J. Symbolic Computation (1993) **15**, 571–606

Constructing Specification Morphisms

Douglas R. Smith

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304 USA

(Received 23 April 1992)

Specification morphisms underlie the refinement of algebraic specifications and provide the logical foundations for algorithm and data structure design. We present four techniques for formally, even mechanically, constructing specification morphisms. The first two techniques, verifying a manually constructed signature morphism and composition of specification morphisms are well-known. The remaining two techniques exploit the axioms of the source specification to help infer the translation of sort and function symbols from the source specification. The third, *unskolemization*, finds the translation of a function symbol by replacing occurrences of it in an axiom by an existentially quantified variable. A constructive proof of the translated axiom yields a witness to the existential that serves as the desired translation of the function symbol. The fourth technique, *connections between specifications*, allows the transfer of structure from one specification morphism to another. The unskolemization and connection techniques arose as abstractions from the algorithm design tactics implemented in the KIDS program transformation system (Smith (1990)). They suggest a more general approach to providing mechanized support for applying design knowledge expressed axiomatically.

1. Introduction

Mathematically-based techniques for software construction will play an increasing, if not critical, role in the future of software engineering. This paper is part of a broader research program to explore a mechanizable model of software development based on algebraic specifications and specification morphisms.

An algebraic specification (or simply a *specification*) defines a language and constrains its possible meanings via axioms and inference rules. Specifications can be used to express many kinds of software-related artifacts, including domain models (Srinivas(1991)), formal requirements (Astesiano and Wirsing (1987), Ehrig and Mahr (1990), Partsch (1990), Sannella and Tarlecki (1985)), programming languages (Broy et al. (1987), Goguen and Winkler (1988), Hoare (1989)), abstract data types (Goguen et al. (1978), Guttag and Horning (1978)), and abstract algorithms (Smith and Lowry (1990)). There has been much work on operations for constructing larger specifications from smaller specifications (Astesiano and Wirsing (1987), Burstall and Goguen (1977), Sannella and Tarlecki (1988)).

A *specification morphism* translates the language of one specification into the language of another specification in a way that preserves theorems. Specification morphisms underlie several aspects of software development, including specification refinement/implementation (Blaine and Goldberg (1991), Partsch (1990), Sannella and Tarlecki (1988), Turski and Maibaum (1987)), algorithm design (Lowry (1987), Smith and Lowry (1990), Veloso (1988)), data structure design (Smith (1992a)), and the binding of parameters in parameterized specifications (Ehrig et al. (1981), Goguen and Winkler (1988)). There has been work on techniques for composing implementations in a way that reflects the structure of the source specification (Astesiano and Wirsing (1987), Sannella and Tarlecki (1988), Turski and Maibaum (1987)); however these composition techniques leave open the problem of constructing simple morphisms.

The problem addressed in this paper is the following. Given a source specification T and a target specification B, construct a specification morphism from T to an extension of B. Most previous work on constructing specification morphisms has adopted a largely verification approach — a language translation is proposed and then the axioms of the source specification are translated and proved in the target specification. In contrast, we seek constructive methods that use the source axioms to help define a language translation such that the source axioms translate to theorems in the target specification.

Four basic techniques for constructing specification morphisms are presented in Sections 3 and 4. The first technique is to compose preexisting (parameterized) specification morphisms. The second technique separates the construction into 'invention' and 'verification' stages. The third is called *unskolemization* and treats the construction of a specification morphism as a constraint satisfaction problem. In particular, the translation of an operator symbol is deduced from an existentially quantified variant of an axiom from the source theory. First-order theorem-proving techniques are used to produce a witness to the existential which then becomes the desired translation. The fourth technique is based on a new concept called *connection between specifications*. A connection between specifications can be thought of as a general set of conditions under which a specification morphism can be constructed in the following way. Suppose that we want to construct a morphism from specification T to (an extension of) specification B. Assuming that we have a morphism from T into some specification A and there exists (or we can construct) a connection from A to B, then there exists a specification morphism from T to B.

The last two techniques, unskolemization and connection between specifications, arose as abstractions from our experience with automated algorithm design in CYPRESS (Smith (1985)) and KIDS (Smith (1990)). In these systems, and others at Kestrel Institute, we have explored the representation of programming knowledge via algebraic specifications. The possibility of machine support for the construction of specification morphisms suggests a system in which users can supply specifications of programming knowledge (algorithms, data structures, and system architectures) and have generic automated tools that support the correct application of this knowledge.

2. Basic Concepts and Notations

As much as possible we adhere to conventional concepts and notation for algebraic specification (Enderton (1972), Goguen et al. (1977), Wirsing (1990)). A *signature* $\Sigma = \langle S, \Omega \rangle$ consists of a set of sort symbols S and a family $\Omega = \langle \Omega_{v,s} \rangle$ of finite disjoint sets indexed by $S^* \times S$, where $\Omega_{v,s}$ is the set of operation symbols of rank $\langle v, s \rangle$. We write $f : v \to s$ to denote $f \in \Omega_{v,s}$ for $v \in S^*,\ s \in S$ when the signature is clear from context. For each sort $s \in S$ we assume that there is an unbounded supply of distinct variables. We write $x : s$ to indicate that variable x has sort s. More generally, if $x = x_1, \ldots, x_n$ is a sequence of variables of sorts $v = v_1, \ldots, v_n$ respectively, then we write $x : v$ to indicate the aggregate sort of variables x.

As far as possible in this paper we treat truth-values as any other sort. Letting *boolean* be the sort symbol for truth values, then $\Omega_{v,boolean}$ is a set of predicate symbols for each $v \in S^*$. The usual logical connectives $\wedge$, $\vee$, $\neg$, $\Longrightarrow$, and $\Longleftrightarrow$ are treated as boolean operations.

For any signature Σ, the Σ-*terms* are defined inductively in the usual way as the well-sorted composition of operator symbols and variables. A Σ-*formula* is a boolean-valued term built from Σ-terms and the quantifiers $\forall$ and $\exists$. A Σ-*sentence* is a closed formula. The generic term *expression* is used to refer to a term, formula, or sentence.

A *signature morphism* $\sigma : \langle S, \Omega \rangle \to \langle S', \Omega' \rangle$ maps S to S' and Ω to Ω' such that the rank of operations are preserved: if $f : v \to s$ in Ω and $v = v_1, \ldots v_n$ then $\sigma(f) : \sigma(v_1) \ldots \sigma(v_n) \to \sigma(s)$ in Ω'. A signature morphism extends in a unique way to a translation of expressions (as a homomorphism between term algebras). For Σ-expression e, let $\sigma(e)$ denote its translation to a Σ'-expression or, when σ is understood, we write $e_{\Sigma'}$.

Let $\langle \mathcal{A}_s \rangle_{s \in S}$ be an S-indexed family of sets. If $v \in S^*$ where $v = v_1 \ldots v_n$ then $\mathcal{A}^v$ denotes the product $\mathcal{A}_{v_1} \times \ldots \times \mathcal{A}_{v_n}$. Letting ϵ denote the empty string, $\mathcal{A}^\epsilon$ denotes the set consisting of the 0-tuple, $\{\langle\rangle\}$. Let $\langle h_s \rangle_{s \in S}$ be an S-indexed family of operator symbols or functions, and $v \in S^*$, then h^v denotes the product $h_{v_1} \times \ldots \times h_{v_n}$. h^ϵ denotes the unique function on $\mathcal{A}^\epsilon$. Let $\Sigma = \langle S, \Omega \rangle$ be a signature. A Σ-*structure* $\mathcal{A}$ consists of an S-indexed collection of sets $\langle \mathcal{A}_s \rangle_{s \in S}$ and for each operator $f : v \to s$ a function

$$f_{\mathcal{A}} : \mathcal{A}^v \to \mathcal{A}_s.$$

A Σ-*homomorphism* from Σ-*structure* $\mathcal{A}$ to Σ-*structure* $\mathcal{B}$ is an S-indexed collection of functions $\langle h_s : \mathcal{A}_s \to \mathcal{B}_s \rangle_{s \in S}$ such that for any $a \in \mathcal{A}^v$,

$$f_{\mathcal{A}}(a) = a' \implies f_{\mathcal{B}}(h^v(a)) = h_s(a').$$

A *specification* $T = \langle S, \Omega, Ax \rangle$ comprises a signature $\Sigma = \langle S, \Omega \rangle$ and a set of Σ-sentences Ax called *axioms*. We assume that axioms are in prenex form (i.e. all quantifiers to the left). Specification $T' = \langle S', \Omega', Ax' \rangle$ *extends* specification $T = \langle S, \Omega, Ax \rangle$ if $S \subseteq S'$, $\Omega_{v,s} \subseteq \Omega'_{v,s}$ for each $v \in S^*$, $s \in S$, and $Ax \subseteq Ax'$. Alternatively, we say T is a *subspecification* of T'. A *model* for T is a structure for $\langle S, \Omega \rangle$ that satisfies the axioms. We shall use modus ponens, substitution of equals/equivalents and other natural deduction *rules of inference* in T. A sentence e is a *theorem* of T, written $\vdash_T e$, if e is in the closure of the axioms under the rules of inference.

The notion of a signature morphism can be extended to a specification morphism by requiring that the translation preserves theorems. Let $T = \langle S, \Omega, Ax \rangle$ and $T' = \langle S', \Omega', Ax' \rangle$ be specifications and let $\sigma : \langle S, \Omega \rangle \to \langle S', \Omega' \rangle$ be a signature morphism between them. σ *properly translates* axiom $A \in Ax$ if $\sigma(A)$ is a theorem of T' (i.e. $\vdash_{T'} \sigma(A)$). σ is a *specification morphism* if it properly translates each axiom of T. σ is a *partial specification morphism* if it is a specification morphism from a subspecification of T to T'. It is straightforward to show that a specification morphism translates theorems of the source specification to theorems of the target specification. The semantics of a specification morphism is given by a model construction. If $\sigma : T1 \to T2$ is a specification morphism, then every model $\mathcal{M}$ of $T2$ can be made into a model of $T1$ by simply "forgetting" some structure of $\mathcal{M}$.

It will be convenient to generalize the definition of signature morphism slightly so that the translations of operator symbols are allowed to be terms in the target specification and the translation of sort symbol are constructions on the target sorts (e.g. a product of target sorts). A symbol-to-term morphism can be treated as a symbol-to-symbol morphism into an extension by definitions of the target specification. Well-formedness of translations under a symbol-to-term signature morphism follows from well-formedness of translation under the usual symbol-to-symbol morphisms and well-formedness under expanding operator definitions (Schoenfield (1967)) and sort definitions (e.g. (Mere and Veloso (1992))). Composition of symbol-to-term signature morphisms is straightforward (Turski and Maibaum (1987)).

Several examples that will be used throughout this paper illustrate the above concepts and notation. A *problem* P consists of a set of possible inputs (also called *problem instances*) $x \in D$ such that *input condition* $I(x)$ holds, and a set of outputs (also called *feasible solutions*) $z \in R$ such that some *output condition* $O(x, z)$ holds. A *problem specification* can be presented in the following format

Spec *ProblemSpec*
  Sorts D, R
  Operations $I : D \rightarrow Boolean$
    $O : D \times R \rightarrow Boolean$
endspec

A concrete problem can be presented via a signature morphism from *ProblemSpec* into the domain specification of the problem. *ProblemSpec* can be extended to form a simple *program specification* by adding a function symbol plus an axiom asserting that the function solves the specified problem.

Spec *ProgramSpec*
  Sorts D, R
  Operations $I : D \rightarrow Boolean$
    $O : D \times R \rightarrow Boolean$
    $f : D \rightarrow R$
  Axioms $\forall(x : D)(I(x) \implies O(x, f(x)))$
endspec

A specification morphism from *ProgramSpec* translates the function symbol f into a program in the target theory. The preservation of the axiom of *ProgramSpec* means that the program is correct with respect to the translation of the input and output conditions.

As an example of a problem, consider the problem of sorting a bag of integers. A domain specification for sorting defines the concepts and laws necessary to support the definition of the sorting problem and a method for solving it. The following domain specification is parameterized on a linear order — given any particular set S that is linearly ordered by $\leq$ we obtain a concrete sorting specification.

