to appear in *Calculational System Design*, Proceedings of the International Summer School Marktoberdorf, Ed. M. Broy, NATO ASI Series, IOS Press, Amsterdam, 1999.

Mechanizing the Development of Software

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
20 December 1998

Abstract. This paper presents a mechanizable framework for software development by refinement. The framework is based on a category of specifications. The key idea is representing knowledge about programming concepts, such as algorithm design, datatype refinement, and expression simplification, by means of taxonomies of specifications and morphisms. Examples are drawn from working systems Specware, Designware, and Planware.

Contents

1. Overview

A software system can be viewed as a composition of information from a variety of sources, including

- the application domain,
- the requirements on the system's behavior,
- software design knowledge about system architectures, algorithms, data structures, code optimization techniques, and
- the run-time hardware/software/physical environment in which the software will execute.

This paper presents a mechanizable framework for representing these various sources of information, and for composing them in the context of a refinement process. The framework is founded on a category of specifications. Morphisms are used to structure and parameterize specifications, and to refine them. Colimits are used to compose specifications. Diagrams are used to express the structure of large specifications, the refinement of specifications to code, and the application of design knowledge to a specification.

The framework features a collection of techniques for constructing refinements based on formal representations of programming knowledge. Abstract algorithmic concepts, datatype refinements, program optimization rules, software architectures, abstract user interfaces, and so on, are represented as diagrams of specifications and morphisms. We arrange these diagrams into taxonomies, which allow incremental access to and construction of refinements for particular requirement specifications. For example, a user may specify a scheduling problem and select a theory of global search algorithms from an algorithm library. The global search theory is used to construct a refinement of the scheduling problem specification into a specification containing a global search algorithm for the particular scheduling problem.

The framework is partially implemented in the research systems Specware, Designware, and Planware. Specware provides basic support for composing specifications and refinements, and generating code. Code generation in Specware is supported by inter-logic morphisms that translate between the specification language/logic and the logic of a particular programming language (e.g. CommonLisp or C++). Specware is intended to be general-purpose and has found use in industrial settings. Designware extends Specware with taxonomies of software design theories and support for constructing refinements from them. Planware provides highly automated support for requirements acquisition and synthesis of high-performance scheduling algorithms.

The remainder of this paper covers basic concepts and the key ideas of our approach to software development by refinement, in particular the concept of design by classification [9]. A simple example is carried through algorithm design, datatype refinement, and expression simplification. We also discuss the application of these techniques to domain-specific refinement in Planware [1].

This paper assumes a rudimentary knowledge of logic and category theory, although details about the categories of interest are introduced and motivated by examples.

spec $Container$ is
  sorts E, $Container$
  op $empty : \to Container$
  op $singleton : E \to Container$
  op $\_join\_ : Container, Container \to Container$
  constructors $\{empty, singleton, join\}$ construct $Container$

  axiom $\forall(x : Container)(x\ join\ empty = x\ \wedge\ empty\ join\ x = x)$
  op $\_in\_ : E, Container \to Boolean$
  definition of in is
    axiom $x\ in\ empty\ \ = false$
    axiom $x\ in\ singleton(y) = (x = y)$
    axiom $x\ in\ U\ join\ V = (x\ in\ U\ \vee\ x\ in\ V)$
  end-definition
end-spec

Figure 1: Specification for Containers

2. Basic Concepts

2.1. Specifications

A specification is the finite presentation of a theory. The signature of a specification provides the vocabulary for describing objects, operations, and properties in some domain of interest, and the axioms constrain the meaning of the symbols. The theory of the domain is the closure of the axioms under the rules of inference.

Example: Here is a specification for partial orders, using notation adapted from Specware. It introduces a sort E and an infix binary predicate on E, called le, which is constrained by the usual axioms. Although Specware allows higher-order specifications, first-order formulations are sufficient in this paper.

spec *Partial-Order* is
  sort E
  op $\_le\_ : E, E \to Boolean$
  axiom *reflexivity* is $x\ le\ x$
  axiom *transitivity* is $x\ le\ y\ \wedge\ y\ le\ z \implies x\ le\ z$
  axiom *antisymmetry* is $x\ le\ y\ \wedge\ y\ le\ x \implies x\ =\ z$
end-spec

Example: Containers are constructed by a binary join operator and they represent finite collections of elements of some sort E. The specification shown in Figure 1 includes a definition by means of axioms. Operators are required to be total. The constructor clause asserts that the operators $\{empty, singleton, join\}$ construct the sort *Container*, providing the basis for induction on *Container*.

The generic term *expression* will be used to refer to a term, formula, or sentence.

A model of a specification is a structure of sets and total functions that satisfy the axioms. However, for software development purposes we have a less well-defined notion of semantics in mind: each specification denotes a set of possible implementations in some computational model. Currently we regard these as functional programs. A denotational semantics maps these into classical models.

2.2. Morphisms

A specification morphism translates the language of one specification into the language of another specification, preserving the property of provability, so that any theorem in the source specification remains a theorem under translation.

A *specification morphism* $m : T \to T'$ is given by a map from the sort and operator symbols of the *domain* spec T to the symbols of the *codomain* spec T'. To be a specification morphism it is also required that every axiom of T translates to a theorem of T'. It then follows that a specification morphism translates theorems of the domain specification to theorems of the codomain.

Example: A specification morphism from *Partial-Order* to *Integer* is:

morphism $Partial\text{-}Order\text{-}to\text{-}Integer$ is
  $\{E \mapsto Integer,\ le \mapsto \leq\}$

Translation of an expression by a morphism is by straightforward application of the symbol map, so, for example, the $Partial\text{-}Order$ axiom $x\ le\ x$ translates to $x \leq x$. The three axioms of a partial order remain provable in *Integer* theory after translation.

Example: A parameterized specification can be treated as a morphism. The specification *Bag* in Appendix A can be parameterized on a spec *Triv* with a single sort:

spec $Triv$ is
  sort E
end-spec

via the morphism

morphism $Bag\text{-}Parameterization$ is
  $\{E \mapsto E\}$

The semantics of a specification morphism is given by a contravariant functor Mod that maps each specification to a category of models and each specification morphism $m : T \to T'$ to a reduct functor $\_|_m : Mod(T') \to Mod(T)$.

Morphisms come in a variety of flavors; here we only use two. An *extension* or *import* is an inclusion between specs.

Example: We can build up the theory of partial orders by importing the theory of preorders. The import morphism is $\{E \mapsto E,\ le \mapsto le\}$.

spec $PreOrder$
  sort E
  op $\_le\_ : E, E \to Boolean$

axiom *reflexivity* is $x \ le\ x$
axiom *transitivity* is $x \ le\ y \ \wedge \ y \ le\ z \implies x \ le\ z$
end-spec

spec $Partial\text{-}Order$
import $PreOrder$
axiom *antisymmetry* is $x \ le\ y \ \wedge \ y \ le\ x \implies x = z$
end-spec

A *definitional extension*, written $A \xrightarrow{d} B$, is an import morphism in which any new symbol in B also has an axiom that defines it. Definitions have implicit axioms for existence and uniqueness. Semantically, a definitional extension has the property that each model of the domain has a unique expansion to a model of the codomain.

Example: $Container$ can be formulated as a definitional extension of $Pre\text{-}Container$:

spec $Pre\text{-}Container$ is
sorts E, $Container$
op $empty : \to Container$
op $singleton : E \to Container$
op $\_join\_ : Container, Container \to Container$
constructors $\{empty, singleton, join\}$ construct $Container$
axiom $\forall (x : Container)(x \ join\ empty = x \ \wedge \ empty\ join\ x = x)$
end-spec

spec $Container$ is
imports $Pre\text{-}Container$
definition of *in* is
axiom $x \ in\ empty = false$
axiom $x \ in\ singleton(y) = (x = y)$
axiom $x \ in\ U \ join\ V = (x \ in\ U \ \vee \ x \ in\ V)$
end-definition
end-spec

2.3. The Category of Specs

Specification morphisms compose in a straightforward way as the composition of finite maps. It is easily checked that specifications and specification morphisms form a category SPEC. Colimits exist in SPEC and are easily computed. Suppose that we want to compute the colimit of $B \xleftarrow{i} A \xrightarrow{j} C$. First, form the disjoint union of all sort and operator symbols of A, B, and C, then define an equivalence relation on those symbols:

$$s \approx t \ \textit{iff}\ (i(s) = t \ \vee \ i(t) = s \ \vee \ j(s) = t \ \vee \ j(t) = s).$$

The signature of the colimit (also known as pushout in this case) is the collection of equivalence classes wrt $\approx$. The cocone morphisms take each symbol into its equivalence class. The axioms of the colimit are obtained by translating and collecting each axiom of A, B, and C.

Example: Suppose that we want to build up the theory of partial orders by composing simpler theories.

spec $BinRel$ is
  sort E
  op $\_le\_ : E, E \to Boolean$
  end-spec

$\longrightarrow$

spec $PreOrder$ is
  import $BinRel$
  axiom $reflexivity$ is $x\ le\ x$
  axiom $transitivity$ is
    $x\ le\ y \land y\ le\ z \implies x\ le\ z$
  end-spec

$\downarrow$

spec $Antisymmetry$ is
  import $BinRel$
  axiom $antisymmetry$ is
    $x\ le\ y \land y\ le\ x \implies x = z$
  end-spec

The pushout of $Antisymmetry \leftarrow BinRel \to PreOrder$ is isomorphic to the specification for *Partial-Order* in Section 2.1. In detail: the morphisms are $\{E \mapsto E,\ le \mapsto le\}$ from *BinRel* to both *PreOrder* and *Antisymmetry*. The equivalence classes are then $\{\{E, E, E\},\ \{le, le, le\}\}$, so the colimit spec has one sort (which we rename E), and one operator (which we rename le). Furthermore, the axioms of *BinRel*, *Antisymmetry*, and *PreOrder* are each translated to become the axioms of the colimit. Thus we have *Partial-Order*.

In the category of specifications, colimit acts as a kind of union operator: the pushout collects the symbols of *Antisymmetry* and *PreOrder* where the morphisms from *BinRel* indicate which symbols are shared. The colimit has the universal property that it computes the simplest such specification.

Example: The pushout operation is also used to instantiate the parameter in a parameterized specification. The binding of argument to parameter is represented by a morphism. To form a specification for bags of integers, we compute the pushout of $Bag \leftarrow Triv \to Integer$, where $Bag \leftarrow Triv$ is $\{E \mapsto E\}$, and $Triv \to Integer$ is $\{E \mapsto Integer\}$.

Example: A specification for sequences can be built up from *Container*, also via pushouts. We can regard *Container* as parameterized on a binary operator

spec $BinOp$ is
  sort E
  op $\_bop\_ : E, E \to E$
end-spec

morphism *Container-Parameterization* : $BinOp \to Container$ is
  $\{E \mapsto E,\ bop \mapsto join\}$

and we can define a refinement arrow that extends a binary operator to a semigroup:

spec *Associativity* is
  import *BinOp*
  axiom *Associativity* is $((x\ join\ y)\ join\ z) = (x\ join\ (y\ join\ z))$
end-spec

The pushout of $Associativity \leftarrow BinOp \rightarrow Container$, produces a collection specification with an associative join operator, which is *Proto-Seq*, the core of sequence theory in Appendix A.

2.4. Diagrams

Roughly, a *diagram* is a graph morphism to a category, usually the category of specifications in this paper. For example, the pushout described above started with a diagram comprised of two arrows:

$$\begin{tikzcd} BinRel \arrow[r] \arrow[d] & PreOrder \\ Antisymmetry & \end{tikzcd}$$

and computing the pushout of that diagram produces another diagram:

$$\begin{tikzcd} BinRel \arrow[r] \arrow[d] & PreOrder \arrow[d] \\ Antisymmetry \arrow[r] & Partial\text{-}Order \end{tikzcd}$$

A diagram *commutes* if the composition of arrows along two paths with the same start and finish node yields equal arrows.

2.4.1. The Structuring of Specifications

Colimits can be used to construct a large specification from a diagram of specs and morphisms. The morphisms express various relationships between specifications, including sharing of structure, inclusion of structure, and parametric structure. Several examples will appear later.

Example: The finest-grain way to compose *Partial-Order* is via the colimit of

$$\begin{tikzcd} & BinRel \arrow[dl] \arrow[d] \arrow[dr] & \\ Reflexivity & Transitivity & Antisymmetry \end{tikzcd}$$

Example: Appendix B gives a structured specification for sorting bags over a linear order.

2.4.2. Refinement and Diagrams

As described above, specification morphisms can be used to help *structure* a specification, but they can also be used to *refine* a specification. When a morphism is used as a refinement, the intended effect is to reduce the number of possible implementations when passing from the domain spec to the codomain. In this sense, a refinement can be viewed as embodying a particular design decision or property that corresponds to the subset of possible implementations of the domain spec which are also possible implementations of the codomain.

Often in software refinement we want to preserve and extend the structure of a structured specification (versus flattening it out via colimit). When a specification is structured as a diagram, then the corresponding notion of structured refinement is a diagram morphism. A *diagram morphism* M from diagram D to diagram E consists of a set of specification morphisms, one from each node/spec in D to a node in E such that certain squares commute (a functor underlies each diagram and a natural transformation underlies each diagram morphism). We use the notation $D \implies E$ for diagram morphisms.

