order function and is polymorphic (similar to `project`,. So application is written as `((embed i) ai)` with the `embed` implicitly indexed: ($\texttt{embeds}_{\texttt{S1+...+Sn}}$ `i`).

[7] I.e., if `a1` thru `an` have types `S1` thru `Sn` resp, then the sort of the tuple `<a1 ... an>` is the product `S1,...,Sn`.

[8] Known as *variant records* in some programming languages.

6.5.7 Quotient Sorts.

Given a sort `A` and an equivalence relation `e: A, A -> Boolean` on it, the sort `A/e` denotes the quotient sort of `A` generated by `e`. The elements of the quotient sort `A/e` are equivalence classes of elements of the base sort `A`. For each quotient sort `A/e`, there is a built-in abstraction function which maps elements of the base sort to the equivalence classes containing them. This abstraction function is called `quotient`, and is a higher order polymorphic function:

```
(quotient e) : A -> A/e
```

6.5.8 Subsorts.

Given a sort `A` and a predicate `p: A -> Boolean` on it, the sort `A|p` denotes the subsort of `A` generated by `p`. The subsort `A|p` consists of those elements of the supersort `A` which satisfy the predicate `p`. For each subsort `A|p`, there is a built-in inclusion function which maps elements of the subsort to the corresponding elements of the supersort. This inclusion function is called `relax`, and is a higher order polymorphic function:

```
(relax p) : A|p -> A
```

6.5.9 Implicit Axioms.

Besides the normal congruence axioms for equality, α-equivalence, and the β-rule for application, every specification implicitly contains the following axioms characterizing the various constructed sorts. These axioms are generated by the system before a specification is passed to the prover.

1. For every product sort `s-1,...,s-n`,

   ```
   % one axiom each for i = 1,...,n
   (fa (x-1 : s-1 ... x-n : s-n) (equal ((project i) <x-1...x-n>) x-i))
   (fa (z : s-1,...,s-n) (equal <((project 1) z)...((project n) z)> z))
   ```

2. For every coproduct sort `s-1+...+s-n`,

```
% embeddings are injective
% one axiom each for i = 1,...,n
(fa (x : s-i y : s-i) (implies (equal ((embed i) x) ((embed i) y))
                               (equal x y)))
% embeddings are collectively surjective
(fa (z : s-1+...+s-n)
    (or (ex (x-1 : s-1) (equal ((embed 1) x-1) z))
        ...
        (ex (x-n : s-n) (equal ((embed n) x-n) z))))
% images of embeddings are disjoint
% one axiom each for i = 1,...,n and j = i+1,...,n
(fa (x : s-i y : s-j) (not (equal ((embed i) x) ((embed j) y))))
```

3. For every subsort `s|p`,

```
% (relax p) is injective
(fa (x : s|p  y : s|p) (implies (equal ((relax p) x) ((relax p) y))
                                (equal x y)))
% these two axioms characterize the image of (relax p)
(fa (x : s|p) (p ((relax p) x)))
(fa (y : s) (implies (p x)
                     (ex (x : s|p) (equal ((relax p) x) y))))
```

4. For every quotient sort `s/q`,

```
% (quotient q) is surjective
(fa (x : s/q) (ex (y : s) (equal ((quotient q) y) x)))
% equality on s/q is the equivalence q
(fa (x : s y : s) (iff (q x y)
                       (equal ((quotient q) x) ((quotient q) y))))
```

6.6 Terms and Formulas

Terms are constructed as is usual for typed lambda calculus or higher order logic. Formulas are just terms of sort `Boolean`.

All functions in SLANG accept one argument and return one result. Multiple arguments and multi-valued returns along with functions with no arguments are handled by accepting and returning tuples. The function application notation implicitly builds tuples if there is more than one argument. For example,

`(make-tree left node right)` is parsed as `(make-tree <left, node, right>)`.

When there is only one argument, a tuple is not automatically constructed. As a consequence, the composition `(union (split s))` below is well-formed:

```
op union : Set, Set -> Set
op split : Set -> Set, Set
axiom (equal (union (split s)) s)
```

The following tables summarizes the construction of terms in SLANG, and their sorts.

Given sorted terms –,	– is a term	of sort –
	Constants	
`c:s`	`c`	`s`
	`<>`	`()`
	Products	
`a1:s1,a2:s2`	`<a1 a2>`	`s1,s2`
`a1:s1,...,an:sn`	`<a1 ... an>`	`s1,...,sn`
`a:s1,s2`	`((project 1) a)`	`s1`
`a:s1,s2`	`((project 2) a)`	`s2`
`a:s1,...,sn`	`((project i) a)`	`si`
	Functions and Application	
`f:-> s`	`(f)`	`s`
`f:s -> t, a:s`	`(f a)`	`t`
`f:s1,...,sn -> t, a:s1,...,sn`	`(f a)`	`t`
`f:s1,...,sn -> t, a1:s1,...,an:sn`	`(f a1 ... an)`	`t`
`v:s, e:t`	`(lambda (v:s) e)`	`s -> t`
`v1:s1,...,vn:sn, e:t`	`(lambda (v1:s1...vn:sn) e)`	`s1,...,sn -> t`
	Coproducts	
`a1:s1`	`((embed 1) a1)`	`s1+s2`
`a2:s2`	`((embed 2) a2)`	`s1+s2`
	Subsorts and Quotient Sorts	
`a:s|p`	`((relax p) a)`	`s`
`a:s`	`((quotient e) a)`	`s/e`
	Quantifiers	
`v:s, e:Boolean`	`(fa (v:s) e)`	`Boolean`
`v1:s1,...,vn:sn, e:Boolean`	`(fa (v1:s1 ... vn:sn) e)`	`Boolean`
`v:s, e:Boolean`	`(ex (v:s) e)`	`Boolean`
`v1:s1,...,vn:sn, e:Boolean`	`(ex (v1:s1 ... vn:sn) e)`	`Boolean`

6.7 Axioms and Theorems

Axioms and theorems in a specification are closed formulas that use the symbols (sorts and operations) appearing in the signature of that specification. The distinction between axioms and theorems is that theorems can be proved[9] from the rest of the specification, and thus do not add to the theory generated by the specification.

Here are some examples:

```
axiom (equal (plus zero x) x)

axiom commutativity-of-plus is
  (fa (x y) (equal (plus x y) (plus y x)))

theorem (fa x (equal (plus x zero) x))
```

Note that the name is optional as is the initial universal quantifier. If you omit the inital universalquantifier, it will be added internally; however, the system will print it in the original form without the quantifier prefix.

6.8 Definitions

A definition for an operation `f:A -> B` in SLANG is a set of axioms which generates a provably[10] functional relation from `A` to `B`. Here is an example from `NAT-SPEC`:

```
definition of times is
   axiom (equal (times x zero) zero)
   axiom (equal (times x one) x)
   axiom (equal (times x (plus y z)) (plus (times x y) (times x z)))
end-definition
```

Syntactically, a definition is a set of axioms enclosed by the pair of keywords `definition` and `end-definition`. Optionally, the definition may have a name and/or the name of the operation being defined. See the BNF grammar for the precise syntax of the various options.

[9]This is not currently verified in the system.

[10]This is not currently checked in the system.

6.9 Constructor Sets

A constructor set is a set of operations with the same range sort, and implicitly introduces an induction axiom for that range sort.

Consider the specification NAT-SPEC introduced at the beginning of Section 6. The constructor set

```
constructors {zero, one, plus} construct NAT
```

implicitly introduces the following induction axiom:

```
(fa (P) (implies
         (and (and (P zero) (P one))
              (fa (x y) (implies (and (P x) (P y)) (P (plus x y)))))
         (fa (n) (P n))))
```

Freeness and reachability. Note that a constructor declaration does not imply that the images of the constructors are disjoint, nor does it imply that all elements in the carrier of a constructed sort are representable by some term. Sometimes, these properties can be asserted by explicit axioms, e.g.,

```
(not (equal zero one))
```

7 Overview of Specification Constructing Operations

There are four ways of constructing a specification in SLANG:

basic—form a basic specification as a set of specification elements (sorts, sort-axioms, operations, axioms/theorems, definitions, and constructor sets).

translate—copy a specification while renaming[11] some symbols (see Section 10.1).

colimit—take the colimit of a diagram of specifications (see Section 10.2),

import—enrich an imported specification with a set of specification elements (see Section 10.3)

[11] A renaming is a set of rules of the form { **<name>** -> **<name>**, .. } which indicates how the symbols of a specification are to be renamed.

The first form explicitly constructs a specification, while the next three are specification constructing operations which take as arguments specifications and diagrams, and yield specifications. The four ways of constructing specifications should be considered as expressions which yield specifications. Wherever a specification is expected in these or other expressions, the name of a specification can be substituted.

The operations `translate`, `colimit` and `import` will be described in Section 10. However, before these operations can be explained, morphisms and diagrams need to be described.

8 Morphisms

A morphism is a mapping from a specification called the *source specification* to a specification called the the *target specification.* Intuitively, it describes how the source specification is "embedded" in the target. A morphism m from a source specification A to a target specification B maps the sorts of A into the sorts of B, and the operations of A into the operations of B such that

1. the signatures of the operations are translated compatibly, and
2. the axioms of A are translated into theorems of B.

Morphisms are described in SLANG by listing the translations of the explicitly declated sorts and operations. The translation of constructed sorts and formulas is then computed inductively.

As an example, consider embedding the `MONOID` spec (just below) into the `SEQ` spec. (See the online library for a version of the `SEQ` spec.)

```
spec MONOID  is
 sort E
 op binop : E, E -> E
 const unit : E

  axiom associativity is
  (equal (binop x (binop y z)) (binop (binop x y) z))
  axiom left-unit-axiom is  (equal (binop x unit) x)
  axiom right-unit-axiom is (equal (binop unit x) x)

 end-spec
```

```
morphism  MONOID-TO-SEQ : MONOID -> SEQ
 is { E -> Seq, binop -> concat, unit -> empty-seq }
```

The keyword `morphism` introduces a morphism definition. As usual it is followed by an optional name. The source specification is an expression yielding a specification, typically just the name of a specification. This is followed by the keyword symbol "`->`" and a target specification. This is followed by the symbol "{" and a comma-separated list of symbol-pair associations. Each symbol-pair association (known as an *sm-rule*) associates a primitive sort (respectively, operation) symbol in the source specification with a primitive sort (respectively, operation) symbol in the target specification. The symbol-pair association is terminated by "}". A morphism is required to map every sort and operation of the source specification to a symbol in the target specification. However, if a symbol of the source specification is not mentioned in the symbol pair association, then it is assumed that the symbol is mapped to a symbol with the same name in the target specification.