Spec $Sorting(\langle S, \leq \rangle :: Linear\text{–}Order)$
  Imports $seq(S), bag(S)$
  Operations $ordered : seq(S) \rightarrow boolean$
    $bagify : seq(S) \rightarrow bag(S)$
  Axioms ... axioms defining the operations ...
endspec

We do not give formal definitions for *ordered*, and *bagify* since they will not be needed later. We can present the sorting problem via a signature morphism of ProblemSpec into expressions of the *Sorting* specification.

$$\begin{array}{lcl} D & \mapsto & bag(S) \\ I & \mapsto & \lambda(x)\ true \\ R & \mapsto & seq(S) \\ O & \mapsto & \lambda(x,z)\ ordered(z)\ \wedge\ x = bagify(z) \end{array}$$

3. Constructing Specification Morphisms: Three Techniques

Specification morphisms can express several aspects of software development, including the refinement/implementation of specifications, algorithm and data structure design, and the binding of parameters in parameterized specifications. In this section we introduce three techniques for formally constructing specification morphisms. The first technique is composing specification morphisms in a library. The second is verifying a manually produced signature morphism. The third, based on the inverse of skolemization, allows us to use theorem-proving technology to construct morphisms in systematic way. A fourth technique, called connection between specifications, is presented in Section 4.

The situation is this: we are given a source specification T and a target specification B, and must construct a specification morphism from T to an extension of B.

3.1. Composition

A specification development environment can be expected to have a library of composition methods and simple specification morphisms. Typical composition methods would include horizontal (parametric) and vertical (sequential) composition. A user would use these to compose specification morphisms that reflect the structure of the source specification.

This approach has been successfully used for datatype refinement in the DTRE system (Blaine and Goldberg (1991)). For each datatype and datatype constructor (i.e. parameterized specification), the DTRE system has one or more (parameterized) specification morphisms that provide standard refinements. DTRE provides a simple language for expressing the horizontal and vertical composition of refinements. Users can annotate datatype declarations in programs with expressions in this language. The DTRE compiler handles the details of effectively composing the refinements and applying them. The system can also automatically generate refinement expressions by exploiting heuristics for selecting refinements based on heuristic measures of data structure size and the execution frequency of operations.

In the sequel we will be concerned with the problem of constructing the specification morphisms that might populate such a library.

3.2. Verification

One approach to constructing a specification morphism is manually to "invent" a signature morphism $\sigma : T \to B$ and then verify that each axiom of T translates to a theorem of B. Roughly speaking, this would correspond to a VDM-style approach to implementing algebraic specifications. Generally useful support tools for this approach would include a language for stating specifications and specification morphisms, a translator for applying a specification morphism to arbitrary expressions of the source language, and a theorem prover.

Example: Given a problem specification (that is, a signature morphism from *Problem-Spec* into a given domain specification), we can extend it to a program specification by supplying a program and then translating the correctness condition and proving it, i.e. *program verification* (Floyd (1967)).

3.3. Unskolemization

We turn now to a technique for developing specification morphisms that are correct by construction. The key idea is to use the axioms of the source specification as constraints on the translations of source symbols. Theorem-proving techniques are used to deduce symbol translations such that the source axioms are properly translated.

Skolemization is the process of replacing an existentially quantified variable z with a Skolem function over the universally quantified variables whose scope includes z. For example, the formula

$$\exists(w)\forall(x,y)\exists(z)\forall(u)H(w,x,y,z,u) \tag{3.1}$$

is skolemized to

$$\forall(x,y)\forall(u)H(a,x,y,f(x,y),u) \tag{3.2}$$

where f is a Skolem function of x and y and a is a Skolem function of no arguments – a Skolem constant. A *simple occurrence* of an operation symbol $g : v \to s$ in a sentence G is a subexpression of G of the form $g(x)$ where $x : v$ is a sequence of distinct variables that are universally quantified in G. Skolemization always replaces an existentially quantified variable with simple occurrences of a fresh operation symbol.

We are interested in the inverse process, *unskolemization*: given a sentence (such as (3.2)) containing identical simple occurrences of operation symbol g, say $g(x)$, replace g by a fresh existentially quantified variable in the scope of x (such as (3.1)).

Suppose that we have a partial specification morphism σ from specification T to specification B and we are trying to complete it. Let $f : v \to s$ be a function symbol of T that has no translation yet under σ. Suppose that F is a prenex normal form axiom in which all occurrences of f are identical and simple, and suppose that all other symbols in F are translatable under σ (i.e. the domain of σ includes all of the sort and operator symbols of F except for the function symbol f). To obtain a candidate translation for a function symbol f, we proceed as follows.

(1) *Unskolemize f in F yielding F'.* Since this has the effect of replacing each occurrence of f by a variable, each symbol in F' can be translated via σ.
(2) *Translate F'.* The translated sentence $\sigma(F')$ need not be an axiom of B. In order for σ to become a specification morphism we need an expression defining the translation of f in B. $\sigma(F')$ can be viewed as a constraint on the possible translations of f.
(3) *Attempt to prove* $\sigma(F')$ *in B.* A constructive proof will yield a (witness) expression $t(x)$ for f that depends only on the variables x. If the proof involves induction (resulting in a recursively defined witness), then we extend the target specification with a fresh operator symbol and an axiom stating its recursive definition.
(4) *Extend the partial morphism* σ *by defining* $\sigma(f)$ *to be* $t(x)$. By construction this translation for f guarantees that σ properly translates the axiom F.

Other axioms that involve f may now be translatable, and if so, then we can attempt to prove that they translate to theorems. In this manner we can incrementally construct a specification morphism. There are several choice points in this procedure. In Step 1 there is the choice of which function symbol and axiom to unskolemize. In Step 3 there may be several alternative proofs, each providing a different translation. Altogether, unskolemization can lead to a tree of specification morphisms from T to B. For example, most sorting algorithms can be treated as alternative specification morphisms from divide-and-conquer theory to a specification of the sorting problem. The variation is due to alternative choices in Step 2 (see Section 3.3.2).

Unskolemization procedures have also been developed for a variety of other applications, including extraction of generalized answers from refutation proofs (Luckham and Nilsson (1971)), finding quantified atomic formulas that are sufficient to prove a given goal formula (Cox and Pietrzykowski (1984)), integrity maintenance in a relational database and paramodulation strategies (McCune (1988)), and inductive inference for machine learning (Chadha (1991)).

We illustrate the unskolemization technique via two examples. The first shows how deductive approaches to program synthesis arise in this setting. The second example shows how unskolemization is a key concept in the algorithm design tactics of KIDS.

3.3.1. Deductive Program Synthesis

As in the previous subsection, suppose that we are given a problem specification and that we wish to construct a program that solves the problem. We have a partial morphism from *ProgramSpec*:

$$\begin{array}{rcl} D & \mapsto & bag(S) \\ I & \mapsto & \lambda(x)\; true \\ R & \mapsto & seq(S) \\ O & \mapsto & \lambda(x,z)\; ordered(z) \;\wedge\; x = bagify(z) \\ f & \mapsto & ? \end{array}$$

The difficulty is finding a translation for f such that the axiom

$$\forall(x : D)\; (I(x) \implies O(x, f(x)))$$

translates to a theorem. The verification approach in Section 3.2 would have a programmer manually supply a sorting program as the image of f. We would then have the obligation of proving that the axiom translates to a theorem (i.e. that the sorting program is correct). Instead we unskolemize f so that it is replaced by an existentially quantified variable $z : R$

$$\forall(x : D)\; \exists(z : R)\; (I(x) \implies O(x, z)).$$

The resulting formula can be translated (since there are no occurrences of f in it) and proved. From a constructive proof there are well-known methods for extracting a R-valued term for z that depends on x; i.e., a function that solves the specified problem.

There is a long history in mathematical logic of using constructive proofs to obtain functions (witnesses) from existentially quantified sentences, going back at least to Brouwer's program of Intuitionism. This approach to constructing programs was first explored in computer science by Green (Green (1979)) and Waldinger (Waldinger (1969)) independently in the late 1960's. They showed that a program could be extracted from the proof generated by a resolution theorem-prover. A few years later Constable (Constable (1971)) emphasized the use of constructive logics. Much work continues on this approach to program construction (see for example (Constable (1986), Manna and Waldinger (1985), Manna and Waldinger (1990))).

3.3.2. Divide-and-Conquer Design Tactic

The algorithm design tactics in KIDS have many applications of unskolemization. For example, the algorithm theories for divide-and-conquer and dynamic programming involve a "soundness axiom" that relates decomposition and composition operators. Given a simple decomposition operator, the soundness axiom is unskolemized and used to derive a corresponding composition operator (Smith (1985), Smith (1991)). The pruning mechanisms of global search algorithms are also derived via unskolemization (Smith (1987)).

The principle of divide-and-conquer algorithms is to solve small problem instances directly, and to solve larger problem instances by decomposing them, solving the pieces, and composing the resulting solutions. Part of a specification for a simple divide-and-conquer theory is given next. It provides the structure for a binary decomposition operator and corresponding composition operator. A general scheme for problem reduction theories (including divide-and-conquer) is given in (Smith (1991)).

Spec *Divide-and-Conquer Theory*
Sorts

D	*input domain*
R	*output domain*

Operations

$I : D \to boolean$	*input condition*
$O : D \times R \to boolean$	*output condition*
$Decompose : D \times D \times D \to boolean$	*decomposition relation*
$Compose : R \times R \times R \to boolean$	*composition relation*
$primitive : D \to boolean$	*primitive predicate*

Axioms

$$\begin{aligned}&\text{(Soundness)}\ \forall(x_0, x_1, x_2 : D)\ \forall(z_0, z_1, z_2 : R)\\ &\quad (I(x_0)\ \wedge\ Decompose(x_0, x_1, x_2)\\ &\qquad \wedge\ O(x_1, z_1)\ \wedge\ O(x_2, z_2)\\ &\qquad \wedge\ Compose(z_0, z_1, z_2)\\ &\qquad \Longrightarrow\ O(x_0, z_0))\end{aligned}$$

. . .
endspec

Here subspecification $\langle\{D, R\}, \{I, O\}, \{\}\rangle$ is *ProblemSpec*. The Soundness axiom asserts that if

(1) nonprimitive problem instance x_0 can decompose into two subproblem instances x_1 and x_2,
(2) subproblem instances x_1 and x_2 have feasible solutions z_1 and z_2 respectively,
(3) z_1 and z_2 can compose to form z_0

then z_0 is a feasible solution to input x_0. The soundness axiom is the key condition relating O, *Decompose*, and *Compose*. We omit the remaining axioms.

Using divide-and-conquer theory we can prove the consistency of a *program theory* containing an abstract divide-and-conquer program parameterized on a divide-and-conquer theory (see (Smith (1985))). For purposes of this paper, we do not need to give such a program theory. However it is important to realize that the program theory is parameterized on a divide-and-conquer theory. This reduces the problem of obtaining a correct, concrete divide-and-conquer program to the problem of constructing a specification morphism from divide-and-conquer theory to a given problem domain theory. Such a specification morphism provides the binding of argument to parameter theory and it is a simple computation to obtain the concrete program.

The construction of a specification morphism from divide-and-conquer theory to Sorting theory can proceed in several ways. One tactic in KIDS is based on the choice of a standard decomposition operator from a library. The tactic then uses unskolemization on the soundness axiom to derive a (specification for a) composition operator. This approach allows the derivations of insertion sort, mergesort, and various parallel sorting algorithms (Smith (1985), Smith (1993)). We proceed in a dual way by choosing a simple composition relation and using unskolemization on the soundness axiom to derive a decomposition operator. Suppose that we choose concatenation as a simple composition relation on the output domain *seq*(*integer*). This gives us the partial signature morphism

$$\begin{array}{rcl} D & \mapsto & \mathit{bag}(S) \\ I & \mapsto & \lambda(x)\ \mathit{true} \\ R & \mapsto & \mathit{seq}(S) \\ O & \mapsto & \lambda(x,z)\ \mathit{ordered}(z)\ \wedge\ x = \mathit{bagify}(z) \\ \mathit{Compose} & \mapsto & \lambda(z_0,z_1,z_2)\ z_0 = \mathit{concat}(z_1,z_2) \\ \mathit{Decompose} & \mapsto & ? \end{array}$$

The soundness axiom

$$\begin{array}{l} \forall(x_0,x_1,x_2:D)\ \forall(z_0,z_1,z_2:R) \\ \quad (I(x_0)\ \wedge\ \mathit{Decompose}(x_0,x_1,x_2)\ \wedge\ O(x_1,z_1)\ \wedge\ O(x_2,z_2)\ \wedge\ \mathit{Compose}(z_0,z_1,z_2) \\ \qquad \Longrightarrow\ O(x_0,z_0)) \end{array}$$

is unskolemized on operator symbol *Decompose* yielding

$$\begin{array}{l} \forall(x_0,x_1,x_2:D)\ \exists(y:\mathit{boolean})\ \forall(z_0,z_1,z_2:R) \\ \quad (I(x_0)\ \wedge\ y\ \wedge\ O(x_1,z_1)\ \wedge\ O(x_2,z_2)\ \wedge\ \mathit{Compose}(z_0,z_1,z_2) \\ \qquad \Longrightarrow\ O(x_0,z_0)). \end{array}$$

This formula can be translated via the partial signature morphism yielding:

$$\begin{array}{l} \forall(x_0,x_1,x_2:\mathit{bag}(\mathit{integer}))\ \exists(y:\mathit{boolean})\ \forall(z_0,z_1,z_2:\mathit{seq}(\mathit{integer})) \\ \quad (\mathit{true}\ \wedge\ y \\ \qquad \wedge\ \mathit{ordered}(z_1)\ \wedge\ x_1 = \mathit{bagify}(z_1) \\ \qquad \wedge\ \mathit{ordered}(z_2)\ \wedge\ x_2 = \mathit{bagify}(z_2) \\ \qquad \wedge\ z_0 = \mathit{concat}(z_1,z_2) \\ \qquad \Longrightarrow\ \mathit{ordered}(z_0)\ \wedge\ x_0 = \mathit{bagify}(z_0)) \end{array}$$

A straightforward proof of this formula in Sorting theory, yields a constructive definition of *Decompose* (see (Smith (1985))):

$$x_0 = x_1 \bigcup x_2 \ \wedge\ x_1 \leq x_2$$

where $\bigcup$ is bag-union and $x_1 \leq x_2$ means that each element of bag x_1 is less-than-or-equal to each element of bag x_2. This is, of course, a specification for the partition operation of a quicksort. If we take this as the translation of *Decompose*, then by construction we know that the soundness axiom translates to a theorem in Sorting theory.

