A Rewrite Rule Based Approach for Synthesizing Abstract Data Types

Deepak Kapur
Computer Science Branch
General Electric R & D Center, KWC264,
Schenectady, NY 12345, U.S.A.

Mandayam Srivas[1]
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794, U.S.A.

Abstract

An approach for synthesizing data type implementations based on the theory of term rewriting systems is presented. A specification is assumed to be given as a system of equations; an implementation is derived from the specification as another system of equations. The proof based approach used for the synthesis consists of reversing the process of proving theorems (i.e. searching for appropriate theorems rather than proving the given ones). New tools and concepts to embody this reverse process are developed. In particular, the concept of expansion, which is a reverse of rewriting (or reduction), is defined and analyzed. The proposed system consists of a collection of inference rules - instantiation, simplification, expansion and hypothesis tesing, and two strategies for searching for theorems depending upon whether the theorem being looked for is in the equational theory or in the inductive theory of the specification.

1. Introduction

In this paper we develop a formal system for automatically synthesizing implementations of abstract data types from their algebraic specifications. In our approach, the *implemented* data type (i.e., the data type which is being synthesized) and the *representing* data types (i.e., the data types used to represent the implemented type) are specified as algebraic axioms. In addition, a mapping, called the *abstraction function*, that relates the values of representing data types to the values of the implemented data type is also specified. The output of the synthesis procedure consists of implementations for the operations of the implemented data type in terms of the operations of the representing data types. Thus, the operations of the representing types are used as primitive functions in the implementation being synthesized. This approach to synthesis can be applied hierarchically to as many levels of abstraction as necessary until we obtain an implementation in terms of the operations of data types, such as arrays, that are directly supported by a programming language system.

Our approach is based on the theory of term rewriting systems developed recently in the context of reasoning and proving theorem automatically about algebraic structures and data types from their specifications [KnB70], [HuH80], [Mus80b], [Hsi82]. Systems built using rewrite rule based approach for abstract data types such as AFFIRM [Mus80a], OBJ [GoT79],

[1]Research supported in part by the National Science Foundation under grants DCR-8401624 and DCR-8319966

and other systems for manipulating general term rewriting systems such as REVE [Les83] and FORMEL [HuH80], have provided encouraging signs for using the approach for theorem proving applications. In our work on generating implementations from specifications, we use many of the same tools and concepts used in theorem proving. Since we view theory-based synthesis as reversing the process of proving theorems (i.e., searching for appropriate theorems rather than proving known ones), we develop new tools and concepts to embody this reverse process. Our system consists of a set of *inference rules*, and strategies for using the inference rules to synthesize implementations. Our experience in working out many examples by hand have suggested that program transformation, optimization and synthesis based on rewrite-rule-based theorem proving is highly promising.

A major advantage of using term rewriting theory is that we can address all issues related to data type synthesis within a uniform framework. We are able to provide formal justification of the soundness of the rules of inference of our system since the consistency of data type specifications can be characterized using term rewriting concepts. We characterize the conditions under which a strategy would successfully synthesize implementations. We present two strategies - one that synthesizes implementations in the *equational theory*, and another that operates in the *inductive theory* of the specification. Both the strategies require that the data type specifications be organized as *canonical* term rewriting systems. This requirement in turn means that there exists a well-founded ordering on terms (as in [Der82]) that can be used to guarantee the *uniform termination* property of the specification term rewriting system. Although the strategies work for any such ordering it is assumed that one such ordering is available. The use of such an ordering assures the termination of the programs synthesized.

The equational strategy is completely automatic in the sense that it does not need any human intervention for it to succesfully synthesize an implementation. It is guaranteed to synthesize an implementation provided there exists an implementation (in the equational theory) the termination of which can be demonstrated using the termination ordering being employed by the strategy. The equational strategy may not terminate unless such an implementation exists. The inductive strategy is only semi-automatic since it needs prompts from the user at strategic points. Furthermore, the strategy is not complete for the inductive theory in the sense it is not guaranteed to produce an implementation even if one exists in the inductive theory.

The rest of the introduction contains an overview of related works in the area. In the next section, we illustrate our approach on a small example. Section 3 gives the reader a background in the use of algebraic techniques for specifying and implementing data types. Section 4 describes our synthesis system: Section 4.1 presents the rules of inference of our system, their justification along with their theoretical basis. A detailed discussion of the strategies for putting various rules of inferences together is given in Section 4.2. This is followed by a section that illustrates the approach in detail on a couple of examples. (For lack of space we present only simple examples in the paper. For more detailed examples see [KaS84].)