Here is a simple example of a morphism that does not map signatures compatibly:

```
spec FOO is
sort A, B
op f: A -> B
end
```

```
spec BAR
sort C, D, E
op g: C -> D
op h: C -> E
end
```

```
morphism M1: FOO -> BAR is {A -> C, B -> D, f -> h}

morphism M2: FOO -> BAR is {A -> C, B -> D, f -> g}
```

Morphism `M1` is not well-formed since the sort `B` is mapped to `D` but the function `f` whose codomain is `B` is mapped to an operation whose codomain is `E`. Morphism `M2` is a well-formed morphism: it translates signatures compatibly.

Morphisms and Built-in Constructs. The translations for built-in sorts and operations cannot be specified in a morphism. These entities are automatically translated to the corresponding built-in entities in the target. Examples of built-in entities are the sort `Boolean`, the Boolean operations (`and`, `or`, `not`, etc.), quantifiers (`fa`, `ex`, `lambda`, etc.), and equality. This latter is an important point: if the morphism m maps the sort S to the sort T, *then m maps the built-in equal on S to the built-in equal on T. You cannot use a morphism to map an equality to a congruence on the target sort.*

Morphisms and Constructed Sorts. Morphisms are defined as symbol maps of the basic sorts and are extended to maps on sort expressions in the natural way. This means that since morphism `M1` above maps sorts `{A -> C, B -> D}` then it maps, for example, the constructed sort `A, B -> A` to `C, D -> C`.

8.1 Local Morphisms

In contexts where a morphism needs to be mentioned and the domain and the codomain of the morphism can be inferred, it is only necessary to specify the rules which make up the morphism. Here is an example where the morphism labeling an arc in a diagram[12] is specified by just listing the rules; the specifications labeling the nodes at either end of the arc determine the domain and codomain of the morphism, and hence they need not be specified.

```
diagram BASIC-BAG-IMPORT-DIAGRAM is
  nodes BIN-OP, COMMUTATIVE, BASIC-SEQ
  arcs  BIN-OP -> COMMUTATIVE : {}
     ,  BIN-OP -> BASIC-SEQ   : {E -> Seq, binop -> concat}
end-diagram
```

8.2 Morphism Terms

The specification-building operations which are introduced briefly in Section 7 and are fully described in Section 10 not only construct specifications but also construct one or more morphisms which relate the constructed specifications and their components. See the relevant subsections of section 10 for more on the morphisms constructed by the various specification-building operations. These morphisms can be mentioned (referred to) using the keywords below, provided the context determines the domain and codomain.

For example, given the spec `IDEMPOTENT` (below), we can use `import-morphism` to refer to its import morphism in the `BASIC-SET-IMPORT-DIAGRAM` diagram (see below).

```
spec IDEMPOTENT is
  import BIN-OP
  axiom idempotence is (equal (binop x x) x)
 end-spec

spec BASIC-BAG is colimit of BASIC-BAG-IMPORT-DIAGRAM
```

[12]See Section 9 for a description of diagrams.

```
diagram BASIC-SET-IMPORT-DIAGRAM is
  nodes BIN-OP, IDEMPOTENT, BASIC-BAG
  arcs  BIN-OP -> IDEMPOTENT : import-morphism
     ,  BIN-OP -> BASIC-BAG  : cocone-morphism from BIN-OP
end-diagram
```

Given that the domain and codomain are determined by the surrounding context, the keywords `translation-morphism` and `import-morphism` can be used to specify the corresponding morphisms (as above). The keyword `cocone-morphism` requires a node as an additional parameter because there is one such morphism from each node of a diagram to the colimit specification (see Section 10.2).

To illustrate, we continue the example above.

```
spec BASIC-BAG is colimit of BASIC-BAG-IMPORT-DIAGRAM

diagram BASIC-SET-IMPORT-DIAGRAM is
  nodes BIN-OP, IDEMPOTENT, BASIC-BAG
  arcs  BIN-OP -> IDEMPOTENT : import-morphism
     ,  BIN-OP -> BASIC-BAG  : cocone-morphism from BIN-OP
end-diagram
```

Identity morphisms. Associated with every specification is an *identity* morphism which maps every sort and operation to itself. This morphism can be mentioned using the keyword `identity-morphism` (again, assuming that the domain and codomain are determined by the context). Note that the difference between using `identity-morphism` and `{}` is that the former uses actual identity whereas the latter utilizes name identity to specify a morphism. I. e., it maps source object with name **n** to target object with the same name. Hence, if there are (say) two sorts with the same name, `{}` will fail to denote a morphism.

9 Diagrams

A *diagram* is a directed multi-graph whose nodes are labeled with specifications and whose arcs are labeled with morphisms. A multi-graph differs from a graph in that there may be more than one arc between nodes. For a diagram to be well formed, the obvious condition that must be met is that the source (target) specification of the morphism labeling an arc must be the same as the specification labeling the source (target) node of the arc. Here are two examples of diagrams: one very simple diagram used to specify a specification as a colimit and another rather more complex diagram

used to specify the embedding relations in a hierarchy of orders. The are many more examples in the library.

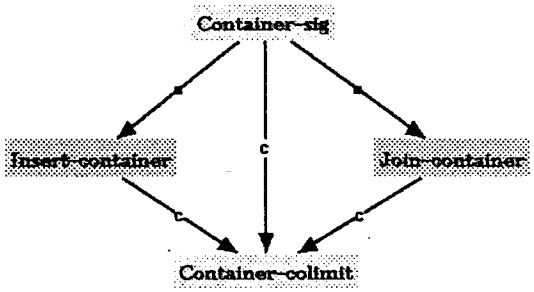


Figure 10: SLANG Diagram for CONTAINER-COLIMIT

```
spec CONTAINER-COLIMIT is
  colimit of diagram
              nodes CONTAINER-SIG, JOIN-CONTAINER, INSERT-CONTAINER
       arcs  CONTAINER-SIG -> JOIN-CONTAINER   : {}
          ,  CONTAINER-SIG -> INSERT-CONTAINER : {}
     end-diagram
```

In this example, the diagram is presented in two forms: in a graphical form[13] and in a textual form. It is a diagram for a colimit.[14] There are four nodes: CONTAINER-SIG, JOIN-CONTAINER, INSERT-CONTAINER, and CONTAINER-COLIMIT. There are also five arcs: two from CONTAINER-SIG to JOIN-CONTAINER and INSERT-CONTAINER, resp. Both these arcs arc specified by {} (map a symbol in CONTAINER-SIG to the identical symbol in JOIN-CONTAINER or INSERT-CONTAINER as appropriate). There are also three cocone arcs: one from each of CONTAINER-SIG, JOIN-CONTAINER, INSERT-CONTAINER to CONTAINER-COLIMIT.

[13]This diagram was created in SPECWARE from standard online library files.

[14]See section 10.2 for a dicussion of colimits, cocones, and cocone morphisms.

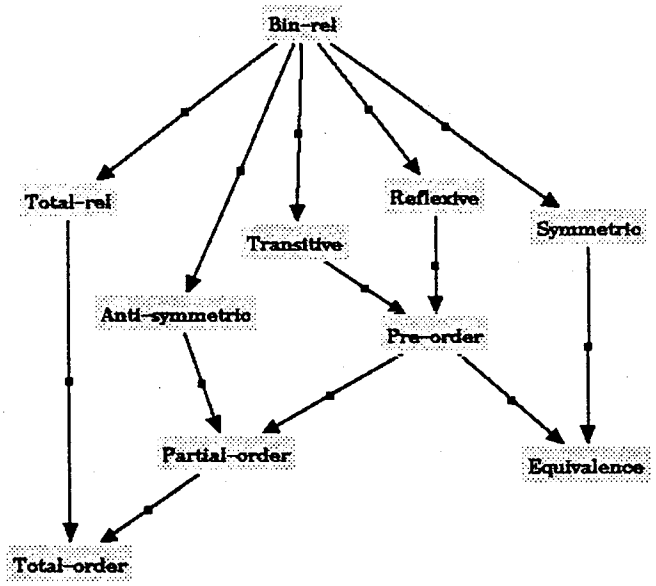


Figure 11: ORDER-HIERARCHY diagram

```
diagram ORDER-HIERARCHY is
  nodes
    BIN-REL, REFLEXIVE, TRANSITIVE, ANTI-SYMMETRIC, TOTAL-REL,
    PRE-ORDER, PARTIAL-ORDER, TOTAL-ORDER, SYMMETRIC,
    EQUIVALENCE
  arcs BIN-REL -> REFLEXIVE: {}, BIN-REL -> TRANSITIVE: {},
       BIN-REL -> ANTI-SYMMETRIC: {}, BIN-REL -> TOTAL-REL: {},
       REFLEXIVE -> PRE-ORDER: {E -> E, BINREL -> LE},
       TRANSITIVE -> PRE-ORDER: {E -> E, BINREL -> LE},
       PRE-ORDER -> PARTIAL-ORDER: {},
       ANTI-SYMMETRIC -> PARTIAL-ORDER: {E -> E, BINREL -> LE},
       PARTIAL-ORDER -> TOTAL-ORDER: {},
       TOTAL-REL -> TOTAL-ORDER: {E -> E, BINREL -> LE},
       BIN-REL -> SYMMETRIC: {},
       SYMMETRIC -> EQUIVALENCE: {E -> E, BINREL -> EQUIV},
       PRE-ORDER -> EQUIVALENCE: {E -> E, LE -> EQUIV}
  end-diagram
```

The diagram in this example specifies the embedding relationships in a hierarchy of orders. Each arc has an associated morphism specifying how its domain is embedded in its codomain. Again the diagram is presented in two forms: in a graphical form and in a textual form.

In a diagram, the (optional) name is followed by the keyword **nodes** which is followed by a list of nodes. A node may be optionally named, and is then followed by a

specification (as usual, this means an expression that yields a specification, be it a specification name or an expression).

If an explicit node name is not given, then the name of the specification at the node is used as the name of the node. This convention may lead to two nodes having the same name; the remedy, of course, is to explicitly name at least one of the nodes. Note that it is illegal to provide a name for a specification explicitly defined as the label of a node of a diagram.

