8 Constructors Can Be Partial, Too

Deepak Kapur[1]
State University of New York at Albany

8.1 Introduction

The behavior of *regular* aggregate data structures (for example, square matrices, complete binary trees, and powerlists [20]), as well as semantic data structures (for example, sorted lists and priority lists), cannot be captured elegantly by using existing specification techniques and languages such as Larch, Z, OBJ, VDM, or LOTOS. Typically such a data structure is defined as a subsort of a larger data structure. For instance, complete binary trees can be defined as a subsort of binary trees, and powerlists as well as sorted lists can be defined as subsorts of lists. For reasoning about algorithmic descriptions using such data structures, it then becomes necessary to explicitly carry subsort conditions. As shown in [14], this approach makes reasoning about them cumbersome, and the resulting proofs get complex.

It is argued in this article that regularity should be exploited while reasoning about specifications based on regular data structures (see also [20]). That does not seem to be possible if regular data structures are modeled using subsorts, perhaps because of incompatibility between regularity and subsorts. Instead of being a benefit, regularity becomes a burden. This situation is somewhat similar to an experience one would have in expressing a specification involving many data types in an unsorted (single-sorted) notation. Different types would have to be characterized by different unary predicates, and conditions expressed using these unary predicates would have to be carried around wherever terms of particular data types are used. A multisorted notation and logic are instead preferred.

The main idea of this article is to propose a simple extension to the algebraic specification method that allows constructors of a data structure to be *partial* in the sense that their behavior may be meaningful only on a subset of the syntactically specified input domain. An *applicability condition* is associated with every operation, including a constructor, of a data type. An operation is intended to be invoked only if the associated applicability condition is satisfied. As will be shown below, with applicability conditions, regular data structures such as complete binary trees

[1]Partially supported by the National Science Foundation Grant nos. CCR-9303394 and CCR-9308016.

and powerlists can be nicely axiomatized, often by using equations. Semantic data structures including priority lists and sorted lists can also be elegantly axiomatized using this proposal.

An applicability condition of a constructor is expressed in terms of other operations of the data structure being defined as well as the operations of other data structures used in the definition. Even though operations are specified using (conditional) equational axioms in the usual way, these axioms are interpreted differently. Instead of variables appearing in an axiom being assumed to be universally quantified, constraints are imposed on them by the applicability conditions of the function symbols to which these variables appear as arguments. Each term has a *constraint set* that can be mechanically generated from the applicability conditions of the function symbols occurring in the term; this constraint set is called the *context* of the term. The constraint set of an equational axiom is the union of the constraint sets of the left side and right side (and of the condition, in case of conditional equations). If axioms can be oriented into terminating rewrite rules such that the context of the left side of a rewrite rule implies the context of its right side (such rules are later called *context preserving*), then rewrite rules can be used for simplifications, like the rewrite rules for conventional algebraic equational axioms, without having to do any bookkeeping for storing contexts of intermediate results. Because of these properties of context-preserving rules, simplification can be efficiently mechanized.

The focus of this article is to propose a framework for specifying data structures with partial functions and to show how reasoning from the proposed specification can be mechanized in this framework using a rewrite-rule based prover for equations and induction. In a companion paper, semantical considerations of the proposed approach will be discussed [13].

Treatment of partial function symbols in the framework of algebraic specification methods has been a source of considerable technical difficulty. Many different approaches have been proposed in the literature (see [21] for a survey as well as the subsection on related work below), in which the focus has been mostly on theoretical issues such as initiality and uniqueness of models, consistency of implementations with respect to a specification, and capturing the exceptional behavioral of operations as supported in a programming language. We are unaware of any automated reasoning systems/proof checkers that can be used effectively for reasoning about specifications with partial constructors other than by capturing their semantics using subsorts in a logical setting. Modeling regular data structures using subsorts leads to carrying conditions about the regular structure of the values of the data structure, resulting in complicated proofs and intricate reasoning.

8.1.1 Informal Overview of the Approach Using Powerlists

Powerlists, a data structure introduced in [20], will be used for illustration. Powerlists have been used to specify many data-parallel algorithms in [20], employing the divide-and-conquer strategy and recursion. They include Batcher's merge sort and fast Fourier transform, parallel prefix computation, and hardware architectures, as well as hardware circuits, including carry look-ahead and ripple carry adders [1].

Informally speaking, (flat) powerlists are lists of length $2^k, k \geq 0$, of the same type of elements, called *scalars* by Misra, with a binary constructor `tie` (which is the same as the usual *append* function on lists) and a unary constructor `s` (that generates a singleton list from a scalar). This is in contrast to `nil` and `cons` being the standard constructors for lists. The main reason for having `tie` as a constructor is to facilitate the specification of data-parallel algorithms expressed using the divide-and-conquer strategy. Unlike `append`, however, `tie` is to be applied only to lists of equal length, thereby implying that `tie(s(3), tie(s(2), s(1)))` is not a meaningful term; it does not stand for any powerlist.

Constructors are specified as follows. If the applicability condition of a function symbol is different from `true`, it is written to the right of its signature specifying the types of its input as well as its result type; otherwise, it can be omitted, since `true` is the default applicability condition. Constructor `tie(u, v)` below has the applicability condition `length(u) = length(v)`, whereas the applicability condition for `s` and `length` is `true`.

```
[ s(item):  powerlist ]
[ tie(u: powerlist, v: powerlist): powerlist ]   length(u)=length(v)
[ length(powerlist): naturals ]

 1. length(s(i)) = 1,
 2. length(tie(u, v)) = length(u) + length(v).
```

Another useful binary function on powerlists is `zip`, which is also defined on powerlists of equal length. Below, a recursive definition of `zip` is given in terms of `s` and `tie`.

```
[ zip(u: powerlist, v:powerlist): powerlist]    length(u)=length(v)

 3. zip(s(i), s(j)) = tie(s(i), s(j)),
 4. zip(tie(x, y), tie(u, v)) = tie(zip(x, u), zip(y, v)).
```

Consider Axiom 4 above. The interpretation is that instead of variables `x, y,` `u`, and `v` being arbitrary powerlists, they must satisfy the constraints imposed by the function symbols to which they are arguments. For instance, since `x` and `y` are arguments to `tie`, they must satisfy its applicability condition, namely, `length(x) = length(y)`. The situation is similar for `u, v,` as well as `x, u` and `y, v`, since `zip` has the same applicability condition as `tie`. In addition,

$$\texttt{length}(\texttt{tie}(x,y)) = \texttt{length}(\texttt{tie}(u,v))$$

because of the applicability condition of `zip` from the left side of the axiom as well as

$$\texttt{length}(\texttt{zip}(x,u)) = \texttt{length}(\texttt{zip}(y,v))$$

because of the applicability condition of `tie` from the right side of the axiom. The constraint set associated with Axiom 4 is thus

```
{ length(x) = length(y), length(u) = length(v),
  length(tie(x, y)) = length(tie(u, v)),
  length(x) = length(u), length(y) = length(v),
  length(zip(x, u)) = length(zip(y, v)) }.
```

The *context* of Axiom 4 is the conjunction of all the literals in its constraint set. The two phrases "context" and "constraint set" are used interchangeably in the rest of the article. Instead of the variables in Axiom 4 being universally quantified, they must satisfy its context.

Using properties of the `length` function, one can show that `length(zip(x, u)) = 2 * length(x)` for meaningful `zip` subterms (i.e., `zip` subterms in which the arguments satisfy the applicability condition); similarly, `length(tie(u, v)) = 2 * length(u)`. From these properties, it can be proved that the context associated with the left side of the axiom

```
length(x)=length(y) and length(u)=length(v) and
   length(tie(x,y))=length(tie(u,v))
```

implies the context of the right side of the axiom

```
length(x)=length(u) and length(y)=length(v) and
   length(zip(x,u))=length(zip(y,v))
```

since

`length(x)` = `length(y)` = `length(u)` = `length(v)`.

As discussed later, if axioms can be oriented as terminating rewrite rules such that (i) the context of the left side of the rewrite rule implies the context of its right side and (ii) the rewrite rule preserves the monotonicity of applicability conditions, then they can be efficiently used for reasoning without having to store intermediate contexts as well as check for conditions. Such a rewrite rule is called *context preserving*.

Below, we reproduce more function definitions on powerlists from [20]; they are needed to define Batcher's merge sort `pbm` recursively. As the definition of `pbm` below illustrates, the use of powerlists results in concise and elegant specifications. Some of these defining axioms could have been given using `tie` instead of `zip`. (The built-in primitive `cond` in the theorem prover *RRL*, Rewrite Rule Laboratory, used below in axioms is to support case analysis so that some conditional equations can be written as equations; see Appendix A for the axiomatization of `cond`.)