The remaining steps in the KIDS divide-and-conquer tactic include unskolemizing another axiom to obtain a translation for the *primitive* predicate, and translating and proving other axioms. The resulting algorithm is a variant of Quicksort (for details see (Smith (1985))). Various selection sort algorithms, such as heapsort, are also derivable by starting with a choice of composition operator.

4. Connections Between Specifications

Unskolemization is particularly effective when there is not a strong coupling between operators in the axioms of a specification. When a specification morphism must be constructed from a specification in which the axioms relate the specification's operators in intricate ways, then another concept, called connections between specifications or simply connections, can often be applied.

By way of motivation, let us preview an application of connections. Suppose that we need a scheduling algorithm and that we have a specification (called an algorithm theory (Smith and Lowry (1989))) for the general concept of backtrack. A specification morphism from this backtrack specification to a specification of the scheduling problem would articulate the components necessary to construct a backtrack scheduler. The axioms of backtrack are complex enough that it is difficult to construct this morphism via unskolemization, so instead we exploit a preexisting specification morphism from backtrack to a specification of the problem of enumerating sequences. This morphism effectively provides the components for a backtrack enumerator of all sequences over a given finite set. A connection between the images of the backtrack specification in the sequence and scheduling specifications effectively transfers the backtrack structure for enumerating sequences to scheduling. The result is a specification morphism from backtrack to scheduling that allows the enumeration of schedules.

A connection between specifications can be thought of as a general set of conditions under which a specification morphism can be constructed in the following way. Suppose that we want to construct a specification morphism from specification $T = \langle S, \Omega, Ax \rangle$ to (an extension of) specification B. Assuming that we have a specification morphism from T into some specification A and there exists (or we can construct) a connection from A to B, then there exists (or we can construct) a specification morphism from T to B.

Intuitively, the connection allows us to construct a proof of $G_A \implies G_B$ for each axiom G of T. Then, since G_A is assumed to be a theorem, we have a proof of G_B. To prove $G_A \implies G_B$ we systematically transform G_A by replacing A-symbols in G_A with corresponding symbols in B, until G_A has been transformed into G_B itself. The key is to guarantee that each such replacement *weakens* the sentence, so by chaining we ultimately obtain $G_A \implies G_B$; i.e. we must ensure that each intermediate sentence is monotonic in the symbol replacements.

Some introductory concepts and analytic tools must be presented before we can define a connection. The following sections present a generalized notion of an ordered-sorted specification (cf. (Goguen and Meseguer (1988))) that we call po-specifications. Polarity analysis of positive and negative occurrences of operator and sort symbols in axioms is used to define connections.

4.1. Polarity Algebra

Most common types come equipped with a natural partial order and equality. For example, the integers have $\leq$ and its converse $\geq$ as partial orders. Equality on integers is also a partial order and it is the intersection of $\leq$ and $\geq$ (by antisymmetry). Sets come with the subset $\subseteq$ and superset $\supseteq$ partial orders and the usual set equality that is the intersection of these orders. Booleans come with implies $\Longrightarrow$ and implied-by $\Longleftarrow$ as partial orders and equivalence $\Longleftrightarrow$ as their intersection.

Furthermore there is a natural partial ordering between sorts themselves, often referred to as the subsort hierarchy (Goguen and Meseguer (1988)). If we let $\leq$ denote the subsort relation between sorts, then for example

$$Nat \leq Integer \leq Rational \leq Real.$$

The only assumption that we are making about the operations on subsorts is that values of the subsort may participate in any computation defined in the supersort (perhaps after conversion).

Generally we will be interested in specifications in which sorts are interpreted as partially-ordered sets — a set (called the carrier or domain) plus partial order and equality. Furthermore the set of sorts is itself partially-ordered under the subsort relation. Such a specification will be called a *po-specification*.

To capture these observations and extend them we define a simple polarity algebra, called POLARITY, and give constraints on its intended interpretations. The constants of POLARITY are $+$ (positive), $-$ (negative), and $\pm$ (neutral). Our intent is that these constants be interpreted as partial orders over some set. Furthermore $+$ and $-$ are to be interpreted as partial orders that are converses to one another, and $\pm$ is interpreted as the equality that is the intersection of these partial orders. There is an order $\sqsubseteq$ on the constants defined by the Hasse diagram

$$\begin{tikzcd}
- \arrow[dr, dash] & & + \arrow[dl, dash] \\
& \pm &
\end{tikzcd}$$

This order is intended to be interpreted as the subset relation between partial orders. It is easy to check that POLARITY is itself a partial order [†]. Two other operations are needed on polarities: the *converse* operation, written $\tilde{p}$, defined by

[†] In fact POLARITY under $\sqsubseteq$ is a meet-semilattice. If we add another constant, say 1, that is the least upper bound of $+$ and $-$, then POLARITY becomes a lattice. The natural interpretation of 1 would be the comparability relation.

p	$\tilde{p}$
$+$	$-$
$\pm$	$\pm$
$-$	$+$

and a *meet* operation $\sqcap$ defined as the greatest lower bound under the $\sqsubseteq$ order. Note that *meet* is associative, commutative, and idempotent. Furthermore, since *meet* computes a lower bound, we have for any polarities p and q

$$p \sqsupseteq p \sqcap q \sqsubseteq q. \tag{4.1}$$

4.2. Sorts as Posets

In subsequent developments, each sort of a specification will be enriched with a collection of partial orders that is an image of POLARITY: two conversely related partial orders and equality. Some standard interpretations of POLARITY can be given;

for integers:

$+$	$\mapsto$	$\leq$
$\pm$	$\mapsto$	$=$
$-$	$\mapsto$	$\geq$

for sets:

$+$	$\mapsto$	$\subseteq$
$\pm$	$\mapsto$	$=$
$-$	$\mapsto$	$\supseteq$

for booleans:

$+$	$\mapsto$	$\Longrightarrow$
$\pm$	$\mapsto$	$\Longleftrightarrow$
$-$	$\mapsto$	$\Longleftarrow$

Let $\xrightarrow[s]{p}$ denote the partial order that is the interpretation of polarity p over sort s. For example, with respect to the interpretations given above,

$$\xrightarrow[Integer]{+} \quad \text{denotes} \quad \leq$$

$$\xrightarrow[Set]{-} \quad \text{denotes} \quad \supseteq$$

$$\xrightarrow[Boolean]{+} \quad \text{denotes} \quad \Longrightarrow$$

$$\xrightarrow[Boolean]{\pm} \quad \text{denotes} \quad \Longleftrightarrow$$

There are several simple properties of polarity relations on sorts that will be useful later.

PROPOSITION 4.1. *Suppose that a, b, c are expressions of sort $s \in S$.*

$$\text{If } a \xrightarrow[s]{p} b \text{ and } b \xrightarrow[s]{p} c$$
$$\text{then } a \xrightarrow[s]{p} c.$$

PROOF. The proposition reflects the transitivity of the partial order that $\xrightarrow[s]{p}$ denotes. $\square$

PROPOSITION 4.2. *Suppose that a and b are expressions of sort $s \in S$ and p and q are polarities.*

$$\text{If } p \sqsubseteq q \text{ and } a \xrightarrow[s]{p} b$$
$$\text{then } a \xrightarrow[s]{q} b.$$

PROOF. When $p \sqsubseteq q$ is interpreted with respect to partial orders R_p and R_q on a set A, this means that $R_p \subseteq R_q$. Consequently, $a \xrightarrow[s]{p} b$ means $\langle a, b\rangle \in R_p$ which implies $\langle a, b\rangle \in R_q$, which is $a \xrightarrow[s]{q} b$. $\square$

COROLLARY 4.1. *Suppose that a and b are expressions of sort $s \in S$ and $p_1, ..., p_n$ are polarities where $1 \leq n$:*

$$\text{If } a \xrightarrow[s]{p_1 \sqcap ... \sqcap p_n} b$$
$$\text{then } a \xrightarrow[s]{p_1} b.$$

PROOF. Since $p_1 \sqsubseteq p_1 \sqcap ... \sqcap p_n$ we can apply Proposition 4.2. $\square$

4.2.1. POLARITY ANALYSIS FOR OPERATORS

Now that sorts come equipped with partial orders, we can explore how operations are affected by the orders. For each operation we give laws that specify monotonicity properties with respect to the orders on its parameters. The laws provide the basis for inference and analysis rules that show how a syntactic change to a subexpression that preserves a local partial order affects the semantics of the whole expression.

Suppose that we have a function $f : v \to s$ in T that has the following *monotonicity law*

$$\text{If } a_i \xrightarrow[v_i]{p_i(p)} b_i \text{ for } i = 1, ..., n$$
$$\text{then } f(a_1, ..., a_n) \xrightarrow[s]{p} f(b_1, ..., b_n)$$

where p_i is a polarity-valued function for $i = 1, ..., n$. This law provides the basis for a *polarity analysis rule for operator f* written (using the notation in (Manna and Waldinger (1986)))

$$[f(a_1, ..., a_n)]^p \Rrightarrow f(a_1^{p_1(p)}, ..., a_n^{p_n(p)}).$$

The analysis rule is used to infer *derived polarities* for arguments from the polarity of the function application. In other words, if we desire to make a syntactic change to $f(a_1, ..., a_n)$ that produces a $\xrightarrow[s]{p}$ result, then we can do so by bringing about a $\xrightarrow[v_i]{p_i(p)}$ change to a_i for $i = 1, ..., n$. Polarity analysis rules will be used top-down to infer derived polarities on subexpressions given an assigned polarity (usually +) to the top expression.

For example, consider the monotonicity law for the operator $size : set(\alpha) \rightarrow integer$ (size of a finite set):

$$\begin{array}{l}\text{If } S \xrightarrow[set]{p} T \\ \text{then } size(S) \xrightarrow[integer]{p} size(T).\end{array}$$

This law splits into three cases ($p = +, \pm, -$):

$$\begin{array}{l}\text{If } S \subseteq T \\ \text{then } size(S) \leq size(T).\end{array}$$

$$\begin{array}{l}\text{If } S = T \\ \text{then } size(S) = size(T).\end{array}$$

$$\begin{array}{l}\text{If } S \supseteq T \\ \text{then } size(S) \geq size(T).\end{array}$$

The polarity analysis rule for *Size* is

$$[size(S)]^p \Rrightarrow size(S^p).$$

As another example, the monotonicity law for $\subseteq$: $set(\alpha) \times set(\alpha) \rightarrow Boolean$ is

$$\begin{array}{l}\text{If } R \xrightarrow[set]{\tilde{p}} R' \ \wedge S \xrightarrow[set]{p} S' \\ \text{then } R \subseteq S \xrightarrow[Boolean]{p} R' \subseteq S'.\end{array}$$

In the case that $p = +$ this law is

$$\begin{array}{l}\text{If } R \supseteq R' \ \wedge \ S \subseteq S' \\ \text{then } R \subseteq S \implies R' \subseteq S'.\end{array}$$

The corresponding analysis rule is:

$$[R \subseteq S]^p \Rrightarrow R^{\tilde{p}} \subseteq S^p$$

A standard library of polarity rules can be listed. The following list is representative, but not exhaustive.