Following the nodes is the keyword `arcs` and a list of arcs. Each arc may be optionally named. It is then followed by the name of the source node followed by a "`->`" and the name of the target node. This is followed by "`:`" and a morphism. This morphism can be given either by its name, a set of rules, or a morphism term.

10 Specification Building Operations

10.1 The Translate Operation

The translate operation creates a copy of a specification with the option of renaming some components. Here is an example:

EXAMPLE 10.1. The expression

```
translate
  spec
    sort E
    op le : E, E -> Boolean
    axiom reflexivity is (fa (x) (le x x))
  end-spec
by { E -> F, le -> ge }
```

evaluates to the specification

```
spec
  sort F
    op ge : F, F -> Boolean
    axiom reflexivity is (fa (x) (ge x x))
  end-spec
```

Note that the axioms are also translated to reflect the new names of the sorts and operations; however, the names of axioms, theorems, etc., remain the same. □

A translation is given by the keyword `translate` followed by a specification and a set of renaming rules that indicate how the symbols of a specification are to be renamed.

A renaming map is a one-to-one map used for copying a specification. Thus, if a renaming maps two sorts onto the same sort name, or two operations onto the same operation name, then there will be multiple sorts or operations with the same name in the copied specification. Although this is not illegal, it is inconvenient in that references to these sorts or operations will be ambiguous.

A common use of the translate operation is to rename colimit specifications (see Section 10.2 below):

```
spec BASIC-SET is
  translate co-limit of IMPORT-DIAGRAM-FOR-BASIC-SET
  by {C -> Set, empty -> empty-set, join -> union}
```

Translation morphisms. The `translate` operation also constructs a morphism (actually, an isomorphism) which maps the elements of the original specification to the corresponding elements of the copied specification. This morphism can be accessed using the syntax `translation-morphism` in a context where the domain and codomain can be inferred (see Section 8.2).

10.2 The Colimit Operation

The colimit operation is fundamental to the SLANG system.

The colimit operation takes a diagram as input and yields a specification, commonly referred to as the colimit of the diagram.

EXAMPLE 10.2. Here is a simple example in which the `REFLEXIVE` and `TRANSITIVE` specs are glued together on `BIN-REL` to construct a spec for pre-orders.

```
spec BIN-REL is
  sorts E
  op binrel : E, E -> Boolean
 end-spec

spec REFLEXIVE is
  import BIN-REL
  axiom reflexivity-axiom is (binrel x x)
 end-spec
```

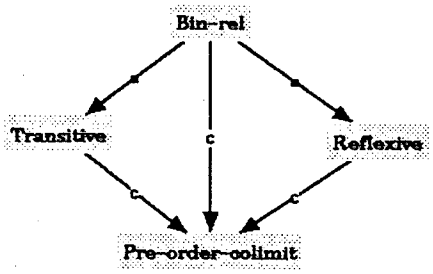


Figure 12: Colimit for PRE-ORDER

```
spec TRANSITIVE is
  import BIN-REL
  axiom transitivity-axiom is
  (implies (and (binrel x y) (binrel y z)) (binrel x z))
 end-spec

spec PRE-ORDER is
  colimit of diagram
              nodes BIN-REL, REFLEXIVE, TRANSITIVE
              arcs  BIN-REL -> REFLEXIVE  : {E -> E, binrel -> binrel}
                 ,  BIN-REL -> TRANSITIVE : {E -> E, binrel -> binrel}
              end-diagram
```

The resulting PRE-ORDER spec is equivalent to

```
spec  PRE-ORDER  is
  sorts E
  op binrel : E, E -> Boolean
  axiom reflexivity-axiom is (binrel x x)
   axiom transitivity-axiom is
  (implies (and (binrel x y) (binrel y z)) (binrel x z))
 end-spec
```

□

Informally, the colimit specification is a "shared" union of the specifications associated with each node of the original diagram. Shared here means that, based on morphisms in the diagram, sorts (respectively, operations) appearing in specifications labeling nodes of the diagram are identified as a single sort (respectively, operation) in the colimit specification.

Formally, given a diagram, the colimit operation creates a new specification, the *colimit* or *apex* specification, and a *cocone*, which assigns a new cocone morphism to each node in the given diagram, such that the domain (source) of that morphism is the specification labeling the node and the codomain (target) is the new colimit specification. The colimit specification and the cocone morphisms leading into it satisfy the property that, for every node in the diagram and for every sort or operation in the specification labeling that node, the translation of the sort or operation along any path leading from the node to the colimit specification is the same. Moreover, the colimit specification only contains those sorts and operations which arise as the translations of some sort or operation in the specification attached to some node in the diagram.

Cocone morphisms. As dicussed just above, for each node the `colimit` operation constructs a cocone morphism from the specification labeling that node in the source diagram to the colimit specification. These morphisms can be accessed using the syntax `cocone-morphism from <node-name>` in a context where the domain and codomain can be inferred (see Section 8.2).

10.2.1 The Colimit Construction Algorithm

The colimit specification and the associated cocone morphisms are constructed using the standard union-find algorithm for computing the connected components of a graph. The disjoint union of the sorts and operations contained in the specifications attached to all nodes in the diagram[15] is partitioned into equivalence classes according to the mappings given by the morphisms labeling the arcs in the diagram.

To be precise, let the disjoint union U of all signatures in a diagram D be

$$U = \{ \langle n, x \rangle \mid n \in \text{nodes}(D) \wedge n\colon S \wedge (x \in \text{sorts}(S) \vee x \in \text{operations}(S)) \},$$

where S is the specification labeling the node n.

Define an equivalence[16] relation $\equiv$ on the set U by

$$\langle n_1, x \rangle \equiv \langle n_2, y \rangle \iff (\exists a)\, a \in \text{arcs}(D) \wedge a\colon n_1 \to n_2\colon m \wedge m(x) = y,$$

[15]Note that, if the same specification labels two different nodes in a diagram, then two copies of the sorts and operations in that specification are generated in the disjoint union.

[16]The relation defined is an equivalence if we consider all the composition morphisms that are implicitly present in a diagram.

where m is the morphism labeling the arc a. That is, two sorts (operations) are equivalenced iff there is an arc whose morphism maps the one to the other.

This equivalence relation partitions the disjoint union U into equivalence classes of sorts or operations (since morphisms map sorts to sorts and operations to operations, each equivalence class will contain only one kind of object). The colimit specification contains one sort or operation corresponding to each equivalence class. The cocone morphism from the specification labeling each node in the diagram is obtained as that map which takes each sort or operation to the the sort or op corresponding to the equivalence class containing it.

In the presence of sort axioms, it is possible for the basic equivalence classes to contain constructed sorts. Hence, when using sort axioms you must ensure that no two distinct constructed sorts are equivalenced: this would violate the restriction that the sort algebra be a free algebra–see the discussion of sort axioms in Section 6.3.

As a special case of the colimit operation, if a diagram consists of just nodes with no arcs between them, the colimit is the disjoint union of the specifications labeling the nodes of the diagram. I.e., the equivalence classes are all singletons.

It is time for an example.

EXAMPLE 10.3. Consider the following diagram whose purpose is to produce a basic specification for sets by combining a specification of containers with the specifications describing properties of idempotence and commutative monoids.[17]

```
diagram IMPORT-DIAGRAM-FOR-BASIC-SET is
   nodes MONOID-SIG, BINOP,
         COMMUTATIVE-MONOID, CONTAINER, IDEMPOTENT
   arcs  MONOID-SIG -> COMMUTATIVE-MONOID
                                : {A -> M, b -> plus, u -> unit}
       , MONOID-SIG -> CONTAINER : {A -> C, b -> join, u -> empty}
       , BINOP -> CONTAINER : {B -> C, f -> join}
       , BINOP -> IDEMPOTENT : {B -> X, f -> idemop}
end-diagram
```

Here are the specifications attached to the nodes in the diagram. Most of the axioms are omitted.

```
spec MONOID-SIG is          spec BINOP is
  sort A                      sort B
```

[17]Some of the sort and operation names have been altered from the corresponding specifications in the SPECWARE online library to more clearly illustrate the action of the colimit operation.

```
            op b : A, A -> A          op f : B, B -> B
            op u : A                end-spec
          end-spec

spec COMMUTATIVE-MONOID is  spec CONTAINER is          spec IDEMPOTENT is
  sort M                     sorts C, E                 sort X
  op plus : M, M -> M        op empty     : C           op idemop : X, X -> X
  op unit : M                op singleton : E   -> C     ...
  ...                        op join      : C,C -> C  end-spec
  axiom                       ...
  (equal (plus x unit) x)   end-spec
  ...
end-spec
```

The colimit specification generated by the operation `colimit of IMPORT-DIAGRAM-FOR-BASIC-SET` is the following (again most of the axioms are omitted):

```
spec
  sorts {M,A,C,B,X}, E
  op {plus,b,join,f,idemop} : {M,A,C,B,X}, {M,A,C,B,X} -> {M,A,C,B,X}
  op {unit,u,empty}         : {M,A,C,B,X}
  op singleton              : E -> {M,A,C,B,X}
  ...
  axiom (equal ({plus,b,join,f,idemop} x {unit,u,empty}) x)
  ...
end-spec
```

The sorts in the five specifications in our diagram are partitioned into two equivalence classes; the operations get partitioned into three. Note that the signatures of the operations in the colimit specification are relinked to refer to the sorts in the colimit specification, and the operations in the axioms are relinked to refer to the operations in the colimit.

The qualified names example (example 10.4) below illustrates the case where two nodes in a diagram are labeled by the same specification. □

10.2.2 Qualified Names

As explained above, the sorts and operations in a colimit specification are equivalence classes. Each such sort or operation inherits all the names of its elements as aliases, and may be referred to (in a specification which imports the colimit) by any one of

these aliases. However, it is frequently the case that the name of an element of an equivalence class does not uniquely determine the class.

Thus, to denote these equivalence classes *qualified names* are used. A simple qualified name is a name of the form `<qualifier>.<name>`. The qualifier is the name of a node in the diagram used to construct the colimit. The denotation of such a qualified name is the equivalence class that contains the sort or operation denoted by the unqualified name in the specification attached to the qualifier node. Qualified names need not be used if a sort (or operation) name alone uniquely identifies an equivalence class. This is true even if the equivalence class contains many names.