1.1. Related Work

Burstall and Darlington [BuD77] and Manna and Waldinger [MaW80] proposed a set of general purpose transformations for refining programs from their high level specifications. Feather [Fea82] has extended Burstall and Darlington's approach in his ZAP system by providing the ability for a user to specify metaprograms to direct the transformation process. Manna and Waldinger have adopted strategies based on theorem proving methods in the first order predicate calculus in their DEADLAUS [MaW80] system. Our method is related to that of Burstall and Darlington. The algebraic specification language is similar to the recursive equational language used by Burstall and Darlington for specifying the behavior of functions in their program transformation work.

Our system offers several advantages over the one proposed by Darlington (([BuD77], [Dar82]) all of which arise as result of the use of the term rewriting framework. The inference mechanisms of our system subsume all the machinery developed by Darlington and Burstall; in this sense, we not only provide a theoretical basis for their approach but also extend it. (Kott [Kot82] has also independently provided theoretical justification for their method by giving conditions under which *fold/unfold* can be applied.) Our system operates in a richer theory (the *inductive theory*) of the specification. This enables us to derive a richer class of implementations. Our approach also appears more promising in developing intelligent strategies for synthesis. This is primarily because our framework is conducive to adapting theorem proving strategies (such as the inference mechanism based on the Knuth-Bendix completion procedure [KnB70], [Mus80b], [HuH80], [Lan81]) for synthesis.

2. Illustration of the Approach

Before getting into the details of our approach, we will illustrate some of our ideas informally on an example. A function **union** is defined on the data type **multiset**, which is constructed using two constructors - a constant function **nullset** that creates an empty multiset and a binary function **insert**: **int X multiset → multiset** that inserts an element into a multiset. The function **union**, which returns the union of two multisets, has the following primitive recursive definition expressed as a set of *rewrite rules*:

(T1) $\mathbf{union(nullset, s2) \rightarrow s2}$

(T2) $\mathbf{union(insert(el, s1), s2) \rightarrow insert(el, union(s1, s2))}$

The data type **multiset** is to be implemented using another data type, **sequence**, whose values are constructed using the constant function [ ], and a binary constructor **+ : int X sequence → sequence**. The *abstraction function h* specifying how sequences represent multisets is also given as rewrite rules:

(H1) $h([\,]) \rightarrow \mathbf{nullset}$

(H2) $h(\mathbf{e + v}) \rightarrow \mathbf{insert(e,} \ h\mathbf{(v))}$

This example is interesting because (1) a **multiset** can be constructed in more than one way using its constructors, and (2) several different sequences may represent the same multiset. Using the abstraction function *h*, the implementation of **union** denoted as **UNION**, can be derived as follows. First, we introduce the rewrite rule that completely characterizes **UNION** in terms of **union** and *h*. This rewrite rule is called the *specification rule* of **UNION**. (Note that for **UNION** to be correctly implementing **union** the latter has to be a homomorphic image of the former.)

(S1) h **(UNION(v1, v2)) → union(**h **(v1),** h **(v2))**

The objective is to derive an implementation for **UNION** independent of **union** and h. This is done by deriving enough rules of the form h **(UNION(t1, t2)) →** h **(t3)** so that we can derive the implementation of **UNION** (as a total function on *sequence*) by dropping h from both sides. Rules (H1) and (H2) defining h suggest that we can instantiate the right hand side of rule (S1) - by first instantiating **v1** to be **[ ]**, and then to be **insert(e1 + v1)**. (Our strategy will discover these instantiations automatically such that they completely cover the domain of the function being implemented.)

h **(UNION([ ], v2)) → union(**h **([ ]),** h **(v2))** *Instantiate (S1)*
h **(UNION([ ], v2)) →** h **(v2)** *Simplify using (H1), (T1)*

By dropping h on both sides, we get

(I1) UNION([ ], v2) → v2

Similarly, by instantiating **v1** to be **e1 + v1** in rule (S1) as suggested by rule (H2) of h, we have

h **(UNION(e1 + v1, v2))**
  → union(h **(e1 + v1),** h **(v2))** *Instantiation of (S1)*
  → insert(e1, union(h **(v1),** h **(v2)))** *Simplify using (H2), (T2)*