For boolean operators:

$$\begin{array}{l}[P \ \wedge \ Q]^p \Rrightarrow P^p \ \wedge \ Q^p \\ [P \ \vee \ Q]^p \Rrightarrow P^p \ \vee \ Q^p \\ [\neg P]^p \Rrightarrow \neg P^{\tilde{p}} \\ [P \implies Q]^p \Rrightarrow P^{\tilde{p}} \implies Q^p\end{array}$$

$$
\begin{array}{l}
[P \iff Q]^p \Rrightarrow P^{\pm} \iff Q^{\pm} \\
[\forall(x:D)\ P(x)]^p \Rrightarrow \forall(x:D)\ P(x)^p \\
[\exists(x:D)\ P(x)]^p \Rrightarrow \exists(x:D)\ P(x)^p
\end{array}
$$

For integer operators:

$$
\begin{array}{l}
[a+b]^p \Rrightarrow a^p + b^p \\
[a-b]^p \Rrightarrow a^p - b^{\tilde{p}} \\
[a \le b]^p \Rrightarrow a^{\tilde{p}} \le b^p \\
[a = b]^p \Rrightarrow a^{\pm} = b^{\pm}
\end{array}
$$

For set operators:

$$
\begin{array}{l}
[R \bigcup S]^p \Rrightarrow R^p \bigcup S^p \\
[R \bigcap S]^p \Rrightarrow R^p \bigcap S^p \\
[R \subseteq S]^p \Rrightarrow R^{\tilde{p}} \subseteq S^p \\
[R = S]^p \Rrightarrow R^{\pm} = S^{\pm}
\end{array}
$$

In the absence of any other information about the monotonicity properties of an operator, there is a default rule:

$$[f(a_1, ..., a_n)]^p \Rrightarrow f(a_1^{\pm}, ..., a_n^{\pm}).$$

This rule is always correct (since we can substitute equals for equals and partial orders are reflexive), and it is consistent with all other possible rules.

The polarity rules can be applied to an expression E to determine the polarity of all subexpressions as a function of the polarity of E. If t is a subexpression of E, let vp_t denote the derived *value polarity* for t (relative to the assignment of polarity $+$ to E). For example, if we assign a polarity of $+$ to the expression

$$size(\{x \mid \neg P \wedge Q\})$$

then recursive application of polarity rules yields the following derived polarities on each subexpression:

$$size(\{x^{\pm} \mid ((\neg P^{-})^{+} \wedge Q^{+})^{+}\}^{+})^{+}.$$

Here we have $vp_{\neg P \wedge Q} = +$ and $vp_P = -$ and so on. As a result of this analysis there are various derived polarity laws (see Proposition 4.1), such as:

If $P \impliedby R$
then $size(\{x \mid \neg P \wedge Q\}) \le size(\{x \mid \neg R \wedge Q\})$.

4.2.2. Polarity Assignment for Operators

The definition of a connection between specifications relies on the calculation of a polarity for each operator symbol in T. A slight difficulty is that if an operator symbol occurs more than once in an expression then polarity analysis may assign different polarities to each occurrence. For example, the operator f has both positive and negative occurrences in the (partially analyzed) expression

$$f^{-}(0) \le^{+} f^{+}(1).$$

Consider specification $T = \langle S, \Omega, Ax \rangle$ and let $Ax' \subseteq Ax$ be a subset of the axioms. Define the polarity map $\pi : \Omega \rightarrow POLARITY$ as follows. Let the polarity of each axiom in Ax' be $+$, then recursively apply polarity analysis. For operator symbol f, collect the polarities of all subexpressions with a leading occurrence of f in Ax', say $vp_{f1}, ..., vp_{fk}$, then let

$$\pi_f = vp_{f1} \sqcap vp_{f2} \ldots \sqcap vp_{fk}.$$

By Corollary 4.1 $\xrightarrow[s]{\pi_f}$ is consistent with the polarity of each occurrence of f in E.

Example: Consider *ProgramSpec* from Section 2:

Spec $ProgramSpec$
  Sorts D, R
  Operations $I : D \rightarrow Boolean$
    $O : D \times R \rightarrow Boolean$
    $f : D \rightarrow R$
  Axiom $\forall (x : D)(I(x) \implies O(x, f(x)))$
endspec

Polarity analysis of the operators yields

$\pi_I = -$ (since I occurs in the antecedent of an implication which is a negative context)
$\pi_O = +$ (since O occurs in the consequent of an implication which is a positive context)
$\pi_f = \pm$ (in the absence of special monotonicity laws about O the default polarity is $\pm$)

As a convention, we assume that the polarity of boolean connectives is $\pm$; this is consistent with the fixed interpretation of truth values in mathematical logic.

4.3. The Set of Sorts as a Poset

In subsequent developments the set of sorts of a specification will be enriched with a collection of partial orders that interprets POLARITY: two conversely related partial orders (called subsort and supersort), and equality.

4.3.1. Polarity Analysis of Sorts

Previous sections dealt with how terms are monotonic in subterms. We now explore how quantified expressions are monotonic with respect to the subtype ordering on sorts.

The *sort polarity* of an expression is indicated by a subscripted polarity (superscripted polarities indicate value polarity). The following polarity analysis rules infer a sort polarity for each subexpression:

$$[\forall (x : D)\ P(x)]^\pi \Rrightarrow \forall (x : D)\ P(x_{\tilde{\pi}})$$
$$[\exists (x : D)\ P(x)]^\pi \Rrightarrow \exists (x : D)\ P(x_\pi)$$
$$[f(a_1, \ldots a_n)]^\pi \Rrightarrow f(a_1, \ldots a_n)_+$$

The sorts of universally quantified variables of a formula F have converse polarity to the polarity of F and the sorts of existentially quantified variables have the same polarity. All nonvariable terms have sort polarity $+$.

Sort polarity rule 3 derives from rule 2. Let $QG[f(a)]$ be an axiom with quantifiers Q and matrix G containing an occurrence of a nonvariable term $f(a)$. $QG[f(a)]$ is equivalent to $Q\ \exists(z : s)\ (f(a) = z\ \wedge\ G[z])$, thus the derived sort polarity on s is the same as the polarity of the axiom (by rule 2) and since we will always assign axioms a polarity of $+$, the derived sort polarity on the s-valued term $f(a)$ is $+$.

We present the polarity laws for sorts in two stages. In the first stage, the subsort relation is interpreted as the subset relation $\subseteq$ between sorts as sets. In the second stage we adopt the more general view that subsorts may require conversion, thus the subsort relation must be interpreted by a conversion map from subsort to supersort.

Sort polarity rule 1 is based on the following law:

$$\begin{array}{l}\text{If } D \xrightarrow[S]{\tilde{p}} D' \ \wedge\ domain(P) = D \bigcup D' \\ \text{then } \forall(x : D)\ P(x) \xrightarrow[boolean]{p} \forall(y : D')\ P(y).\end{array}$$

This law specializes to three cases ($p = +, \pm, -$):

$$\begin{array}{l}\text{If } D \supseteq D' \\ \text{then } \forall(x : D)\ P(x) \implies \forall(y : D')\ P(y).\end{array}$$

$$\begin{array}{l}\text{If } D = D' \\ \text{then } \forall(x : D)\ P(x) \iff \forall(y : D')\ P(y).\end{array}$$

$$\begin{array}{l}\text{If } D \subseteq D' \\ \text{then } \forall(x : D)\ P(x) \impliedby \forall(y : D')\ P(y).\end{array}$$

Sort polarity rule 2 corresponds to the law

$$\begin{array}{l}\text{If } D \xrightarrow[S]{p} D' \ \wedge\ domain(P) = D \bigcup D' \\ \text{then } \exists(x : D)\ P(x) \xrightarrow[boolean]{p} \exists(y : D')\ P(y).\end{array}$$

This law specializes to three cases ($p = +, \pm, -$):

$$\begin{array}{l}\text{If } D \subseteq D' \\ \text{then } \exists(x : D)\ P(x) \implies \exists(y : D')\ P(y).\end{array}$$

$$\begin{array}{l}\text{If } D = D' \\ \text{then } \exists(x : D)\ P(x) \iff \exists(y : D')\ P(y).\end{array}$$

$$\begin{array}{l}\text{If } D \supseteq D' \\ \text{then } \exists(x : D)\ P(x) \impliedby \exists(y : D')\ P(y).\end{array}$$

In the second stage we adopt the more general view that subsorts may require conversion, thus the subsort relation must be interpreted by a conversion map from subsort to supersort. For example, we can treat integers as a subtype of the reals, but in conventional programming languages these types have distinct representations. This fact requires the convertsion of integers to reals when an integer-valued subexpression is used in a real-valued expression.

Given a pair of sorts s_1 and s_2, a polarity p will be interpreted as a *conversion function*, written

$$h : s_1 \xrightarrow[S]{p} s_2.$$

If $p = +$ then $h : s_1 \rightarrow s_2$ (a map from s_1 to s_2).
If $p = -$ then $h : s_1 \leftarrow s_2$ (a map from s_2 to s_1).
If $p = \pm$ then $s_1 = s_2$ and $h : s_1 \leftrightarrow s_2$ is the identity function.

The fact that the direction of the conversion function can vary causes some notational problems that motivate the following definitions.

$$\begin{aligned} \ell_p &= \begin{cases} id_{s_1} : s_1 \rightarrow s_1 & \text{if } p = +, \pm \\ h : s_2 \rightarrow s_1 & \text{if } p = - \end{cases} \\ r_p &= \begin{cases} h : s_1 \rightarrow s_2 & \text{if } p = + \\ id_{s_2} : s_2 \rightarrow s_2 & \text{if } p = -, \pm \end{cases} \end{aligned}$$

Rule 1 is based on the following law:

$$\begin{aligned} &\text{If } h : D \xrightarrow[S]{\tilde{p}} D' \ \wedge\ domain(P) = codomain(h) \\ &\text{then } \forall(x : D)\ P(\ell_p(x)) \xrightarrow[boolean]{p} \forall(y : D')\ P(r_p(y)). \end{aligned}$$

This law specializes to three cases ($p = +, \pm, -$):

$$\begin{aligned} &\text{If } h : D \leftarrow D' \ \wedge\ domain(P) = D \\ &\text{then } \forall(x : D)\ P(x) \implies \forall(y : D')\ P(h(y)). \end{aligned}$$

$$\begin{aligned} &\text{If } h : D \leftrightarrow D' \ \wedge\ domain(P) = D = D' \\ &\text{then } \forall(x : D)\ P(x) \iff \forall(y : D')\ P(y). \end{aligned}$$

$$\begin{aligned} &\text{If } h : D \rightarrow D' \ \wedge\ domain(P) = D' \\ &\text{then } \forall(x : D)\ P(h(x)) \impliedby \forall(y : D')\ P(y). \end{aligned}$$

Rule 2 derives from the law

$$\begin{aligned} &\text{If } h : D \xrightarrow[S]{p} D' \ \wedge\ domain(P) = codomain(h) \\ &\text{then } \exists(x : D)\ P(\ell_p(x)) \xrightarrow[boolean]{p} \exists(y : D')\ P(r_p(y)). \end{aligned}$$

This law specializes to three cases ($p = +, \pm, -$):

$$\begin{aligned} &\text{If } h : D \rightarrow D' \ \wedge\ domain(P) = D' \\ &\text{then } \exists(x : D)\ P(h(x)) \implies \exists(y : D')\ P(y). \end{aligned}$$

$$\text{If } h : D \leftrightarrow D' \ \wedge\ domain(P) = D = D'$$

then $\exists(x : D)\ P(x) \iff \exists(y : D')\ P(y)$.

If $h : D \leftarrow D' \ \wedge\ domain(P) = D$
then $\exists(x : D)\ P(x) \impliedby \exists(y : D')\ P(h(y))$.

4.3.2. Polarity Assignment on Sorts

The polarity function π (which was defined for operator symbols in Section 4.2.2) can be extended to sort symbols. This assignment is consistent with the inferred polarities of all occurrences of each sort symbol. Consider a specification $T = \langle S, \Omega, Ax\rangle$ and let $Ax' \subseteq Ax$ be a subset of the axioms. The sort polarity map $\pi : S \to \mathit{POLARITY}$ is calculated as follows. Let the polarity of each axiom in Ax' be $+$, then recursively apply sort polarity analysis. For sort symbol s, collect the polarities of all occurrences of s-valued terms in Ax', say $sp_1, ..., sp_k$, then let

$$\pi_s = sp_1 \sqcap \ldots \sqcap sp_k.$$

Example: Consider the axiom from *ProgramSpec*

$$\forall(x : D)(I(x) \implies O(x, f(x))).$$

Polarity analysis of the sorts yields

$\pi_D = -$ (the quantification of x gives D negative polarity)
$\pi_R = +$ (the occurrence of $f(x)$ gives R positive polarity)

As a convention, we assume that $\pi_{boolean}$ is $\pm$. Again, this is consistent with the fixed interpretation of truth values.