Example: A datatype refinement that refines bags to sequences can be presented as the diagram morphism $BtoS : BAG \implies BAG\text{-}AS\text{-}SEQ$:

$$
\begin{tikzcd}
 & Bag \arrow[d, "BtoS_{Bag}"'] & Triv \arrow[l] \arrow[d, "BtoS_{Triv}"] \\
Seq \arrow[r] & Bag\text{-}as\text{-}Seq & Triv \arrow[l] \arrow[ll, bend left]
\end{tikzcd}
\qquad
\begin{tikzcd}
BAG \arrow[d, Rightarrow, "BtoS"'] \\
BAG\text{-}AS\text{-}SEQ
\end{tikzcd}
$$

where the domain and codomain of *BtoS* are shown in boxes, and the (one) square commutes. Here *Bag-as-Seq* is a definitional extension of *Seq* that provides an image for *Bag* theory. Specs for *Bag*, *Seq* and *Bag-as-Seq* and details of the refinement can be found in Appendix A. The interesting content is in spec morphism $BtoS_{Bag}$:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{morphism } BtoS_{Bag} : Bag \to Bag\text{-}as\text{-}Seq \text{ is}} \\
\{Bag & \mapsto & Bag\text{-}as\text{-}Seq, \\
empty\text{-}bag & \mapsto & bag\text{-}empty, \\
empty\text{-}bag? & \mapsto & bag\text{-}empty?, \\
nonempty? & \mapsto & bag\text{-}nonempty?, \\
singleton\text{-}bag & \mapsto & bag\text{-}singleton, \\
singleton\text{-}bag? & \mapsto & bag\text{-}singleton?, \\
nonsingleton\text{-}bag? & \mapsto & bag\text{-}nonsingleton?, \\
in & \mapsto & bag\text{-}in, \\
bag\text{-}union & \mapsto & bag\text{-}union, \\
bag\text{-}wfgt & \mapsto & bag\text{-}wfgt, \\
size & \mapsto & bag\text{-}size\}
\end{array}
$$

Diagram morphisms compose in a straightforward way based on spec morphism composition. It is easily checked that diagrams and diagram morphisms form a category. Colimits in this category can be computed using colimits in SPEC. In the sequel we will generally use the term refinement to mean a diagram morphism.

2.5. Logic Morphisms and Code Generation

Inter-logic morphisms [2] are used to translate specifications from the specification logic to the logic of a programming language. See [10] for more details. They are also useful for translating between the specification logic and the logic supported by various theorem-provers and analysis tools. They are also useful for translating between the theory libraries of various systems.

3. Software Development by Refinement

$$\begin{array}{c} S_0 \\ \Downarrow \\ S_1 \\ \Downarrow \\ S_2 \\ \Downarrow \\ \cdots \\ \Downarrow \\ S_n \\ \rightsquigarrow \\ Code \end{array}$$

The development of correct-by-construction code via a formal refinement process is shown to the left. The refinement process starts with a specification S_0 of the requirements on a desired software artifact. Each S_i, $i = 0, 1, ..., n$ represents a structured specification (diagram) and the arrows $\Downarrow$ are refinements (represented as diagram morphisms). The refinement from S_i to S_{i+1} embodies a design decision which cuts down the number of possible implementations. Finally an inter-logic morphism translates a low-level specification S_n to code in a programming language. Semantically the effect is to narrow down the set of possible implementations of S_n to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification S_0 (and proving its consistency).

Clearly, two key issues in supporting software development by refinement are: (1) how to construct specifications, and (2) how to construct refinements. Most of the sequel treats mechanizable techniques for constructing refinements.

3.1. Constructing Specifications

A specification-based development environment supplies tools for creating new specifications and morphisms, for structuring specs into diagrams, and for composing specifications via importation, parameterization, and colimit. In addition, a software development environment needs to support a large library of reusable specifications, typically including specs for (1) common datatypes, such as integer, sequences, finite sets, etc. and (2) common mathematical structures, such as partial orders, monoids, vector spaces, etc. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific theories, such as resource theories, or generic theories about domains such as satellite control or transportation.

3.2. Constructing Refinements

A refinement-based development environment supplies tools for creating new refinements. One of our innovations is showing how a library of abstract/reusable/generic refinements can be applied to produce refinements for a given specification. In this paper we focus mainly on refinements that embody design knowledge about (1) algorithm design, (2) datatype refinement, and (3) expression optimization. We believe that other types of design knowledge can be similarly expressed and exploited, including interface design, software architectures, domain-specific requirements capture (see Section 4. on Planware), and others. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific refinements.

The key concept of this work is the following: abstract design knowledge about datatype refinement, algorithm design, software architectures, program optimization rules, visualization displays, and so on, can be expressed as refinements (i.e. diagram morphisms). The domain of one such refinement represents the abstract structure that is required in a user's specification in order to apply the embodied design knowledge. The refinement itself embodies a design constraint – the effect is a reduction in the set of possible implementations. The codomain of the refinement contains new structures and definitions that are composed with the user's requirement specification.

$$\begin{tikzcd} A \arrow[r, Rightarrow] \arrow[d, Rightarrow] & S_0 \arrow[d, Rightarrow] \\ B \arrow[r, Rightarrow] & S_1 \end{tikzcd}$$

The figure to the left shows the application of a library refinement $A \implies B$ to a given (structured) specification S_0. First the library refinement is selected. The applicability of the refinement to S_0 is shown by constructing a *classification arrow* from A to S_0 which classifies S_0 as having A-structure by making explicit how S_0 has at least the structure of A. Finally the refinement is applied by computing the pushout in the category of diagrams. The creative work lies in constructing the classification arrow [7, 9].

3.3. Development of Sorting Algorithms

Our goal in this section is to show a simple example that refines a requirement spec to code by applying (1) an algorithm design refinement, (2) a datatype refinement, and (3) an expression optimization refinement. We step through the refinement of a specification for sorting a bag over an arbitrary linear order. A full specification for the import to *Sorting* is given in Appendix B.

spec $Sorting$ is
  import $Bag\text{-}Seq\text{-}over\text{-}LinOrd$
  op $sorted? : Bag, Seq \to Boolean$
  def $sorted?(x, z) = (ordered?(z) \ \wedge \ x = seq\text{-}to\text{-}bag(z))$
  op $sorting : Bag \to Seq$
  axiom $sorted?(x, sorting(x))$
end-spec

The following diagram serves as a roadmap of the design steps covered in the next three subsections. First a divide-and-conquer refinement is applied, then a datatype refinement (bags implemented as sequences), then an expression simplification refinement is applied, and so on. The squares represent the application of library refinements and the main line of refinement descends directly from *Sorting*.

$$\begin{tikzcd}
\mathit{Divide\text{-}and\text{-}Conquer}\text{-}0\text{-}1\text{-}2 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathit{Sorting} \arrow[d, Rightarrow] & \\
\mathit{Divide\text{-}and\text{-}Conquer}\text{-}0\text{-}1\text{-}2\text{-}\mathit{scheme} \arrow[r, Rightarrow] & \mathit{Sorting\text{-}Alg}_1 \arrow[d, Rightarrow] & \mathit{BAG} \arrow[l, Rightarrow] \arrow[d, Rightarrow] \\
\mathit{New\text{-}Expression} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathit{Sorting\text{-}Alg}_2 \arrow[d, Rightarrow] & \mathit{BAG\text{-}AS\text{-}SEQ} \arrow[l, Rightarrow] \\
\mathit{Equational\text{-}Simplification} \arrow[r, Rightarrow] & \mathit{Sorting\text{-}Alg}_3 \arrow[d, Rightarrow] & \\
 & \cdots &
\end{tikzcd}$$

3.3.1. Sorting: Algorithm Design

Before algorithm design can be applying we must have a well-defined problem to solve. *Problem-Theory* in Figure 2 expresses the abstract structure of a problem: given input datum $x : D$, find a feasible solution $z : R$ satisfying the problem requirement constraint: $O(x, z)$. We call the input datum x a problem instance; note that *Problem-Theory* is intended to specify a function.

Suppose that the user first decides to apply a generic refinement for divide-and-conquer algorithms.

$$\begin{tikzcd}
\mathit{empty\text{-}seq} \arrow[r, mapsto, "sort"] \arrow[d, "\mathit{empty\text{-}seq}^{-1}"'] & \mathit{empty\text{-}seq} \\
\langle\rangle \arrow[r, mapsto, "id"] & \langle\rangle \arrow[u, "\mathit{empty\text{-}seq}"']
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{singleton\text{-}seq}(e) \arrow[r, mapsto, "sort"] \arrow[d, "\mathit{singleton\text{-}seq}^{-1}"'] & \mathit{singleton\text{-}seq}(e) \\
e \arrow[r, mapsto, "id"] & e \arrow[u, "\mathit{singleton\text{-}seq}"']
\end{tikzcd}$$

$$\begin{tikzcd}
x_0 \arrow[r, mapsto, "sorting"] \arrow[d, "concat^{-1}"'] & z_0 \\
\langle x_1, x_2\rangle \arrow[r, mapsto, "sorting \times sorting"] & \langle z_1, z_2\rangle \arrow[u, "merge"']
\end{tikzcd}$$

The principle of divide-and-conquer is to solve small problem instances by some direct means, and to solve larger problem instances by decomposing them, solving the pieces, and composing the resulting solutions. The figure to the left shows a mergesort as generalized kind of divide-and-conquer with three cases. The bottommost square shows the familiar divide-and-conquer case: the input x_0 is decomposed into two subproblem instances, x_1 and x_2, which are solved to produce z_1 and z_2, which are in turn composed to form solution z_0. The decomposition operator here is loosely specified as the inverse of a constructor, *concat*.

The first two cases also have the decompose-solve-compose pattern where the decomposition operator is the inverse of a constructor. The datum $\langle\rangle$ is the 0-tuple and it is treated as the output of the inverse of a constant constructor; $\langle\rangle$ is the sole element of the empty product sort, denoted *Unit*. The first two cases differ from the third in that the input decomposes into a subproblem that is easily solved, by the identity function on the appropriate sort.

The domain spec of a simple divide-and-conquer refinement is given in Figure 2 that abstracts from the mergesort example. It supposes three cases (indexed by 0, 1, and 2), each based on the signature of a constructor, similar to a homomorphism between algebras with three constructors. The abstraction lies in treating each operator in the example as a problem theory. Subscripted O's denote problem requirement constraints; e.g. $O_{Decompose-i}$ specifies the decomposition operator in case i, and so on. The discriminator p_i represents the condition under which $O_{Decompose-i}$ can decompose an input. The Soundness-2 axiom relates O, $O_{Decompose2}$, and $O_{Compose2}$. It 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. Similar comments hold for the other Soundness axioms. The operator $>$ is a well-founded order on D to assure termination (axioms are omitted for simplicity).

A general scheme for problem reduction theories (including divide-and-conquer) is given in [6]. The idea is to have a different first-order divide-and-conquer theory for each possible abstract constructor signature.

The codomain spec of the divide-and-conquer refinement (Figure 3) contains a schematic definition for the top-level divide-and-conquer functions and schematic requirement specifications for subalgorithms C_0, C_1, and C_2. Given any morphism from *Divide-and-Conquer-0-1-2* and functions that satisfy the requirement specifications for the subalgorithms, the corresponding instance of the divide-and-conquer function satisfies its requirement specification [4].

There is one complication which needs explaining. Specware requires that all functions be total and deterministic; each operation in a spec has implicit axioms for existence and uniqueness of solutions. This simplifies the logic of Specware, but complicates the treatment of nondeterministic functions which arise naturally in refinement, due to the levels of abstraction. Consider, for example, an operation that decomposes a bag b into an element a and the remainder of the bag b'; i.e. an inverse of $insert\text{-}bag(a, b')$. If the bags are ultimately implemented as lists, then it is natural to use *first* and *rest* to perform the decomposition. However consider two equal bags {1,1,2}= {2,1,1}. If {1,1,2} is represented by [1,1,2] then it will decompose into 1 and {1,2}. If {2,1,1} is represented by [2,1,1], then it will decompose into 2 and {1,1}. Hence the decomposition is not functional on bags, even though it is functional on lists.

There are various approaches to this problem. We observe however that any extra work required to make a naturally nondeterministic operator into a deterministic one is often unnecessary from the programming point of view, when the nondeterminism is of the don't-care variety – any solution will do, so any extra work to force one unique solution is a constraint posed by the programming system rather than by the software requirements. One solution to this problem is to treat operations that may need to be nondeterministic via existential quantification. In place of decomposition operators in Figure 3we have existentially quantified requirement constraints .