It can be easily shown using an argument similar to the one given above for `zip` that each axiom below has the property that the context of its left side implies the context of its right side. Such axioms can be oriented from left to right, resulting in terminating and context-preserving rewrite rules.

```
[ leq(u: powerlist, v:powerlist): boolean]          length(u) = length(v)
[ minp(u: powerlist, v:powerlist): powerlist]       length(u) = length(v)
[ maxp(u: powerlist, v:powerlist): powerlist]       length(u) = length(v)
[ compare(u: powerlist, v: powerlist): powerlist]   length(u) = length(v)
[ pbm(u: powerlist, v: powerlist): powerlist]       length(u) = length(v)

 5. leq(s(i), s(j)) = (i <= j),
 6. leq(zip(x, y), zip(u, v)) = leq(x, u) and leq(y, v),
 7. minp(s(i), s(j)) = cond(i <= j, s(i), s(j)),
 8. minp(zip(x, y), zip(u, v)) =  zip(minp(x, u) , minp(y, v)),
 9. maxp(s(i), s(j)) = cond(i <= j, s(j), s(i))
10. maxp(zip(x, y), zip(u, v)) =  zip(maxp(x, u) , maxp(y, v)),
11. compare(x, y) = zip(minp(x, y), maxp(x, y)),
12. pbm(s(i), s(j)) = compare(s(i), s(j)),
13. pbm(zip(x, y), zip(u, v)) = compare(pbm(x, v), pbm(y, u)).
```

For a simple example, consider proving `leq(minp(x, y), x)` from the definitions of `leq` and `minp`. The function `leq` compares two powerlists elementwise and returns true if every element in the first powerlists is smaller than the corresponding element in the second powerlist. Below, we give the key steps in a proof.

First, the context of the conjecture,

`length(x)` = `length(y)` `and` `length(minp(x,y))` = `length(x)`, is checked for consistency. It can be shown for meaningful `minp(x, y)` terms that

$$\texttt{length}(\texttt{minp}(\texttt{x},\texttt{y})) = \texttt{length}(\texttt{x}) = \texttt{length}(\texttt{y}),$$

so the context simplifies to `length(x) = length(y) = length(minp(x, y))`, which is consistent.

A proof of the above conjecture can be done by induction under its context. Using the cover set method for automating induction [24, 17], one can generate an induction scheme from the definition of `minp`. The basis case occurs when `x = s(i1), y = s(j1),`, which one can reduce to true using the rewrite rules in the definitions of `leq` and `minp`. For this case, the context does not play any role, since the context of the rules used is true.

For the induction step, the subgoal generated is

$$\texttt{leq}(\texttt{minp}(\texttt{zip}(\texttt{x1},\texttt{y1}),\texttt{zip}(\texttt{u1},\texttt{v1})),\texttt{zip}(\texttt{x1},\texttt{y1})),$$

where the induction hypotheses are

$\texttt{leq}(\texttt{minp}(\texttt{x1},\texttt{u1}),\texttt{x1})$ as well as $\texttt{leq}(\texttt{minp}(\texttt{y1},\texttt{v1}),\texttt{y1})$.

The subgoal matches the left side of the rule corresponding to Axiom 8, so the rewrite rule can be applied; we do not have to recheck that the context of the subgoal is consistent and that the context of the subgoal implies the context of the axiom under the matching substitution for the axiom to be applicable. All this is ensured automatically from the context of the original conjecture and the fact that the rewrite rules corresponding to the axioms are context preserving and the induction schemes are generated from context-preserving rules.

By the application of this rule, the subgoal simplifies to

$$\texttt{leq}(\texttt{zip}(\texttt{minp}(\texttt{x1},\texttt{u1}),\texttt{minp}(\texttt{y1},\texttt{v1})),\texttt{zip}(\texttt{x1},\texttt{y1})).$$

Once again, the above matches the left side of the rewrite rule corresponding to Axiom 6 defining `leq`, and there is no need to check that the context of the above formula implies the context of Axiom 6. The result of applying this rule is

$$\texttt{leq}(\texttt{minp}(\texttt{x1},\texttt{u1}),\texttt{x1})\ \texttt{and}\ \texttt{leq}(\texttt{minp}(\texttt{y1},\texttt{v1}),\texttt{y1}),$$

which are indeed the two induction hypotheses. Hence a proof.

As will be shown in a later section, this unconditional rewriting using the rules corresponding to the axioms is possible because the rewrite rules are context preserving. It is sufficient to check at the beginning before rewriting that the context is consistent. The context of the result of rewriting is implied by the original context and is consistent. Further, there is no need to store any intermediate contexts either, or to check whether the context of an intermediate result implies the context of a rewrite rule before its application. Thus, a lot of checking is avoided that is required when manipulations are manually done.

As an exercise, the reader may wish to compare the above proof with a rather tedious proof that would be generated if powerlists were defined as a subsort of lists (as given in Appendix B).

Many such proofs of properties of function definitions used in defining `pbm` as well as properties of `pbm` and other algorithms can be mechanically generated by *RRL*, Rewrite Rule Laboratory, a theorem prover mechanizing induction and equational reasoning using rewriting techniques [17]. Interestingly, these mechanically generated proofs turn out to be similar to carefully handcrafted proofs reported in [20, 1]; for details, see [14, 15].

8.1.2 An Example of a "Semantic" Data Structure

As an example of a "semantic" data structure, consider specifying *sorted lists* of items, where the items admit a total ordering. In most specification techniques, sorted lists would be specified as a subsort (subtype) of a more general data type *lists*. A sorted list can be specified as a list satisfying the property of sortedness. However, it is unclear how sorted lists can be specified directly as a data type in its own right, without having to define it as a special subset of lists.

Using applicability conditions, one can specify sorted lists directly, as follows. A term `consl(i, l)` is not legal if `l` is nonempty and `i` is greater than some element in `l`, similar to a syntactically type-incorrect term if arguments to a function have types different from what the function is specified to take.

```
[ el() : sorted_list ]
[ consl(i:item,l:sorted_list):sorted_list]         empty(l) or
   (not(empty(l)) and (i<=min(l)))
[ empty(l:sorted_list): bool ]
[ min(l:sorted_list): item ]                       not(empty(l))
[ insert(l:sorted_list, i:item): sorted_list ]

1.  empty(el())  =  true,
2.  empty(consl(i, l))  =  false,
3.  min(consl(i, l))  =  i,
4.  insert(el(), i)  =  consl(i, el()),
5.  insert(consl(i, l), j) =
       cond(j <= i, consl(j,consl(i,l)), consl(i,insert(l,j))).
```

Consider Equation 3 above. It says that `min(consl(i, l))` is `i` itself; this is so because the applicability condition of `consl` ensures that `i` is `<=` every element in `l`. Similarly, it can be proved that after the insertion of an element by `insert` into a sorted list, the result is also a sorted list. In the above axioms, variables `i` and `j` range over the items, the variable `l` ranges over sorted lists, and the applicability condition for each function symbol appearing in an axiom is assumed to be satisfied.

If sorted lists were specified as a subsort of lists, then either the axiom defining `min` would have to change considerably, or the axiom would have to be made into a conditional equation, with the condition that `l` is sorted. Similar modifications would have to be made to the definition of `insert`. In doing proofs of properties of algorithms on sorted lists, conditions about sortedness would have to be explicitly included.

Many other examples of data structures exist for which it is meaningful for constructors to be partial, for instance, matrices in linear algebra and geometric constructions. See [22] for additional examples.

In the rest of the article, we formalize the concepts of applicability conditions, contexts, and context-preserving rewrite rules, and we propose a reasoning calculus for equations with implicit contexts and its mechanization. We do not include a discussion of semantical issues—particularly, proof theoretic and model theoretic semantics of specifications with applicability conditions; they will be considered in [13]. In the next subsection, we briefly review how specifications developed with the proposed approach have been used for reasoning about data-parallel algorithms and hardware descriptions. This is followed by a brief discussion of related work on different attempts to extend algebraic specification methods for treating partial functions.

8.1.3 Current Results

Along with developing an approach for axiomatizing regular and semantic data structures using partial constructors, a main objective of this research has been to explore whether existing reasoning tools for algebraic specifications, such as the theorem prover *RRL*, can be adapted to reason about such axiomatizations.

In [14, 15], it is shown how reasoning by using context-preserving rewrite rules can be supported in *RRL*. Mechanically generated proofs using *RRL* about properties of algorithmic and hardware descriptions of powerlists are discussed. In [14], two different approaches for specifying powerlists[2] are contrasted with the proposed approach from the perspective of the nature of mechanically generated proofs (i.e.,

[2] These specifications of flat powerlists are included in Appendix B.

their structure, the number of cases considered, and length) of properties of parallel algorithms defined using powerlists. Batcher's merge sort algorithm (the function `pbm` above) and a nontrivial transformation for embedding a multidimensional array into a hypercube such that adjacency among elements in the multidimensional array is preserved [20, 19] are used for illustration and comparison. An adjacency-preserving embedding transformation can be used to transform normal parallel algorithms operating on multidimensional arrays into equivalent parallel algorithms on a hypercube and other related architectures [19].

The use of applicability conditions and the ability to mechanical generate contexts result in elegant and compact specifications. Context-preserving rules make mechanization of reasoning about such specifications using rewrite rules easier to handle (fewer cases and shorter formulas to be considered, and fewer manipulations), in contrast to viewing these data types as subsorts of more general data types. As an example, in one of the proofs of an adjacency property in a multidimensional array being preserved in a hypercube, the number of cases decreases dramatically with the proposed approach (from over 200 to less than 50). That reduction is not surprising because one observes similar gains in going from a untyped logic to a typed logic for reasoning about algorithms involving many data types.

The proposed approach enables separation of concerns by factoring out applicability conditions from definitions, and instead associating them with the signature. Checks for applicability conditions and contexts are separated from the application of axioms defining functions, thus factoring out routine reasoning from reasoning pertinent to the function definitions.

Proofs generated from specifications expressed using applicability conditions are easier to generate, understand, and present, in contrast to proofs generated from specifications using other methods. Many of the mechanically generated proofs of properties of algorithms on powerlists, for instance, are similar to the well-handcrafted proofs reported in [20, 1].