Now, we want to bring symbol h on the right hand side to its outer most level. This can be done by applying some of our rewrite rules in the reverse direction (called *expansion* later in the paper, which is similar to but, more general than, Burstall and Darlington's *fold* mechanism). In order to use rule (H2) for h in the reverse direction, we must first use rule (S1) in the reverse direction. These two expansion steps will result in the following rewrite rules.

h **(UNION(e1 + v1, v2)) → insert(e1,** h **(UNION(v1, v2)))**
  → h **(e1 + UNION(v1, v2))**

Dropping h on both sides, we have

(I2) UNION(e1 + v1, v2) → e1 + UNION(v1, v2)

(I1) and (I2) constitute an implementation for **UNION** because the two rewrite rules define **UNION** as a total function on **sequence**. Note that in this example the correspondence between **union** and **UNION** is very obvious because of the close correspondence between the constructors (**[ ]**, **+**) of *sequence* and the constructors (*nullset*, *insert*) of *multiset* (except for the relation on the constructors of *multiset*). The examples in Section 6 will demonstrate that the method can synthesize more interesting implementations.

The mechanisms used above are all embodied in our system as *inference rules* that act on a set of rewrite rules to produce a new rewrite rule. The theoretical justification (given later) for each of the inference rules is that it always produces rewrite rules consistent with the original set of rewrite rules. Specifically, the h dropping mechanism can be justified by the *hypothesis checking* inference rule which uses the Knuth-Bendix completion procedure; we need to have a criterion for h dropping especially when h is a many-to-one function. According to this rule of inference, if the new rule being hypothesized does not result in any contradiction along with the existing rewrite rules, then the hypothesized rule can be added to the system.

We organize the rewrite rules in our specification into several groups based on the role they play during the synthesis of an implementation. The specification rules (such as S1) that specify the new function(s) to be implemented are grouped into the *specification set.* The rules that are used for expansion, such as (S1), (H1), and (H2), are grouped as the *expansion set.* The rules that are used for simplifying terms form the *reduction set.* The criteria that determine membership of a rewrite rule in these groups will be given later.

Although the transformation process at least for this simple example is similar to that of Burstall and Darlington [BuD77], there are several fundamental differences. We deduce the instantiations of the rules automatically from the left hand sides of the rules defining h. The inference rule in our system that does the instantiation uses the notion of *derived pairs* [GKM82]. Note that there are several possible sets of instantiations that completely span the domain. Our synthesis strategy enumerates all possible sets of instantiations systematically attempting to find implementation for each of them until it finds one. For instance, in the above example another set of instantiations possible for the arguments of **UNION** is **{(v1, [ ]), (v1, e + v2)}**. In this case, our system may derive the following pair of rewrite rules:

UNION(v1, [ ]) → UNION(v1, UNION([ ], [ ]))
UNION(v1, e + v2) → UNION(v1, INSERT(e, v2))

Our strategy would discard this implementation because the first rewrite rule cannot be ordered under any well-founded ordering.

There are four inference mechanisms being used in our approach:

(1) *Instantiation* of variables in the rewrite rules specifying the function to be implemented. (The instantiation is not arbitrary, but is directed by the definitions of other functions.)

(2) *Simplification* of a term to its irreducible form (a term is said to be irreducible if it cannot be further simplified.)

(3) *Expansion* to introduce recursion or other helping functions in the implementation.

(4) The *Knuth-Bendix completion* procedure for checking whether new hypotheses being made are indeed consistent with the existing definitions.

3. Abstract Data Types

3.1. Specification

Abstract data types are specified using the algebraic technique developed by Guttag [GuH78] and the ADJ group [GTW8.]. Our presentation of the specification is patterned after Guttag et. al. Specifically, abstract data types are defined one by one in a hierarchical way assuming other data types to be specified elsewhere. We use the initial algebra semantics [GTW8.] for our data type specifications. The data type **sequence** is specified in the figure below. (The data type **item** is assumed to be specified elsewhere.)

The operations of a data type are grouped into two classes: (i) *generators*, which generate all the values of the data type, and (ii) *defined functions*, which are defined on the values of the data type constructed by the generators. These classes are explicitly identified in the specification along with the domain and range of every operation.