The development of a divide-and-conquer algorithm for sorting begins with the con-

spec *Problem-Theory* is
  sorts D, R
  op $O : D, R \to Boolean$
end-spec

spec *Divide-and-Conquer*-0-1-2 is
  import *DRO*
  sort E
  op $F : D \to R$
  op $\_ > \_ : D, D \to Boolean$
  axioms $>$ is a well-founded order ...

  op $p_0 : D \to Boolean$
  op $O_{Decompose0} : D, Unit \to Boolean$
  op $O_{Compose0} : R, Unit \to Boolean$
  axiom *Soundness*-0 is
    $O_{Decompose0}(x, \langle\rangle) \ \wedge \ O_{Compose0}(z, \langle\rangle) \implies O(x, z)$
  axiom *discriminator-of-Decompose*0 is
    $p_0(x) \implies O_{Decompose0}(x, \langle\rangle)$

  op $p_1 : D \to Boolean$
  op $O_{Decompose1} : D, E \to Boolean$
  op $O_{Compose1} : R, E \to Boolean$
  axiom *Soundness*-1 is
    $O_{Decompose1}(x, e) \ \wedge \ O_{Compose1}(z, e) \implies O(x, z)$
  axiom *discriminator-of-Decompose*1 is
    $p_1(x) \implies \exists(e)\, O_{Decompose1}(x, e)$

  op $p_2 : D \to Boolean$
  op $O_{Decompose2} : D, D, D \to Boolean$
  op $O_{Compose2} : R, R, R \to Boolean$
  axiom *Soundness*-2 is
    $O_{Decompose2}(x_0, x_1, x_2) \ \wedge \ O(x_1, z_1) \ \wedge \ O(x_2, z_2) \ \wedge \ O_{Compose2}(z_0, z_1, z_2)$
      $\implies O(x_0, z_0)$
  axiom *discriminator-of-Decompose*2 is
    $p_2(x) \implies \exists(x_1, x_2)\, O_{Decompose2}(x, x_1, x_2) \ \wedge \ x > x_1 \ \wedge \ x > x_2$

  axiom $\forall(x : D)\ p_0(x)\ xor\ p_1(x)\ xor\ p_2(x)$
end-spec

Figure 2: A Simple Divide-and-Conquer Algorithm Theory

spec *Divide-and-Conquer*-0-1-2-*scheme* is
  import *Divide-and-Conquer*-0-1-2
  op $C_0 :\rightarrow R$
  axiom $O_{Compose0}(C_0, \langle\rangle)$

  op $C_1 : E \rightarrow R$
  axiom $O_{Compose1}(C_1(e), e)$

  op $C_2 : R, R \rightarrow R$
  axiom $O_{Compose2}(C_2(x_1, x_2), x_1, x_2)$

  definition of F is
    axiom $p_0(x) \implies O_{Decompose0}(x, \langle\rangle) \wedge F(x) = C_0$
    axiom $p_1(x) \implies \exists(e)(O_{Decompose1}(x, e) \wedge F(x) = C_1(e))$
    axiom $p_2(x) \implies \exists(x_1, x_2)(O_{Decompose2}(x, x_1, x_2)$
      $\wedge F(x) = C_2(F(x_1), F(x_2)))$
  end-definition
  theorem $O(x, F(x))$
end-spec

Figure 3: Parameterized Divide-and-Conquer Algorithm

struction of a morphism from *Problem-Theory* to *Sorting* theory:

$$
\begin{array}{ccc}
D & \mapsto & Bag \\
R & \mapsto & Seq \\
O & \mapsto & sorted?
\end{array}
$$

Since the morphism from *Problem-Theory* to *Divide-and-Conquer* is an inclusion, we can use straightforward propagation to obtain translations for the components of *Problem-Theory* in *Divide-and-Conquer*:

$$
\begin{array}{rcl}
D & \mapsto & Bag \\
R & \mapsto & Seq \\
O & \mapsto & sorted? \\
F & \mapsto & ? \\
E & \mapsto & ? \\
> & \mapsto & ? \\
p_0 & \mapsto & ? \\
O_{Decompose0} & \mapsto & ? \\
O_{Compose0} & \mapsto & ? \\
\ldots & &
\end{array}
$$

To complete the classification arrow we attempt to translate the remaining operators into expressions of *Sorting*. Alternative translations give rise to different sorting algorithms. There are several ways to proceed. One approach is based on the choice of a set of standard decomposition operators from a library. The tactic then uses unskolemization on the soundness axioms to derive specifications for the composition operators. This approach allows the derivation of insertion sort, mergesort, and various parallel sorting algorithms [4, 8]. A dual approach is to choose a set of standard composition operators from a library and use the soundness axioms to derive the decomposition operators (leading to selections sort, heapsort, and quicksort). We present the first approach in detail, then sketch the second.

Suppose that we choose the constructor set $\{empty\text{-}bag, singleton\text{-}bag, bag\text{-}union\}$ as the basis for the decomposition relation on the input domain Bag. This gives us the partial signature morphism

$$
\begin{array}{rcl}
D & \mapsto & Bag \\
R & \mapsto & Seq \\
O & \mapsto & sorted? \\
F & \mapsto & sorting \\
E & \mapsto & E \\
> & \mapsto & bag\text{-}wfgt \\
p_0 & \mapsto & empty\text{-}bag? \\
O_{Decompose0} & \mapsto & \lambda(x)\ x = empty\text{-}bag \\
O_{Compose0} & \mapsto & ? \\
p_1 & \mapsto & singleton\text{-}bag? \\
O_{Decompose1} & \mapsto & \lambda(x, e)\ x = singleton\text{-}bag(e) \\
O_{Compose1} & \mapsto & ? \\
p_2 & \mapsto & nonsingleton\text{-}bag? \\
O_{Decompose2} & \mapsto & \lambda(x_0, x_1, x_2)\ x_0 = bag\text{-}union(x_1, x_2) \\
O_{Compose2} & \mapsto & ?
\end{array}
$$

The soundness axiom

$$
\begin{array}{l}
\forall(x_0, x_1, x_2 : D)\ \forall(z_0, z_1, z_2 : R) \\
\quad (O_{Decompose2}(x_0, x_1, x_2)\ \wedge\ O(x_1, z_1)\ \wedge\ O(x_2, z_2)\ \wedge\ O_{Compose2}(z_0, z_1, z_2) \\
\qquad \implies O(x_0, z_0))
\end{array}
$$

cannot be translated into *Sorting* because $O_{Compose2}$ has no translation yet. However, a technique called *unskolemization* allows us to use inference tools to deduce a suitable translation for $O_{Compose2}$ [7]. Unskolemizing operator symbol $O_{Compose2}$ replaces the occurrence of $O_{Compose2}$ by a fresh existentially quantified variable in the scope of the quantifiers for z_0, z_1, z_2:

$$
\begin{array}{l}
\forall(z_0, z_1, z_2 : R)\ \exists(y : Boolean)\ \forall(x_0, x_1, x_2 : D) \\
\quad (O_{Decompose2}(x_0, x_1, x_2)\ \wedge\ O(x_1, z_1)\ \wedge\ O(x_2, z_2)\ \wedge\ y \implies O(x_0, z_0)).
\end{array}
$$

This formula has the same satisfiability properties as the original and it can be translated to *Sorting* via the partial morphism yielding:

$$
\begin{array}{l}
\forall(z_0, z_1, z_2 : Seq)\ \exists(y : Boolean)\ \forall(x_0, x_1, x_2 : Bag) \\
\quad (x_0 = bag\text{-}union(x_1, x_2)\ \wedge\ sorted?(x_1, z_1)\ \wedge\ sorted?(x_2, z_2)\ \wedge\ y \\
\quad \implies sorted?(x_0, z_0))
\end{array}
$$

A straightforward proof of this formula in *Sorting* finds a witness for y which results in a translation for $O_{Compose2}$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{sorted?(x_0, z_0)} \\
= & \qquad \text{by def of } sorted? \\
 & ordered(z_0) \ \wedge \ x_0 = seq\text{-}to\text{-}bag(z_0) \\
= & \qquad \text{by assumption } x_0 = bag\text{-}union(x_1, x_2) \\
 & ordered(z_0) \ \wedge \ bag\text{-}union(x_1, x_2) = seq\text{-}to\text{-}bag(z_0) \\
= & \qquad \text{by assumption } x_i = seq\text{-}to\text{-}bag(z_i), \ i = 1, 2 \\
 & ordered(z_0) \ \wedge \ bag\text{-}union(seq\text{-}to\text{-}bag(z_1), seq\text{-}to\text{-}bag(z_2)) = seq\text{-}to\text{-}bag(z_0).
\end{array}
$$

which can then unify with the assumption y, which must be a term over variables z_0, z_1, z_2. More generally, we pick up as part of the witness any assumptions that are expressed over the variables z_0, z_1, z_2, in this case yielding

$$
O_{Compose2} \ \mapsto \ \begin{cases} \lambda \ (z_0, z_1, z_2) \ ordered(z_1) \ \wedge \ ordered(z_2) \\ \Longrightarrow \ ordered(z_0) \\ \wedge \ bag\text{-}union(seq\text{-}to\text{-}bag(z_1), seq\text{-}to\text{-}bag(z_2)) = seq\text{-}to\text{-}bag(z_0). \end{cases}
$$

This is, of course, a specification for a merge operation. If we take this as the translation of $O_{Compose2}$, then we know that the Soundness-2 axiom translates to a theorem in *Sorting* by construction.

The remaining steps in constructing this classification arrow are similar. We unskolemize $O_{Compose1}$ in Soundness-1 and then derive a witness:

$$
\begin{array}{ll}
\multicolumn{2}{l}{sorted?(x, z)} \\
= & \qquad \text{by def of } sorted? \\
 & ordered(z) \ \wedge \ x = seq\text{-}to\text{-}bag(z) \\
= & \qquad \text{using assumption } x = singleton\text{-}bag(e) \\
 & ordered(z) \ \wedge \ singleton\text{-}bag(e) = seq\text{-}to\text{-}bag(z) \\
= & \qquad Bag\text{-}and\text{-}Seq\text{-}Conv \text{ axiom} \\
 & ordered(z) \ \wedge \ seq\text{-}to\text{-}bag(singleton\text{-}seq(e)) = seq\text{-}to\text{-}bag(z) \\
\Longleftarrow & \qquad \text{Leibnitz: equality is a congruence} \\
 & ordered(z) \ \wedge \ singleton\text{-}seq(e) = z \\
= & \qquad \text{simplifying using } ordered(singleton\text{-}seq(e)) = true \\
 & singleton\text{-}seq(e) = z.
\end{array}
$$

yielding the translation

$$
O_{Compose1} \ \mapsto \ \lambda(z, e) \ singleton\text{-}seq(e) = z.
$$

We can unskolemize $O_{Compose0}$ in Soundness-0 and then derive a witness:

$$
\begin{array}{ll}
\multicolumn{2}{l}{sorted?(x, z)} \\
= & \qquad \text{def of } sorted? \\
 & ordered(z) \ \wedge \ x = seq\text{-}to\text{-}bag(z) \\
= & \qquad \text{assumption } x = empty\text{-}bag
\end{array}
$$

$$
\begin{array}{ll}
 & ordered(z) \ \wedge \ empty\text{-}bag = seq\text{-}to\text{-}bag(z) \\
= & \qquad \textit{Bag-and-Seq-Conv} \text{ axiom} \\
 & ordered(z) \ \wedge \ seq\text{-}to\text{-}bag(empty\text{-}seq) = seq\text{-}to\text{-}bag(z) \\
\Longleftarrow & \qquad \text{Leibnitz: equality is a congruence} \\
 & ordered(z) \ \wedge \ empty\text{-}seq = z \\
= & \qquad \text{simplifying using } \textit{Sorting} \text{ axiom } ordered(empty\text{-}seq) = true \\
 & empty\text{-}seq = z.
\end{array}
$$

yielding the translation

$$O_{Compose0} \ \mapsto \ \lambda(z) \ empty\text{-}seq = z.$$

The classification arrow is now complete:

$$
\begin{array}{rcl}
D & \mapsto & Bag \\
R & \mapsto & Seq \\
O & \mapsto & sorted? \\
F & \mapsto & sorting \\
E & \mapsto & E \\
> & \mapsto & bag\text{-}wfgt \\
p_0 & \mapsto & empty\text{-}bag? \\
O_{Decompose0} & \mapsto & \lambda(x) \ x = empty\text{-}bag \\
O_{Compose0} & \mapsto & \lambda(z) \ z = empty\text{-}seq \\
p_1 & \mapsto & singleton\text{-}bag? \\
O_{Decompose1} & \mapsto & \lambda(x, e) \ x = singleton\text{-}bag(e) \\
O_{Compose1} & \mapsto & \lambda(z, e) \ z = singleton\text{-}seq(e) \\
p_2 & \mapsto & nonsingleton\text{-}bag? \\
O_{Decompose2} & \mapsto & \lambda(x_0, x_1, x_2) \ x_0 = bag\text{-}union(x_1, x_2) \\
O_{Compose2} & \mapsto & \lambda \ (z_0, z_1, z_2) \ ordered(z_1) \ \wedge \ ordered(z_2) \implies ordered(z_0) \\
 & & \ \wedge \ seq\text{-}to\text{-}bag(z_0) = bag\text{-}union(seq\text{-}to\text{-}bag(z_1), seq\text{-}to\text{-}bag(z_2))
\end{array}
$$

Note that we have a classification arrow that translates a symbol to an expression (rather than a symbol). This commonly occurring situation is treated by extending the codomain diagram with definitional extensions as necessary. In the sequel we will assume this treatment whenver the need for a symbol-to-expression translation arises.

We can compute the pushout

$$
\begin{tikzcd}
\textit{Divide-and-Conquer}\text{-0-1-2} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \textit{Sorting} \arrow[d, Rightarrow] \\
\textit{Divide-and-Conquer}\text{-0-1-2-}\textit{scheme} \arrow[r, Rightarrow] & \textit{Sorting-Alg}_1
\end{tikzcd}
$$

to obtain a refinement of *Sorting* that contains a definition for a mergesort algorithm, shown in Figure 4.