EXAMPLE 10.4. To illustrate the need for qualified names, consider the following specification in which two partial order relations are defined on the same sort. This is done by taking the colimit of a diagram which contains two nodes labeled by the same specification, that of a partial order. The diagram also contains another node and two arcs labeled with morphisms which ensure that the two sorts in the two copies of the partial order specification are collapsed into one.

```
spec TRIV is    spec PARTIAL-ORDER is
  sorts E         sorts P
 end-spec         op le : P, P -> Boolean
                  axiom (le x x)
                  axiom (implies (and (le x y) (le y x)) (equal x y))
                  axiom (implies (and (le x y) (le y z)) (le x z))
                end-spec

morphism TRIV-TO-PO: TRIV -> PARTIAL-ORDER = { E -> P }

spec DOUBLE-PARTIAL-ORDER is
colimit of diagram
             nodes A: TRIV,
                   B: PARTIAL-ORDER,
                   C: PARTIAL-ORDER
             arcs  A -> C: triv-to-po,
                   A -> B: triv-to-po
```

```
end-diagram
```

The colimit specification will contain a single sort {E,P} with aliases E and P and *two* operations with the same name: `le : {E,P}, {E,P} -> Boolean`. If, in another specification which imports DOUBLE-PARTIAL-ORDER, we want to refer to these operations, we have to use qualified names: `B.le` and `C.le`. For example, we could require that the two orders be converses of each other:

```
axiom (iff (B.le x y) (C.le y x))
```

Or, we could rename these operations using:

```
spec DOUBLE-PARTIAL-ORDER-1 is
  translate DOUBLE-PARTIAL-ORDER by { B.le -> le, C.le -> ge }
```

□

In general, to handle the case of the specification attached to a node being itself a colimit, cascaded qualifiers are allowed. That is, the most general form of a reference in SLANG is:

```
<qualifier>.<qualifier>....<qualifier>.<name>
```

Such a reference is resolved by starting with the outermost qualifier and proceeding inwards. I.e., the outermost qualifier must be the name of a node in the diagram used to construct the current colimit, etc. To resolve a qualifier, there must be a colimit specification containing the node denoted by the qualifier.

While qualified names can be used to refer to a sort or operation of a colimit specification, the system does not display specifications using qualified names. If an equivalence class with more than one element is formed in a colimit specification it is printed as an equivalence class, i.e., as the set containing all of the names of the sorts (operations) in the class. If the class contains just a single name, the name is printed and the set brackets are suppressed.

10.3 Imports

The fourth operation for constructing specifications in SLANG is `import`. Although it is not technically necessary, i.e., the specification generated using `import` can also be generated using the other operations, it is convenient. The purpose of the `import` operation is to enrich a specification with new sorts, operations, axioms, etc.

EXAMPLE 10.5. Here is a specification in which we import the DOUBLE-PARTIAL-ORDER specification defined in Example 10.4 above and extend it with an axiom which asserts that the two orders are converses of each other.

```
spec DOUBLE-PARTIAL-ORDER-2 is
  import DOUBLE-PARTIAL-ORDER
  axiom (iff (B.le x y) (C.le y x))
end-spec
```

□

There can be only one `import` in a specification. The denotation of a spec term containing an import declaration (as in the example above) is a specification which contains all the elements of the imported specification together with any sorts, operations, axioms added in the term.

SLANG extends the notation for `import` with the following syntax to accommodate two frequently occurring constructs for building specifications:

1. `spec import <diagram> ... end-spec`

   which expands to

   `spec import colimit of diagram ... end-spec`

2. `spec import <spec-1>, <spec-2>, ..., <spec-n> ... end-spec`

   which expands to

```
spec
 import colimit of diagram
                    nodes <spec-1>, <spec-2>, ..., <spec-n>
                  end-diagram
 ...
end-spec
```

Import morphisms. The `import` operation also constructs a morphism which maps the elements of the imported specification to the corresponding elements of the importing specification. This morphism can be accessed using the syntax `import-morphism` in a context where the domain and codomain can be inferred (see Section 8.2).

Part III

Refinement Constructs in Slang

The development process of SPECWARE is intended to support the refinement of a problem (source) specification into a solution (target) specification. Refinements introduce additional components and behavioral constraints. Source and target specification as well as the refinement between them are precise, formal objects. SLANG's refinement constructs, introduced below and defined in subsequent chapters, address three important aspects of refinement: (1) construction of a solution relative to some base (problem reduction); (2) sequential (vertical) composition of refinements (refinement layers); and (3) parallel (horizontal) composition of refinements (refinement components).

11 Overview of Refinement

In SPECWARE refinement of specifications proceeds by induction on the specification structure. The next section gives an overview of refinement of basic specs and the following section gives an overview of the refinement of structured specs. The notions are more fully discussed in section 12. In the section on interpretations it will be seen that refinement of structured specs requires a systematic lifting of specification notions (specifications, spec morphisms, and specification constructing operations) to corresponding refinement notions (interpretations, interpretation morphisms, and interpretation constructing operations). See especially section 12.5.

11.1 Refinement of Basic Specifications

The basic refinement construct in SLANG is an *interpretation* (see section 12). Interpretations generalize morphisms as follows. A morphism from A to B specifies an "embedding" of the spec A into the spec B; an interpretation from A to B specifies an "embedding" of A into a *definitional extension* of B, i.e. a specification consisting of B and *definitions* of further sorts and operations. Both morphisms and interpretations are closed under sequential composition–this allows us to follow one refinement by another.

Semantically speaking, a morphism corresponds to a simple construction of models of A from models of B: from any model of B, we obtain a model of A by simply "forgetting" those sorts and operations of B that have no counterpart in A, i.e. that are outside the image of the morphism. An interpretation from A to B corresponds to a more complicated construction: first, expand the given model of B by the sorts and operations defined in the definitional extension component of the interpretation, then reduce the resulting model to the signature of A (along the embedding morphism).

From a semantic point of view, refinement (morphism or interpretation) of a spec A to spec B amounts to a restriction of the model class of A: there is an associated construction that yields an A-model for each B-model, but not every A-model can (in general) be constructed from a B-model.

11.2 Refinement of Structured Specifications

We systematically exploit the specification structure to construct interpretations for complex specs.

Colimit Refinement The colimit of a diagram of interpretations yields an interpretation from the colimit of the interpretations sources to the colimit of the interpretation targets.

Translation Refinement If spec B is a translation of spec A, then there is a translation of any interpretation with source A into an interpretation with source B.

Import Refinement If spec B imports spec A, it is not in general possible to construct an interpretation for B from an interpretation for A. However, it is possible if the import morphism is a *definitional extension.*

Translation morphisms and definitional extensions can be seen as degenerate interpretations (the source embedding morphism is the identity). Propagation of an interpretation along a translation or definitional extension is then a special case of sequential composition of interpretations.

12 Interpretations

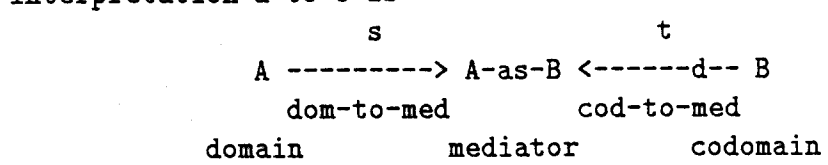
Interpretations generalize morphisms to capture a more general notion of specification refinement (for an overview of refinement, see 11). We first introduce the syntax for interpretations in SLANG and then characterize their semantics as model constructions. Subsequently, we discuss the sequential (vertical) and parallel (horizontal) composition of interpretations. The horizontal compositions, i.e. the gluing of interpretations from pieces leads us to interpretation morphisms, and to a generalization of interpretations, *interpretation schemes.* Finally, we show how to lift specification construction operations to interpretation construction operations.

12.1 Interpretations in SLANG

A prototypical example of a named SLANG interpretation is of the form

```
interpretation a-to-b: A => B is
            mediator   A-as-B
            dom-to-med s
            cod-to-med t
```

A possible graphical rendering of this construct is as follows:



Just like spec morphisms, interpretations have a domain (A) and codomain (B), both specifications. We refer to the domain and codomain also as the *source* and *target* specification, respectively. The interpretation is defined by a *source morphism* (s) whose domain is the source of the interpretations, and a target morphism (t), whose domain is the target of the interpretation. Both morphisms have the same codomain, the *mediator* specification (A-*as*-B) of the interpretation.

The target morphism must be a *definitional extension.* The mediator A-*as*-B "mediates" between source A and target B by adding definitions to B so that "A can be expressed as B" via source morphism s.

EXAMPLE 12.1. Here is a prototypical example of the use of an interpretation to refine a data type specification: the representation of sets by bags with no duplicate elements.

```
interpretation SET-TO-BAG-SUBSORT : BASIC-SET => BASIC-BAG is
  mediator SET-AS-BAG-SUBSORT
  domain-to-mediator {Set       -> Set-as-Bag,
                      empty-set -> empty-set,
                      singleton -> singleton-set,
                      union     -> set-union,
                      insert    -> set-insert,
                      empty?    -> set-empty?,
                      in        -> set-in}
  codomain-to-mediator import-morphism
```

where the mediating specification is

```
spec SET-AS-BAG-SUBSORT is
  import BASIC-BAG

  sort Set-as-Bag
  sort-axiom Set-as-Bag = Bag | no-dup?

  op no-dup? : Bag -> Boolean
```

```
definition of no-dup? is
  axiom (equal (no-dup? empty-bag) true)
  axiom (equal (no-dup? (insert x B))
               (and (no-dup? B) (not (in x B))))
end-definition

%%%%%%%%%%%%%

op empty-set     : Set-as-Bag
op singleton-set : E                        -> Set-as-Bag
op set-union     : Set-as-Bag, Set-as-Bag -> Set-as-Bag
op set-insert    : E, Set-as-Bag            -> Set-as-Bag
op set-empty?    : Set-as-Bag               -> Boolean
op set-in        : E, Set-as-Bag            -> Boolean

definition of empty-set is
  axiom (equal ((relax no-dup?) empty-set) empty-bag)
end-definition

definition of singleton-set is
  axiom (equal ((relax no-dup?) (singleton-set x)) (singleton x))
end-definition

definition of set-union is
  axiom (equal (set-union empty-set S2) S2)
  axiom (equal
         (set-union (set-insert x S1) S2)
         (set-insert x (set-union S1 S2)))
end-definition

definition of set-insert is
  axiom (implies
         (in x ((relax no-dup?) S))
         (equal ((relax no-dup?) (set-insert x S))
                ((relax no-dup?) S)))
  axiom (implies
         (not (in x ((relax no-dup?) S)))
         (equal ((relax no-dup?) (set-insert x S))
                (insert x ((relax no-dup?) S))))
end-definition

definition of set-empty? is
  axiom (equal (set-empty? S) (empty? ((relax no-dup?) S)))
end-definition
```

```
definition of set-in is
  axiom (equal
          (set-in x S)
          (in x ((relax no-dup?) S)))
end-definition

end-spec
```

□

12.1.1 Definitional Extensions

A morphism $m: B \rightarrow C$ is a definitional extension if (1) m is injective; (2) each sort and operation of C outside the image of m C is defined in terms of sorts and operations within the image m, and (3) every axiom of C outside the image of m is provable from the definitions plus the translations of the axioms of B along m.