8.1.4 Related Work

The applicability condition of an operation can also be thought of as a precondition á la Floyd and Hoare's axiomatic method. A data structure can be specified by using the so-called abstract model approach proposed in [7], in which constructors and nonconstructors can have nontrivial preconditions. It is unclear how preconditions can be used to efficiently and mechanically reason about the interaction among the operations of the data structure when they are used for defining algorithms such as Batcher's merge sort (the function `pbm`). In [9], an approach for showing correctness of implementations with respect to such specifications is discussed based on partial algebras and partial homomorphism. The approach appears to be quite complicated and not easy to automate.

In [6] and [10], preconditions on nonconstructors are used to specify their exceptional behavior. In this article we have proposed the use of an applicability condition to characterize the subdomain of inputs on which the function behavior is of interest. The use of a precondition for specifying exceptional behavior is orthogonal to the issue of intentional incompleteness captured by an applicability condition. We have not considered the specification of exceptional behavior; the proposed approach can be extended to specify exceptional behavior also by associating a precondition with a function for specifying exceptional behavior, and an applicability condition for specifying intentional partiality.

Mosses [21] surveyed numerous extensions to the algebraic specification approach for handling sorts and subsorts, exceptions, partial operations, etc.; notable among these extensions are error algebras, exceptional algebras, partial algebras, hierarchical specifications, and order-sorted specifications. None of these extensions appears to model, elegantly and adequately, regular data structures such as powerlists and complete binary trees because such structural information needs to be specified semantically. Methods for reasoning about specifications expressed using these approaches have yet to be explored and automated.

The approach discussed in [22] based on partial algebras appears to be the most closely related to our proposed approach. However, Reichel's approach seems to be complicated because of its attempts to develop a theory analogous to the initial algebra semantics for total algebras. Moreover, many requirements are imposed on the generalized substitution rule (pp. 102–105). Most important, we are unaware of any automated tools for reasoning about specifications written using this approach.

In contrast, the approach proposed in this article is simple and mechanizable, and it has an elegant semantics. A standard model can be developed by first translating axioms in a specification into conditional axioms using contexts, constructing a model of conditional axioms, and restricting the initial model to domains of interest as specified by the applicability conditions of the operations. The denotation of a function symbol in the model is a *minimally defined* function (that is, a function is undefined on an input unless the specification requires it to be defined). A detailed semantics will be given in [13]. Most important, it is demonstrated in [14, 15] how rewrite-rule–based provers such as *RRL* developed for total algebras can be extended to reason about partial algebras efficiently if axioms can be oriented into context-preserving rewrite rules. Automated induction methods and heuristics generalize to work on partial algebras. Under these assumptions, reasoning about partial functions is as simple and elegant as reasoning about total functions.

8.2 Specifications with Applicability Conditions

The algebraic specification method for abstract data types as introduced in [5, 25, 4] involves giving the syntactic specification (signature) of operations (function symbols) of the data type and a finite set of equations (or conditional equations) relating different sequences of operations to express the semantics of the operations. All operations are assumed to be total functions. This approach is well studied, and several books explain the approach with numerous examples [2, 8]. For simplicity, it is assumed below that a specification defines a single abstract data type, called D, in a bottom-up hierarchical manner, using other data types already defined elsewhere. Let F be the set of operations of D, and let $C \subseteq F$ constitute the set of constructors of D that generate all the values of the data type. Once the constructors are identified, any relation among them can be specified using equations.

8.2.1 Applicability Conditions

We extend the algebraic specification method by including in the signature of each function symbol an *applicability condition* that specifies the subset of the input domain (as given by its signature) on which the function is applicable. The default applicability condition is `true`, and is often omitted. Function symbols used in specifying applicability conditions can also be partial and have nontrivial applicability conditions associated with them (e.g., `min` on `sorted_list`).

An applicability condition must satisfy certain conditions. (i) Each applicability condition is a quantifier-free and-or combination of equalities (without negation) expressed using the function symbols associated with D and the function symbols associated with other data types; negated equalities are not allowed, since models cannot be easily constructed. (ii) Applicability conditions of functions symbols in F are specified acyclically; that is, the applicability condition of a function symbol f of D is not expressed (directly or indirectly) using another function symbol g of D if the applicability condition of g is expressed using f. The second condition is especially important to ensure that the domain of values generated by partial constructors is nonempty.

We assume that the applicability conditions used in a specification are already fully expanded: either the functions used in the applicability conditions have `true` as their applicability condition, or the nontrivial applicability conditions of these functions have already been incorporated. For example, the applicability condition of `consl` in sorted lists above already includes `not(empty(l))`, the applicability condition of `min`.[3]

To keep the presentation simple, mutually recursively defined function symbols will not be considered.

8.2.2 Interpreting Axioms

An algebraic specification is a nonlogical theory in a multisorted (conditional) equational logic. Axioms are equations of the form $l = r$ relating two terms of the same type, or conditional equations of the form $l = r\ if\ b$, where b is a conjunction of Boolean literals. As illustrated above, the special primitive *cond* can sometimes be used to transform some conditional equations to unconditional equations. Variables appearing in the axioms are assumed to be universally quantified.

The economy of expression and reasoning in extending the specification method are achieved because of an appropriate interpretation of axioms. Much like type constraints on variables and subterms, which require that the arguments to every function symbol be of appropriate types, the arguments to function symbols in the proposed extension additionally satisfy the applicability condition of the function symbol. Universally quantified variables must satisfy the constraints imposed because of the applicability conditions. As an example, consider Equation 13 from the above definition of Batcher's merge sort function `pbm`.

13. $\mathtt{pbm}(\mathtt{zip}(\mathtt{x},\mathtt{y}),\mathtt{zip}(\mathtt{u},\mathtt{v})) = \mathtt{compare}(\mathtt{pbm}(\mathtt{x},\mathtt{v}),\mathtt{pbm}(\mathtt{y},\mathtt{u}))$.

Since `zip, compare`, and `pbm` have the applicability condition stating that their arguments must be of the same length, it can be derived that these applicability conditions require that powerlists `x, y, u, v` be of the same length. The context of this equation is implicit, since it can be generated as discussed in the next subsection.

Equations with implicit contexts can be viewed as abbreviations for equivalent conditional equations in which contexts are made explicit. If one uses contexts implicitly, most axioms defining functions of a data type can be written as equations. This economy of expression not only results in compact and elegant specifications, but also facilitates reasoning because equations can be oriented as unconditional rewrite rules, which can be efficiently mechanized in contrast to conditional rewrite

[3]The proposed approach handles cases in which functions used in expressing applicability conditions themselves have nontrivial applicability conditions. While building contexts as discussed in a later subsection, one has to recurse with the acyclicity requirement guaranteeing the termination of recursive generation of contexts.

rules. In proofs by induction also, fewer cases need to be considered when functions are defined using terminating unconditional rewrite rules [24, 17, 14]. In the next section, we discuss a rewrite-rule–based approach for reasoning using equations with implicit contexts.

If a data type such as powerlists is modeled as a subsort, conditional equations have to be used. See Appendix B, for instance, in which flat powerlists are specified as a subsort of lists. Axioms have conditions explicitly included. While using such axioms with explicit conditions for reasoning, it becomes necessary to establish the conditions repeatedly whenever such axioms are used.

If the applicability condition of a function symbol is a disjunction, then an equational axiom with an implicit context serves as a compact representation for many conditional equations in which contexts are made explicit. As an example, in the above specification of sorted lists, `consl(i, l)` has

$\mathtt{empty}(\mathtt{l})$ or $(\mathtt{not}(\mathtt{empty}(\mathtt{l}))$ and $(\mathtt{i} <= \mathtt{min}(\mathtt{l})))$

as its applicability condition. Equation 5 in that specification

```
insert(consl(i,l),j) =
   cond(j<=i, consl(j,consl(i,l)), consl(i,insert(l,j)))
```

would result in two conditional equations when context is explicitly included in axioms (unless conditions in conditional equations are allowed to be expressed using disjunction). In proofs involving an equation with a disjunctive implicit context, many subcases (two in the above example) are reduced to a single case.

Contexts Constraints imposed on axioms due to applicability conditions are collected as *contexts* that can be generated mechanically. Given a function $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k})$ with an applicability condition ϕ_f, the context of a nonvariable term $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{k})$, written as $\mathtt{context}(\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{k}))$, can be recursively constructed:

$\mathtt{context}(\mathtt{x}) = \mathtt{true}$

where $\mathtt{x}$ is a variable. Let $\mathtt{context}(\mathtt{t}_1) = \phi_{\mathtt{t}_1}, \ldots, \mathtt{context}(\mathtt{t}_\mathtt{k}) = \phi_{\mathtt{t}_\mathtt{k}}$;

$\mathtt{context}(\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{k})) \;=\; \phi_{\mathtt{t}_1}$ and $\ldots$ and $\phi_{\mathtt{t}_\mathtt{k}}$ and $\sigma(\phi_\mathtt{f})$

where $\sigma = \{\mathtt{x}_1 \leftarrow \mathtt{t}_1, \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{t}_\mathtt{k}\}$.[4] Of course, the context of a term can be inconsistent.

[4]This construction assumes that the applicability condition ϕ_f is fully expanded; that is, the nontrivial applicability conditions of functions used in expressing ϕ_f are already considered in ϕ_f. If that is not the case, $\mathtt{context}(\phi_\mathtt{f})$ would have to be recursively constructed first, and the substitution σ would be applied on $\mathtt{context}(\phi_\mathtt{f})$.