4.4. Remarks about Polarity

A straightforward inductive argument shows that polarity rules compose in a natural way.

Proposition 4.3. *Suppose $G[e] : r$ is an expression of sort r containing an occurrence of expression $e : t$ of sort t. If $G[e]$ has polarity π_G and the occurrence of e in G has derived polarity $p(\pi_G)$ and $e \xrightarrow[t]{p(\pi_G)} e'$ then $G[e] \xrightarrow[r]{\pi_G} G[e']$.*

Polarity analysis of the propositional structure of first-order sentences has a long history in mathematical logic. For example, Lyndon's theorem (Lyndon (1959)) is a classical result based on polarity analysis of axioms. Manna and Waldinger (Manna and Waldinger (1986)) use polarity analysis to enrich the resolution format with special inference rules for various relations. Our extension of polarity analysis to operators in arbitrary (partially ordered) sorts and the (partially ordered) set of sorts seems to be new. The RAINBOW inference system (Smith (1982), Smith (1985), Smith (1990)) in KIDS allows inference within the (terms of the) partially ordered sorts of a po-specification. In particular, in *boolean* it allows the derivation of sufficient conditions, necessary conditions, or equivalent conditions; in *integer* it allows the derivation of lower bounds, upper bounds, and

$$\begin{tikzcd}
 & T \arrow[dl, "\sigma_A"'] \arrow[dr, "\sigma_B"] & \\
A \arrow[rr, dashed, "H"] & & B
\end{tikzcd}$$

0acci

Figure 1. *T*-Connection from *A* to *B*

simplified expressions, and so on. Polarity analysis is used to determine applicability of rules that preserve a specified ordering.

4.5. Connections

A connection between specifications can be thought of as a generalized homomorphism. In both cases, there is a function for each sort symbol of the common signature, and a condition for each function/predicate symbol. One difference is that homomorphisms are defined in terms of structures for some give signature, whereas connections are defined in terms of specifications and signature morphisms from a common specification. Connections can be given a model-level definition, just as homomorphisms between specifications can be formalized. Connections generalize the notion of a homomorphism in two essential ways. First, in a homomorphism the functions between sorts all map in one direction — from the source to the target, whereas in connections the functions can map in either direction depending on polarity analysis of the sorts in the axioms. Second, a homomorphism condition asserts the preservation of an equality relation under the homomorphism. In contrast, the connection conditions assert the preservation of an ordering relation under the conversion functions.

Let $T = \langle S, \Omega, Ax \rangle$ be a po-specification. Let π be a polarity map on T obtained by analysis of axioms $Ax' \subseteq Ax$. Let A and B be specifications with signature morphisms $\sigma_A : T \to A$ and $\sigma_B : T \to B$. We will write A_s instead of $\sigma_A(s)$ for sort symbol $s \in S$ and f_A instead of $\sigma_A(f)$ for function symbol f in T. A T-*connection* from A to B is a collection of *conversion functions* $H = \{h_s : A_s \xrightarrow[S]{\pi_s} B_s \mid s \in S\}$ such that for each operator symbol $f : v \to s$ in T the following *connection condition* holds (notation is explained below)

$$\forall (x : D^v, z : D_s)\ (\ell_s(z) \xrightarrow[A_s]{\pi_f} f_A(\ell^v(x))) \implies (r_s(z) \xrightarrow[B_s]{\pi_f} f_B(r^v(x))).$$

(see Figure 1). The dependence of the direction of each conversion function on polarity analysis of axioms motivates the following definitions. These are similar to the notations defined in Section 4.3.1. The functions ℓ and r are indexed on sorts here versus polarities earlier.

$$\ell_s \; = \; \begin{cases} id_{A_s} : A_s \to A_s & \text{if } \pi_s = +, \pm \\ h_s : B_s \to A_s & \text{if } \pi_s = - \end{cases}$$

$$r_s = \begin{cases} h_s : A_s \to B_s & \text{if } \pi_s = + \\ id_{B_s} : B_s \to B_s & \text{if } \pi_s = -, \pm \end{cases}$$
$$D_s = \begin{cases} A_s & \text{if } \pi_s = +, \pm \\ B_s & \text{if } \pi_s = - \end{cases}$$

The *specification C of a T-connection from A to B obtained via analysis of the axioms $Ax' \subseteq Ax$* is an extension of both A and B that is defined as follows. The sorts of C are the union of the sorts of A and B. The operations of C are the union of the operations of A and B together with symbols for the conversion functions. The axioms of C are the union of the axioms of A and B together with the connection conditions. When T, A, B, σ_A, σ_B, and Ax' are clear in context we will simply call C a (T-)*connection specification.* Note that C is not necessarily a conservative extension of B. The connection conditions place extra constraints on the symbols of B that need not be theorems of B.

The connection conditions clearly state a kind of order-preserving relationship between A and B. The unusual aspect of a connection is that the conversion maps h_s do not always map from A to B, but may go in the reverse direction, depending on the polarity analysis.

The following result asserts that the connection conditions hold for arbitrary nonvariable subexpressions of the analyzed axioms.

THEOREM 4.1. *Let C be a specification of a T-connection from A to B obtained via analysis of the axioms $Ax' \subseteq Ax$. Let $f(a)[X]$ be a subexpression of an axiom of Ax' with free variables $X : u$ where $f : v \to s$ and $a : u \to v$. If*
(i) $\sigma_A : T \to A$ is a specification morphism;
(ii) $\sigma_B : \langle S, \Omega\rangle \to B$ is a signature morphism;
(iii) σ_B properly translates the preorder and monotonicity laws of T into B; [†]
then
$$\vdash_C \forall(X : D^u, z : D_s)\ (\ell_s(z) \xrightarrow[A_s]{vp_{f(a)}} f_A(a_A)[\ell^u(X)]) \implies (r_s(z) \xrightarrow[B_s]{vp_{f(a)}} f_B(a_B)[r^u(X)]).$$

PROOF. The proof uses structural induction on $f(a)$. To simplify the presentation assume that f takes one argument. The generalization to zero or more arguments is straightforward. Let X and z be arbitrary constants in D^u and D_s respectively. Because of the sort polarity rule for nonvariable terms we have $\pi_s \in \{\pm, +\}$; therefore ℓ_s is the identity function, and r_s is $h_s : A_s \to B_s$ and we must establish

$$(z \xrightarrow[A_s]{vp_{f(a)}} f_A(a_A)[\ell^u(X)]) \implies (h_s(z) \xrightarrow[B_s]{vp_{f(a)}} f_B(a_B)[r^u(X)])$$

To proceed, we assume the antecedent $z \xrightarrow[A_s]{vp_{f(a)}} f_A(a_A)[\ell^u(X)]$ and apply a sequence of $\xrightarrow[B_s]{vp_{f(a)}}$ -preserving transformations to $h_s(z)$.

† I.e., the reflexivity and transitivity laws for $\xrightarrow[s]{p}$ where $p \in \{+, \pm, -\}$ and $s \in S$, must properly translate into B and similarly for the polarity laws for each operator in T.

$h_s(z)$

$\xrightarrow[B_s]{vp_{f(a)}}$ by the connection condition for $\xrightarrow[s]{vp_{f(a)}}$; see Note 1 below

$h_s(f_A(a_A))[\ell^u(X)]$

$\xrightarrow[B_s]{\pm}$ same expression, different notation; see Note 2

$h_s(f_A(\ell_v(a'_A)))$

$\xrightarrow[B_s]{\pi_f}$ by the connection condition for f; see Note 3

$f_B(r_v(a'_A))$

$\xrightarrow[B_s]{vp_{f(a)}}$ by induction and the polarity theorem for f; see Note 4

$f_B(a_B[r^u(X)])$

$\xrightarrow[B_s]{\pm}$ same expression, different notation

$f_B(a_B)[r^u(X)]$.

According to Proposition 4.2 each arrow above can be relabeled $\xrightarrow[B_s]{vp_{f(a)}}$ since $\pm \sqsubseteq \pi_f \sqsubseteq vp_{f(a)}$. By transitivity (Proposition 4.1) we obtain the result

$$h_s(z) \xrightarrow[B_s]{vp_{f(a)}} f_B(a_B)[r^u(X)].$$

Notes:

1. Since $\pi_s \in \{\pm, +\}$, the connection condition for $\xrightarrow[s]{vp_{f(a)}}$ is

$$\forall(y_1 : A_s, y_2 : A_s)\ ((y_1 \xrightarrow[A_s]{vp_{f(a)}} y_2) \implies (h_s(y_1) \xrightarrow[B_s]{vp_{f(a)}} h_s(y_2))).$$

Unifying the antecedent with the assumption $z \xrightarrow[A_s]{vp_{f(a)}} f_A(a_A)[\ell^u(X)]$, we get the substitution $\{y_1 \mapsto z, y_2 \mapsto f_A(a_A)[\ell^u(X)]\}$, allowing us to infer

$$h_s(z) \xrightarrow[B_s]{vp_{f(a)}} h_s(f_A(a_A))[\ell^u(X)].$$

2. With respect to $f_A(a_A)$ let

$$a' \text{ denote } \begin{cases} x & \text{if } a_A \text{ is a variable } x \\ a_A[\ell^u(X)] & \text{otherwise} \end{cases}$$

Note that $a_A[\ell^u(X)]$ has the form $\ell_v(a')$ in $f_A(a_A)[\ell^u(X)]$: if a is a variable $x : v$, then $a_A[\ell^u(X)]$ is $\ell_v(a')$; and otherwise a_A is a nonvariable term and polarity analysis on sorts will result in $\pi_v \in \{\pm, +\}$ and thus ℓ_v is the identity function, so again $a_A[\ell^u(X)]$ is $\ell_v(a')$. This step is a slight change in our meta-level description of an expression, not a change in the expression itself.

3. The connection condition for f

$$\forall(w : D_v, y : D_s)\ (y \xrightarrow[A_s]{\pi_f} f_A(\ell_v(w))) \implies (h_s(y) \xrightarrow[B_s]{\pi_f} f_B(r_v(w)))$$

can be applied with instantiation $\{y \mapsto f_A(\ell_v(a')),\ w \mapsto a'\}$.

4. We wish to apply the polarity law for f in B

$$\text{If } c \xrightarrow[B_v]{r(p)} d$$

$$\text{then } f_B(c) \xrightarrow[B_s]{p} f_B(d)$$

using the substitutions $\{c \mapsto r_v(a'_A),\ d \mapsto a_B[r^u(X)]\}$ and letting p be $vp_{f(a)}$ and $r(p)$ be vp_a. To establish the condition of the law, i.e.

$$r_v(a'_A) \xrightarrow[B_v]{vp_a} a_B[r^u(X)],$$

we reason inductively as follows. Consider $r_v(a'_A)$. If a is a variable $x : v$, then $a'_A = x = a'_B$ so

$$r_v(a'_A) \xrightarrow[B_v]{\pm} r_v(x) \xrightarrow[B_v]{\pm} a_B[r^u(X)].$$

But then, since $\pm \sqsubseteq vp_a$ we have

$$r_v(a'_A) \xrightarrow[B_v]{vp_a} a_B[r^u(X)]$$

by Proposition 4.2. If, on the other hand, a is a nonvariable term, then by the induction hypothesis we have

$$\forall(X : D^u, z : D_v)\ (\ell_v(z) \xrightarrow[A_v]{vp_a} a_A[\ell^u(X)]) \implies (r_v(z) \xrightarrow[B_v]{vp_a} a_B[r^u(X)])$$

or, since $\pi_v \in \{\pm, +\}$ by sort polarity analysis and thus ℓ_v is the identity function,

$$\forall(X : D^u, z : D_v)\ (z \xrightarrow[A_v]{vp_a} a_A[\ell^u(X)]) \implies (r_v(z) \xrightarrow[B_v]{vp_a} a_B[r^u(X)]).$$

Using the instantiation $\{z \mapsto a'_A\}$, the antecedent follows by reflexivity of $\xrightarrow[A_v]{vp_a}$, thus we again infer $r_v(a'_A) \xrightarrow[B_v]{vp_a} a_B[r^u(X)]$. Finally, after discharging the assumption in the polarity law for f we infer

$$f_B(r_v(a'_A)) \xrightarrow[B_s]{vp_{f(a)}} f_B(a_B[r^u(X)]).$$

□

Before proceeding on to the main result of this section, one last bit of additional notation is needed. Let $G[U, E]$ denote a closed formula in prenex normal form where

$U : v$ denotes the sequence of variables that are universally quantified and $E : w$ denotes the sequence of variables that are existentially quantified. For example, the formula

$$\forall(x : D)\ \exists(y : R)\ \forall(z : W)(I(x, z) \implies P(x, y, z))$$

can be described in this notation via the correspondence

$$\begin{array}{lll} U & \mapsto & [x, z] \\ E & \mapsto & [y] \end{array}$$

Let δ be a specification that extends specifications α, β, and γ; and let there be signature morphisms from specification T into α, β, and γ. Let ${}^{\beta}_{\alpha}G_{\gamma}[U, E]$ denote the translation of $G[U, E]$ in δ in which the sorts of universally quantified variables are interpreted in α, the sorts of existentially quantified variables are interpreted in β, and G is interpreted in γ. Returning to the example above, ${}^{A}_{B}G_{B}[[h_D(x), h_W(z)], [y]]$ denotes

$$\forall(x : D_B)\ \exists(y : R_A)\ \forall(z : W_B)\ (I_B(h_D(x), h_W(z)) \implies P_B(h_D(x), y, h_W(z))).$$

Simplifying slightly, Theorem 4.2 states that if $\sigma_B : T \rightarrow B$ is a signature morphism and $\sigma_A : T \rightarrow A$ is a specification morphism and C is a connection from A to B, then σ_B is a specification morphism (from T to C). This result reduces the problem of constructing a specification morphism to the problem of constructing a connection. As can be seen in the examples in Section 5, connections can be constructed using a series of unskolemize-and-prove steps.