If $m: B \rightarrow C$ is a definitional extension, we also say that C is a definitional extension of B. If C is a definitional extension of B, then C is consistent if and only if B is consistent.

In the present implementation, the test whether a morphism m is a definitional extension checks properties (1) and (2) above,[18] but will fail if additional theorems are present (as allowed by (3)).

A definitional extension will graphically be shown as

$$B \dashrightarrow C$$

Pushouts (base form of colimit) "preserve" definitional extensions. Consider the following pushout (colimit) diagram:

$$\begin{array}{ccc} B & \overset{m}{\dashrightarrow} & C \\ {\scriptstyle n}\downarrow & & \downarrow{\scriptstyle n'} \\ D & \underset{m'}{\overset{\text{po}}{\dashrightarrow}} & E \end{array}$$

If m is a definitional extension, then the corresponding (cocone) morphism m' is also a definitional extension. Furthermore, definitional extensions are closed under (sequential) composition. Both properties, preservation by pushouts and closure under composition, are needed for sequential composition of interpretations.

[18]The system only makes a syntactic check on operation definitions.

EXAMPLE 12.2. The target morphism in example 12.1 is an import morphism which is a definitional extension (as is the case with all interpretations). Here is another example of a definitional extension.

```
spec SET is
  import BASIC-SET

  op delete : E, Set -> Set

  definition of delete is
    axiom (equal (delete x empty-set) empty-set)
    axiom (equal (delete x (insert x S)) S)
    axiom (implies (not (equal x1 x2))
                   (equal (delete x1 (insert x2 S))
                          (insert x2 (delete x1 S))))
  end-definition

  definition set-equal-def of equal is
    axiom (iff (equal S T)
               (fa (x) (iff (in x S) (in x T))))
  end-definition

end-spec
```

□

12.2 Interpretations as Model Constructions

12.2.1 Semantics of Morphisms

The semantics of a specification A is a class of models, $\mathbf{Mod}[A]$. The semantics of a morphism $\sigma: A \rightarrow B$ is a mapping $\_|_\sigma$:$\mathbf{Mod}[B] \rightarrow \mathbf{Mod}[A]$, called the σ-reduct of $\mathbf{Mod}[B]$ to $\mathbf{Mod}[A]$. Note that the morphism σ and its σ-reduct are mappings in opposite directions.

Each model m_B of a spec B is an assignment of semantic objects, (e.g. sets) and operations (e.g. functions on sets) to the sorts and operations of B. The σ-reduct of a model m_B is then defined as follows: for any sort s_A in spec A,

$$m_B|_\sigma(s_A) = m_B(\sigma(s_A))$$

and likewise for each operation f_A in spec A,

$$m_B|_\sigma(f_A) = m_B(\sigma(f_A))$$

The σ-reduct indeed "reduces" m_B to a model m_A of A by picking out those components present in A and by "forgetting" all other components of B.

Note that in general the σ-reduct transforms each model of B into a model of A, but not every model of A can necessarily be generated from a model of B. Thus, in general the image of the σ-reduct is a proper subclass of $\mathbf{Mod}[A]$. In this sense refinement amounts to a restriction of the class of models considered.

12.2.2 Semantics of Definitional Extensions

For an arbitrary spec morphism $\sigma: A \rightarrow B$ the σ-reduct $\_|_\sigma$:$\mathbf{Mod}[B] \rightarrow \mathbf{Mod}[A]$ is in general neither injective (one-to-one) nor surjective (onto). If σ is a definitional extension then $\_|_\sigma$ is bijective, i.e. both injective and surjective, and therefore has an inverse $\_+_\sigma$:$\mathbf{Mod}[A] \rightarrow \mathbf{Mod}[B]$, called σ-expansion with

$$(m_B|_\sigma)+_\sigma = m_B$$

$$(m_A+_\sigma)|_\sigma = m_A$$

for all models m_A of A and all models m_B of B.

12.2.3 Semantics of Interpretations

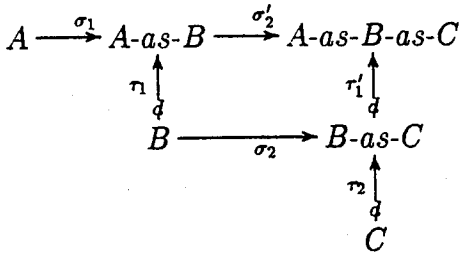
Given the semantics of morphisms and definitional extensions above, we define the semantics of an interpretation $\pi: A \Rightarrow B$. If σ is the source morphism and τ the target morphism (a definitional extension), then the model construction $\_\dagger_\pi$:$\mathbf{Mod}[B] \rightarrow \mathbf{Mod}[A]$ corresponding to π is the composition of the τ-expansion followed by the σ-reduction, i.e.

$$\_\dagger_\pi = \_|_\sigma \circ \_+_\tau$$

In words: Given a model m_B of B, we construct a model of A by first expanding m_B along τ and then reducing the result along σ.

12.3 Sequential (Vertical) Composition of Interpretations

Given two interpretations $\pi_1: A \Rightarrow B$ and $\pi_2: B \Rightarrow C$, the sequential composition $\pi = \pi_1; \pi_2$ of π_1 and π_2 is obtained as follows (see the diagram below).



Let σ_i and τ_i be the source and target morphism of π_i, respectively for $i = 1, 2$. The pushout of τ_1 and σ_2 yields two morphisms τ_1' and σ_2'. Since pushouts preserve definitional extensions (see 12.1.1), τ_1' is a definitional extension. Definitional extensions are closed under (sequential) composition. Therefore we can define the composition of π_1 and π_2 as the interpretation with source morphism $\sigma_1; \sigma_2'$ and target morphism $\tau_1' \circ \tau_2$[19].

Sequential composition of interpretations facilitates incremental, layered refinement.

EXAMPLE 12.3. As an example of the sequential composition of two interpretations, consider the interpretation of sets as bags in Example 12.1 together with the following interpretation which refines bags to sequences.

```
interpretation BAG-TO-SEQ-QUOTIENT : BASIC-BAG => SEQ is
  mediator BAG-AS-SEQ-QUOTIENT
  domain-to-mediator {Bag       -> Bag-as-Seq,
                      empty-bag -> empty-bag,
                      singleton -> singleton-bag,
                      bag-union -> bag-union,
                      insert    -> bag-insert,
                      empty?    -> bag-empty?,
                      in        -> bag-in}
  codomain-to-mediator import-morphism

spec BAG-AS-SEQ-QUOTIENT is
  import SEQ

  sort Bag-as-Seq
  sort-axiom Bag-as-Seq = Seq / bag-equal

  ...
```

[19] "$\sigma; \tau$" is sequential composition in diagrammatic order: read "σ then τ"
"$\tau \circ \sigma$" is sequential composition in application order: read "τ after σ"

```
 end-spec
```

These two interpretations can be composed to yield an interpretation from sets to sequences; here are the relevant constructions.

```
diagram SET-AS-BAG-AS-SEQ-DIAGRAM is
  nodes BASIC-BAG, SET-AS-BAG-SUBSORT, BAG-AS-SEQ-QUOTIENT
  arcs  BASIC-BAG -> SET-AS-BAG-SUBSORT : import-morphism,
        BASIC-BAG -> BAG-AS-SEQ-QUOTIENT :
        {Bag       -> Bag-as-Seq,
         empty-bag -> empty-bag,
         singleton -> singleton-bag,
         bag-union -> bag-union,
         insert    -> bag-insert,
         empty?    -> bag-empty?,
         in        -> bag-in}
end-diagram

spec SET-AS-BAG-AS-SEQ is
  colimit of SET-AS-BAG-AS-SEQ-DIAGRAM

interpretation SET-TO-SUBSORT-OF-SEQ-QUOTIENT : BASIC-SET => SEQ is
  mediator SET-AS-BAG-AS-SEQ
  domain-to-mediator {Set       -> Set-as-Bag,
                      empty-set -> empty-set,
                      singleton -> singleton-set,
                      union     -> set-union,
                      insert    -> set-insert,
                      empty?    -> set-empty?,
                      in        -> set-in}
  codomain-to-mediator {in        -> BAG-AS-SEQ-QUOTIENT.in,
                        empty?    -> BAG-AS-SEQ-QUOTIENT.empty?,
                        singleton -> BAG-AS-SEQ-QUOTIENT.singleton}
```

□

12.4 Parallel (Horizontal) Composition of Interpretations

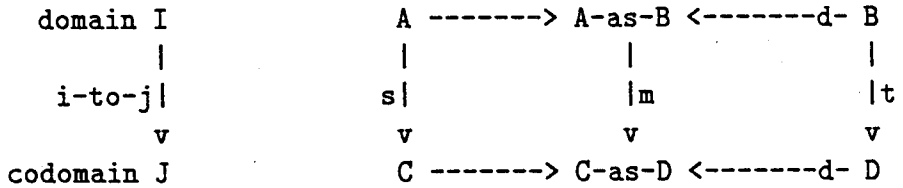
Analogous to the colimit operation on diagrams over specs and spec morphisms, we define a colimit operation on diagrams of *interpretations* and *interpretation morphisms*. I.e., the nodes of the diagram are interpretations and the arcs are interpretation morphisms (morphisms between interpretations).

12.4.1 Interpretation Morphisms

A prototypical definition of a named interpretation morphism takes the form

```
ip-scheme-morphism i-to-j: I -> J is
            domain-sm s
          mediator-sm m
          codomain-sm t
```

A depiction of this structure is shown below.



An interpretation morphism has a domain and codomain, both interpretations. An interpretation morphism is defined by a triple of morphisms between the domains, mediators, and codomains of its source and target interpretation such that the diagram above commutes.[20]

12.4.2 Interpretation Colimits[21]

In figure 13, the interpretation for $D0$ is the colimit (pushout) of the interpretations for $B0$ and $C0$ glued on the interpretation for $A0$. This view can be seen by transforming the diagram in figure 13 into the diagram in figure 14 which is a diagram whose nodes are (labeled with) interpretations and whose arcs are (labeled with) interpretation morphisms.