It should be easy to see from the above construction that

$\texttt{context}(\texttt{t}) \Rightarrow \texttt{context}(\texttt{t}')$

where `t'` is a subterm of `t`.

The context of an equation $\texttt{s} = \texttt{t}$, written as $\texttt{context}(\texttt{s} = \texttt{t})$, is the conjunction of $\texttt{context}(\texttt{s})$ and $\texttt{context}(\texttt{t})$; similarly, the context of a conditional equation includes, in addition, the context of its condition. For example, the context of the left side of Axiom 13 above about the function `pbm` is

```
length(x)=length(y) and length(u)=length(v) and
   length(zip(x,y))=length(zip(u,v));
```

the context of the right side is

```
length(x)=length(v) and length(y)=length(u) and
   length(pbm(x,v))=length(pbm(y,u)).
```

The context of Axiom 13 is the conjunction of these formulas.

Similarly, the context of Axiom 5 of sorted lists is

```
(empty(l) or (not(empty(l)) and (i <= min(l)))) and
(empty(consl(i,l)) or (not(empty(consl(i,l))) and
   (j<=min(consl(i,l))))) and
(empty(insert(l,j)) or (not(empty(insert(l,j))) and
   (i<=min(insert(l,j))))).
```

Contexts can be simplified using properties of functions to express applicability conditions, for example, `length` for powerlists and `empty`, and `min` for sorted lists. The above context simplifies using the properties that

$\texttt{empty}(\texttt{consl}(\texttt{i},\texttt{l})) = \texttt{false}, \qquad \texttt{empty}(\texttt{insert}(\texttt{l},\texttt{j})) = \texttt{false}$

to

```
(empty(l) or (not(empty(l)) and (i <= min(l)))) and
(j <= min(consl(i,l)) and i <= min(insert(l, j))).
```

Similarly, the context of Axiom 13 about `pbm` simplifies to

```
length(x) = length(y) = length(u) = length(v).
```

using the properties that `length(zip(x, y)) = length(x) + length(y)` and `length(pbm(x, y)) = length(x) + length(y)`.

8.3 Equational Reasoning Using Rewrite Rules

Below, we briefly review an equational calculus with implicit contexts; more details will be given in [13]. Most of the inference rules are the same as in the classical equational calculus, except that for transitivity, one needs to be careful, as explained below.

Given a set E of equations with implicit contexts, the following rules of inference can be used to generate equational consequences with their implicit contexts.

Membership: $E \vdash s = t$, for any $s = t \in E$.

Reflexivity: $E \vdash t = t$, for any term t.

Symmetry: From $E \vdash s = t$, $E \vdash t = s$.

Application preservation: From $E \vdash s_1 = t_1, \cdots, s_k = t_k$, $E \vdash f(s_1, \cdots, s_k) = f(t_1, \cdots, t_k)$.

Instantiation: Given a substitution σ with a consistent context for each variable assignment, and $E \vdash s = t$, $E \vdash \sigma(s) = \sigma(t)$.

Inconsistent context: $E \vdash s = t$ if $E \cup \{context(s = t)\}$ is inconsistent.

The transitivity property does not hold for equations with implicit contexts; in other words, given $E \vdash s = t$, $E \vdash t = u$, it is not sound to deduce that $E \vdash s = u$, since t may not occur in $s = u$. This becomes clear if implicit contexts are made explicit and are included in the formulas. $E \vdash s = t$ means that $(context(s) \ \wedge \ context(t)) \Rightarrow s = t$ can be inferred from E, Similarly, $E \vdash t = u$, means that $(context(t) \wedge context(u)) \Rightarrow t = u$ can be inferred from E. From these, it follows that $(context(s) \ \wedge \ context(t) \ \wedge \ context(u)) \Rightarrow s = u$ can be inferred from E. The context of $s = u$ is, however, $context(s) \ \wedge \ context(u)$. Hence, it is not always the case that $E \vdash s = u$.

In the next subsection, we impose a condition on axioms to deal with this technicality so that the transitivity property holds. Transitivity is required to use equations as rewrite rules. The concept of context-preserving rewrite rules is introduced. With context-preserving rewrite rules, it is possible to use term-rewriting–based methods, particularly, normalization and the cover set induction method, supported in *RRL* [17, 24]). Except for ensuring that a given goal has a consistent context, no additional overhead is involved while simplifying the goal by rewriting, since contexts of intermediate terms do not have to be stored, and intermediate contexts need not be checked for consistency. These properties enable reasoning to be mechanized efficiently, since considerable checking is avoided.

This approach also gives a nice way to reason about partial functions using standard term rewriting methods without extensive modifications to the theorem prover *RRL*. Axiomatization of partial functions and mechanically reasoning about them have been a source of considerable technical difficulty [21, 22].

8.3.1 Context-Preserving Rules

Unconditional equations can be manipulated as rewrite rules whenever it is possible to orient them using termination orderings [3]. This is the way reasoning about equations is automated in *RRL*. Conditional equations can also be handled using conditional rewrite rules, but the rewriting process is expensive because at every rewrite, the condition of a conditional rewrite rule must be satisfied before it can be applied. Checking whether the condition of a conditional rewrite rule is satisfied for a particular substitution itself involves rewriting recursively.

An equation $s = t$ is oriented as either $s \rightarrow t$ or $t \rightarrow s$, depending upon whether $s > t$ or $t > s$ in a termination ordering $>$ used to orient equations. A termination ordering is a well-founded partial ordering that is preserved under substitutions ($s > t \Rightarrow \sigma(s) > \sigma(t)$ for any σ) and has the subterm property ($f(\cdots s \cdots) > s$) and the monotonicity property ($s > t \Rightarrow f(\cdots s \cdots) > f(\cdots t \cdots)$). (For a conditional equation $s = t \ if \ c$, a conditional rule $s \rightarrow t \ if \ c$ is made if $s > t$ and $s > c$.) A finite set of axioms defining a function symbol is oriented into terminating rewrite rules, thus also guaranteeing that the function computation terminates.

To orient axioms with implicit contexts into terminating rewrite rules using a termination ordering, axioms are required to satisfy additional conditions. Below, we show how to arrive at these conditions.

Consider a rewrite step obtained by a single application of a context-preserving rule. An equation $s = t$ rewrites to $s' = t$ (without any loss of generality) in one step using a context-preserving rule $l \rightarrow r$ at position p if and only if

$$s|p = \sigma(l), \quad s' = s[p \leftarrow \sigma(r)].$$

How can one ensure that $context(s = t) \Rightarrow context(s' = t)$ so that there is no need to check for consistency of the context of the new equation and so that there is no need to compute and store the context of the new equation?

First, $context(\sigma(l))$ should imply $context(\sigma(r))$, which can be ensured by requiring that for a context-preserving rewrite rule $l \rightarrow r$, $context(l) \Rightarrow context(r)$. Then, a monotonicity property is needed.

A rewrite rule $l \rightarrow r$ is called *monotonic* with respect to an applicability condition $\phi_f(x_1, \cdots, x_k)$ associated with a function $f(x_1, \cdots, x_k)$ if for every x_i such that x_i and l have the same type, $\{x_i \leftarrow l\}(\phi_f) \Rightarrow \{x_i \leftarrow r\}(\phi_f)$, that is, the result of substituting l for x_i in ϕ_f implies the result of substituting r for x_i in ϕ_f. Obviously, this should be established without assuming the axiom corresponding to the rule under consideration.

A rewrite rule $l \rightarrow r$ is called *monotonic* if it is monotonic with respect to the applicability condition of every function symbol under consideration.

For the powerlist example, the applicability conditions of a binary function `f(x, y)` is either true or `length(x) = length(y)`. To show that a rule relating terms of type powerlists is monotonic, the length of the left side of the rule should be proved equal to the length of its right side.

A rewrite rule $l \rightarrow r$ is called *context preserving* if

1. $context(l)$ is consistent,
2. $context(l) \Rightarrow context(r)$, and finally,
3. $l \rightarrow r$ is monotonic.

If the built-in function symbol `cond` appears in the right side r of a rule, it is too restrictive to require that $context(l)$ implies $context(r)$. Instead, $context(l) \wedge c_i$ is required to imply $context(r_i)$, where r_i is the simplified form of r under the condition c_i constructed from arguments of `cond` in r.[5] For instance, to show that Axiom 5 of sorted lists can be oriented into a context-preserving rule, it suffices to show that

```
((empty(l) or (not(empty(l)) and (i <= min(l)))) and (j <= i))
         implies  (j <= min(consl(i,l))),
```

as well as

```
((empty(l) or (not(empty(l)) and (i <= min(l)))) and not(j <= i))
        implies  (i <= min(insert(l, j))).
```

It now follows by induction that when an equation $s = t$ is rewritten to $s' = t$ in one step using a context-preserving rule, then

$context(s = t) \Rightarrow context(s' = t)$.

[5]Rewriting by conditional context-preserving rewrite rules can also be defined accordingly.

Thus, if $s = t$ has a consistent context, so does $s' = t$.

Given two rewrite steps $s \rightarrow t$ and $t \rightarrow u$ using context-preserving rules, it is sound to deduce that $s \rightarrow^+ u$. This is because $context(s) \Rightarrow context(t)$ and $context(t) \Rightarrow context(u)$, which means that $context(s) \Rightarrow context(u)$. Unlike in the case of arbitrary rules, where it would be necessary to carry the intermediate context $context(t)$ along with the deduction $s = u$, there is no need to do so for context-preserving rules. This property allows us to use context-preserving rewrite rules for simplification much like ordinary rewrite rules for conventional algebraic specifications (in which applicability conditions are true).