Example – sketch of a derivation for Quicksort:

spec $Sorting\text{-}Alg_1$ is
  import $Bag\text{-}Seq\text{-}over\text{-}LinOrd$

  op $sorted? : Bag, Seq \to Boolean$
  definition $sorted?(x,z) = (ordered?(z) \ \wedge \ x = seq\text{-}to\text{-}bag(z))$

  op $sorting : Bag \to Seq$
  theorem $sorted?(x, sorting(x))$
  definition of $sorting$ is
    axiom $empty\text{-}bag?(x) \implies (x = empty\text{-}bag \ \wedge \ sorting(x) = C_0)$
    axiom $singleton\text{-}bag?(x) \implies \exists(e)(x = singleton\text{-}bag(e) \ \wedge \ sorting(x) = C_1(e))$
    axiom $nonsingleton\text{-}bag?(x)$
      $\implies \exists(x_1, x_2)(x = bag\text{-}union(x_1, x_2)$
        $\wedge \ sorting(x) = C_2(sorting(x_1), sorting(x_2)))$
  end-definition

  op $C_0 : \to Seq$
  axiom $C_0 = empty\text{-}seq$

  op $C_1 : E \to Seq$
  axiom $C_1(e) = singleton(e)$

  op $C_2 : Seq, Seq \to Seq$
  axiom $ordered?(z_1) \ \wedge \ ordered?(z_2)$
    $\implies (ordered?(C_2(z_1, z_2))$
      $\wedge \ seq\text{-}to\text{-}bag(C_2(z_1, z_2)) = bag\text{-}union(seq\text{-}to\text{-}bag(z_1), seq\text{-}to\text{-}bag(z_2)))$
  end-spec

Figure 4: Divide-and-Conquer Sorting Algorithm

The derivation of a variant of quicksort is dual to the derivation for mergesort: rather than select a simple set of constructors as the basis for the decomposition operations, we select a simple set of constructors as a basis for the composition operators. In particular if we return to the stage of filling in the partial morphism

$$\begin{array}{rcl}
D & \mapsto & Bag \\
R & \mapsto & Seq \\
O & \mapsto & sorted? \\
F & \mapsto & sorting \\
E & \mapsto & E \\
> & \mapsto & bag\text{-}wfgt \\
p_0 & \mapsto & ? \\
O_{Decompose0} & \mapsto & ? \\
O_{Compose0} & \mapsto & ? \\
\ldots & &
\end{array}$$

and we choose the set of sequence constructors $\{empty\text{-}seq, singleton\text{-}seq, concat\}$ as a basis for composition, we get

$$\begin{array}{rcl}
D & \mapsto & Bag \\
R & \mapsto & Seq \\
O & \mapsto & sorted? \\
F & \mapsto & sorting \\
E & \mapsto & E \\
> & \mapsto & bag\text{-}wfgt \\
p_0 & \mapsto & ? \\
O_{Decompose0} & \mapsto & ? \\
O_{Compose0} & \mapsto & \lambda(z)\ z = empty\text{-}seq \\
p_1 & \mapsto & ? \\
O_{Decompose1} & \mapsto & ? \\
O_{Compose1} & \mapsto & \lambda(z, e)\ z = singleton\text{-}seq(e) \\
p_2 & \mapsto & ? \\
O_{Decompose2} & \mapsto & ? \\
O_{Compose2} & \mapsto & \lambda(z_0, z_1, z_2)\ z_0 = concat(z_1, z_2)
\end{array}$$

Now we use the Soundness axioms to deduce translations for the decomposition requirement constraints. The most interesting case is unskolemizing operator symbol $O_{Decompose2}$ in axiom Soundness-2:

$$\begin{array}{l}
\forall(x_0, x_1, x_2 : D)\ \exists(y : Boolean)\ \forall(z_0, z_1, z_2 : R) \\
\quad (y\ \wedge\ O(x_1, z_1)\ \wedge\ O(x_2, z_2)\ \wedge\ O_{Compose2}(z_0, z_1, z_2) \\
\quad\quad \Longrightarrow\ O(x_0, z_0)).
\end{array}$$

This formula can be translated via the partial morphism yielding:

$$\begin{array}{l}
\forall(x_0, x_1, x_2 : Bag)\ \exists(y : Boolean)\ \forall(z_0, z_1, z_2 : Seq) \\
\quad (y\ \wedge\ sorted?(x_1, z_1)\ \wedge\ sorted?(x_2, z_2)\ \wedge\ z_0 = concat(z_1, z_2) \\
\quad\quad \Longrightarrow\ sorted?(x_0, z_0))
\end{array}$$

A straightforward proof of this formula in *Sorting*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{sorted?}(x_0, z_0)} \\
= & \qquad \text{by def of } \mathit{sorted?} \\
 & \mathit{ordered}(z_0) \ \wedge \ x_0 = \mathit{seq\text{-}to\text{-}bag}(z_0) \\
= & \qquad \text{by assumption } z_0 = \mathit{concat}(z_1, z_2) \\
 & \mathit{ordered}(\mathit{concat}(z_1, z_2)) \ \wedge \ x_0 = \mathit{seq\text{-}to\text{-}bag}(\mathit{concat}(z_1, z_2)) \\
= & \qquad \text{distributing using axioms from } \mathit{Sorting} \text{ and } \mathit{Bag\text{-}and\text{-}Seq\text{-}Conv} \\
 & \mathit{ordered}(z_1) \ \wedge \ \mathit{all\text{-}le}(z_1, z_2) \ \wedge \ \mathit{ordered}(z_2) \\
 & \wedge \ x_0 = \mathit{bag\text{-}union}(\mathit{seq\text{-}to\text{-}bag}(z_1), \mathit{seq\text{-}to\text{-}bag}(z_2)) \\
= & \qquad \text{simplifying using } \mathit{seq\text{-}to\text{-}bag}(z_i) = x_i, \ i = 1, 2 \\
 & \mathit{all\text{-}le}(x_1, x_2) \ \wedge \ x_0 = \mathit{bag\text{-}union}(x_1, x_2)
\end{array}
$$

results in a translation for $O_{Decompose2}$:

$$O_{Decompose2} \ \mapsto \ \lambda(x_0, x_1, x_2) \ \mathit{all\text{-}le}(x_1, x_2) \ \wedge \ x_0 = \mathit{bag\text{-}union}(x_1, x_2)$$

That is, each element of bag x_1 is less-than-or-equal to each element of bag x_2 and x_0 decomposes into two exhaustive subbags of itself. This is, of course, a specification for a partition operation in a quicksort. If we take this as the translation of $O_{Decompose2}$, then we know that the soundness axiom translates to a theorem in *Sorting-theory* by construction. The other two cases are similar to the base cases in mergesort.

The divide-and-conquer theory allows a pleasing derivational symmetry between algorithms with simple decomposition operations and complex composition operations and dually, algorithms with simple composition operations and complex decomposition operations.

3.3.2. Sorting: Datatype Refinement

$$
\begin{tikzcd}
\mathit{BAG} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mathit{Sorting} \\
\mathit{BAG\text{-}AS\text{-}SEQ} &
\end{tikzcd}
$$

Next, suppose that the user decides to bring the mergesort algorithm closer to implementation by refining bags into sequences. This datatype refinement was discussed in Section 2.4.2. and it is presented in detail in Appendix A. The classification arrow from BAG to *Sorting* is essentially an inclusion, comprised of the identity morphisms from BAG specs to corresponding specs in the diagram of the *Sorting* spec.

The classification arrow together with the pushout is shown in Figure 5. The two cocone arrows in the figure are shown with dashes to help distinguish them. Also, to save space the figure abbreviates *Bag-as-Seq* to *BaS*.

The pushout gives us a refinement of *Sorting*, shown in Figure 6. The effect on the mergesort definition arises mainly from the translations of the $BtoS_{Bag}$ morphism in Section 2.4.2.. Note that not all bag operators are directly in the image of the classification arrow. In particular, the operators *all-le* and *seq-to-bag* are extensions to bag theory added during the construction of sorting theory. The fact that they are defined however

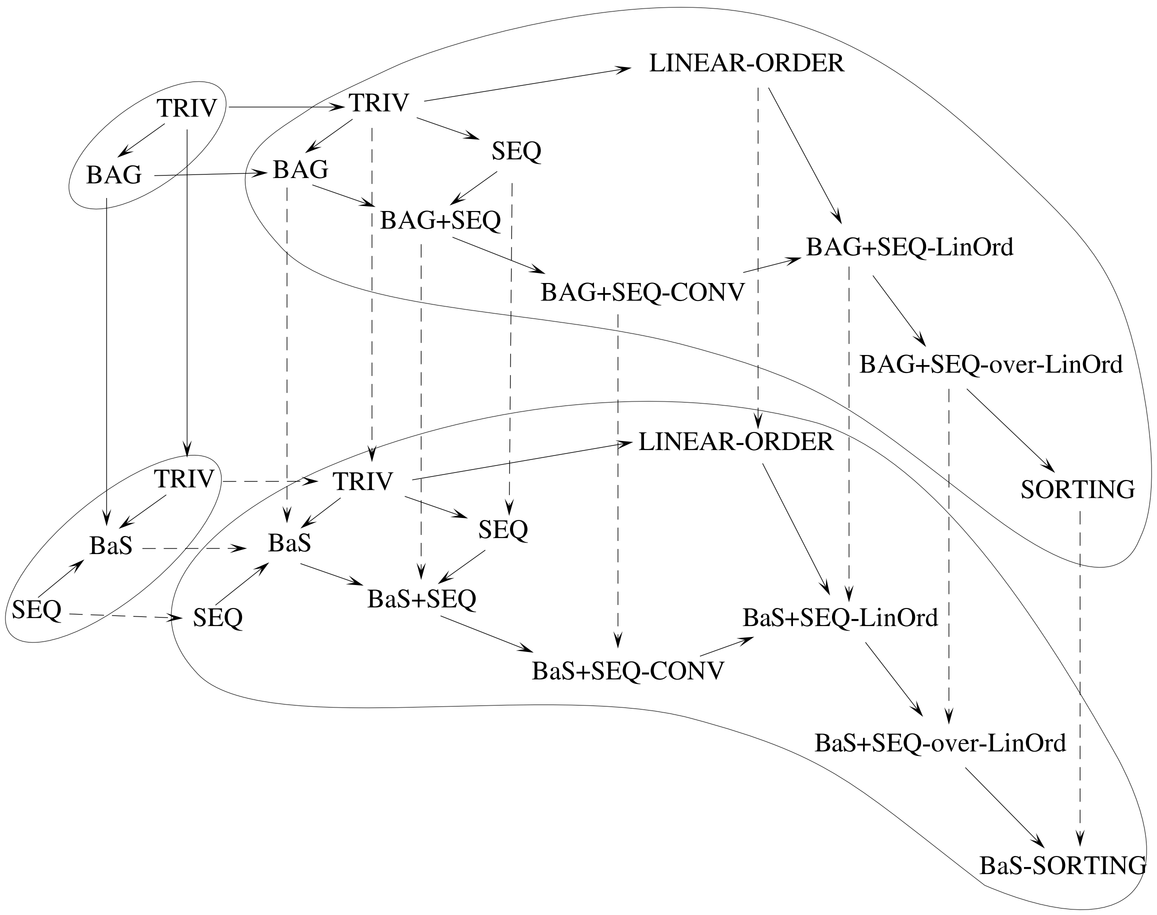

Figure 5: Refining Bags to Seqs in Sorting

means that the pushout will translate the definitions too, so they remain defined after translation. If some of the auxiliary sorts or operators on the datatype are not in the image of the classification arrow, then the pushout will simply translate whatever constraints there are on such sorts and operators. If the constraints are definitions, then the translation will be definitional, otherwise the symbols will be constrained but not necessarily uniquely, and further refinement will be required later.

This datatype refinement mainly serves to bring the spec closer to the programming language level. Other datatype refinements have a more dramatic effect by refining to a complicated data structure that efficiently implements the more abstract type. The same machinery applies. Leverage on the programming task will be most apparent when a relatively small domain specification refines to a relatively large codomain spec – the extra bits of information in the codomain are added to the user's spec essentially for free once the classification arrow has been constructed. Some examples would include Boolean expressions refining to BDDs, and sets over a fixed finite universe refining to bit vectors or hash tables, and finite relations refining to B-trees.

spec $Sorting\text{-}Alg_2$ is
  import $Bag\text{-}as\text{-}Seq\text{-}Seq\text{-}over\text{-}LinOrd$
  op $sorted? : Bag\text{-}as\text{-}Seq, Seq \rightarrow Boolean$
  def $sorted?(x, z) = (ordered?(z) \ \wedge \ x = seq\text{-}to\text{-}bag(z))$

  op $sorting : Bag\text{-}as\text{-}Seq \rightarrow Seq$
  theorem $sorted?(x, sorting(x))$
  definition of $sorting$ is
    axiom $bag\text{-}empty?(x) \implies (x = bag\text{-}empty \ \wedge \ sorting(x) = C_0)$
    axiom $bag\text{-}singleton?(x) \implies \exists(e)(x = bag\text{-}singleton(e) \ \wedge \ sorting(x) = C_1(e))$
    axiom $bag\text{-}nonsingleton?(x)$
      $\implies \exists(x_1, x_2)(x = bag\text{-}union(x_1, x_2)$
        $\wedge \ sorting(x_0) = C_2(sorting(x_1), sorting(x_2)))$
  end-definition

  op $C_0 :\rightarrow Seq$
  axiom $C_0 = empty\text{-}seq$

  op $C_1 : E \rightarrow Seq$
  axiom $C_1(e) = singleton(e)$

  op $C_2 : Seq, Seq \rightarrow Seq$
  axiom $ordered?(z_1) \ \wedge \ ordered?(z_2)$
    $\implies (ordered?(C_2(z_1, z_2))$
      $\wedge \ seq\text{-}to\text{-}bag(C_2(z_1, z_2)) = bag\text{-}union(seq\text{-}to\text{-}bag(z_1), seq\text{-}to\text{-}bag(z_2)))$
end-spec

Figure 6: Sorting Algorithm after Datatype Refinement

3.3.3. Sorting: Expression Optimization

After performing algorithm design and datatype refinement, there are often opportunities for optimizing various subexpressions. Transformation rules and optimization tactics, such as equational rewriting, context-dependent simplification [5], finite differencing [3], partial evaluation, and other more specialized techniques can result in dramatic improvements in the complexity of the final implementation. We show how at least some optimization techniques can be captured as refinements, and can be applied in the same manner as refinements that express algorithm design and datatype refinement knowledge.

The general approach to representing expression optimization techniques is a two stage process: first, an equation is deduced with the expression-to-be-simplified on the left-hand-side, and second, the spec S is reformulated to exploit the new theorem. The applicable metatheorem here is: if spec $S1$ has theorem $e1 = e2$ and $S2$ is the same spec as $S1$ except that some occurrences of $e1$ are replaced by $e2$, then $S1$ and $S2$ are isomorphic specifications. The metatheorem justifies a substitution operator on specs that creates a spec and the isomorphism between them.

Equational rewriting is a basic optimization technique that uses equational inference in the user's spec to deduce an equal term. However, to determine which side is simpler requires a metric. The approach used in KIDS [5] and in Specware/Designware is to assign a weight to each operator, measuring the complexity of an expression by summing the weights of each operator occurrence.