To compute the colimit interpretation (from $D0$ to $D1$ in the diagram in figure 14), we transpose the diagram of interpretations into a diagram whose nodes are specification diagrams of the same shape and whose arcs are diagram morphisms.[22] (figure 15).

[20]To say that a diagram commutes means that for any two nodes n_1 and n_2, and any two paths p_1 and p_2 between the nodes n_1 and n_2, the functions obtained by composing the functions along the arcs of the paths p_1 and p_2 are equal.

[21]This section presupposes some familiarity with category theory.

[22]I.e. natural transformations between the diagrams (viewed as functors from the common shape category into the category of specs and spec morphism).

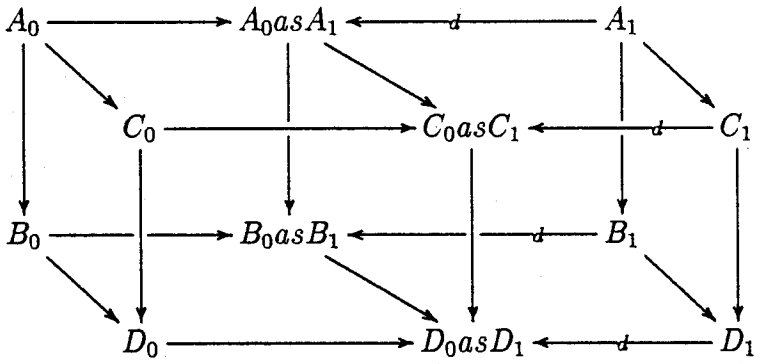


Figure 13: Interpretation Colimit—Spec Diagram

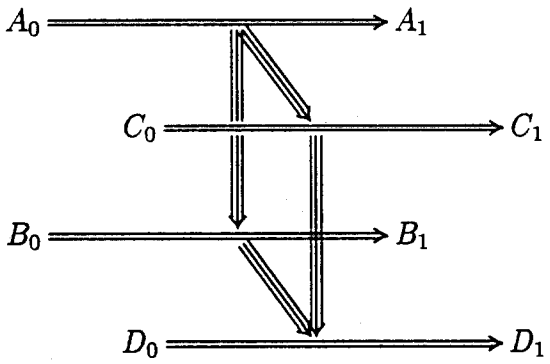


Figure 14: Interpretation Colimit—Interpretation Diagram

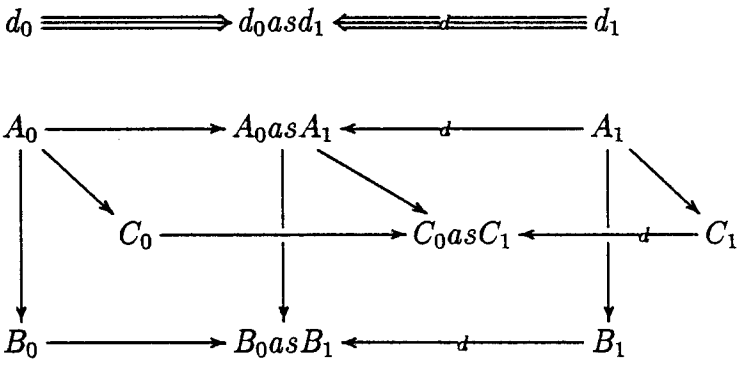


Figure 15: Interpretation Colimit—Transposed Diagram

In this transposed diagram, there are three nodes labeled with spec diagrams and two arcs labeled with diagram morphisms. That is, $d0$ is a spec diagram with nodes $A0$, $B0$, $C0$, etc. If $d1$ has a colimit, then we can compute an interpretation for $D0$ by taking the colimit of each of the diagrams $d0$, $d0$-*as*-$d1$, and $d1$ to yield the specs $D0$, $D0$-*as*-$D1$, and $D1$ (as in the diagram in figure 13 above). These three specs, together the with witness arrows for the universality of $D0$ and $D1$ with respect to $D0$-*as*-$D1$ form an interpretation from $D0$ to $D1$. We use the following facts about the category of spec diagrams with common shape:

1. If a diagram $d1$ has a colimit, and $t1: d1 \Rightarrow d2$ is a diagram morphism with all pieces definitional extensions, then $d2$ has a colimit and the unique arrow between the colimit of $d1$ and the colimit of $d2$ is a definitional extension.

2. If a diagram $d2$ has a colimit, and $t0: d0 \Rightarrow d2$ is a diagram morphism, then $d0$ has a colimit.

12.4.3 Interpretation Schemes and Morphisms

For practical purposes it is not sufficient to cover interpretations by pieces that are themselves interpretations. We therefore introduce *interpretation schemes* as a suitable generalization of interpretations. Interpretation schemes can be be thought of as interpretations with holes, or as interpretation specifications.

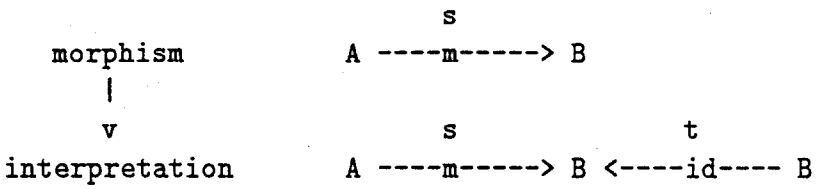
An interpretation scheme has the same structure as an interpretation, but the target morphism can be an arbitrary morphism; it need not be a definitional extension. *Interpretation scheme morphisms (ip-scheme morphisms)* are as defined in 12.4.1 with ip-schemes as domains and targets.

Colimits of ip-scheme diagrams exist provided the underlying spec diagrams have colimits. In general, the colimit of an ip-scheme diagram is an interpretation scheme, not an interpretation.

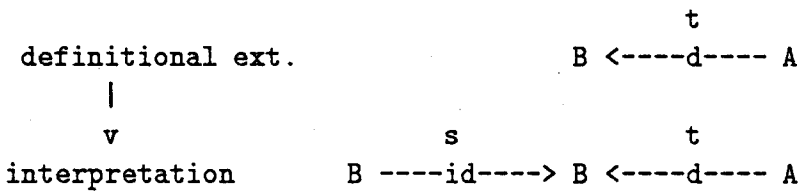
Note that for arbitrary source spec A and target spec B there is no most general interpretation but there is a most general interpretation scheme, namely the one consisting of the inclusion morphisms from A and B into the coproduct (disjoint union) of A and B.

12.5 Lifting Spec Operations to Interpretation Operations

Any specification morphism `m` can be "lifted" to an interpretation by taking `m` as the source morphism and the identity on the codomain of `m` as the target morphism. (Note that the identity is a definitional extension.)

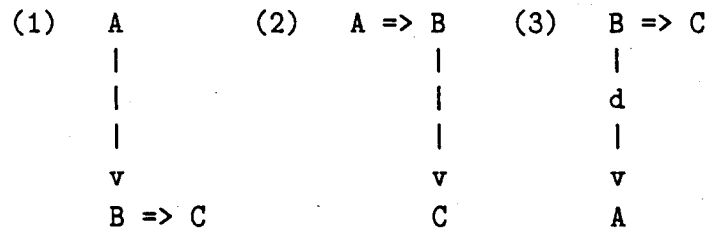


A definitional extension can be lifted to an interpretation by taking the identity on its domain as the source morphism:



Translation of a spec A results in a spec B and a spec isomorphism τ from A to B. Isomorphisms are trivial definitional extensions. Hence, by the above on definitional extensions, translations can be lifted to interpretations.

Sequential composition of interpretations then yields rules for propagating interpretations along morphisms in the following situations:



In each case, sequential composition produces an interpretation from A to C.

13 Putting Code Fragments Together

When specifications are sufficiently refined, they can be converted into programs which realize them. This involves a switching of logics. We use the theory of logic morphisms described in [10]. We will confine our attention to entailment systems and their morphisms, rather than logics (which include models and institutions). Entailment systems are sufficient for the purpose of code generation.

13.1 Entailment Systems and their Morphisms

An entailment system is a 3-tuple $\langle \mathbf{Sig}, sen, \vdash \rangle$ consisting of a category **Sig** of signatures and signature morphisms, a functor sen: **Sig** $\rightarrow$ **Set** which assigns to each signature a set of "sentences", and a function $\vdash$ which associates to each signature an entailment relation (satisfying the expected axioms).

To map one entailment system into another, we map the syntax (i.e., signatures and sentences) while preserving entailment. Preservation of entailment represents the relevant correctness criterion for translating specifications from one logic to another. Note that this is similar to the correctness criterion for refinement within a single logic.

13.2 Translating from Slang to Lisp

The specification language used in SPECWARE is called SLANG. We distinguish SLANG because SPECWARE may have multiple back-ends, Lisp, C, Ada, etc., each with its own logic.

We consider a sub-logic of SLANG, called the *abstract target language* (for LISP); there is one sub-logic for each language into which SLANG specifications can be translated. We will denote this sub-logic by SLANG^{--}. The sub-logic SLANG^{--} is defined by starting with a set of basic specifications, such as integers, sequences, etc., which have direct realizations in the target language. All specifications which can be constructed from the base specifications, with the following restrictions, are then included in the sub-logic:

- for colimit specifications, only injective morphisms are allowed in the diagram;[23]
- all definitions must be constructive, i.e., they must either be explicit definitions (e.g., `(equal (square x) (times x x))`), or, if they are recursive, they must be given as conditional equations using a constructor set.

The goal of the refinement process is to arrive at a sufficiently detailed specification which satisfies the restrictions above.

The sub-logic SLANG^{--} will be translated into a functional subset of LISP. To facilitate this translation, we couch this subset as an entailment system, denoted LISP^{--}. The signatures of this entailment system are finite sets of untyped operations and the sentences are function definitions of the form

```
(defun f (x)
  (cond ((p x) (g x))
   ...))
```

and generated conditional equations of the form

```
(if (p x) (equal (f x) (g x))).
```

The entailment relation is that of rewriting, since theories in LISP^{--} can be viewed as conditional-equational theories over the simply-typed λ-calculus.

In Figure 16, we show a fragment of an entailment system morphism from SLANG^{--} to LISP^{--}. Note, in particular, the translations from and to empty specifications.