Given an equation $s = t$ with a set $\mathcal{R}$ of context-preserving rewrite rules, if a normal form of s with respect to $\mathcal{R}$ is equal to a normal form of t, then $s = t$ is an *equational consequence* of $\mathcal{R}$ (but not vice versa; for the converse to hold, additional conditions, such as confluence, on $\mathcal{R}$ must be imposed). This definition is identical to the one for the case of rewrite rules made from equations without any implicit contexts. Consequently, the normalization inference rule of *RRL* can be freely used for proving equations and conditional equations.

8.3.2 Checking for Context Preservation

Checking whether a rule is context preserving typically involves the use of definitions and properties of function symbols appearing in the applicability conditions as well as other function symbols in a rule.

For powerlists, it involves first proving properties such as

```
length(zip(x, y)) = length(x) + length(y),
length(minp(x, y)) = length(x),
length(maxp(x, y)) = length(x),
length(compare(x, y)) = length(x) + length(y),
length(pbm(x, y)) = length(x) + length(y).
```

For instance, the following rule

4. `zip(tie(x,y),tie(u,v))` $\rightarrow$ `tie(zip(x,u),zip(y,v))`

in the definition of `zip` can be shown to be context preserving as follows.

The context of the left side is

```
length(x)=length(y) and length(u)=length(v) and
   length(tie(x,y))=length(tie(u,v)).
```

Under the condition that `length(x) = length(y)`, `length(tie(x, y))` can be rewritten to `length(x) + length(y)`. This further rewrites to `length(x) + length(x)`. Similarly, under the condition that `length(u) = length(v)`, `length(tie(u, v))` rewrites to `length(u) + length(u)`. The context of the left side now simplifies, using the property that `length(x) > 0` and properties of `+`, to

$$\texttt{length}(\mathtt{x}) = \texttt{length}(\mathtt{y}) = \texttt{length}(\mathtt{u}) = \texttt{length}(\mathtt{v}).$$

If one uses the lemma about the length of a zipped powerlist,

$$\texttt{length}(\texttt{zip}(\mathtt{x}, \mathtt{u})) = \texttt{length}(\mathtt{x}) + \texttt{length}(\mathtt{u}),$$

this formula implies the context of the right side, which is

```
length(x)=length(u) and length(y)=length(v) and
   length(zip(x,u))=length(zip(y,v)).
```

We now show that Rule 4 above is also monotonic. Every function has either the applicability condition `length(x) = length(y)` or `true`. For Rule 4 to be monotonic with respect to `length(x) = length(y)`, it is enough to show that `length` of its left side is equal to the length of its right side, namely,

$$\texttt{length}(\texttt{zip}(\texttt{tie}(\mathtt{x}, \mathtt{y}), \texttt{tie}(\mathtt{u}, \mathtt{v}))) = \texttt{length}(\texttt{tie}(\texttt{zip}(\mathtt{x}, \mathtt{u}), \texttt{zip}(\mathtt{y}, \mathtt{v})))$$

assuming its context. This follows from the properties of `length, tie,` and `zip`.

This establishes that Rule 4 is indeed context preserving.

Proofs showing that rules are context preserving have to be done only once; they can be done when axioms are oriented into context-preserving rewrite rules. This check can be viewed to be similar to type-checking a formula, except that the check involves the use of semantic information and reasoning, typically proofs by induction.

All equations and conditional equations in the specifications in this paper can be shown to transform into context-preserving rules. The proof of this property for each rule can be mechanically carried out in *RRL*. For nested powerlists (whose specification is given in Appendix C), it can be shown, similar to flat powerlists, that

```
similar(zip(x, y), tie(x, y)),
similar(minp(x, y), x),
similar(maxp(x, y), x),
similar(zip(u, v), tie(x, y)) = similar(u, x) and similar(v, y).
```

and so on.

This requirement on axioms to be orientable to context-preserving rules does not seem to be restrictive at all. In fact, we have been unsuccessful in constructing an example of a meaningful data structure with partial constructors that is axiomatized using rewrite rules and in which the rules are not context preserving.

8.4 Context-Preserving Definitions

For mechanically proving properties of functions from their definitions, axioms in a specification are input to *RRL* in two parts: definitions and properties. For axioms in a definition, `:=` is used to separate two sides of the equality, instead of `=`. Each axiom in a definition of a function symbol `f` is such that its left side is of the form $\mathtt{f}(\mathtt{s}_1, \cdots, \mathtt{s}_\mathtt{k})$, in which typically $\mathtt{s}_\mathtt{i}$'s are constructor terms in constructor-based definitions. But that need not be the case; as we saw above, most functions on powerlists, such as `leq, minp, maxp, compare,` and `pbm`, are defined using the nonconstructor `zip`.

A function definition is called *context preserving* if all the axioms in the definition can be oriented as context-preserving rewrite rules.

The applicability condition associated with a function is used in determining whether a given set of rewrite rules completely defines the function. For instance, `zip` on powerlists is intended to be defined only on powerlists of equal length. To test whether the rules corresponding to two equations for `zip` (Rules 3 and 4 in Subsection 1.1) completely define `zip`, we check whether their left sides cover all cases of pairs of powerlists of equal length. Equation 3 covers the case of singleton powerlists. Equation 4 covers the case of pairs of tied lists: since `tie` operates on lists of equal length, the second equation covers the case when the length of the first argument is `2 * length(x)` and the length of the second argument is `2 * length(u)`. Because of the recursive call to `zip` on the right side of Equation 4, `x` and `u` must have the same length. Hence, Equation 4 indeed covers the case of powerlists of equal length > 1.

In [11], a method for checking completeness of function definitions with applicability conditions is discussed. Using this method, one can distinguish intentional incompleteness (as is the case here) from omissions of cases due to mistakes and/or oversight. One can prove that all the definitions in the specifications given here are complete.

8.5 Inductive Reasoning about Recursive Definitions

One of the main reasons for writing a formal specification is to pin down the assumptions made in the design of an abstraction and enhance confidence in the design process. Formal specifications serve many useful purposes. In particular, a specification is a contract between a designer and an implementor, and it can be used to check the correctness of an implementation. Further, specifications of data structures are used to reason about algorithms using such data structures. Earlier, we reviewed how equational reasoning can be mechanized by rewriting using context-preserving rules generated from equations and conditional equations with implicit contexts. Below, we focus on the use of a specification for reasoning about recursive algorithmic descriptions by induction.

For reasoning about algorithms, proofs by induction are usually employed. First, we need to precisely define the model(s) being used for doing proofs by induction. The model(s) of interest is the minimal model generated using the constructors of the data type. Even in the presence of nontrivial applicability conditions, such a model can be incrementally constructed insofar as the applicability conditions are expressed acyclically. A detailed construction will be given in [13]. Here we briefly illustrate the construction for flat powerlists.

Powerlists are constructed incrementally using `s` and `tie`, but since `tie` uses `length` in its applicability condition, a model is constructed by successively approximating the denotations of `s, tie`, and `length`. The denotation of `length` is defined on singleton powerlists. With that, `tie` can be defined on singleton lists, thus constructing powerlists `tie(s(a), s(b))` for arbitrary scalars `a, b`. Now `length` is defined on these powerlists, and their length is 2. Then, `tie` is applied on powerlists of length 2 to generate powerlists of the form `tie(tie(s(a), s(b)), tie(s(c), s(d)))`, on which `length` can be defined. This process is repeated, giving powerlists of lengths 4, 8, 16, and so on. The model will not have any denotation for a ground term `tie(tie(s(a), s(b)), s(a))`, for instance, since for `tie` to be applicable, its arguments must be of equal length, which is not the case here. In this way, a standard model for powerlists can be constructed.

This model gives rise to an induction principle expressed in terms of the constructors `s` and `tie`, namely, that if a property $\alpha(\mathtt{x})$ with a single free variable `x` of type (flat) powerlist is such that $\alpha(\mathtt{s(a)})$ can be proved for all scalars `a`, and $\alpha(\mathtt{tie(u,v)})$ can be proved for all powerlists `u, v` such that `length(u) = length(v)`, then $\alpha(\mathtt{x})$ is proved.

For a data type D with a finite set C of constructors, the induction principle is essentially that of structural induction with the modification that arguments to a constructor must satisfy the associated applicability condition. To prove a property $\alpha(x)$, where x is a variable of type D, it suffices to prove for every constructor ground term g satisfying its context, $\alpha(g)$. For illustration, assume that (i) all constructors in C are either unary or binary, (ii) unary constructors do not take any argument of type D, and (iii) both arguments of each binary constructor are of type D (as in the case of flat powerlists). Then, a structural induction scheme would be

$$\frac{\begin{array}{ccc} \phi_a(x) & \Rightarrow & \alpha(a(x)) \\ \phi_b(x_1, x_2) \wedge \alpha(x_1) \wedge \alpha(x_2) & \Rightarrow & \alpha(b(x_1, x_2)) \end{array}}{\alpha(x).}$$

Above, $\phi_a(x)$ is the applicability condition of a unary constructor $a(x)$; $\phi_b(x_1, x_2)$ is the applicability condition of a binary constructor $b(x_1, x_2)$.