Theorem 4.2. *Let $T = \langle S, \Omega, Ax\rangle$ be a po-specification; let C be the specification of a T-connection from A to B obtained via analysis of the axioms $Ax' \subseteq Ax$. If*
(i) $\sigma_A : T \rightarrow A$ is a signature morphism that properly translates the axioms of Ax' into A;
(ii) $\sigma_B : \langle S, \Omega\rangle \rightarrow B$ is a signature morphism;
(iii) σ_B properly translates the preorder and polarity laws of T into B;
then σ_B properly translates the axioms of Ax' into C. In particular, if $Ax' = Ax$ then σ_B is a specification morphism.

Proof. Let $G \in Ax'$. A proof of ${}^{B}_{B}G_{B}[U, E]$ in C can be constructed as follows. Let $U : v$ and $E : w$. By assumption (i), ${}^{A}_{A}G_{A}[U, E]$ is a theorem of A and therefore $\vdash_C {}^{A}_{A}G_{A}[U, E]$.

${}^{A}_{A}G_{A}[U, E]$

$\implies$ changing the sort for universally quantified variables; see Note 1

${}^{A}_{B}G_{A}[h^v(U), E]$

$\implies$ applying Theorem 4.1; see Note 2

${}^{A}_{B}G_{B}[U, h^w(E)]$

$\implies$ changing the sort for existentially quantified variables; see Note 3

${}^{B}_{B}G_{B}[U, E]$.

Then by modus ponens we infer $\vdash_C {}^B_B G_B[U, E]$.

Notes

1. If G contains no variables that are universally quantified then the step holds vacuously. Suppose that G contains the subexpression $\forall(x : D)P[x]$. By polarity analysis, $\pi_D \in \{-, \pm\}$ so $h_D : D_A \leftarrow D_B$ (where h_D may be the identity map). Also, since G is assumed to be in prenex form, the derived polarity of $\forall(x : D)P[x]$ in G must be positive. By the polarity law for universally quantified sorts (see Section 4.3.1),

$$\forall(x : D_A)P_A[x] \implies \forall(y : D_B)P_A[h_D(y)]$$

and then by Proposition 4.3

$$G[\forall(x : D_A)P_A[x]] \implies G[\forall(y : D_B)P_A[h_D(y)]].$$

Continue in this manner for each variable that is universally quantified.

2. Since G is in prenex form, let $G = QH$ where Q denotes the quantifiers, H denotes the matrix. Axioms are assigned polarity $+$, so the derived polarity on H is also $+$. By Theorem 4.1 we have

$$\begin{aligned}&\forall(U : D^v, E : D^w, z : \mathit{boolean})\\ &\qquad (z \xrightarrow[\mathit{Boolean}]{+} H_A[\ell^v(U), \ell^w(E)]) \implies (z \xrightarrow[\mathit{Boolean}]{+} H_B[r^v(U), r^w(E)]).\end{aligned}$$

This formula simplifies as follows: each universally quantified sort in G (i.e. each component of D^v) has nonpositive sort polarity, thus $\ell^v(U)$ is $h^v(U)$ and $r^v(U)$ is simply U; each existentially quantified sort in G (i.e. each component of D^w) has nonnegative sort polarity, thus $\ell^v(E)$ is E and $r^v(E)$ is simply $h^w(E)$. Thus the connection condition simplifies to

$$\forall(U : B^v, E : A^w, z : \mathit{boolean})\ (z \implies H_A[h^v(U), E]) \implies (z \implies H_B[U, h^w(E)]).$$

and then to

$$\forall(U : B^v, E : A^w)\ H_A[h^v(U), E] \implies H_B[U, h^w(E)].$$

By substitution we obtain

$${}^A_B Q\ H_A[h^v(U), E] \implies {}^A_B Q\ H_B[U, h^w(E)].$$

3. If G contains no existentially quantified variables then the step holds vacuously. Suppose that G contains the subexpression $\exists(x : D)P[x]$, so D has nonnegative sort-polarity. From the connection we have $h_D : D_A \to D_B$. By polarity analysis, the polarity of $\exists(x : D)P[x]$ in G must be positive. By the polarity theorems for existentials (see Section 4.3.1)

$$\exists(x : D_A)P[h_D(x)] \implies \exists(y : D_B)P[y]$$

and then by Proposition 4.3

$$G[\exists(x : D_A)P[h_D(x)]] \implies G[\exists(y : D_B)P[y]].$$

Continuing in this manner for each existential quantification, we achieve the desired result.

□

Theorem 4.2 corresponds to the following fact about models: if M_A is a model for specification T and S_B (a Σ-reduct of S_C) is a structure for T and there exists a connection (between structures) from M_A to S_B, then S_B is a model of T.

Again, connections provide a proper generalization of homomorphisms. A strong homomorphism is a connection in which all sort polarities are nonnegative and the polarities of all function and predicate symbols are $\pm$. A weak homomorphism allows predicate symbols to have polarity $+$. Under this polarity assignment the connection conditions simplify to the homomorphism conditions.

Regarding condition *(iii)* in Theorems 4.1 and 4.2, it can be shown that if the preorder and polarity laws are included in the subset of axioms that undergo polarity analysis, then condition *(iii)* is unnecessary.

4.6. Example: Program Specification

The proof construction in the proof of Theorem 4.2 can be illustrated (abstractly) using *ProgramSpec*.

Spec $ProgramSpec$
  Sorts D, R
  Operations $I : D \to Boolean$
  $O : D \times R \to Boolean$
  $f : D \to R$
  Axiom $\forall(x : D)(I(x) \implies O(x, f(x)))$
endspec

Polarity analysis of the axiom yields (see previous sections):

$\pi_{boolean} = \pm$
$\pi_D = -$
$\pi_R = +$
$\pi_I = -$
$\pi_O = +$
$\pi_f = \pm$

Suppose that there is a specification morphism from *ProgramSpec* into specification A and a signature morphism from *ProgramSpec* into specification B. Based on the the preceding polarity analysis, a specification C of a *ProgramSpec*-connection from A to B would have conversion maps

$h_D : A_D \leftarrow B_D$
$h_R : A_R \to B_R$

and the simplified connection conditions[†]

$$\begin{array}{ll} (I) & \forall(x : B_D)\ I_A(h_D(x)) \Longleftarrow I_B(x) \\ (O) & \forall(x : B_D, z : A_R)\ O_A(h_D(x), z) \implies O_B(x, h_R(z)) \\ (f) & \forall(x : B_D)\ h_R(f_A(h_D(x))) = f_B(x) \end{array}$$

Note that the connection condition (f) gives a definition for f_B by showing how to invoke f_A and translate its results. The connection effectively defines a simple problem reduction from problem B to problem A.

Note also that Booleans remain unchanged under the connection, therefore boolean operators translate to themselves under a connection. The proof that the axiom of *Program-Spec* translates (via the signature morphism into B) into a theorem of C is constructed as follows. By assumption

$$\vdash_A \forall(x : A_D)(I_A(x) \implies O_A(x, f_A(x)))$$

which implies (since C is an extension of A)

$$\vdash_C \forall(x : A_D)(I_A(x) \implies O_A(x, f_A(x))).$$

We proceed by reasoning within C as follows.

$\forall(x : A_D)(I_A(x) \implies O_A(x, f_A(x)))$

$\implies$ changing the quantification of x

$\forall(x : B_D)(I_A(h_D(x)) \implies O_A(h_D(x), f_A(h_D(x))))$

$\implies$ by the connection condition for I

$\forall(x : B_D)(I_B(x) \implies O_A(h_D(x), f_A(h_D(x))))$

$\implies$ by the connection condition for O

$\forall(x : B_D)(I_B(x) \implies O_B(x, h_R(f_A(h_D(x)))))$

$\iff$ by the connection condition for f

$\forall(x : B_D)(I_B(x) \implies O_B(x, f_B(x)))$

thus $\vdash_C \forall(x : B_D)(I_B(x) \implies O_B(x, f_B(x)))$.

[†] The connection conditions usually collapse to a simpler form. For example, the condition for (I) is

$$\forall(x : B_D, z : \mathit{boolean})\ ((\ell_{boolean}(z) \xrightarrow[\mathit{boolean}]{\pi_I} I_A(\ell_D(x))) \implies (r_{boolean}(z) \xrightarrow[\mathit{boolean}]{\pi_I} I_B(r_D(x))))$$

or

$$\forall(x : B_D, z : \mathit{boolean})\ ((z \Longleftarrow I_A(h_D(x))) \implies (z \Longleftarrow I_B(x)))$$

which simplifies to

$$\forall(x : B_D)\ (I_A(h_D(x)) \Longleftarrow I_B(x)).$$

4.7. Generalizations

Straightforward application of polarity analysis sometimes leads to connection conditions and conversion maps that are too strong to be useful. By adroitly distinguishing uses of operators and sorts, we can weaken the connection conditions and sort relations. The problem is that the same operator may be used for several purposes in different contexts. By renaming the operator for its different uses (without changing its meaning), we can weaken and thus generalize the connection condition for that operator. Analogously, a sort may be used for different purposes in different contexts. Again, by renaming the sort for its different uses we get a weaker polarity.

There is a dual formulation of sort polarity analysis that is also useful. We conjecture that it allows a generalization of Lyndon's theorem that homomorphisms preserve models iff the source theory has positive axioms (Lyndon (1959)). It also seems to subsume previous work on datatype implementation via abstraction and refinement functions (Smith (1992a)).

5. Applications

Theorem 4.2 supports the following method for constructing a specification morphism from T to B. Suppose σ is a partial specification morphism from T to B. We can extend σ to a (total) signature morphism by extending B with fresh symbols to obtain B' and defining $\sigma(q) = q'$ for each symbol q in T that is untranslatable under *sigma* and fresh symbol q'. We then construct a connection from A to B' using unskolemization on the connection conditions to define the conversion functions. The connection conditions provide definitions for the new symbols of B' such that the axioms of T are properly translated by σ into C (which extends B').

We illustrate the unskolemization and connection techniques through two concrete examples.