The construction of values of a data type in terms of its generators is not necessarily unique. The generators - **nullset** and **insert** - of the data type **multiset** we saw in section 2 can be used in more than one way to construct the same multiset. For instance, **insert(insert(nullset,1),2)** and **insert(insert(nullset,2),1)** both construct the same multiset **{1, 2}**. This equivalence relation is characterized by the equations relating the generators. We refer to the terms like **insert(insert(nullset, 2), 1)** constructed solely using the generators as *generator terms*. Generator terms that do not contain any variables are referred to as *generator constants*.

We require that the specification of every data type involved in the synthesis be complete and consistent. By completeness ([GuH78], [Kap80].), we mean that the equations in the specification are such that every defined function is defined for every generator constant of the data type. That is, every term that applies a defined function to generator constants, such as **union(insert(nullset, 1), nullset)**, can be shown to be equivalent to a generator constant (or the distinguished element **error**) of the range type of the function. Consistency ensures that every function of the data type forms a well-defined function. One way to guarantee the completeness and consistency of a specification is to ensure that it can be organized as a term rewriting system that is *well-spanned* [Sri82] and *canonical*[2]

Data Type Sequence

Generators

[ ] : → **sequence**
+ : **item X sequence → sequence**

Defined Functions

first : **sequence → item ∪ { Error }**
rest : **sequence → sequence ∪ {Error}**
rotate : **sequence → sequence**

Axioms

(1) **first([ ]) ≡ Error**
(2) **first(e + v) ≡ e**

(3) **rest([ ]) ≡ Error**
(4) **rest(e + v) ≡ v**

(7) **rotate([ ]) ≡ [ ]**
(8) **rotate(e + [ ]) ≡ e + [ ]**
(9) **rotate(e1 + (e2 + v)) ≡ e2 + rotate(e1 + v)**

Figure 1. Algebraic Specification of Data Type Sequence

[2]A term rewriting system is canonical if every sequence of rewrites emanating form a term α terminates with the same term β; β is said to be the normal form of α.

[HuH80]. We briefy describe the two properties below.

A term rewriting system is *well-spanned* if it can be structured such that every term of the form **F**(**g**), where **F** is a defined function and **g** is a generator constant, is an instance of the left hand side of an equation in the specification. A canonical term rewriting system for a given set of equational axioms can be derived using the Knuth-Bendix completion procedure [HuH80]. It is assumed that the equational axioms specifying the data types can be oriented and made into rewrite rules using some termination ordering [Der82]. For instance, all the axioms in the specifications of **sequence** and **queue** can be oriented as rewrite rules from left to right using a *recursive path ordering* [Der82] in which the defined operation symbols are assigned more weight than the generators. In case any of the operations have the associative and/or commutative property or any other property which needs special handling, then appropriate termination ordering [DHJ83] incorporating such properties must be used. The Knuth-Bendix completion procedure or its generalization developed by Peterson and Stickel [PeS81] or Lankford and Ballantyne [LaB77] to deal with special properties such as associativity, commutativity is then applied to rules to obtain a canonical term rewriting system which gives the decision procedures for the equational theories of these data types. For instance, the specifications of **sequence** and **queue** are canonical because the Knuth-Bendix completion procedure when run on them does not generate any new rewrite rules.

The *equational theory* of a data type specified by equational axioms are all equational formulas that can be derived using the axioms of equality - reflexivity, symmetry, transitivity, substitution property and replacement. When a set of equations E can be organized into a canonical term rewriting system **R** , the equational theory of E contains formulas $\alpha = \beta$ such that α and β have the same normal form. The *inductive theory*, which contains the equational theory, is the set of all equational formulas that can be derived using the rules of equality and the following principle of induction. In the following, a generator-constant substitution is a substitution in which variables are substituted by generator constants.

$$\frac{\forall \textit{ generator-constant substitution } \sigma,\ \sigma(a) = \sigma(b) \in \textit{equational theory}}{a = b \in \textit{inductive theory}}$$

For example, consider the function **f** defined on natural numbers (with generators 0 and **S**) by the rewrite rules: $\mathbf{f}(\mathbf{0}) \rightarrow 0$ and $\mathbf{f}(\mathbf{S}(\mathbf{x})) \rightarrow \mathbf{f}(\mathbf{x})$. The equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is not in the equational theory of natural numbers with **f** but is in the inductive theory. For the data types **sequence** and **queue** for example, the equational formula expressing the associativity of append is in the inductive theory. It should be noted that this inference rule is not effective. Generally weaker forms of this inference rule are used which are practically powerful enough.