As a simple illustration, consider the following function `rev` defined on powerlists.

```
[ rev(powerlist): powerlist ]

14. rev(s(i)) = s(i),
15. rev(tie(x, y)) = tie(rev(y), rev(x)).
```

It can be shown by induction that the above axioms can be oriented into context-preserving rewrite rules; the lemma

$$\texttt{length}(\texttt{rev}(\texttt{u})) = \texttt{length}(\texttt{u})$$

needs to be proved first to show that the two arguments to `tie` on the right side of Equation 15 are of equal length using `length(x) = length(y)`.

A simple property of `rev`,

$$\texttt{rev}(\texttt{rev}(\texttt{u})) = \texttt{u}$$

can be proved by the above induction principle.

The basis case is easy: `rev(rev(s(i)))` simplifies to `s(i)`, which is also the right side.

The induction step is

$$\texttt{rev}(\texttt{rev}(\texttt{tie}(\texttt{x}, \texttt{y}))) = \texttt{tie}(\texttt{x}, \texttt{y})$$

assuming the induction hypotheses

$$\texttt{rev}(\texttt{rev}(\texttt{x})) = \texttt{x}, \quad \texttt{rev}(\texttt{rev}(\texttt{y})) = \texttt{y}$$

and

$\texttt{length}(\texttt{x}) = \texttt{length}(\texttt{y})$.

The term `rev(rev(tie(x, y)))` rewrites by the rule from Axiom 15 to

$\texttt{tie}(\texttt{rev}(\texttt{rev}(\texttt{x})), \texttt{rev}(\texttt{rev}(\texttt{y})))$,

which, using the induction hypotheses, simplifies further to `tie(x, y)`. Notice that we did not have to check the condition that the two arguments to `tie` be of equal length; this is taken care because of the applicability condition.

8.6 Extending the Cover Set Method

When function definitions are not constructor based, that is, they are given using nonconstructors, such as `zip`, as is the case of most function definitions on powerlists given earlier, the above induction principle may not be useful. *RRL* supports the *cover set* method for mechanizing inductive reasoning about functions defined on data structures [17, 24]. This method generalizes the structural induction method by allowing a well-founded termination ordering used for generating induction schemes. The method works for constructor-based definitions also. Below, an informal overview of the cover set method is given, focusing on how it can be extended to consider definitions given using context-preserving terminating rewrite rules. More details about the cover set method can be found in [24, 17, 16, 12].

A cover set $\mathcal{C}$ for a function symbol f with a complete definition D, consisting of a finite set of terminating rewrite rules, is a finite set of triples. Each triple in $\mathcal{C}$ is derived from a rewrite rule in D. For a conditional rule $l \rightarrow r\ if\ cond$, where $l = f(s_1, \ldots, s_k)$ and f does not occur in any of s_i's, the first component of the triple is the k-tuple $< s_1, \ldots, s_k >$ of arguments of f in l, the second component is a set consisting of all k-tuples serving as arguments to f in r, and the third component is the set of all literals in *cond*. A triple could have its second and third components be the empty set. For an unconditional rewrite rule, the third component of the triple is empty. If the right side of a rewrite rule does not have a recursive call to f, the second component is empty. In a cover set generated from a constructor-based definition, the first and second components of each triple in it contain terms only over constructors.

An induction scheme for a conjecture C containing a subterm $f(x_1, \cdots, x_k)$ can be easily generated from a cover set $\mathcal{C}$ of f. An induction scheme is a finite set of induction cases of the form

$<< \sigma_c, cond_c >, \{\cdots, < \theta_i, cond_i >, \cdots\} >,$

where $< \sigma_c, cond_c >$ is used to generate the induction conclusion and each triple in $\{\cdots, < \theta_i, cond_i >, \cdots\}$ is used to generate an induction hypothesis. Each of these components of an induction case is obtained from a cover set triple

$<< s_1, \cdots, s_k >, \ \{\cdots, < s_1^i, \cdots, s_k^i >, \cdots\}, cond >$

in $\mathcal{C}$ as follows: $\sigma_c = \{x_1 \rightarrow s_1, \cdots, x_k \rightarrow s_k\}$, and $cond_c = cond$. Similarly, $\theta_i = \{x_1 \rightarrow s_1^i, \cdots, x_k \rightarrow s_k^i\}$ and $cond_i = cond$.[6]

The left sides of a definition are thus used to generate all the subgoals of a conjecture to be proved, and the recursive calls to f on the right sides of the rules, if any, generate the induction hypotheses for each subgoal. A rewrite rule in a definition without any recursive call in its right side generates a basis case of the induction scheme.

The cover set method enables the use of different induction schemes for the same data structure depending upon how different function definitions on the data structure are expressed. The cover set method has been successfully used to prove many nontrivial theorems by induction on numbers, lists, sequences, arrays, and other recursive data structures [24].

The cover set method can be extended to be applicable to function definitions given using context-preserving rewrite rules (with implicit contexts). As in a conditional rewrite rule in a definition, the subgoal corresponding to the left side of a rewrite rule (unconditional or conditional) with implicit context is applicable only if the implicit context is satisfied in addition to any explicit condition expressed in the rewrite rule. So the third component of a triple in a cover set now also includes literals from the implicit context of the rewrite rule (the context of its left side of a context-preserving rule), in addition to literals in the condition of a conditional rewrite rule. Since the hypotheses are generated from the recursive calls on the right side of a rule, the context of a conclusion implies the context of each of the hypotheses. This extension is illustrated below.

To prove a simple property

`leq(minp(x,y),maxp(x,y))`,

the heuristics in *RRL* would pick a cover set generated from the definition of `minp` (or `maxp`, since they are the same).

[6] The definition of an induction scheme generated from a cover set given in [16, 12] is more general; it includes the position of the subterm $f(x_1, \cdots, x_k)$ in C as well as replacements for the subterm in the induction conclusion and the induction hypotheses.

Two rules define minp: Rule 7, whose right side does not have any recursive calls to minp, and Rule 8, whose right side has recursive calls to minp. The cover set of minp is

$$\{<< \mathtt{s(i)}, \mathtt{s(j)} >, \{\}, \{\} >,$$
$$<< \mathtt{zip(x_1, y_1)},\ \ \mathtt{zip(u_1, v_1)} >, \{< \mathtt{x_1}, \mathtt{u_1} >,\ \ < \mathtt{y_1}, \mathtt{v_1} >\},$$
$$\{\mathtt{length(x_1)} = \mathtt{length(u_1)} = \mathtt{length(y_1)} = \mathtt{length(v_1)}\} >\}.$$

The first element is contributed by Rule 7; its condition

$$\mathtt{length(s(i))} = \mathtt{length(s(j))} = \mathtt{1}$$

trivially evaluates to true and is thus omitted. The second element is contributed by Rule 8, and its condition is the context of the rule.

The method for generating an induction scheme from a cover set is extended to include in the condition of each induction hypothesis, the literals of the context of the subterm used to generate the induction hypothesis. The induction scheme generated from the above cover set is

$$\{< \{\mathtt{x} \to \mathtt{s(i)}, \mathtt{y} \to \mathtt{s(j)}\}, \{\}, \{\} >,$$
$$< \{\mathtt{x} \to \mathtt{zip(x_1, y_1)}, \mathtt{y} \to \mathtt{zip(u_1, v_1)}\},$$
$$\{\mathtt{length(x_1)} = \mathtt{length(y_1)} = \mathtt{length(u_1)} = \mathtt{length(v_1)}\},$$
$$\{< \{\mathtt{x} \to \mathtt{x_1}, \mathtt{y} \to \mathtt{u_1}\}, \{\mathtt{length(x_1)} = \mathtt{length(u_1)}\} >,$$
$$< \{\mathtt{x} \to \mathtt{y_1}, \mathtt{y} \to \mathtt{v_1}\}, \{\mathtt{length(y_1)} = \mathtt{length(v_1)}\} >\} >\}.$$

The first element gives the basis case. The second element is the induction step with two induction hypotheses. The condition for the conclusion is the same as from the corresponding cover set element; the condition for each induction hypothesis is generated from the context of the subterm of the recursive call to minp in the right side of rule 8, giving rise to the hypothesis. It is easy to see that the condition associated with the conclusion substitution implies the condition associated with each hypothesis substitution.

We use the above induction scheme to prove the above conjecture about leq, minp, maxp as follows. The first substitution in the induction scheme is the basis case in which the variables are instantiated to be singleton powerlists.

$$\mathtt{leq(minp(s(i), s(j)), maxp(s(i), s(j))},$$

which is true by case analyses and rewriting using the rules corresponding to Axioms 7, 9, and 5 in the specifications of powerlists.

The subgoal obtained using the second element of the above induction scheme is

$$\mathtt{leq}(\mathtt{minp}(\mathtt{zip}(\mathtt{x_1},\mathtt{y_1}),\mathtt{zip}(\mathtt{u_1},\mathtt{v_1})),\mathtt{maxp}(\mathtt{zip}(\mathtt{x_1},\mathtt{y_1}),\mathtt{zip}(\mathtt{u_1},\mathtt{v_1}))),$$

under the condition

$$\mathtt{length}(\mathtt{x_1}) = \mathtt{length}(\mathtt{y_1}) = \mathtt{length}(\mathtt{u_1}) = \mathtt{length}(\mathtt{v_1}),$$

using the induction hypotheses

$$\mathtt{leq}(\mathtt{minp}(\mathtt{x_1},\mathtt{u_1}),\mathtt{maxp}(\mathtt{x_1},\mathtt{u_1})) \text{ and } \mathtt{leq}(\mathtt{minp}(\mathtt{y_1},\mathtt{v_1}),\mathtt{maxp}(\mathtt{y_1},\mathtt{v_1})).$$

This can also be proved by rewriting using the rules corresponding to Axioms 8, 10, and 6 of powerlists.