Here is a refinement to express equational simplification:

spec *Expression* is
  sorts D, Q
  op $expr : D \to Q$
end-spec

spec *New-Expression* is
  import *Expression*
  op $new\text{-}expr : D \to Q$
  axiom $expr(x) = new\text{-}expr(x)$
end-spec

spec *Equational-Simplification* is
  import *New-Expression*
  theorem $expr(x) = new\text{-}expr(x)$
end-spec

Equational–Simplification (*ES*) is a degenerate case of our approach to representing expression optimization techniques. The presence of the theorem in ES is required in order that it be included in the pushout spec when the refinement is applied.

Context-dependent simplification is a generalization of equational simplification. Suppose that we wish to simplify expression *expr* in some context C within spec S. To simplify *expr*, we want to use not only the equations and other theorems of the spec S, but also whatever contextual properties hold when *expr* is evaluated. For example, when simplifying the else-branch of a conditional, we can assume the negation of the test of the conditional.

spec *Expression-and-Context* is
  import *Expression*
  op $C : D \to Boolean$
  op $new\text{-}expr : D \to Q$
  axiom $C(x) \implies expr(x) = new\text{-}expr(x)$
  end-spec

spec *Context-Dependent-Simplification* is
  import *Expression-and-Context*
  theorem $C(x) \implies expr(x) = new\text{-}expr(x)$
end-spec

Finite Differencing [3] is a less basic optimization and it has nondegenerate structure in the codomain of its represention as a refinement. The idea in finite differencing is to replace an expensive expression in a loop by incremental computation, making use of some extra variables. The following refinement expresses the correspondence between these concepts and the symbols in the refinement: e is the expensive expression, which may occur in the definition of operator f; h is the body of the definition of f. g is a new operator which is an abstraction of f with an extra parameter c that is maintained to satisfy the invariant $c = e(x)$. The theorem asserts equality between f and g when g is invoked with an argument that satisfies the invariant. After developing the classification arrow and pushing out, the substitution (meta)operator replaces occurrences of f with g. To realize the implicit incremental benefits of this change, context-dependent simplification must be applied to exploit the local invariant $c = e(x)$.

spec *Expression-and-Function* is
  import *Expression*
  sort R
  op $f : D \to R$
  op $h : D \to R$
  def $f(x) = h(x)$
  end-spec

spec *Abstracted-Op* is
  import *Expression-and-Function*
  sort $DQ = (D, Q \mid \lambda(x, c)\ c = e(x))$
  op $g : DQ \to R$
  def $g(x, c) = h(x)$
  theorem $f(x) = g(x, e(x))$
end-spec

We show how the equational simplification refinement works by simplifying the clauses of the definition of *sorting* in $Sorting\text{-}Alg_2$. The simplification often serves a larger goal. Following algorithm design there are usually opportunities to simplify expressions due to the juxtaposition of subexpressions from different sources that have been assembled by means of a definitional scheme. More pertinent here, when datatype refinement is performed and the codomain has a constructed sort, then simplification can reexpress definitions in terms of the base sorts of the construction. The definitional axioms of *sorting* in $Sorting\text{-}Alg_2$ are expressed in terms of a quotient sort construction

sort-axiom $Bag\text{-}as\text{-}Seq = Seq/perm?$

Simplification will have the effect of reexpressing the axioms in terms of sequences, using coercion functions.

We show the simplification of the singleton case since it most clearly exemplifies the essential reasoning pattern. The classification arrow can be partially filled in once a user has selected an expression to simplify:

$$
\begin{array}{rcl}
D & \mapsto & \mathit{Unit} \\
Q & \mapsto & \mathit{Boolean} \\
\mathit{expr} & \mapsto & \forall(x : \mathit{Bag\text{-}as\text{-}Seq})(\mathit{bag\text{-}singleton?}(x) \implies \\
& & \qquad \exists(e)(x = \mathit{bag\text{-}singleton}(e) \ \wedge \ \mathit{sorting}(x) = C_1(e))) \\
\mathit{new\text{-}expr} & \mapsto & ?
\end{array}
$$

To obtain a translation for *new-expr*, we unskolemize it in the *New-Expression* axiom, then translate the result and find a witness in *Sorting* theory as shown in Figure 7. The proof draws on axioms from *Bag-as-Seq* and *Seq* and the quantifier change theorems:

$$(\forall(x : D/ \equiv)\ A(x)) \ = \ (\forall(y : D)\ A(q(y))) \qquad \text{where } q = \mathit{quotient}(\equiv)$$

$$(\forall(x : D|p?)\ A(r(x))) \ = \ (\forall(y : D)(p?(y) \implies A(y))) \qquad \text{where } r = \mathit{relax}(p?)$$

The completed the classification arrow is:

$$
\begin{array}{rcl}
D & \mapsto & \mathit{Unit} \\
Q & \mapsto & \mathit{Boolean} \\
\mathit{expr} & \mapsto & \forall(x : \mathit{Bag\text{-}as\text{-}Seq})(\mathit{bag\text{-}singleton?}(x) \implies \\
& & \qquad \exists(e)(x = \mathit{bag\text{-}singleton}(e) \ \wedge \ \mathit{sorting}(x) = C_1(e))) \\
C & \mapsto & \lambda(x)\ \mathit{singleton\text{-}seq?}(x) \\
\mathit{new\text{-}expr} & \mapsto & \forall(y : \mathit{Seq})\ \mathit{singleton\text{-}seq?}(y) \implies \mathit{sorting}(q(y)) = y
\end{array}
$$

The effect of computing the pushout of the refinement and the classification arrow is to add the theorem asserting equality of these two expressions in context. We then apply a substitution (meta)operator that creates an isomorphic spec. After simplifying all three axioms of *sorting*, these refinements have the effect shown in Figure 8. The special form *restrict* coerces a value to a subsort, provided that the subsort predicate holds in context. $\mathit{Sorting\text{-}Alg}_4$, also shown in Figure 8 shows the effect of aggregating the clauses of the *sorting* definition into a conditional using another refinement.

3.3.4. Sorting: Summary

One remaining step is to design a definition for C_2. This spec can be refined either by applying divide-and-conquer to create one of several possible simple merge operators: the straightforward one is the usual linear time sequential merge [4]; an alternate derivation yields the log time parallel merge underlying Batcher's Sort [8]. Another way to synthesize a definition for C_2 is to reduce it to "legacy code"; that is, an existing library routine that can satisfy the C_2 requirement specification. This reduction process uses a general mechanism called connections betweeen theories [7].

The code generation process uses library refinements from the specification language to a programing language (currently in Specware these are CommonLISP and C++).

$$\forall(x : \mathit{Bag\text{-}as\text{-}Seq})(\mathit{bag\text{-}singleton?}(x) \\ \implies \exists(e)(x = \mathit{bag\text{-}singleton}(e) \ \wedge \ \mathit{sorting}(x) = C_1(e)))$$

$=$ quantifier change via $q : \mathit{Seq} \rightarrow \mathit{Bag\text{-}as\text{-}Seq} \ = \ \mathit{quotient}(\mathit{perm?})$

$$\forall(y : \mathit{Seq})(\mathit{bag\text{-}singleton?}(q(y)) \\ \implies \exists(e)(q(y) = \mathit{bag\text{-}singleton}(e) \ \wedge \ \mathit{sorting}(q(y)) = C_1(e)))$$

$=$ simplifying, using the defs of *bag-singleton?* and *bag-singleton*

$$\forall(y : \mathit{Seq})(\mathit{singleton\text{-}seq?}(y) \\ \implies \exists(e)(q(y) = q(\mathit{singleton\text{-}seq}(e)) \ \wedge \ \mathit{sorting}(q(y)) = C_1(e)))$$

$=$ quantifier change via the subsort coercion
$r1 : 1\text{-}\mathit{Seq} \rightarrow \mathit{Seq} \ = \ \mathit{relax}(\mathit{singleton\text{-}seq?})$

$$\forall(z : 1\text{-}\mathit{Seq})\exists(e)(q(r1(z)) = q(\mathit{singleton\text{-}seq}(e)) \ \wedge \ \mathit{sorting}(q(r1(z))) = C_1(e))$$

$=$ unifying with $q(r1(w)) = q(\mathit{singleton\text{-}seq}(\mathit{singleton\text{-}seq\text{-}inv}(w)))$
which is inferred from the inverse axiom
$r1(w) = \mathit{singleton\text{-}seq}(\mathit{singleton\text{-}seq\text{-}inv}(w))$

$$\forall(z : 1\text{-}\mathit{Seq}) \ \mathit{sorting}(q(r1(z))) = C_1(\mathit{singleton\text{-}seq\text{-}inv}(z))$$

$=$ unfolding the def of C_1

$$\forall(z : 1\text{-}\mathit{Seq}) \ \mathit{sorting}(q(r1(z))) = \mathit{singleton\text{-}seq}(\mathit{singleton\text{-}seq\text{-}inv}(z))$$

$=$ simplifying, again using the inverse axiom

$$\forall(z : 1\text{-}\mathit{Seq}) \ \mathit{sorting}(q(r1(z))) = r1(z)$$

$=$ quantifier change in reverse, using subsort coercion $r1 : 1\text{-}\mathit{Seq} \rightarrow \mathit{Seq}$

$$\forall(y : \mathit{Seq}) \ \mathit{singleton\text{-}seq?}(y) \ \implies \ \mathit{sorting}(q(y)) = y$$

Figure 7: Simplification of a *sorting* axiom

spec $Sorting\text{-}Alg_3$ is
  import $Bag\text{-}as\text{-}Seq\text{-}Seq\text{-}over\text{-}LinOrd$
  ...
  op $sorting : Bag\text{-}as\text{-}Seq \to Seq$
  theorem $sorted?(x, sorting(x))$
  definition of $sorting$ is
    axiom $empty\text{-}seq?(x) \implies sorting(q(x)) = x$
    axiom $singleton\text{-}seq?(x) \implies sorting(q(x)) = x$
    axiom $nonsingleton\text{-}seq?(x) \implies$
        let $(x1 = q(left\text{-}split(restrict(x))),$
          $x2 = q(right\text{-}split(restrict(x))))$
        $sorting(q(x)) = C_2(sorting(x1), sorting(x2)))$
  end-definition
  ...
end-spec

spec $Sorting\text{-}Alg_4$ is
  import $Bag\text{-}as\text{-}Seq\text{-}Seq\text{-}over\text{-}LinOrd$
  ...
  op $sorting : Bag\text{-}as\text{-}Seq \to Seq$
  theorem $sorted?(x, sorting(x))$
  def $sorting(q(x)) =$
      (if $empty\text{-}seq?(x)$ then x
      elseif $singleton\text{-}seq?(x)$ then x
      else (let $(x1 = q(left\text{-}split(restrict(x))),$
            $x2 = q(right\text{-}split(restrict(x))))$
        $C_2(sorting(x1), sorting(x2)))$
  ...
end-spec

Figure 8: Sorting Algorithm after Simplifications

4. Domain-Specific Software Development

The sorting example shows how to exploit *domain-independent* design knowledge that is uniformly represented by refinements. In this section the Planware system provides examples of using refinements to represent *domain-specific* design knowledge.

Planware [1] is a synthesis system that is specialized to the production of scheduling algorithms. It extends Specware/Designware with libraries of theories and refinements about scheduling and with a specialized tactic for controlling the application of such design knowledge. Planware applies library refinements to transform information from the user into a formal requirement specification, which is typically thousands of line of specification text. The refinement to code via algorithm design, datatype refinement, and expression optimization is completely automatic.

4.1. Constructing a Requirement Specification

Planware provides an answer to the question of how to help automate the acquisition of requirements from the user and to assemble a formal requirement specification for the user. The key idea is to focus on a narrow well-defined class of programs and to prebuild an abstract specification that covers the class. Interaction with the user is only required in order to obtain the refinement from the abstract spec to a specification of the requirements of the user's particular problem.

The scheduling problem in general is to compute a set of reservations on a given set of resources in order to accomplish a given set of tasks subject to certain constraints. Optionally we may want to optimize an objective function. A sketch of an abstract scheduling specification follows, where the import *Scheduling-Import* combines specifications for *Time*, *Quantity*, *Task*, *Resource*, *Reservation*, *Set-of-Task*, *Set-of-Resource*, *Set-of-Reservation*:

spec $Task$
  sort $Task$
end-spec

spec $Resource$
  sort $Resource$
end-spec

spec $Abstract\text{-}Scheduling$ is
  imports $Scheduling\text{-}Import$
  sorts $Reservation = Resource, Task, Time$
    $Schedule = Set\text{-}of\text{-}Reservation$
  op $scheduler : Set\text{-}of\text{-}Task, Set\text{-}of\text{-}Resource \rightarrow Schedule$

  op $all\text{-}tasks\text{-}scheduled? : Set\text{-}of\text{-}Task, Schedule \rightarrow Boolean$
  def $all\text{-}tasks\text{-}scheduled?(tasks, schedule)$
    $= \forall(tsk)(tsk\ in\ tasks = \exists!(rsvn)(rsvn\ in\ schedule\ \wedge\ task(rsvn) = tsk))$
  axiom $all\text{-}tasks\text{-}scheduled?(tasks, scheduler(tasks, resources))$

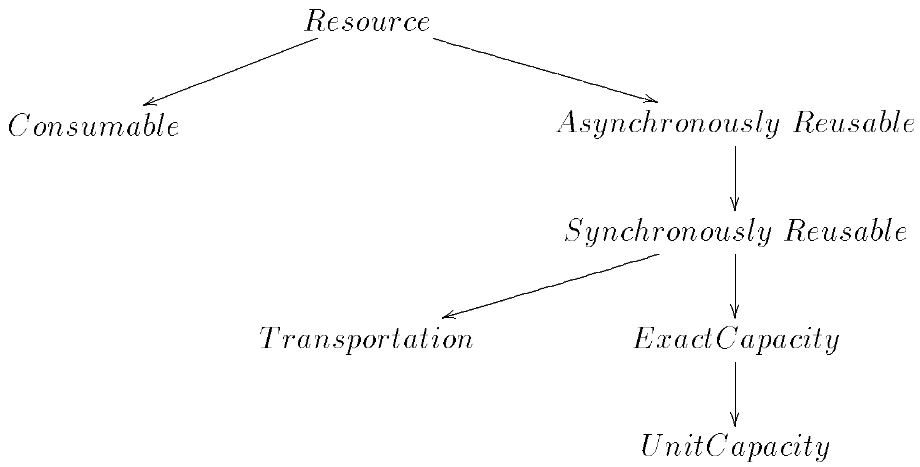

Figure 9: Taxonomy of Resource Theories

op *only-given-resources-used?* : $Set\text{-}of\text{-}Resource, Schedule \to Boolean$
def *only-given-resources-used?*$(resources, schedule)$
$= \forall(rsvn)(rsvn\ in\ schedule \implies resource(rsvn)\ in\ resources)$
axiom *only-given-resources-used?*$(resources, scheduler(tasks, resources))$
end-spec

This specifies an abstract scheduler whose input is a set of tasks (each with unspecified structure) and a set of resources (each with unspecified structure). Two generic problem constraints are imposed on the output of the *Scheduler* operator: *all-tasks-scheduled?* asserts that it must schedule all (and only) the input tasks, and *only-given-resources-used?* asserts that it uses only the supplied resources. So, for example, if the input is 100 cargo items to fly on 10 aircraft, then the schedule generated by *Scheduler* must schedule all 100 items (and no more) and use only the 10 given aircraft.