5.1. Simple Problem Reduction

In this example, we use unskolemization and connections to reduce a given sorting problem to a library sorting program. Suppose that we have a presentation of the domain of sorting that includes definitions for operations such as $ordered : seq(integer) \to Boolean$ (which decides if a sequence of integers is ordered) and $Quicksort : bag(integer) \to seq(integer)$ which sorts a bag of integers. Quicksort can be presented as a correct program via a specification morphism from *ProgramSpec* into the domain theory of sorting:

$$\begin{array}{rcl} D & \mapsto & bag(integer) \\ I & \mapsto & \lambda(x)\ true \\ R & \mapsto & seq(integer) \\ O & \mapsto & \lambda(x,z)\ x = bagify(z)\ \wedge\ ordered(z) \\ f & \mapsto & \lambda(x)\ Quicksort(x) \end{array}$$

The function *bagify* maps a sequence to the bag of its elements. Suppose now that we are given the problem of sorting sequences, i.e., we obtain a specification *Seq-Sorting* for sorting sequences (rather than bags):

$$\begin{array}{rcl}
D & \mapsto & seq(integer) \\
I & \mapsto & \lambda(x)\ true \\
R & \mapsto & seq(integer) \\
O & \mapsto & \lambda(x,z)\ bagify(x) = bagify(z)\ \wedge\ ordered(z) \\
f & \mapsto & \lambda(x)\ SeqSort(x)
\end{array}$$

where *SeqSort* is the name of our desired sequence-sorting program. The problem is to provide a definition for *SeqSort* such that this signature morphism is a specification morphism from *ProgramSpec*. To do so we first develop a *ProgramSpec*-connection from *Sorting* to *Seq-Sorting*. Reusing the polarity analysis from Section 4.6, we need conversion maps with signatures

$$\begin{array}{l}
h_D : bag(integer) \leftarrow seq(integer) \\
h_R : seq(integer) \rightarrow seq(integer)
\end{array}$$

such that the following simplified connection conditions are theorems:

$$\begin{array}{ll}
(I) & \forall(x : seq(integer))\ true \implies true \\
(O) & \forall(x : seq(integer),\ z : seq(integer)) \\
& \qquad h_D(x) = bagify(z)\ \wedge\ ordered(z) \\
& \qquad \implies bagify(x) = bagify(h_R(z))\ \wedge\ ordered(h_R(z)) \\
(f) & \forall(x : seq(integer))\ h_R(Quicksort(h_D(x))) = SeqSort(x)
\end{array}$$

To establish the connection we must derive expressions for the conversion maps such that the connection conditions hold. To do that we can use unskolemization! Condition (I) provides no information. However, focusing on (O) and replacing h_D by $y : bag(integer)$ and h_R by $w : seq(integer)$ and noting dependencies we obtain the unskolemized formula

$$\begin{array}{l}
\forall(x : seq(integer))\ \exists(y : bag(integer))\ \forall(z : seq(integer))\ \exists(w : seq(integer)) \\
\qquad (y = bagify(z)\ \wedge\ ordered(z) \\
\qquad \implies \\
\qquad bagify(x) = bagify(w)\ \wedge\ ordered(w)).
\end{array}$$

Proof of this formula results in the substitutions

$$\{y \mapsto \lambda(x)\ bagify(x),\ w \mapsto \lambda(z)z\}$$

giving us h_D and h_R respectively and simultaneously ensuring that the connection condition (O) is a theorem. Next we establish condition (f) by unskolemizing *SeqSort* (replacing it with variable $z : seq(integer)$ and using the derived expressions for h_D and h_R)

$$\forall(x : seq(integer))\ \exists(z : seq(integer))\ Quicksort(bagify(x)) = z$$

which trivially yields the substitution

$$\{z \mapsto \lambda(x)\ Quicksort(bagify(x))\}.$$

This gives us a definition for *SeqSort* that simultaneously ensures that the connection condition (f) is a theorem. Putting the pieces together and applying Theorem 4.2 we have that

$$\begin{array}{rcl} D & \mapsto & seq(integer) \\ I & \mapsto & \lambda(x)\ true \\ R & \mapsto & seq(integer) \\ O & \mapsto & \lambda(x,z)\ bagify(x) = bagify(z)\ \wedge\ ordered(z) \\ f & \mapsto & \lambda(x)\ Quicksort(bagify(x)) \end{array}$$

is a specfication morphism. In words, the expression $Quicksort(bagify(x))$ would correctly sort any given sequence x.

5.2. Global Search Theory

In (Smith (1987), Smith and Lowry (1990)) we present a formal theory of backtrack algorithms, called *global search theory* (or simply gs-theory). Global search theory provides the general concepts, operations, and laws that underlie concrete backtrack programs.

The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *space descriptors* (denoted by hatted symbols). In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor.

Formally, *global search theory* is presented as follows:

Spec *Global Search Theory*
Sorts

D	*input domain*
R	*output domain*
$\hat{R}$	*space descriptors*

Operations

$I : D \to boolean$	*input condition*
$O : D \times R \to boolean$	*input/output condition*
$\hat{I} : D \times \hat{R} \to boolean$	*subspace descriptors condition*
$\hat{r}_0 : D \to \hat{R}$	*initial space*
$Satisfies : R \times \hat{R} \to boolean$	*denotation of descriptors*
$Split : D \times \hat{R} \times \hat{R} \to boolean$	*split relation*
$Extract : R \times \hat{R} \to boolean$	*extractor of solutions from spaces*

Axioms

GS0. $I(x) \implies \hat{I}(x, \hat{r}_0(x))$

GS1. $I(x)\ \wedge\ \hat{I}(x,\hat{r})\ \wedge\ Split(x,\hat{r},\hat{s}) \implies \hat{I}(x,\hat{s})$

GS2. $I(x) \wedge O(x,z) \implies \mathit{Satisfies}(z, \hat{r}_0(x))$
GS3. $I(x) \wedge \hat{I}(x,\hat{r})$
$\implies (\mathit{Satisfies}(z,\hat{r}) \iff \exists(\hat{s})\ (\ \mathit{Split}^*(x,\hat{r},\hat{s}) \wedge \mathit{Extract}(z,\hat{s})))$

where the subspecification $\langle\{D,R\},\{I,O\},\{\}\rangle$ is *ProblemSpec*, $\hat{R}$ is the type of space descriptors, $\hat{I}$ defines legal space descriptors, $\hat{r}$ and $\hat{s}$ vary over descriptors, $\hat{r}_0(x)$ is the descriptor of the initial set of candidate solutions, $\mathit{Satisfies}(z,\hat{r})$ means that z is in the set denoted by descriptor $\hat{r}$ or that z satisfies the constraints that $\hat{r}$ represents, $\mathit{Split}(x,\hat{r},\hat{s})$ means that $\hat{s}$ is a subspace of $\hat{r}$ with respect to input x, and $\mathit{Extract}(z,\hat{r})$ means that z is directly extractable from $\hat{r}$. Axiom GS0 asserts that the initial descriptor $\hat{r}_0(x)$ is a legal descriptor. Axiom GS1 asserts that legal descriptors split into legal descriptors. Axiom GS2 constrains the denotation of the initial descriptor — all feasible solutions are contained in the initial space. Axiom GS3 gives the denotation of an arbitrary descriptor $\hat{r}$ — an output object z is in the set denoted by $\hat{r}$ if and only if z can be extracted after finitely many applications of *Split* to $\hat{r}$ where

$$\mathit{Split}^*(x,\hat{r},\hat{s}) \iff \exists(k:Nat)\ (\ \mathit{Split}^k(x,\hat{r},\hat{s})\ )$$

and

$$\mathit{Split}^0(x,\hat{r},\hat{t}) \iff \hat{r} = \hat{t}$$

and for all natural numbers k

$$\mathit{Split}^{k+1}(x,\hat{r},\hat{t}) \iff \exists(\hat{s}:\hat{R})\ (\ \mathit{Split}(x,\hat{r},\hat{s}) \wedge \mathit{Split}^k(x,\hat{s},\hat{t})).$$

In (Smith (1987)) we show the consistency of an abstract global search program in a program theory parameterized on global search theory. Consequently, construction of a correct global search program reduces to the problem of constructing a specification morphism from global search theory to a given problem specification.

As an example of a concrete gs-theory, consider the problem of enumerating sequences over a given finite set S. A space is a set of sequences with common prefix *part_sol* and is represented by *part_sol*. The descriptor for the initial space is just [] (the empty sequence). Splitting is performed by appending an element from S onto the end of the common prefix *part_sol*. The sequence *part_sol* itself is directly extractable from the space. This global search theory for enumerating sequences, called *gs-sequences*, can be presented via a specification morphism:

D	$\mapsto$	$\mathit{set}(\alpha)$
I	$\mapsto$	$\lambda(S)\ \mathit{true}$
R	$\mapsto$	$\mathit{seq}(\alpha)$
O	$\mapsto$	$\lambda(S,q)\ \mathit{range}(q) \subseteq S$
$\hat{R}$	$\mapsto$	$\mathit{seq}(\alpha)$
$\hat{I}$	$\mapsto$	$\lambda(S,\mathit{part\_sol})\ \mathit{range}(\mathit{part\_sol}) \subseteq S$
Satisfies	$\mapsto$	$\lambda(q,\mathit{part\_sol})\ \exists(r)\ (q\ =\ \mathit{concat}(\mathit{part\_sol},r))$
$\hat{r}_0$	$\mapsto$	$\lambda(S)\ [\,]$
Split	$\mapsto$	$\lambda(S,\mathit{part\_sol},\mathit{part\_sol}')$ $\exists(i:\alpha)\ (i \in S \wedge \mathit{part\_sol}' = \mathit{append}(\mathit{part\_sol},i))$
Extract	$\mapsto$	$\lambda(q,\mathit{part\_sol})\ q = \mathit{part\_sol}$

Suppose that we obtain a specification for a simple scheduling problem: given a set of jobs (*Jobs*) and a precedence relation on jobs (*PRECEDES*) that is a partial order, enumerate all feasible schedules, where a schedule is a sequence of jobs that satisfy the precedence constraints.

A domain specification for scheduling might be parameterized on *JOB* (the parameter *TRIV* requires a single sort plus equality on the sort) and be presented

Spec $Scheduling(JOB :: TRIV)$
 Operations $Partial\text{–}Order : set(JOB \times JOB) \rightarrow Boolean$
 $Bijective : seq(JOB) \times set(JOB) \rightarrow Boolean$
 $Consistent : seq(JOB) \times set(JOB \times JOB) \rightarrow Boolean$
 Axioms ... axioms defining the operations ...
endspec

We do not give formal definitions for *Partial–Order*, *Bijective*, and *Consistent* since they will not be needed later. Informally, a sequence S is *bijective* into a set R iff the elements of the sequence are in one-to-one correspondence with R. A sequence S is *consistent* with a partial order $\sqsubseteq$ if $S(i) \sqsubseteq S(j)$ whenever $1 \leq i \leq j \leq length(S)$. We can present the scheduling problem via a signature morphism of *ProblemSpec* into expressions of the *Scheduling* specification.

The problem can be presented via a signature morphism from *ProblemSpec* into *Scheduling*:

$$\begin{array}{rcl}
D & \mapsto & set(JOB) \times set(JOB \times JOB) \\
I & \mapsto & \lambda(Jobs, Precedes)\ Partial\text{–}Order(Precedes) \\
R & \mapsto & seq(JOB) \\
O & \mapsto & \lambda(Jobs, Precedes, S)\ Bijective(S, Jobs)\ \wedge\ Consistent(S, Precedes)
\end{array}$$

Once we have a global search theory of scheduling, then it is a simple mechanical step to obtain a correct global search program (Smith (1987), Smith (1990)). The problem then is to develop a specification morphism from global search theory into *Scheduling*. We do so by constructing a connection from *gs-sequences* to *Scheduling*.