In [14, 15], many such examples are discussed. Several properties of recursively defined functions on powerlists are proved using rewriting and the cover set induction method. These properties are then used to prove the correctness of Batcher's merge sort algorithm (the function `pbm` in the above specification). Using the specification of nested powerlists given in Appendix C for modeling multidimensional arrays, we discussed in [14] the design and verification of a transformation to embed a multidimensional array into a hypercube such that the embedding function preserves the adjacency of elements in the array in the hypercube. In [15], carry look-ahead adders and ripple carry adders on bit vectors of size 2^k are described using powerlists; their behavior correctness (i.e., they indeed compute addition on numbers) is mechanically established using *RRL*.

Our experience suggests that the mechanical proofs obtained using the proposed approach are less verbose and more elegant than the proofs obtained when powerlists are defined as a subsort of lists. In the automatic generation of these proofs, the number of cases to be considered is dramatically reduced (in some cases from over 200 to less than 50), since constraints in the context of equations arising in the proof do not have to be explicitly considered.

8.7 Conclusion

A simple extension of the algebraic specification method is proposed for axiomatizing regular data structure such as complete binary trees and powerlists (flat as well as nested), as well as semantic data structures such as priority lists and sorted lists.

As a byproduct of the approach, we get a simple method for treating and reasoning about partial functions including partial constructors. Every function symbol has an associated applicability condition, a semantic property that input must satisfy for the function to be applicable. Axiomatization typically consists of equations in which variables are subject to constraints imposed by the applicability conditions collected as contexts.

In order to reason using axioms with implicit contexts, a condition on axioms is identified. If axioms can be oriented into context-preserving rewrite rules, these rewrite rules can be used for simplification, much as in the case of conventional algebraic specification methods.

The approach has been implemented in our theorem prover *RRL* for equational and inductive reasoning, and it has been used effectively to reason about the properties of many data parallel programs including FFT and Batcher's merge sort [14], as well as hardware descriptions including adders [15]. Mechanical proofs generated by *RRL* are compact and simple and—surprisingly—closely mimic the handcrafted proofs reported in [20, 1].

Many issues related to specification with applicability condition still need to be further investigated. Identifying (speculating) lemmas that would be useful for establishing that rules are context preserving is an interesting challenge. This is likely to require less user intervention as new properties of function definitions are proved and transformed into context-preserving rewrite rules. Another important issue is that of hard-wiring such checks in order to make reasoning more efficient. Efficient automated methods for showing correctness of an implementation need to be explored.

The reader will have noticed that the notion of a regular data structure is discussed informally. No attempt has been made to formalize this concept in this article. Developing a formal definition of a regular data structure is still an open question.

Acknowledgments

I thank Jay Misra for suggesting that we attempt proofs of algorithms in his paper using *RRL*. Proofs reported in this paper were done together with M. Subramaniam. I also thank him and Sam Kamin for many useful comments on an earlier draft of this paper.

Appendix A: Axiomatization of Cond

RRL supports `cond` as a polymorphic function parameterized by the type of its second and third arguments for doing case analysis. It can be used to express some conditional equational axioms as equational axioms, as discussed in the body of the paper. The following axiomatization of `cond` is built in *RRL*.

```
[ cond(b: bool, x: t, y: t): t ]

 1. cond(true, x, y) = x,
 2. cond(false, x, y) = y,
 3. cond(b, x, x) = x.
```

If the second and third arguments are also of the Boolean type, there are additional axioms:

```
 4. cond(x, true, false) = x,
 5. cond(x, false, true) = not(x).
```

Terms involving `cond` are simplified first using the rewrite rules corresponding to the above axioms (equations are oriented from left to right). If `cond` occurs in a formula, the case analysis is done on the first argument of `cond`, which is a Boolean term. Let

$\mathtt{f}(..., \mathtt{cond}(\mathtt{b}, \mathtt{s}, \mathtt{t}), ...) \;=\; ... \;\mathtt{if}\; \mathtt{C}$

be a goal being attempted. Then this goal is split into two subgoals:

$\mathtt{f}(..., \mathtt{s}, ...) \;=\; ... \;\mathtt{if}\; \mathtt{C}\; \mathtt{and}\; \mathtt{b}$

$\mathtt{f}(..., \mathtt{t}, ...) \;=\; ... \;\mathtt{if}\; \mathtt{C}\; \mathtt{and}\; \mathtt{not}(\mathtt{b}).$

Appendix B: Specification of Flat Powerlists as a Subsort of Lists

Alternative ways are possible for axiomatizing flat powerlists. One obvious choice is to define powerlists as a subsort of lists—lists of length $2^k, k \geq 0$. Note below that the predicate `plp` is defined by an axiom that is existentially quantified. It can also be equationally axiomatized using auxiliary predicates. Below, we give some definitions.

```
[ nl(): list ]
[ cons(i:item, l: list): list ]
[ length (l:list): naturals ]
[ append(l1: list, l2: list): list ]
[ plp (l: list): powerlist ]
[ s(i:item):  powerlist ]
[ tie(u: powerlist, v: powerlist): powerlist ]
[ zip (u: powerlist, v:powerlist): powerlist]
[ minp(u: powerlist, v:powerlist): powerlist]

 1. length(nl()) = 0,
 2. length(cons(i, l)) = length(l) + 1,
 3. append(nl(), l) = l,
 4. append(cons(i, l1), l2) = cons(i, append(l1, l2)),
 5. plp(l) = (exist k >= 0 length(l) = 2^k),
 6. s(i) = cons(i, nl()),
 7. tie(u, v) = append(u, v) if length(u) = length(v),
 8. zip(s(i), s(j)) = tie(s(i), s(j)),
 9. zip(tie(x,y),tie(u,v)) = tie(zip(x,u),zip(y,v))
       if (length(x)=length(y) and length(u) = length(v) and
       length(x) = length(u) and length(y) = length(v)),
10. minp(s(i), s(j)) = cond(i <= j, s(i), s(j)),
11. minp(zip(x,y),zip(u,v)) = zip(minp(x,u),minp(y,v))
       if (length(x)=length(y) and length(u) = length(v) and
       length(x) = length(u) and length(y) = length(v)).
```

The reader will notice the use of conditions expressed using `length` to specify the regularity of powerlists. In every definitional axiom dealing with nonsingleton powerlists, such conditions on variables must be enforced.

The above axiomatization was our first attempt at specifying powerlists in *RRL*. We soon observed that when such an axiomatization is used to reason about algorithms, it becomes necessary to carry conditions around and then repeatedly discharge them for completing proofs every time the axioms are applied. Careful case analysis must be done; most of the cases are trivial, with one condition expressing the regularity property of an argument and another condition violating the regularity property of another argument. Nevertheless, these extra conditions lead to long and tedious proofs with many cases. Proofs by induction are done using the constructors of lists, instead of those of powerlists. See [14] for more details about proofs done using this specification.

A simple order-sorted specification [23] of powerlists would perhaps be similar to the above specification. Another possible way is to define infinitely many subsorts of lists by parameterizing subsorts by the length of lists. But then functions such as `tie, zip,` and `minp` would have to be represented by families of infinitely many functions, one for each subsort corresponding to length 2^k. It is unclear how order-sorted specifications of either kind would be helpful in efficiently reasoning about recursively defined functions such as `pbm`.

A slight variation over the above formalization is the following axiomatization without using `nl, cons, append`. Instead `s, tie` are used as free constructors, and nonconstructors such as `zip, minp`, etc., are defined only for those powerlists generated by `tie` with arguments of equal length. This requirement is expressed as a condition in the definitions below.

```
[ length (l: powerlist): naturals ]
[ s(i:item):  powerlist ]
[ tie(u: powerlist, v: powerlist): powerlist ]
[ zip (u: powerlist, v:powerlist): powerlist]
[ minp(u: powerlist, v:powerlist): powerlist]

1. length(s(i)) = 1,
2. length(tie(u, v)) = length(u) + length(v),
3. zip(s(i), s(j)) = tie(s(i), s(j)),
4. zip(tie(x,y), tie(u,v)) = tie(zip(x,u), zip(y,v))
      if (length(x)=length(y) and length(u) = length(v) and
      length(x) = length(u) and length(y) = length(v)),
5. minp(s(i), s(j)) = cond(i <= j, s(i), s(j)),
6. minp(zip(x,y),zip(u,v)) = zip(minp(x,u), minp(y,v))
      if (length(x)=length(y) and length(u) = length(v) and
      length(x) = length(u) and length(y) = length(v)).
```

The reader will again notice the use of conditions expressed using `length` for defining `zip, minp`, etc., on nonsingleton powerlists.

This was our second attempt at axiomatization of powerlists before we developed the approach discussed in this paper. In [14], different proofs of the property of the `rev` function on powerlists are discussed for an illustrative comparison using each of the above three formalizations on *RRL*. The first formalization using applicability conditions leads to a simple proof on *RRL* involving two cases. The second formalization, modeling powerlists as a subsort of lists, produces the most complex proof involving nine cases. The third formalization (the one above in which `s, tie` are free constructors) results in a proof involving four cases. Such reductions in proof lengths are typical. The factor of reduction in cases to be considered goes up for nontrivial function definitions given in terms of other function definitions.