We briefly describe two domain-specific refinements that are applied to refine this abstract scheduling spec to a spec for a particular problem, such as a transportation problem, a power plant maintenance problem, or a processor scheduling problem.

Figure 9 depicts a taxonomy of resource theories that covers many commonly occurring types of resources. Abstract *Resource* theory refines into *Consumable* resources (e.g. fuel, crew duty time) and *Reusable* resources (e.g. aircraft, trucks, parking lots, printers). *Reusable* resources further refine into *Synchronously Reusable* resources (e.g. ships, aircraft) where all reservations of a resource instance must be synchronized (e.g. passenger reservations on a particular airline flight are all for the same start and finish time). *Synchronously Reusable* resources further refine into *Transportation* resources (which have spatial translation between starting and ending points), and *Unit Capacity* resources, which can handle at most one reservation at a time.

The axiomatic formulation has several benefits over an object-oriented classification hierarchy. Both provide the signature of the relevant operations on the resource. A specification for a resource type also provides the axiomatic characterization of all properties

spec *Semilattice-Structured-Attribute* is
  sorts $Task$, *Attr-sort*
  op $attribute : Task \to Attr\text{-}sort$
  op $rsvn\text{-}attr : Reservation \to Attr\text{-}sort$
  op $\_le\_ : Attr\text{-}sort, Attr\text{-}sort \to Boolean$
  axioms ... *le* is a bounded semilattice ...
end-spec

spec *Create-Constraint-from-Task-Attribute* is
  import *Semilattice-structured-attribute*
  op $consistent\text{-}attr : Schedule \to Boolean$
  def $consistent\text{-}attr(sched) =$

$$\forall(rsvn, tsk)(rsvn \in sched \ \wedge \ tsk\ in\ tasks(rsvn) \implies attr(tsk)\ le\ rsvn\text{-}attr(rsvn))$$

  axiom $consistent\text{-}attr(scheduler(tasks, resources))$
end-spec

Figure 10: Refinement for Constructing Requirement Constraints on a Scheduler

shared by instances of the resource class. Furthermore, important theorems that provide the basis for important algorithmic concepts can be recorded in the resource spec.

To particularize the abstract scheduling specification, Planware requests the user to select a resource theory from the taxonomy. The composed arrow from *Resource* theory to the selected theory is the arrow used for refinement.

In the next step, the user interacts with a spreadsheet-like interface that simultaneously gathers information about task attributes and how they are ordered as a semilattice – this information is critical for algorithm design later. We skip the details of this process in order to focus on how task attributes are lifted into requirement constraints on *Scheduler* (see [1]). As an example, consider the *release-time* attribute on *Task* which we suppose the user has selected as a lower bound on the *start-time* attribute which is required on reservations of a transportation resource. A domain-specific refinement is used to create problem requirement constraints is shown in Figure 10:

The codomain of the refinement creates a new predicate which is asserted as a requirement constraint on *Scheduler*. The axiom asserts that the scheduler must produce an output that satisfies this new constraint. From context, Planware automatically constructs the classification arrow:

$$\begin{array}{rcl} Task & \mapsto & Task \\ Attr\text{-}sort & \mapsto & Time \\ attribute & \mapsto & release\text{-}time \\ rsvn\text{-}attr & \mapsto & start\text{-}time \\ le & \mapsto & \leq \end{array}$$

The pushout then automatically adds the new structure, including the requirement constraint on *Scheduler*, to the Scheduling specification:

spec $\mathit{Abstract\text{-}Scheduling}_1$ is
  imports *Scheduling-Import*
  sorts $\mathit{Reservation} = \mathit{Resource}, \mathit{Task}, \mathit{Time}$
      $\mathit{Schedule} = \mathit{Set\text{-}of\text{-}Reservation}$
  op $\mathit{scheduler} : \mathit{Set\text{-}of\text{-}Task}, \mathit{set\text{-}of\text{-}Resource} \to \mathit{Schedule}$

  op $\mathit{all\text{-}tasks\text{-}scheduled?} : \mathit{Set\text{-}of\text{-}Task}, \mathit{Schedule} \to \mathit{Boolean}$
  def $\mathit{all\text{-}tasks\text{-}scheduled?}(\mathit{tasks}, \mathit{schedule})$
      $= \forall(\mathit{tsk})(\mathit{tsk\ in\ tasks} = \exists!(\mathit{rsvn})(\mathit{rsvn\ in\ schedule}\ \wedge\ \mathit{task}(\mathit{rsvn}) = \mathit{tsk}))$
  axiom $\mathit{all\text{-}tasks\text{-}scheduled?}(\mathit{tasks}, \mathit{scheduler}(\mathit{tasks}, \mathit{resources}))$

  op $\mathit{only\text{-}given\text{-}resources\text{-}used?} : \mathit{Set\text{-}of\text{-}Resource}, \mathit{Schedule} \to \mathit{Boolean}$
  def $\mathit{only\text{-}given\text{-}resources\text{-}used?}(\mathit{resources}, \mathit{schedule})$
      $= \forall(\mathit{rsvn})(\mathit{rsvn\ in\ schedule} \implies \mathit{resource}(\mathit{rsvn})\ \mathit{in\ resources})$
  axiom $\mathit{only\text{-}given\text{-}resources\text{-}used?}(\mathit{resources}, \mathit{Scheduler}(\mathit{tasks}, \mathit{resources}))$

  op $\mathit{rsvn\text{-}start\text{-}time} : \mathit{Reservation} \to \mathit{Time}$
  op $\mathit{consistent\text{-}start\text{-}time} : \mathit{Schedule} \to \mathit{Boolean}$
  def $\mathit{consistent\text{-}start\text{-}time}(\mathit{sched}) =$
      $\forall(\mathit{rsvn}, \mathit{tsk})(\mathit{rsvn\ in\ sched}\ \wedge\ \mathit{tsk\ in\ tasks}(\mathit{rsvn})$
        $\implies \mathit{release\text{-}time}(\mathit{tsk}) \leq \mathit{rsvn\text{-}start\text{-}time}(\mathit{rsvn}))$
  axiom $\mathit{consistent\text{-}start\text{-}time}(\mathit{scheduler}(\mathit{tasks}, \mathit{resources}))$
end-spec

This new requirement constraint on the scheduler asserts that the actual start time of a reservation must be no earlier than the release date of any task in the reservation. Applying this domain-specific yet abstract refinement to all attributes of *Task* gathered from dialog with the user allows Planware to automatically generate the requirements constraints on *Scheduler* (without the user having to know, see, or write advanced mathematics).

4.2. Datatype Refinement and Problem Reformulation

Datatype refinement plays several important roles in Planware. We briefly mention one use at the problem formulation stage that is critical to the development of good algorithms at a later stage.

$$\mathit{Set}(\mathit{Resource} \times \mathit{Task} \times \mathit{Time})$$
$$\Downarrow$$
$$\mathit{Map}(\mathit{Resource}, \mathit{Set}(\mathit{Resource} \times \mathit{Task} \times \mathit{Time}))$$
$$\Downarrow$$
$$\mathit{Map}(\mathit{Resource},\ \mathit{Ordered\text{-}Seq}(\mathit{Resource} \times \mathit{Task} \times \mathit{Time}))$$

At the most abstract level in Planware, a schedule is presented as a set of reservations (which in turn are effectively tuples). This abstraction is well-suited for gathering and formalizing user requirements, but quite poor if implemented in a straightforward or naive way (e.g. as lists). Planware exploits refinements shown above, where the second refinement exploits the linear ordering of time to refine a set to a sorted sequence (ordered by increasing time). The effect is to refine schedule-as-a-relation into schedule-as-a-Gantt-chart (or time-function) – a common representation in scheduling algorithms. The main benefit is that several of the output constraints on the scheduler function now simplify tremendously. Put another way, the refined data structures satisfy some of the constraints implicitly, they are built-in, so they need not be explicit.

A clear example of this phenomenon arises in the k-queens problem. A straightforward specification of this problem would define a $k \times k$ bit matrix to represent placement of queens on a $k \times k$ chessboard. In this representation we have four constraints to satisfy: no-two-queens-per-row, no-two-queens-per-column, no-two-queens-per-ascending-diagonal, no-two-queens-per-descending-diagonal. The constraint that there is at most one queen per column allows us to refine the bit matrix to a sequence where the i^{th} entry represents the row number of the queen in that column, if there is one. The effect is to cut down the space of possible solutions by building in the constraint no-two-queens-per-column. Furthermore, the specification of the queens problem can be condensed since the no-two-queens-per-column constraint simplifies away in the refined representation.

This phenomenon points out another benefit of a refinement approach: it supports users in exploring alternative formulations of their problem. A complicated (i.e. lots of information) but efficient (i.e. tighter representation of the problem so that there's less junk in the search space) representation can be derived from a simpler, more understandable formulation. Oftentimes in current software development approaches the inertia of the lots-of-bits formulation may discourage users from backing up and rethinking/reformulating their approach. The refinement approach encourages a carefully staged design process wherein the user first expresses the most abstract essential requirements on the desired software, and only then begins to explore decisions about formulation, algorithms, architectures, data structures, and so on.

5. Scaling up

The process of refining specification S_0 described above has three basic steps:

1. select a refinement $A \Longrightarrow B$ from a library,
2. construct a classification arrow $A \Longrightarrow S_0$, and
3. compute the pushout S_1 of $B \Longleftarrow A \Longrightarrow S_0$.

The resulting refinement is the cocone arrow $S_0 \Longrightarrow S_1$. This basic refinement process is repeated until the relevant sorts and operators of the spec have sufficiently explicit definitions that they can be easily translated to a programming language, and then compiled.

In this section we address the issue of how this basic process can be further developed in order to scale up as the size and complexity of the library of specs and refinements

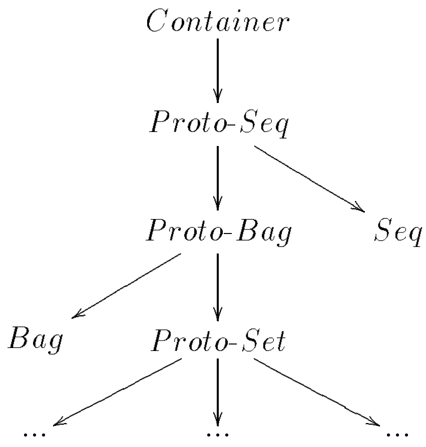

Figure 11: Taxonomy of Container Datatypes

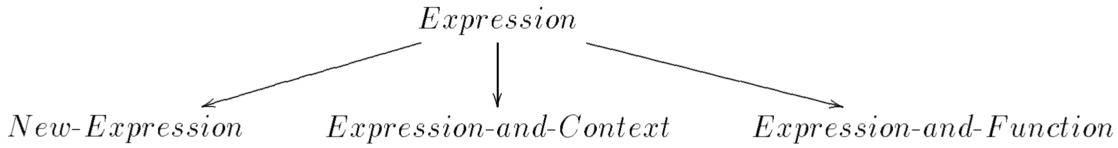

Figure 12: Taxonomy of Expression Optimization Refinements

grows. The first key idea is to organize libraries of specs and refinements into *taxonomies*. The second key idea is to support *tactics* at two levels: theory-specific tactics for constructing classification arrows, and task-specific tactics that compose common sequences of the basic refinement process into a larger refinement step.

5.1. Design by Classification: Taxonomies of Refinements

A productive software development environment will have a large library of reusable refinements, letting the user (or a tactic) select refinements and decide where to apply them. The need arises for a way to organize such a library, to support access, and to support efficient construction of classification arrows. A library of refinements can be organized into *taxonomies* where refinements are indexed on the nodes of the taxonomies, and the nodes include the domains of various refinements in the library. The taxonomic links are refinements, indicating how one refinement applies in a stronger setting than another.

Figure 11 sketches a taxonomy of abstract datatypes for collections. Details are given in Appendix A. The arrows between nodes express the refinement relationship; e.g. the morphism from *Proto-Seq* to *Proto-Bag* is an extension with the axiom of commutativity applied to the join constructor of *Proto-Seq*s. Datatype refinements are indexed by the specifications in the taxonomy; e.g. a refinement from (finite) bags to (finite) sequences is indexed at the node specifying (finite) bag theory. Figure 12 shows the beginnings of a taxonomy of expression optimization refinements that includes the ones presented in Section 3.3.3.. Figure 13 shows a taxonomy of algorithm design theories. The refinements indexed at each node correspond to (families of) program schemes. The algorithm theory associated with a scheme is sufficient to prove the consistency of any instance of the scheme.