[23] For colimit specifications which can be construed as "instantiations" of a "generic" specification, the morphisms from the formal to the actual may be non-injective.

The set of sentences in the SLANG specification INT translates to the empty set; this is because integers are primitive in LISP. Similarly, the empty SLANG specification translates to a non-empty LISP specification; this is because some built-in operations of SLANG are not primitive in LISP.

13.2.1 Translating Constructed Sorts

There are numerous details in entailment system morphisms such as that from SLANG^{--} to LISP^{--}. We will briefly consider the translation of constructed sorts. Subsorts can be handled by representing elements of a subsort by the corresponding elements of the supersort. Similarly, quotient sorts can be handled by representing their elements by the elements of the base sort. Sentences have to be translated consistently with such representation choices: e.g., injections associated with subsorts (`(relax p)`) and the surjections associated with quotient sorts (`(quotient e)`) must be dropped. Also, the equality on a quotient sort must be replaced by the equivalence relation defining the quotient sort.

In Figure 17, we show the representation of coproduct sorts by variant records. This translation exploits the generality of entailment system morphisms: a signature is mapped into a theory.

13.3 Translation of Colimits: Putting Code Fragments Together

If an entailment system morphism is defined in such a way that it is co-continuous, i.e., colimits are preserved, then we obtain a recursive procedure for translation, which is similar to that of refinement: the code for a specification can be obtained by assembling the code for smaller specifications which cover it.

The entailment system morphism from SLANG^{--} to LISP^{--} briefly described above does preserve colimits because of our restriction to injective morphisms. In general, this is true for most programming languages because they only allow imports, which are inclusion morphisms.

SLANG^{--}	$\longrightarrow$	LISP^{--}
$\emptyset$	$\longmapsto$	spec SLANG-BASE is ops implies, iff (defun implies (x y) (or (not x) y)) (defun iff (x y) (or (and x y) (and (not x) (not y)))) end-spec
INT	$\longmapsto$	SLANG-BASE
spec FOO is import INT op abs : Int -> Int definition of abs is axiom (implies (ge x zero) (equal (abs x) x)) axiom (implies (lt x zero) (equal (abs x) (minus zero x))) end-definition end-spec	$\longmapsto$	spec FOO' is import SLANG-BASE op abs (defun abs (x) (cond ((>= x 0) x) ((< x 0) (- 0 x)))) end-spec

Figure 16: Fragment of entailment system morphism from SLANG^{--} to LISP^{--}

```
spec STACK is                          ⟼  spec STACK' is
import INT                                 import SLANG-BASE
...                                        op size, E-Stack?, NE-Stack?
sort-axiom                                 (defun E-Stack? (s)
  Stack = E-Stack + NE-Stack                 (= (car s) 1))
...                                        ...
op size  : Stack -> Int                    (defun size (s)
definition of size is                        (cond
  axiom                                        ((E-Stack? s) 0)
   (equal (size ((embed 1) s))                 ((NE-Stack? s)
          zero)                                 (1+ (size (pop (cdr s)))))
  axiom                                       ))
   (equal (size ((embed 2) s))               end-definition
          (succ (size (pop s))))             end-spec
end-definition
end-spec
```

Figure 17: The representation of coproduct sorts as variant records

Part IV

Assessment

14 Conclusions

14.1 Focus and Results

The primary focus of this effort was on the basic concepts and operations that enable a scalable technology for system synthesis. Synthesis technology is inherently complex; the primitive concepts must therefore be both simple and general so that tractable implementations of powerful, but usable tools can be constructed.

SPECWARE is intended as a system for composing software: specifications, refinements, and code. Composition is crucial both for complexity management and reuse. SPECWARE therefore provides operations for composing specifications from pieces (smaller specifications), for constructing refinements of specification from refinements of the pieces, and for constructing system code from code modules.

The presence of both parallel and sequential refinement composition allows system development from parts through several architectural layers, from composition of an initial specification to generation of a system of code modules, e.g. in Ada, linked by a precise, richly structured design record. Such design records enable accurate requirements traceability and controlled system evolution as well as high degree of reuse.

14.2 Beyond DTRE, KIDS, and REACTO

At present, not all of the capabilities for which the earlier Kestrel prototypes present a proof of concept have been realized in SPECWARE. SPECWARE, however, does present a carefully constructed, coherent framework for the integration of these capabilities, often in more general yet simpler form.

DTRE The data type refinement capabilities of DTRE have been fully recreated in a generalized form in SPECWARE. The underlying theory has been significantly refined. The notions of parallel composition of interpretations, interpretation schemas, and parameterized interpretations are original results of this effort. It would useful to revisit and incorporate the automated data structure selection ideas explored in DSS.

KIDS The theories in KIDS are theories about the REFINE language; in SPECWARE, the specification logic is completely independent of any target language. While the development process in KIDS includes data refinement in principal, such a capability does not exist in the implemented KIDS system. The specification framework of SPECWARE is richer than that of KIDS, and provides a much richer framework for the representation of taxonomic design knowledge. SPECWARE is now at a stage at which, after careful analysis, adding KIDS' automated algorithm design and optimization capabilities will lead to a cleaner and more general re-incarnation of the KIDS technology. In particular, first attempts to perform algorithm design in the presence of data type refinement have led to the discovery of subtle interactions.

REACTO There are several ways of adding a state machine formalism to SPECWARE. One of the most attractive approaches is to specify state machines as SLANG theories. This creates the opportunity to use the refinement machinery of SPECWARE to derive different representations for the finite-state control. This degree of system design flexibility is not present in the original prototype.

We believe that SPECWARE represents a significant advance toward scalable technology for system synthesis.

References

[1] BALZER, R., CHEATHAM, T. E., AND GREEN, C. Software technology in the 1990's: Using a new paradigm. *IEEE Computer 16*, 11 (November 1983), 39–45.

[2] BLAINE, L. Semi-automatic data structure selection. Tech. rep., Kestrel Institute, August 1990. Kestrel Institute Internal Report.

[3] BLAINE, L., AND GOLDBERG, A. Verifiably correct data type refinement. Tech. rep., Kestrel Institute, November 1990.

[4] BLAINE, L., AND GOLDBERG, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[5] GILHAM, L.-M., GOLDBERG, A., AND WANG, T. C. Toward reliable reactive systems. In *Proceedings of the 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, May 1989).

[6] GOLDBERG, A. Reusing software developments. In *Proceedings of the ACM SIGSOFT 4th Symposium on Software Development Environments* (Irvine, CA, December 6–8, 1990), pp. 107–119.

[7] HAREL, D. Statecharts: A visual approach to complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.

[8] JÜLLIG, R. Applying formal software synthesis. *IEEE Software 10*, 3 (May 1993), 11–22. (also Technical Report KES.U.93.1, Kestrel Institute, May 1993).

[9] JÜLLIG, R., AND SRINIVAS, Y. V. Diagrams for software synthesis. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference* (Chicago, IL, September 20–23, 1993), IEEE Computer Society Press, pp. 10–19.

[10] MESEGUER, J. General logics. In *Logic Colloquium'87*, H.-D. Ebbinghaus et al., Eds. North-Holland, 1989, pp. 275–329.

[11] SMITH, D. R. Derived preconditions and their use in program synthesis, LNCS 138. In *Sixth Conference on Automated Deduction* (Berlin, 1982), D. W. Loveland, Ed., Springer-Verlag, pp. 172–193.

[12] SMITH, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[13] SRINIVAS, Y. V., AND JÜLLIG, R. Specware:^{tm} formal support for composing software. Tech. Rep. KES.U.94.5, Kestrel Institute, December 1994. To appear in Proceedings of the Conference on Mathematics of Program Construction, Kloster Irsee, Germany, July 1995.

[14] WANG, T. C. Shd-prover at University of Texas at Austin. In *Proceedings of 8th Conference on Automated Deduction*, J. H. Siekmann, Ed. Springer-Verlag, Berlin, 1986, pp. 707–708. Lecture Notes in Computer Science, Vol. 230.

[15] WANG, T. C., AND BLEDSOE, W. W. Hierarchical deduction. *Journal of Automated Reasoning 3*, 1 (March 1987), 35–77.

[16] ZIMMERMANN, P., AND ZIMMERMANN, W. The automatic complexity analysis of divide-and-conquer algorithms. Tech. Rep. INRIA-RR-1134, INRIA, December 1989.

Part V

Appendices

A SLANG Syntax

In this appendix, we define the syntax of SLANG. Readers familiar with REFINE may also wish to consult the file `<slang-top-level>/core4/code/language/spec-grammar.re`. As with the rest of the language manual, the grammar is divided into a core part and an extension for refinements.

A.1 Notation

To describe the syntax, we use BNF augmented with regular expression constructs. Non-terminals are enclosed in angle brackets, "⟨...⟩". Terminals are indicated by `typewriter` font. Syntax alternatives are separated by "|". Parentheses, "(...)", are used for grouping, e.g., for inline alternatives. Optional entities are enclosed in square brackets, "[...]". A "*" after a syntactic element indicates zero or more repetitions of that element; a "+" indicates one or more repetitions.

A.2 Core Slang Grammar

A.2.1 Top-Level Objects

The top-level objects of CORE SLANG are specifications, morphisms, and diagrams. Each such object class appears twice in the grammar, once with the prefix "global-" and once with the prefix "local-". Global objects must be named; local objects must not be named. Global objects can only appear at the top-level; local objects can appear within other expressions.