We proceed by extending *Scheduling* with fresh symbols so that the signature morphism above can be extended to a signature morphism from gs-theory to *Scheduling*.

$$\begin{array}{rcl}
D & \mapsto & set(JOB) \times set(JOB \times JOB) \\
I & \mapsto & \lambda(Jobs, Precedes)\ Partial\text{–}Order(Precedes) \\
R & \mapsto & seq(JOB) \\
O & \mapsto & \lambda(Jobs, Precedes, S)\ Bijective(S, Jobs)\ \wedge\ Consistent(S, Precedes) \\
\hat{R} & \mapsto & \hat{R}_{sched} \\
\hat{I} & \mapsto & \hat{I}_{sched} \\
Satisfies & \mapsto & Satisfies_{sched} \\
\hat{r}_0 & \mapsto & \hat{r}_{0\,sched} \\
Split & \mapsto & Split_{sched} \\
Extract & \mapsto & Extract_{sched}
\end{array}$$

Next we develop a gs-theory-connection from *gs-sequences* to *Scheduling*. Polarity analysis results in the following polarity assignments based on the four axioms of gs-theory:

$$\begin{aligned}
&\pi_D = -\\
&\pi_R = +\\
&\pi_{\hat{R}} = \pm\\
\\
&\pi_I = -\\
&\pi_{\hat{I}} = \pm\\
&\pi_O = -\\
&\pi_{Satisfies} = \pm\\
&\pi_{\hat{r}_0} = \pm\\
&\pi_{Split} = \pm\\
&\pi_{Extract} = \pm
\end{aligned}$$

A connection specification has conversion function signatures

$$\begin{aligned}
&h_{boolean} : boolean \leftrightarrow boolean\\
&h_D : set(JOB) \leftarrow set(JOB) \times set(JOB \times JOB)\\
&h_R : seq(JOB) \rightarrow seq(JOB)\\
&h_{\hat{R}} : seq(JOB) \leftrightarrow seq(JOB)
\end{aligned}$$

and the simplified connection conditions

$$\begin{array}{ll}
(I) & \forall(Jobs, Precedes)\ (true \impliedby Partial\text{–}Order(Precedes))\\
(O) & \forall(Jobs, Precedes, Sched)(range(h_R(Sched)) \subseteq h_D(Jobs, Precedes)\\
& \quad \impliedby Bijective(Sched, Jobs)\ \wedge\ Consistent(Sched, Precedes))\\
(\hat{I}) & \forall(Jobs, Precedes, part\_sol)\ (range(part\_sol) \subseteq h_D(Jobs, Precedes)\\
& \quad \iff \hat{I}_{sched}(Jobs, Precedes, part\_sol))\\
(\hat{r}_0) & \forall(Jobs, Precedes)\ [] = \hat{r}_{0\ sched}(h_D(Jobs, Precedes))\\
(Satisfies) & \forall(Sched, part\_sol)\ \exists(r)\\
& \quad (h_R(Sched) = concat(part\_sol, r) \iff Satisfies_{sched}(Sched, part\_sol))\\
(Split) & \forall(Jobs, Precedes, part\_sol, part\_sol')\\
& \quad \exists(i)\ (i \in h_D(Jobs, Precedes)\ \wedge\ part\_sol' = append(part\_sol, i))\\
& \qquad \iff Split_{sched}(Jobs, Precedes, part\_sol, part\_sol')\\
(Extract) & \forall(Sched, part\_sol)\ h_R(Sched) = part\_sol \iff Extract_{sched}(Sched, part\_sol)
\end{array}$$

As in the previous example, we derive expressions for the conversion maps such that the connection conditions hold. Condition (I) is universally valid so it provides no information. However, focusing on (O) and replacing h_D by $y : set(JOB)$ and h_R by $w : seq(JOB)$ and noting dependencies we obtain the unskolemized formula

$$\begin{aligned}
&\forall(Jobs, Precedes)\ \exists(y : set(JOB))\ \forall(Sched)\ \exists(w : seq(JOB))\\
&\quad (Bijective(Sched, Jobs)\ \wedge\ Consistent(Sched, Precedes) \implies range(w) \subseteq y)
\end{aligned}$$

Proving this formula requires the definition

$$Bijective(sq, st) \iff Injective(sq, st)\ \wedge\ range(sq) = st.$$

After expanding the term $Bijective(Sched, Jobs)$ in the antecedent and unifying, we obtain the substitutions

$$\{w \mapsto \lambda(Sched)\ Sched,\ y \mapsto \lambda(Jobs, Precedes)\ Jobs\}$$

giving us h_D and h_R respectively and simultaneously ensuring that the connection condition (O) is a theorem.

Next we establish condition $(\hat{I})$ by unskolemizing $\hat{I}_{sched}$ (replacing it with variable $z : seq(JOB)$ and using the derived expressions for h_D and h_R)

$$\forall(Jobs, Precedes, part\_sol)\ \exists(z : seq(JOB))\ (range(part\_sol) \subseteq Jobs \iff z)$$

which trivially yields the substitution

$$\{z \mapsto \lambda(Jobs, Precedes, part\_sol)\ range(part\_sol) \subseteq Jobs\}.$$

This gives us a definition for $\hat{I}_{sched}$ that simultaneously ensures that the connection condition $(\hat{I})$ is a theorem. The remaining connnection conditions are treated in the same way.

Putting the pieces together and applying Theorem 4.2 we have a specification morphism from Global Search to Scheduling.

$$\begin{array}{rcl}
D & \mapsto & set(JOB) \times set(JOB \times JOB) \\
I & \mapsto & \lambda(Jobs, Precedes)\ Partial\text{–}Order(Precedes) \\
R & \mapsto & seq(JOB) \\
O & \mapsto & \lambda(Jobs, Precedes, Sched) \\
 & & \quad Bijective(Sched, Jobs)\ \wedge\ Consistent(Sched, Precedes) \\
\hat{R} & \mapsto & seq(JOB) \\
\hat{I} & \mapsto & \lambda(Jobs, Precedes, part\_sol)\ range(part\_sol) \subseteq Jobs \\
Satisfies & \mapsto & \lambda(Sched, part\_sol)\ \exists(r)\ (Sched\ =\ concat(part\_sol, r)) \\
\hat{r}_0 & \mapsto & \lambda(Jobs, Precedes)\ [\,] \\
Split & \mapsto & \lambda(Jobs, Precedes, part\_sol, part\_sol') \\
 & & \qquad \exists(i)\ (i \in Jobs\ \wedge\ part\_sol' = append(part\_sol, i)) \\
Extract & \mapsto & \lambda(Sched, part\_sol)\ Sched = part\_sol
\end{array}$$

This morphism specifies a global search theory for generating sequences over a given set of *Jobs*. It is used to instantiate a global search program scheme to obtain a concrete scheduling program. Other steps in the global search design tactic use unskolemization to derive pruning tests and constraint propagation mechanisms (see (Smith (1987), Smith (1990), Smith (1992b))).

6. Concluding Remarks

We have presented several methods for constructing specification morphisms. The two new methods, unskolemization and connections, carefully exploit the axioms of the source theory to derive symbol translations such that the source axioms translate to theorems.

Most of the steps in the algorithm design tactics of KIDS can be viewed as applying either the unskolemization or connection techniques. We have designed and optimized over fifty algorithms using KIDS. Despite the apparently complex machinery, our experience is that the unskolemization and connection techniques tend to break the design task into a series of relatively simple deduction problems that are tractable with respect to current theorem-proving technology.

One goal of the research described in this paper is to develop a mechanized environment that supports the acquisition, development, and implementation of specifications. In addition to modelling application domains and developing formal requirement specifications, users could supply design knowledge in the form of specifications (for abstract algorithms, abstract datatypes, system architectures, etc.). Tools for constructing specification morphisms would support the application of design knowledge during the implementation of requirements specifications.

We are currently implementing the techniques described in this paper within the KIDS system. The immediate aim is to support algorithm design directly from a hierarchy of algorithm theories (Smith and Lowry (1990)), rather than relying on a collection of manually coded special-purpose design tactics. Longer term goals are to explore the design of data structures and the application of software architecture theories to system design.

Acknowledgements

I would like to thank Lee Blaine, Jose Meseguer, Y.V. Srinivas, and the referees for their helpful comments on versions of this paper. I would also like to thank Peter Pepper for suggesting the term “unskolemization” during a presentation of this concept. This research was supported by the Air Force Office of Scientific Research under Contract F49620-91-C-0073, by the Office of Naval Research under Grant N00014-90-J-1733, and by DARPA and Rome Laboratories under Contract F30602-91-C-0043.

References

Astesiano, E. and Wirsing, M. (1987). An introduction to ASL. In Meertens, L., editor, *Program Specification and Transformation*, pages 343–365. North-Holland, Amsterdam.

Blaine, L. and Goldberg, A. (1991). DTRE – a semi-automatic transformation system. In Möller, B., editor, *Constructing Programs from Specifications*, pages 165–204. North-Holland, Amsterdam.

Broy, M., Wirsing, M., and Pepper, P. (1987). On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54–99.

Burstall, R. M. and Goguen, J. A. (1977). Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, Cambridge, MA. IJCAI.

Chadha, R. (1991). *Applications of Unskolemization*. PhD thesis, University of North Carolina at Chapel Hill, NC.

Constable, R. L. (1971). Constructive mathematics and automatic program writers. In *Information Processing 71*, pages 229–233, Ljubljana, Yugoslavia. IFIP.

Constable, R. L. (1986). *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, New York.

Cox, P. and Pietrzykowski, T. (1984). A complete, nonredundant algorithm for reversed skolemization. *Theoretical Computer Science*, 28:239–261.

Ehrig, H., Kreowski, H. J., Thatcher, J., Wagner, E., and Wright, J. (1981). Parameter passing in algebraic specification languages. In *Proceedings, Workshop on Program Specification*, volume 134, pages 322–369, Aarhus, Denmark.

Ehrig, H. and Mahr, B. (1990). *Fundamentals of Algebraic Specification, vol. 2: Module Specifications and Constraints*. Springer-Verlag, Berlin.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press, New York.

Floyd, R. W. (1967). Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics XIX*, pages 19–32. American Mathematical Society.

Goguen, J. A. and Meseguer, J. (1988). Order-sorted algebra I: Equational deduction for multiple inheritance, polymorphism, and partial operations. Draft technical report, Computer Science Laboratory, SRI International.

Goguen, J. A., Thatcher, J. W., and Wagner, E. G. (1978). An initial algebra approach to the specification, correctness and implementation of abstract data types. In Yeh, R., editor, *Current Trends in Programming Methodology, Vol. 4: Data Structuring*. Prentice-Hall, Englewood Cliffs, NJ.

Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. (1977). Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95.

Goguen, J. A. and Winkler, T. (1988). Introducing OBJ3. Technical Report SRI-CSL-88-09, SRI International, Menlo Park, California.

Green, C. (1979). *The Application of Theorem Proving to Question Answering Systems.* Garland Publishing, Inc., New York, New York. (PhD Thesis, Department of Computer Science, Stanford University, June 1969).

Guttag, J. V. and Horning, J. J. (1978). The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52.

Hoare, C. (1989). Varieties of programming languages. Technical report, Programming Research Group, University of Oxford, Oxford, UK.

Lowry, M. R. (1987). Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence*, Seattle, WA.

Luckham, D. C. and Nilsson, N. (1971). Extracting information from rersolution proof trees. *Artificial Intelligence*, 2:27–54.

Lyndon, R. C. (1959). Properties preserved under homomorphism. *Pacific Journal of Mathematics*, 9:143–154.

Manna, Z. and Waldinger, R. (1985). *Logical Basis for Computer Programming, Vol. 1: Deductive Reasoning.* Addison-Wesley, Reading, MA.

Manna, Z. and Waldinger, R. (1986). Special relations in automated deduction. *Journal of the ACM*, 33(1):1–59.

Manna, Z. and Waldinger, R. (1990). *Logical Basis for Computer Programming, Vol. 2: Deductive Systems.* Addison-Wesley, Reading, MA.

McCune, W. W. (1988). Un-skolemizing clause sets. *Information Processing Letters*, 29:257–263.

Meré, M. and Veloso, P. A. (1992). On extension by sorts. Technical Report 38/92, Departamento de Informatica, Pontifícia Universidade Católica, Rio de Janiero, Brazil.

Partsch, H. (1990). *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, New York.

Sannella, D. and Tarlecki, A. (1985). Extended ML: An institution-independent framework for formal program development. In *Category Theory and Computer Programming, LNCS 240*, pages 364–389.

Sannella, D. and Tarlecki, A. (1988). Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25(3):233–281.

Shoenfield, J. R. (1967). *Mathematical Logic.* Addison-Wesley, Reading, MA.

Smith, D. R. (1982). Derived preconditions and their use in program synthesis, LNCS 138. In Loveland, D. W., editor, *Sixth Conference on Automated Deduction*, pages 172–193, Berlin. Springer-Verlag.

Smith, D. R. (1985). Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

Smith, D. R. (1987). Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute.

Smith, D. R. and Lowry, M. R. (1990). Algorithm theories and design tactics. Science of Computer Programming, 14(2-3): 305–321.

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

Smith, D. R. (1991). Structure and design of problem reduction generators. In Möller, B., editor, *Constructing Programs from Specifications*, pages 91–124. North-Holland, Amsterdam.

Smith, D. R. (1992b). Transformational approach to scheduling. Technical Report KES.U.92.2, Kestrel Institute.

Smith, D. R. (1992a). Data structure design. in preparation.

Smith, D. R. (1993). Automated derivation of parallel sorting algorithms. In Paige, R., Reif, J., and Wachter, R., editors, *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic Publishers, New York.

Srinivas, Y. V. (1991). Algebraic specification for domains. In Prieto-Diaz, R. and Arango, G., editors, *Domain Analysis: Acquisition of Reusable Information for Software Construction*, pages 90–124. IEEE Computer Society Press, Los Alamitos, CA.

Turski, W. M. and Maibaum, T. E. (1987). *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England.

Veloso, P. A. (1988). Problem solving by interpretation of theories. In *Contemporary Mathematics*, volume 69, pages 241–250. American Mathematical Society, Providence, Rhode Island.

Waldinger, R. (1969). *Constructing Programs Automatically Using Theorem Proving.* PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, PA.

Wirsing, M. (1990). Algebraic specification. In van Leeuwen, J., editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 675–788. MIT Press/Elsevier.