The equations in the inductive theory are not theorems in the logical sense because they do not hold good in all the models of the specification. They do hold good in the initial algebra model of the specification. They are useful for our purpose because we use the initial algebra semantics for data types. The inductive theory is the basis for our method since the programs synthesized by our inference rules lie in the inductive theory of the specification.

Henceforth, we will assume that the specifications of all data types being used in the synthesis procedure are well-spanned and each specification has a canonical term rewriting system associated with it (modulo an equivalence relation specified by special properties such as associativity, commutativity, etc.).

3.2. Specifying the Desired Implementation

Implementing a data type consists of choosing a representation for the data type, and implementing every operation of the data type in terms of the operations of the representation type. The implementation for the operations can, in general, be expressed in an arbitrary language. In the present work we express the implementation in a language identical to the one used to express the specification. That is, an implementation for an operation is expressed as a set of well-formed rewrite rules that defines the operation as a function on the chosen representation type.

In order to synthesize interesting implementations for a data type **D**, we require the user to furnish information about how the values of the representing type(s) **R** are used to represent the values of **D**. This information is specified by the user as an abstraction function *h* from **R** to **D** again as a set of rewrite rules. In general, not all the values of **R** may be used to represent the values of **D**; the subset of values of **R** that are used is specified by an *invariant predicate*. In this paper we will assume that the set of values of **R** used for representing **D** is identical to the domain of the abstraction function *h*. Some of the issues concerning data type synthesis in the presence of nontrivial invariants are discussed in [Sri82].

The abstraction mapping can be complex and may use additional auxiliary functions on **D** and/or **R** which are specified using the operations of **D** and **R** again as rewrite rules such that they are completely defined. Further, we will assume that these rewrite rules for *h* and auxiliary functions also satisfy the completeness and consistency conditions stated earlier. Our experience suggests that the more complex the abstraction funcion is, the more difficult it is to generate implementations for the operations of **D**. The rewrite rules (H1) through (H2) of the example in section 2 specify the abstraction function for an implementation of **multiset**. Specified below is another abstraction function for representing queues in terms of sequences. The empty queue is represented by the empty sequence. A nonempty queue is represented by a sequence whose elements are identical to the ones in the queue, but are arranged in the reverse order. The motivation for such a scheme is that the reading and deletion of elements from a queue can be performed efficiently. The specification of *h* uses an auxiliary function **add_at_head** on queues. This function adds an element at the front end of a queue.

(H1) h**([ ]) → nullq**
(H2) h**(e + v) → add_at_head(**h**(e),** h**(v))**

(H3) **add_at_head(e, nullq) → enqueue(e, nullq)**
(H4) **add_at_head(e1, enqueue(e2, q)) → enqueue(e2, add_at_head(e1, q))**

An implementation **F** for an operation **f** of **D** is then completely characterized by the following homomorphism property:

(*) h**(F(x1, ..., xn)) → f(**h**(x1), ...,** h**(xn))**

The mapping h is assumed to behave like an identity function on values of data types other than **R** . This is natural because we generate implementations for data types one at a time and in a hierarchical way. For each **f** , the above rewrite rule that specifies the implementing function **F** is said to be the *specification rule* of **F**.

4. Proof Based Approach for Synthesis

Our approach to synthesis is proof based because (1) we employ concepts used in theorem proving based on term rewriting systems, and (2) an implementation is derived as a theorem of the specification. The goal of our synthesis task, however, is fundamentally different from that of theorem proving. In theorem proving the goal is to establish that a given property is a theorem of a set of axioms (specification). In synthesis we have to search for a rewrite rule of an appropriate form that is known to be a theorem of the specification. The rewrite rules we are looking for are to constitute an implementation. Note that our characterization of the synthesis task is different from that of Waldinger [MaW80]. In [MaW80] a program is derived as a proof of a theorem which is an input/output specification of the program. Our approach is better suited for taking advantage of theorem proving ideas based on term rewriting systems.

In our approach, synthesis is performed with help of a system of *inference rules.* Every inference rule acts on a set of rewrite rules **R** and produces a new rewrite rule that is guaranteed to be consistent (i.e., a theorem in the inductive theory) with **R** . The set **R** initially consists of

(1) the specification of all data types,

(2) the specification of the abstraction function h , and

(3) the specification rule for the implementing functions.