⟨top-level-slang-object⟩ ⟶
  ⟨global-spec⟩ | ⟨global-signature-morphism⟩ | ⟨global-sm-diagram⟩

A.2.2 Specifications

⟨global-spec⟩ ⟶
  `spec` ⟨symbol⟩ (`is` | `=`)
    [⟨import-declaration⟩]
    ⟨development-element⟩*
  `end-spec`
  |
  `spec` ⟨symbol⟩ (`is` | `=`) ⟨spec-operation⟩

A.2.3 Import Declarations

⟨import-declaration⟩ ⟶
  `import` ⟨spec-term⟩ (`,` ⟨spec-term⟩)* |
  `import` ⟨diagram-term⟩

A.2.4 Specification Elements

⟨development-element⟩ ⟶
  ⟨sort-declaration⟩ | ⟨sort-axiom⟩ | ⟨op-declaration⟩ |
  ⟨constructor-set⟩ | ⟨theorem⟩ | ⟨definition⟩

⟨sort-declaration⟩ ⟶ (`sorts` | `sort`) ⟨spec-sort⟩ (`,` ⟨spec-sort⟩)*

⟨spec-sort⟩ ⟶ ⟨symbol⟩

⟨sort-axiom⟩ ⟶ `sort-axiom` ⟨spec-sort-ref⟩ `=` ⟨spec-sort-term⟩

⟨op-declaration⟩ ⟶ (`op` | `const`) ⟨symbol⟩ `:` ⟨spec-sort-term⟩

⟨constructor-set⟩ ⟶
  `constructors` `{` ⟨spec-op-ref⟩ (`,` ⟨spec-op-ref⟩)* `}` `construct` ⟨spec-sort-term⟩

⟨theorem⟩ ⟶ (`axiom` | `theorem`) [⟨symbol⟩ (`is` | `=`)] ⟨spec-op-term⟩

⟨definition⟩ ⟶
  `definition` [⟨symbol⟩ [`of` ⟨spec-op-ref⟩] (`is` | `=`)]
    ⟨definition-clause⟩$^{+}$
  `end-definition`

⟨definition-clause⟩ ⟶ ⟨theorem⟩

A.2.5 Sort Terms

⟨spec-sort-term⟩ ⟶
  ⟨spec-sort-ref⟩ | ⟨spec-sort-function⟩ | ⟨spec-sort-subsort⟩ |
  ⟨spec-sort-quotient⟩ | ⟨spec-sort-coproduct⟩ | ⟨spec-sort-product⟩

⟨spec-sort-ref⟩ ⟶ ⟨qualified-name⟩

⟨spec-sort-function⟩ ⟶ [⟨spec-sort-term⟩] `->` ⟨spec-sort-term⟩

⟨spec-sort-subsort⟩ ⟶ ⟨spec-sort-term⟩ `|` ⟨spec-op-term⟩

⟨spec-sort-quotient⟩ ⟶ ⟨spec-sort-term⟩ `/` ⟨spec-op-term⟩

⟨spec-sort-coproduct⟩ ⟶ `[ ]` | ⟨spec-sort-term⟩ (`+` ⟨spec-sort-term⟩)$^+$

⟨spec-sort-product⟩ ⟶ `( )` | ⟨spec-sort-term⟩ (`,` ⟨spec-sort-term⟩)$^+$

A.2.5.1 Precedence and associativity for sort terms. The different operators for constructing sort terms are listed below in the order of increasing precedence. Precedence can be overridden with parentheses.

precedence for ⟨spec-sort-term⟩
  brackets `(` matching `)`
  same-level `->` associativity right
  same-level `,` `+` associativity none
  same-level `|` `/`

A.2.6 Terms and Formulas

⟨spec-op-term⟩ ⟶
  ⟨spec-op-ref⟩ | ⟨spec-op-operation⟩ | ⟨spec-op-binding-operation⟩ | ⟨spec-op-product⟩

⟨spec-op-ref⟩ ⟶ ⟨qualified-name⟩ [`:` ⟨spec-sort-term⟩]

⟨spec-op-operation⟩ ⟶
  `(` ⟨spec-op-term⟩ ⟨spec-op-term⟩* `)` |
  `(` (`project` | `embed`) ⟨positive-integer⟩ `)`

⟨spec-op-binding-operation⟩ ⟶
  (⟨spec-op-binding-rator⟩ (⟨bound-var⟩*) ⟨spec-op-term⟩)

⟨spec-op-binding-rator⟩ ⟶ (`fa` | `ex` | `lambda`)

⟨bound-var⟩ ⟶ ⟨symbol⟩ [: ⟨spec-sort-term⟩]

⟨spec-op-product⟩ ⟶ `<` ⟨spec-op-term⟩* `>`

A.2.7 Specification Terms

Specification terms are terms which denote specifications. Generally, terms are of three kinds: references to named objects, operations, and explicit terms for anonymous (or local) objects.

⟨spec-term⟩ ⟶ ⟨spec-ref⟩ | ⟨local-spec⟩ | ⟨spec-operation⟩

⟨spec-ref⟩ ⟶ ⟨symbol⟩

⟨local-spec⟩ ⟶
  `spec`
    [⟨import-declaration⟩]
    ⟨development-element⟩*
  `end-spec`

⟨spec-operation⟩ ⟶ ⟨spec-translation⟩ | ⟨spec-colimit⟩

⟨spec-translation⟩ ⟶ `translate` ⟨spec-term⟩ `by` { [⟨sm-rules⟩] }

⟨spec-colimit⟩ ⟶ `colimit of` ⟨diagram-term⟩

A.2.8 Specification Morphisms

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

⟨sm-rules⟩ ⟶ ⟨sm-rule⟩ (, ⟨sm-rule⟩)*

⟨sm-rule⟩ ⟶ ⟨sort-or-op-ref⟩ `->` ⟨sort-or-op-ref⟩

⟨sort-or-op-ref⟩ ⟶ ⟨qualified-name⟩ | (⟨qualified-name⟩ : ⟨spec-sort-term⟩)

A.2.9 Specification Morphism Terms

⟨sm-term⟩ ⟶ ⟨sm-ref⟩ | ⟨local-signature-morphism⟩ | ⟨sm-operation⟩

⟨sm-ref⟩ ⟶ ⟨symbol⟩

⟨local-signature-morphism⟩ ⟶
`[morphism` ⟨spec-term⟩ `->` ⟨spec-term⟩`]` `{` `[`⟨sm-rules⟩`]` `}`

⟨sm-operation⟩ ⟶
`identity-morphism` | `translation-morphism` | `import-morphism` |
`cocone-morphism from` ⟨symbol⟩

A.2.10 Diagrams

⟨global-sm-diagram⟩ ⟶
`diagram` ⟨symbol⟩ `(is` | `=)`
`[nodes` ⟨sm-node⟩ `(,` ⟨sm-node⟩`)*]`
`[arcs` ⟨sm-arc⟩ `(,` ⟨sm-arc⟩`)*]`
`end-diagram`

A.2.11 Diagram Terms

⟨diagram-term⟩ ⟶ ⟨diagram-ref⟩ | ⟨local-sm-diagram⟩

⟨diagram-ref⟩ ⟶ ⟨symbol⟩

⟨local-sm-diagram⟩ ⟶
`diagram`
`[nodes` ⟨sm-node⟩ `(,` ⟨sm-node⟩`)*]`
`[arcs` ⟨sm-arc⟩ `(,` ⟨sm-arc⟩`)*]`
`end-diagram`

A.2.12 Diagram Elements

⟨sm-node⟩ ⟶ `[`⟨symbol⟩ `:]` ⟨spec-term⟩

⟨sm-arc⟩ ⟶ `[`⟨symbol⟩ `:]` ⟨sm-node-ref⟩ `->` ⟨sm-node-ref⟩ `:` ⟨sm-term⟩

⟨sm-node-ref⟩ ⟶ ⟨symbol⟩

A.2.13 Qualified Names

⟨qualified-name⟩ ⟶ (⟨node-name⟩ .)* ⟨sort-or-op-name⟩

⟨node-name⟩ ⟶ ⟨symbol⟩

⟨sort-or-op-name⟩ ⟶ ⟨symbol⟩

A.2.14 Simple Names

⟨symbol⟩ ⟶ ⟨symbol-start-char⟩ ⟨symbol-continue-char⟩*

⟨symbol-start-char⟩ ∈
`*abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`

⟨symbol-continue-char⟩ ∈
`-*abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890?!`

A.2.15 Comments

The character "`%`" indicates the start of a comment; everything which follows until the end of the line is ignored. Larger pieces of text can be commented out by enclosing them in "`#||...||#`"; these characters function as brackets and can be nested.

A.3 Refinement Constructs in Slang

In addition to specifications, morphisms, and diagrams, there are two additional top-level constructs in SLANG: interpretation schemes and interpretation scheme morphisms.

⟨top-level-slang-object⟩ ⟶ ⟨global-ip-scheme⟩ | ⟨global-ips-morphism⟩

A.3.1 Interpretations and Interpretation Schemes

⟨global-ip-scheme⟩ ⟶
`(interpretation | ip-scheme)` ⟨symbol⟩ : ⟨spec-term⟩ `=>` ⟨spec-term⟩ `(is | =)`
`mediator` ⟨spec-term⟩
`(dom-to-med | domain-to-mediator)` ⟨sm-term⟩
`(cod-to-med | codomain-to-mediator)` ⟨sm-term⟩

A.3.2 Interpretation (Scheme) Terms

⟨ips-term⟩ ⟶ ⟨ips-ref⟩ | ⟨local-ip-scheme⟩

⟨ips-ref⟩ ⟶ ⟨symbol⟩

⟨local-ip-scheme⟩ ⟶
(`interpretation` | `ip-scheme`) [⟨spec-term⟩ `=>` ⟨spec-term⟩]
`mediator` ⟨spec-term⟩
(`dom-to-med` | `domain-to-mediator`) ⟨sm-term⟩
(`cod-to-med` | `codomain-to-mediator`) ⟨sm-term⟩

A.3.3 Interpretation (Scheme) Morphisms

⟨global-ips-morphism⟩ ⟶
`ip-scheme-morphism` ⟨symbol⟩ : ⟨ips-term⟩ `->` ⟨ips-term⟩ (`is` | `=`)
`domain-sm` ⟨sm-term⟩
`mediator-sm` ⟨sm-term⟩
`codomain-sm` ⟨sm-term⟩

A.3.4 Interpretation (Scheme) Morphism Terms

⟨ipsm-term⟩ ⟶ ⟨ipsm-ref⟩ | ⟨local-ips-morphism⟩

⟨ipsm-ref⟩ ⟶ ⟨symbol⟩

⟨local-ips-morphism⟩ ⟶
`ip-scheme-morphism`
`domain-sm` ⟨sm-term⟩
`mediator-sm` ⟨sm-term⟩
`codomain-sm` ⟨sm-term⟩

Index

This index contains entries for the reserved words and nonterminals in the BNF description of SLANG in addition to entries for the main body of the text. The entries for the nonterminals and reserved words precede the standard entries.

There are three levels of indexing: main entry; subentry; and subsubentry. Also, "-" is sometimes used as a surrogate for the main entry.

Rome Laboratory

Customer Satisfaction Survey

RL-TR-_____

Please complete this survey, and mail to RL/IMPS, 26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

_____

_____

Organization Name:_____(Optional)

Organization POC: _____(Optional)

Address:_____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful 1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No_____

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.

*U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20104

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.