Nodes that are deeper in a taxonomy correspond to specifications that have more structure than those at shallower levels. Generally, we wish to select refinements that are indexed as deeply in the taxonomy as possible, since the maximal amount of structure in the requirement specification will be exploited. In the algorithm taxonomy, the deeper the node, the more structure that can be exploited in the problem, and the more problem-solving power that can be brought to bear. Roughly speaking, narrowly scoped but faster algorithms are deeper in the taxonomy, whereas widely applicable general algorithms are at shallower nodes.

Two problems arise in using a library of refinements: (1) selecting an appropriate refinement, and (2) constructing a classification arrow. If we organize a library of refinements into a taxonomy, then the following *ladder construction* process provides incremental access to applicable refinements, and simultaneously, incremental construction of classification arrows.

$$\begin{tikzcd}
A_0 \arrow[r, Rightarrow, "I_0"] \arrow[d, Rightarrow] & Spec_0 \arrow[d, Rightarrow] \\
A_1 \arrow[r, Rightarrow, "I_1"] \arrow[d, Rightarrow] & Spec_1 \arrow[d, Rightarrow] \\
A_2 \arrow[r, Rightarrow, "I_2"] \arrow[d, Rightarrow] & Spec_2 \arrow[d, Rightarrow] \\
\cdots \arrow[d, Rightarrow] & \cdots \arrow[d, Rightarrow] \\
A_n \arrow[r, Rightarrow, "I_n"] & Spec_n
\end{tikzcd}$$

The process of incrementally constructing a refinement is illustrated in the *ladder construction* diagram to the left. The left side of the ladder is a path in a taxonomy starting at the root. The ladder is constructed a rung at a time from the top down. The initial interpretation from A_0 to $Spec_0$ is often simple to construct. The rungs of the ladder are constructed by a constraint solving process that involves user choices, the propagation of consistency constraints, calculation of colimits, and constructive theorem proving [7, 9]. Generally, the rung construction is stronger than a colimit – even though a cocone is being constructed. The intent in contructing $I_i : A_i \Longrightarrow Spec_i$ is that $Spec_i$ has sufficient *defined* symbols to serve as the codomain. In other words, the *implicitly* defined symbols in A_i are translated to *explicitly* defined symbols in $Spec_i$.

$$\begin{tikzcd}
A_n \arrow[r, Rightarrow, "I_n"] \arrow[d, Rightarrow] & Spec_n \arrow[d, Rightarrow] \\
B_n \arrow[r, Rightarrow] & Spec_{n+1}
\end{tikzcd}$$

Once we have constructed a classification arrow $A_n \Longrightarrow Spec_n$ and selected a refinement $A_n \Longrightarrow B_n$ that is indexed at node A_n in the taxonomy, then constructing a refinement of $Spec_0$ is straightforward: compute the pushout, yielding $Spec_{n+1}$, then compose arrows down the right side of the ladder and the pushout square to obtain $Spec_0 \Longrightarrow Spec_{n+1}$ as the final constructed refinement.

Again, rung construction is *not* simply a matter of computing a colimit. For example,

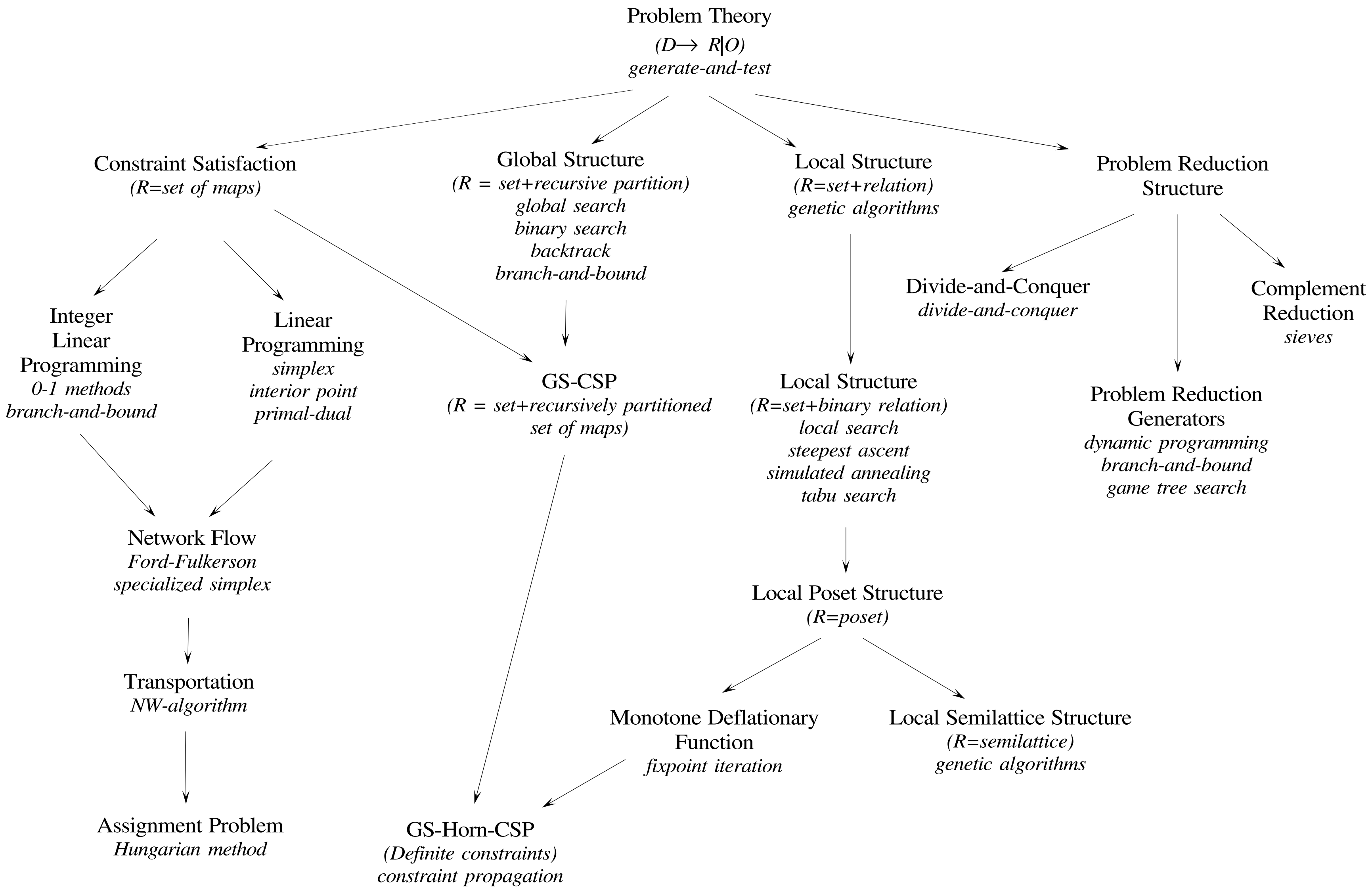

Figure 13: Taxonomy of Algorithm Theories

there are at least two distinct arrows from *Divide-and-Conquer* to *Sorting*, corresponding to a mergesort and a quicksort – these are distinct cocones and there is no universal sorting algorithm corresponding to the colimit. However, applying the refinement that we select at a node in the taxonomy *is* a simple matter of computing the pushout. For algorithm design the pushout simply instantiates some definition schemes and other axiom schemes.

It is unlikely that a general automated method exists for constructing rungs of the ladder, since it is here that creative decisions can be made. For general-purpose design it seems that users must be involved in guiding the rung construction process. However in domain-specific settings and under certain conditions it will possible to automate rung construction (as discussed in the next section). Our goal in Designware is to build an interface providing the user with various general automated operations and libraries of standard components. The user applies various operators with the goal of filling out partial morphisms and specifications until the rung is complete. After each user-directed operation, constraint propagation rules are automatically invoked to perform sound extensions to the partial morphisms and specifications in the rung diagram. Constructive theorem-proving provides the basis for several important techniques for constructing classification arrows [7, 9].

5.2. Tactics

The design process described so far uses primitive operations such as (1) selecting a spec or refinement from a library, (2) computing the pushout/colimit of (a diagram of) diagram morphisms, and (3) unskolemizing and translating a formula along a morphism, (4) witness-finding to derive symbol translations during the construction of classification arrows, and so on. These and other operations can be made accessible through a GUI, but inevitably, users will notice certain patterns of such operations arising, and will wish to have macros or parameterized procedures for them, which we call *tactics*. They provide higher level (semiautomatic) operations for the user.

The need for at least two kinds of tactics can be discerned.

1. *Classification tactics* control operations for constructing classification arrows. The divide-and-conquer theory admits at least two common tactics for constructing a classification arrow which we illustrated in Section 3.3.1.. One tactic can be procedurally described as follows: (1) the user selects a operator symbol with a DRO requirement spec, (2) the system analyzes the spec to obtain the translations of the DRO symbols, (3) the user is prompted to supply a standard set of constructors on the input domain D, (4) the tactic performs unskolemization on the composition relation in each Soundness axiom to derive a translations for O_{Ci}, and so on. This tactic was followed in the mergesort derivation.

   The other tactic is similar except that the tactic selects constructors for the composition relations on R (versus D) in step (3), and then uses unskolemization to solve for decomposition relations in step (4). This tactic was followed in the quicksort derivation.

A classification tactic for context-dependent simplification provides another example. Procedurally: (1) user selects an expression *expr* to simplify, (2) type analysis is used to infer translations for the input and output sorts of *expr*, (3) a context analysis routine is called to obtain contextual properties of *expr* (yielding the translation for C), (4) unskolemization and witness-finding are used to derive a translation for *new-expr*.

2. *Refinement tactics* control the application of a collection of refinements; they may compose a common sequence of refinements into a larger refinement step. Planware has a code-generation tactic for automatically applying spec-to-code interlogic morphisms. Another example is a refinement tactic for context-dependent simplification; procedurally, (1) use the classification tactic to construct the classification arrow, (2) compute the pushout, (3) apply a substitution operation on the spec to replace *expr* with its simplified form and to create an isomorphism. Finite Differencing requires a more complex tactic that applies the tactic for context-dependent simplification repeatedly in order to make incremental the expressions set up by applying the *Expression-and-Function* $\rightarrow$ *Abstracted-Op* refinement.

We can also envision the possibility of metatactics that can construct tactics for a given class of tasks. For example, given an algorithm theory, there may be ways to analyze the sorts, ops and axioms to determine various orders in constructing the translations of classification arrows. The two tactics for divide-and-conquer mentioned in Section 3.3.1. are an example.

6. Summary

Perhaps the main message of this paper is that a formal software refinement process can be supported by automated tools, and in particular that libraries of design knowledge can be brought to bear in constructing refinements for a given requirement specification. One goal of this paper has been to show that diagram morphisms are adequate to capture design knowledge about algorithms, data structures, and expression optimization techniques, as well as the refinement process itself. We showed how to apply a library refinement to a requirement specification by constructing a classification arrow and computing the pushout. We discussed how a library of refinements can be organized into taxonomies and presented techniques for constructing classification arrows incrementally. The examples and most concepts described are working in the Specware, Designware, and Planware systems.

Acknowledgements: The work reported here is the result of extended collaboration with my colleagues at Kestrel Institute. I would particularly like to acknowledge the contributions of David Espinosa, LiMei Gilham, Junbo Liu, Dusko Pavlovic, and Stephen Westfold. I would also like to thank Lambert Meertens for his suggestions on treating nondeterministic functions in the Specware context.

References

[1] Blaine, L., Gilham, L., Liu, J., Smith, D., and Westfold, S. Planware – domain-specific synthesis of high-performance schedulers. In *Proceedings of the Thirteenth Automated Software Engineering Conference* (October 1998), IEEE Computer Society Press, pp. 270–280.

[2] Meseguer, J. General logics. In *Logic Colloquium 87*, H. Ehrig et al., Ed. North Holland, Amsterdam, 1989, pp. 275–329.

[3] Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.

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

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

[6] Smith, D. R. Structure and design of problem reduction generators. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 91–124.

[7] Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.

[8] Smith, D. R. Derivation of parallel sorting algorithms. In *Parallel Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter, Eds. Kluwer Academic Publishers, New York, 1993, pp. 55–69.

[9] Smith, D. R. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96* (1996), vol. LNCS 1101, Springer-Verlag, pp. 62–84.

[10] Srinivas, Y. V., and Jüllig, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.