Using the proofs of the correctness of the `pbm` function as well, it is shown in [14] that elegant and compact proofs, very similar to carefully hand-crafted proofs reported in [20], can be generated using *RRL* from specifications expressed using applicability conditions and implicit contexts. Doing these proofs about the properties of `pbm` using the above two axiomatization instead leads to complicated and long proofs with many cases that can be avoided using implicit contexts.

Appendix C: A Specification of Nested Powerlists

To exhibit the expressive power and elegance of our specification method, we extend the specification of flat powerlists using applicability conditions to axiomatize nested powerlists. Nested powerlists are used in [20] to model multidimensional arrays. An additional constructor, `n`, stands for nesting. The applicability condition is changed from `length(u) = length(v)` to `similar(u, v)`. Below, we give definitions of the some of the functions.

```
[ s(item):  powerlist ]
[ tie(u: powerlist, v: powerlist): powerlist ]     similar(u, v)
[ n(u: powerlist): powerlist]
[ length(powerlist): naturals ]
[ depth(powerlist): naturals ]
[ zip(u: powerlist, v:powerlist): powerlist]      similar(u, v)
[ minp(u: powerlist, v:powerlist): powerlist]      similar(u, v)
[ similar(u: powerlist, v:powerlist): bool]

 1. length(s(i)) = 1,
 2. length(tie(u, v)) = length(u) + length(v),
 3. length(n(x)) = 1,
 4. depth(s(i)) = 1,
 5. depth(tie(u, v)) = depth(u),
 6. depth(n(x)) = depth(x) + 1,
 7. zip(s(i), s(j)) = tie(s(i), s(j)),
 8. zip(tie(x, y), tie(u, v)) = tie(zip(x, u), zip(y, v)),
```

```
 9. zip(n(x), n(y)) = n(zip(x, y)),
10. minp(s(i), s(j)) = cond(i <= j, s(i), s(j)),
11. minp(zip(x, y), zip(u, v)) =  zip(minp(x, u) , minp(y, v)),
12. minp(n(x), n(y)) = n(minp(x, y)),
13. similar(s(i), s(j)) = true,
14. similar(n(x), n(y)) = similar(x, y),
15. similar(tie(x, y), tie(u, v)) = similar(x, u) and similar(y, v),
16. similar(s(i), n(y)) = false,
17. similar(s(i), tie(u, v)) = false,
18. similar(n(x), s(i)) = false,
19. similar(n(x), tie(u, v)) = false,
20. similar(tie(x, y), s(i)) = false,
21. similar(tie(x, y), n(u)) = false.
```

This specification is a natural generalization of the specification of flat powerlists given earlier. A specification exploiting divide-and-conquer strategy turns out to be especially elegant for such data structures. In contrast, a specification of nested powerlists as a subsort of nested lists as we did for flat powerlists in Appendix B turns out to be complicated. At every level, it must be stated that all elements in the list are similar, that is, they are of the same depth d and same length 2^k, for some d and k.

In [14], we discuss a mechanically generated proof, using *RRL*, of the property that the embedding transformation defined in [20, 19] from a multidimensional array of dimensions that are powers of 2 to a hypercube preserves adjacency of neighbors in the array. This property is useful in establishing a transformation from algorithms running on a multidimensional array of processors to a hypercube. A proof of this property turns out to be quite nontrivial and exhibits how the use of context-preserving rules can result in compact proofs with relatively fewer cases. We conjecture that a proof from an axiomatization of nested powerlists as a subsort of nested lists is likely to be involved, long, and difficult to understand, because of numerous cases that might have to be considered.

References

[1] W. Adams. *Verifying Adder Circuits Using Powerlists.* Technical Report CS-TR-94-02, Computer Science Department, University of Texas at Austin, 1994.

[2] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM, 1989.

[3] N. Dershowitz. Termination of rewriting. *J. Symbolic Computation,* 3:69–116, 1987.

[4] J. A. Goguen, J. W. Thatcher, E. G. Wegner, and J. B. Wright. Abstract data types as initial algebras and correctness of data representations. In *Proceedings of the Conference on Computer Graphics, Pattern Recognition, & Data Structures,* pp. 89–93, May 1975.

[5] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types*, Ph.D. thesis, CSRG-59, Department of Computer Science, University of Toronto, Toronto, Canada, 1975.

[6] J. V. Guttag. Notes on type abstractions. *IEEE Trans. on Software Eng.,* SE-6(1):13–23, 1980.

[7] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica,* 4:271–281, 1972.

[8] I. V. Horebeek and J. Lewi. *Algebraic Specifications in Software Engineering.* Springer-Verlag, 1989.

[9] S. Kamin and M. Archer. Partial implementation of abstract data types: A dissenting view on errors. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types,* pp. 317–336, *Lecture Notes in Computer Science*, vol. 173, Springer-Verlag, 1984.

[10] D. Kapur. *Towards a Theory for Data Abstractions.* Technical Report TR-237, MIT LCS, 1980.

[11] D. Kapur. An automatic tool for analyzing completeness of equational specifications. In *Proceedings of ISSTA 94,* Seattle, August 1994.

[12] D. Kapur. Mechanizing induction. In *Proceedings of the National Seminar on Theoretical Computer Science (NSTCS)*, pp. 214–242, Bombay, India, August 1995.

[13] D. Kapur. *Partial data types.* Manuscript in preparation, Department of Computer Science, State University of New York, Albany.

[14] D. Kapur and M. Subramaniam. Automated reasoning about parallel algorithms using powerlists. In V. S. Alagar, editor, *Proceedings of AMAST*, pp. 416–430, *Lecture Notes in Computer Science,* vol. 936, Springer-Verlag, 1995,

[15] D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using powerlists. *J. Formal Methods in System Design* (to appear).

[16] D. Kapur and M. Subramaniam. New Uses of Linear Arithmetic in Automated Theorem Proving for Induction. *J. Automated Reasoning,* 16(1–2):39–78, 1996.

[17] D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *Computers and Math. with Applications,* 29(2):91–114, 1995.

[18] D. Knuth. *The Art of Computer Programming: Sorting and Searching,* vol. 3. Addison Wesley, 1973.

[19] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes.* Morgan Kaufmann, 1992.

[20] J. Misra, Powerlist: A structure for parallel recursion. In *A Classical Mind: Essays in Honor of C. A. R. Hoare,* Prentice-Hall, 1994. A revised version appeared in *TOPLAS,* 16(6):1737–1767, 1994.

[21] P. D. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specifications,* pp. 66–91, *Lecture Notes in Computer Science,* vol. 655, Springer-Verlag, 1991.

[22] H. Reichel. *Initial Computability, Algebraic Specifications and Partial Algebras.* Oxford Univ. Press, 1987.

[23] G. Smolka, W. Nutt, J. A. Goguen, and J. Mesaguer. Order-sorted equational computation. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures* 2, pp. 299–367, Academic Press, 1989.

[24] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction,* pp. 250–265, *Lecture Notes in Computer Science,* vol. 310, Springer-Verlag, 1988.

[25] S. N. Zilles. *Algebraic Specification of Data Types.* Project Mac Progress Report 1974, pp. 52–58. CSG Memo 119, Laboratory for Computer Science, MIT, 1974.

Contributors

Robert S. Boyer
Computer Sciences and Philosophy Departments
University of Texas at Austin
Austin, TX 78712
E-mail: boyer@cs.utexas.edu
URL: http://www.cs.utexas.edu/users/boyer/

Shang-Ching Chou
Department of Computer Science
Wichita State University
Wichita, KS 67260-0083
E-mail: chou@cs.twsu.edu
URL: http://www.cs.twsu.edu/~chou/chou.html

Xiao-Shan Gao
Institute of Systems Science
Academia Sinica
Beijing 100080
People's Republic of China
E-mail: xgao@mmrc.iss.ac.cn

Lawrence Henschen
Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208
E-mail: henschen@ece.nwu.edu
URL: http://www.ece.nwu.edu/~henschen/

Deepak Kapur
Institute for Programming and Logics
LI 67A, 1400 Washington Avenue
State University of New York
Albany, NY 12222
E-mail: kapur@cs.albany.edu

Kenneth Kunen
Department of Mathematics
University of Wisconsin, Madison
Madison, WI 53706
E-mail: kunen@math.wisc.edu
URL: http://www.math.wisc.edu/~kunen/

Ewing Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
E-mail: lusk@mcs.anl.gov
URL: http://www.mcs.anl.gov/people/lusk/

William McCune
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
E-mail: mccune@mcs.anl.gov
URL: http://www.mcs.anl.gov/home/mccune/

J Strother Moore
Computational Logic, Inc
1717 West Sixth Street, Suite 290
Austin, TX 78703
E-mail: moore@cli.com
URL: http://www.cli.com/

Ross Overbeek
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
E-mail: overbeek@mcs.anl.gov/

Lawrence C. Paulson
Computer Laboratory, University of Cambridge
Pembroke Street
Cambridge CB2 3QG, England
E-mail: `lcp@cl.cam.ac.uk`
URL: `http://www.cl.cam.ac.uk/users/lcp/`

Hantao Zhang
The University of Iowa
Department of Computer Science
Iowa City, IA 52242
E-mail: `hzhang@cs.uiowa.edu`
URL: `http://www.cs.uiowa.edu/~hzhang/`

Jing-Zhong Zhang
Chengdu Institute of Computer Application
Academia Sinica, 610015, Chengdu,
People's Republic of China
E-mail: `zjz@ntr.cdb.ac.cn/`

Index