A Containers, Bags, and Sequences

In this section we present parts of the following taxonomy of Container structures and their refinements. This taxonomy is adapted from the Specware/Designware library. One refinement is given in detail: $Bag \implies Seq$.

$$
\begin{tikzcd}
 & Container \arrow[d] & \\
 & Proto\text{-}Seq \arrow[d] \arrow[dr] & \\
 & Proto\text{-}Bag \arrow[dl] \arrow[d] & Seq \\
Bag & Proto\text{-}Set \arrow[dl] \arrow[dr] & \\
\ldots & & \ldots
\end{tikzcd}
$$

```
spec CONTAINER is
  sorts E, Container
  op empty     : -> Container
  op singleton     : E -> Container
  op _join_ : Container, Container -> Container
  axiom unit is
    x join empty = x  &  empty join x = x
  constructors {empty, singleton, join} construct Container

  op empty? : Container -> Boolean
  definition of empty? is
    axiom empty?(empty) = true
    axiom empty?(singleton(y)) = false
    axiom empty?(U join V) = (empty?(U) &  empty?(V))
  end-definition

  op nonempty? : Container -> Boolean
  def nonempty?(b) = ~empty?(b)

  op _in_ : E, Container -> Boolean
  definition of in is
    axiom x in empty = false
    axiom x in singleton(y) = (x = y)
    axiom x in U join V = (x in U or  x in V)
  end-definition
 end-spec
```

```
spec PROTO-SEQ is
 translate
 colimit of
  diagram
   nodes ASSOCIATIVE, CONTAINER, BIN-OP
   arcs BIN-OP -> ASSOCIATIVE : {}
      , BIN-OP -> CONTAINER : {E -> Container, binop -> join}
  end-diagram
 by {Container -> Seq,
     empty -> empty-seq,
     empty? -> empty-seq?,
     singleton -> singleton-seq,
     join -> concat}
```

The definition of PROTO-SEQ given above evaluates to the following "flat" spec:

```
spec PROTO-SEQUENCE is
  sorts E, Seq
  const empty-seq : -> Seq
  op singleton-seq : E -> Seq
  op _concat_ : Seq, Seq -> Seq
  constructors {empty-seq, singleton-seq, concat} construct Seq

  axiom unit is
    forall(x:Seq)(x concat empty-seq = x  &  empty-seq concat x = x)
  axiom associativity is
    x concat (y concat z) = (x concat y) concat z

  op empty-seq? : Seq -> Boolean
  definition of empty-seq? is
    axiom empty-seq?(empty) = true
    axiom empty-seq?(singleton(y)) = false
    axiom empty-seq?(U concat V) = (empty-seq?(U) &  empty-seq?(V))
  end-definition

  op nonempty-seq? : Seq -> Boolean
  def nonempty-seq?(b) = ~empty-seq?(b)

  op _in_ : E, Seq -> Boolean
  definition of in is
    axiom x in empty-seq = false
    axiom x in singleton-seq(y) = (x = y)
    axiom x in U concat V = (x in U or  x in V)
  end-definition
 end-spec
```

```
spec PROTO-BAG is
 translate
 colimit of
  diagram
   nodes BIN-OP, COMMUTATIVE, PROTO-SEQ
   arcs  BIN-OP -> COMMUTATIVE : {}
      ,  BIN-OP -> PROTO-SEQ   : {E -> Seq, binop -> concat}
  end-diagram
 by {Seq -> Bag,
     empty-seq -> empty-bag,
     empty-seq? -> empty-bag?,
     singleton-seq -> singleton-bag,
     concat -> bag-union}

spec PROTO-SET is
 translate
 colimit of
  diagram
   nodes BIN-OP, IDEMPOTENT, PROTO-BAG
    arcs  BIN-OP -> IDEMPOTENT : {}
       ,  BIN-OP -> PROTO-BAG  : {E -> Bag, binop -> bag-union}
  end-diagram
 by {Bag -> Set,
     empty-bag -> empty-set,
     singleton-bag -> singleton-set,
     bag-union -> union}
```

We must extend the PROTO- specs with additional sorts and operations to get useful theories. In practice there would be several extensions to Proto-SEQ yielding a subtaxonomy of Sequence theories, each useful in different contexts.

Note the use of subsort definitions here. Subsorts are necessitated by the requirement that all operators be total. For example, *NE-seq* (short for Non-Empty sequences) is defined as sequences restricted to nonempty sequences. The selectors *head* and *tail* can then be introduced as with signature $NE\text{-}seq \rightarrow Seq$. A subsort construction $E|p?$ comes with an injective coercion function $relax(p?) : E|p? \rightarrow E$ that takes subsort elements into the parent sort. The function r is introduced as an abbreviation of *relax(nonempty-seq?)* which coerces from the subsort (*NE-seq*) to the parent sort (*seq*). Note also how SEQ introduces destructors as inverses of constructors. These play a key role in divide-and-conquer-style algorithms. *Unit* denotes the empty product construction – the sort with one element, $\langle\rangle$.

```
spec SEQ is
  import PROTO-SEQ

  sort NE-Seq
  sort-axiom NE-Seq = Seq | nonempty-seq?
  op r: NE-seq -> Seq
  def r = relax(nonempty-seq?)

  op prepend : E , Seq -> Seq
  op head : NE-Seq -> E
  op tail : NE-Seq -> Seq
  constructors {empty-seq, prepend} construct Seq
  axiom prepend(a,S) = singleton(a) concat S

  op size: Seq -> Nat
  definition of size is
    axiom size(empty-seq) = 0
    axiom size(prepend(a,S)) = 1 + size(S)
  end-definition

  sort 0-Seq
  sort-axiom 0-Seq = Seq | empty-seq?
  op r0: 0-seq -> Seq
  def r0 = relax(empty-seq?)
  op empty-seq-inv : 0-seq -> Unit
  inversion of constructor empty-seq is
    discriminator empty-seq?
    destructors   [empty-seq-inv]
  axiom fa(x:0-seq) empty-seq(empty-seq-inv(x)) = r0(x)

  sort 1-Seq
```

```
  sort-axiom 1-Seq = Seq | singleton-seq?
  op singleton-seq? : Seq -> Boolean
  def singleton-seq?(x) = (size(x)=1)
  op r1: 1-seq -> Seq
  def r1 = relax(singleton-seq?)
  op singleton-seq-inv : 1-Seq -> E
  inversion of constructor singleton-seq is
    discriminator singleton-seq?
    destructors   [singleton-seq-inv]
  axiom fa(x:1-Seq) singleton-seq(singleton-seq-inv(x)) = r1(x)

  sort 2-Seq
  sort-axiom 2-Seq = Seq | nonsingleton-seq?
  op nonsingleton-seq? : Seq -> Boolean
  def nonsingleton-seq?(x) = (size(x)>1)
  op r2: 2-seq -> Seq
  def r2 = relax(nonsingleton-seq?)
  op left-split  : 2-Seq -> Seq
  op right-split : 2-Seq -> Seq
  inversion of constructor concat is
    discriminator nonsingleton-seq?
    destructors   [left-split, right-split]
  axiom fa(x:2-Seq) concat(left-split(x), right-split(x)) = r2(x)
 end-spec


spec BAG is
 import PROTO-BAG
 op size: Bag -> Nat
 definition of size is
    axiom size(empty-bag) = 0
    axiom size(singleton-bag(e)) = 1
    axiom size(bag-union(b1, b2)) = size(b1) + size(b2)
 end-definition

 op bag-wflgt : Bag, Bag -> Boolean
 def bag-wfgt(b1, b2) = (size(b1) > size(b2))

 op singleton-bag? : Bag -> Boolean
 def singleton-bag?(b) = (size(b)=1)

 op nonsingleton-bag? : Bag -> Boolean
 def nonsingleton-bag?(b) = (size(b)>1)

 end-spec
```

The following text gives the refinement of bags to sequences, as described in Section 2.4.3. Note the use of a quotient sort definition here. The sort *Bag-as-Seq* is defined as sequences quotiented by the permutation relation. The purpose of *Bag-as-Seq* is to serve as the translation of *Bag*. The quotienting captures the commutativity of bags: we say two sequences are equal as bags exactly when they are permutations of one another. When we finally translate down to code, equality on bags will translate to the permutation relation on the implementing sequences. A quotient sort construction $E/\equiv$ comes with a surjective coercion function $quotient(\equiv) : E \to E/\equiv$ that takes E elements to their equivalence class. The function q is introduced as an abbreviation of *quotient(perm?)*.

```
spec BAG-AS-SEQ is
  import SEQ

  op remove-1 : E, Seq -> Seq
  definition of remove-1 is
    axiom empty-seq?(S) => remove-1(x S)=S
    axiom x=head(S) => remove-1(x, r(S))=tail(S)
    axiom ~(x=head(S))
          => remove-1(x, r(S))=prepend(head(S), remove-1(x, tail(S)))
  end-definition

  op perm? : Seq, Seq -> Boolean
  definition of perm? is
    axiom empty-seq?(S1)=> (perm?(S1 S2) = empty-seq?(S2))
    axiom perm?(r(S1), S2)
          => head(S1) in  S2 & perm?(tail(S1), remove-1(head(S1), S2))
  end-definition

  sort Bag-as-Seq
  sort-axiom Bag-as-Seq = Seq/perm?
  constructors {bag-empty, bag-singleton, bag-union} construct Bag-as-Seq

  op q : Seq -> Bag-as-Seq
  def q = quotient(perm?)

  op bag-empty     : Bag-as-Seq
  def bag-empty = q(empty-seq)

  op bag-empty?    : Bag-as-Seq -> Boolean
  def bag-empty?(q(s)) = empty-seq?(s)

  op bag-nonempty? : Bag-as-Seq -> Boolean
  def bag-nonempty?(q(s)) = ~empty-seq?(s)

  op bag-singleton : E  -> Bag-as-Seq
```

```
  def bag-singleton(x) = q(singleton-seq(x))

  op bag-singleton? : Bag-as-Seq -> Boolean
  def bag-singleton?(q(s)) = singleton-seq?(s)

  op bag-nonsingleton? : E  -> Bag-as-Seq
  def bag-nonsingleton(q(s)) = nonsingleton-seq?(s)

  op bag-in : E, Bag-as-Seq -> Boolean
  def bag-in(x, q(S)) = x in S

  op bag-union : Bag-as-Seq, Bag-as-Seq -> Bag-as-Seq
  def bag-union(q(S1),q(S2)) = q(concat(S1, S2))

  op bag-size      : Bag-as-Seq -> Nat
  definition of bag-size is
    axiom bag-size(q(s)) = size(s)
  end-definition

 op bag-wflgt : Bag-as-Seq, Bag-as-Seq -> Boolean
 definition of bag-wflgt is
   axiom  bag-wflgt(b1, b2) = gt(bag-size(b1), bag-size(b2))
 end-definition

 end-spec
```

Finally the refinement from BAG to SEQ-AS-BAG is

```
diagram-morphism BtoS is
  {BtoS-Triv : Triv -> Triv,
   BtoS-Bag  : Bag -> Seq-as-Bag}

diagram BAG is
  nodes Bag, Triv
  arcs Triv -> Bag: {E-> E}
end-diagram

diagram BAG-AS-SEQ is
  nodes Seq, Bag-As-Seq, Triv
  arcs Triv -> Bag-As-Seq: {E -> E},
       Triv -> Seq       : {E -> E},
        Seq -> Bag-As-Seq: import-morphism
 end-diagram
```

```
morphism BtoS-Bag  : Bag -> Bag-As-Seq is
 {Bag               -> Bag-As-Seq,
  empty-bag         -> bag-empty,
  empty-bag?        -> bag-empty?,
  nonempty?         -> bag-nonempty?,
  singleton-bag     -> bag-singleton,
  singleton-bag?    -> bag-singleton?,
  nonsingleton-bag? -> bag-nonsingleton?,
  in                -> bag-in,
  bag-union         -> bag-union,
  bag-wfgt          -> bag-wfgt,
  size              -> bag-size}
```

B Specification for Sorting

Listed below are the components of the structured specification for the problem of sorting a bag whose elements are drawn from a linearly ordered set. The specification is parameterized on the linear order.

$$
\begin{tikzcd}
 & \mathit{TRIV} \arrow[dl] \arrow[dr] \arrow[rr] & & \mathit{LINEAR\text{-}ORDER} \arrow[ddd] \\
\mathit{BAG} \arrow[dr] & & \mathit{SEQ} \arrow[dl] & \\
 & \mathit{BAG\text{-}AND\text{-}SEQ} \arrow[d] & & \\
 & \mathit{BAG\text{-}AND\text{-}SEQ\text{-}CONV} \arrow[rr] & & \mathit{BAG\text{-}SEQ\text{-}LinOrd} \arrow[d] \\
 & & & \mathit{BAG\text{-}SEQ\text{-}over\text{-}LinOrd} \arrow[d] \\
 & & & \mathit{SORTING}
\end{tikzcd}
$$

```
spec BAG-AND-SEQ is
 colimit of diagram
  nodes TRIV, BAG, SEQ
  arcs  TRIV -> BAG: {},
        TRIV -> SEQ: {}
 end-diagram

spec BAG-AND-SEQ-CONV is
 import BAG-AND-SEQ
 op seq-to-bag: Seq -> Bag
 definition of seq-to-bag is
  axiom seq-to-bag(empty-seq) = empty-bag
  axiom seq-to-bag(singleton-seq(e)) = singleton-bag(e)
  axiom seq-to-bag(concat(S1, S2))
         = bag-union(seq-to-bag(S1), seq-to-bag(S2))
 end-definition
 end-spec

spec BAG-SEQ-LinOrd is
 colimit of diagram
  nodes TRIV, LINEAR-ORDER, BAG-AND-SEQ-CONV
  arcs  TRIV -> LINEAR-ORDER: {},
```

```
       TRIV -> BAG-AND-SEQ-CONV: {}
 end-diagram

spec BAG-SEQ-over-LinOrd is
  import BAG-SEQ-LinOrd

  op all-le: Bag, Bag -> Boolean
  definition of all-le is
    axiom all-le(empty-bag, empty-bag) = true
    axiom all-le(B, empty-bag) = true
    axiom all-le(singleton-bag(e), empty-bag) = true
    axiom all-le(singleton-bag(d), singleton-bag(e)) = (d le e)
    axiom all-le(singleton-bag(d), bag-union(B1, B2))
        = all-le(singleton-bag(d), B1) & all-le(singleton-bag(d), B2)
    axiom all-le(bag-union(B1, B2), B) = all-le(B1, B) & all-le(B2, B)
  end-definition

  op ordered? : Seq -> Boolean
  definition of ordered? is
    axiom ordered?(empty-seq) = true
    axiom ordered?(singleton(a)) = true
    axiom ordered?(prepend(a, y)) = all-le(singleton(a), seq-to-bag(y))
    axiom ordered?(concat(y, z)) = ordered?(y)
            & all-le(seq-to-bag(y), seq-to-bag(z)) & ordered?(z)
   end-definition
 end-spec

spec SORTING is
 import BAG-SEQ-over-LinOrd

 op  sorted?  : Bag, Seq -> Boolean
 def sorted?(x,z) = (ordered?(z) & x = seq-to-bag(z))

 op sorting : Bag -> Seq
 axiom sorted?(x, sorting(x))
 end-spec
```