Note that (1) and (2) form a well-formed system of rewrite rules. When (3) is added the set **R** remains canonical but is no longer well-spanned. This is because the functions (F) implementing the operations (**f**) are not yet defined on all values of the representation type. The synthesis process consists of repeatedly invoking appropriate inference rules on **R** so as to make it well-spanned. An implementation for an operation **f** of a data type is synthesized by deriving a well-spanned set of rewrite rules that implements **f** as a function on the representation type. The implementation is guaranteed to be correct since every rewrite rule, being derived by one or more application of an inference rule, is consistent with the specification rule of **f** . We first present the inference rules of the system, and then describe the strategies for invoking the inference rules.

4.1. Inference Rules

In the following we state the inference rules of our system. Every inference rule has the general form $\dfrac{c_1,c_2,\ldots,c_n}{\alpha \rightarrow \beta}$, where c_1, ..., c_n are a set of conditions, and $\alpha \rightarrow \beta$ is a new rewrite rule. The inference rule is to be read as "if the conditions c_1, ..., c_n hold good for a rewriting system S, then the rewrite rule $\alpha \rightarrow \beta$ may be added to S." The soundness of the inference rules is guaranteed by ensuring that the new rule $\alpha \rightarrow \beta$ is in the inductive theory of data types and the functions under consideration. In every inference rule it is also assumed that the new rewrite rule $\alpha \rightarrow \beta$ is added to S only if it preserves the uniform termination property of S. This can be ensured by checking that under the termination ordering $>$ being used $\alpha > \beta$. (In all our examples given later we use the recursive path ordering defined by Dershowitz [Der82].)

Instantiation

$$\frac{\alpha_1 \rightarrow \beta_1,\ \alpha_2 \rightarrow \beta_2 \in S, \quad <\gamma,\ \delta> \textit{ is a derived pair of the first rewrite rule on the second}}{\gamma \rightarrow \delta}$$

A *derived pair* [GKM82] $<\gamma, \delta>$ of $\alpha_1 \rightarrow \beta_1$ on $\alpha_2 \rightarrow \beta_2$ obtained by *superposing* [KnB70] β_1 on α_2 is defined as follows: consider a nonvariable subterm t of β_1 which unifies with α_2; let σ be the most general unifier of t and α_2. Then $\gamma = \sigma(\alpha_1)$, and $\delta = \sigma(\beta_{11})$, where β_{11} is obtained by replacing t in β_1 by β_2. For instance, in the case of the **union** example of section 2, the derived pair of rewrite rule (S1) on rewrite rule (H1) produces the rewrite rule **h(UNION([ ], v2)) $\rightarrow$ union(h([ ]), h (v2))**.

Derived pairs are a generalization of reduction applied on rules except that derived pairs are constructed using unification instead of matching. Derived pair generation is similar to narrowing [LaB79]. Clearly, the new rewrite rule derived by this inference rule is in the equational theory of S; An advantage of using the derived pair mechanism rather than arbitrary instantiation (like done in [BuD77] [Fea82]) to instantiate rewrite rules is that it is possible to generate a well-spanned set of instantiations automatically. This is done (as will be explained more clearly in section 4.2) by computing derived pairs between a specification rule and every other possible rewrite rule. The fact that each of the functions in the specification are completely specified ensures that the function being implemented will also be defined completely when all the derived pairs are computed.

Simplification

Let $\beta \rightarrow^* \gamma$ stand for β simplifies to γ (i.e., γ is the normal form of β) using rewrite rules in S.

$$\frac{\alpha \rightarrow \beta \in S,\ \beta \rightarrow^* \gamma}{\alpha \rightarrow \gamma}$$

The justification of this rule of inference is obvious from the definition of reduction; the new rule in this case is also in the equational theory and preserves the termination ordering. The new rewrite rule obtained using this inference rule is put in to the same set of rewrite rules from where $\alpha \rightarrow \beta$ comes.

Expansion

We say γ *expands to* δ *in* S (written $\gamma \Leftarrow \delta$) using a rewrite rule $\alpha \rightarrow \beta \in S$ if the following conditions hold:

(1) There exists a subterm t of γ such that t is unifiable[3] with β.

[3]If any of the functions satisfy special properties such as associativity and commutativity then it is necessary to use unification algorithms under equational theories [Sti81].

(2) $\delta = \sigma(\gamma_1)$, where γ_1 is obtained by replacing t in γ by α.

The rule is

$$\frac{\alpha \rightarrow \beta \in S,\ \beta \Leftarrow \delta \textit{ with } \sigma \textit{ being the unifier used for expansion}}{\sigma(\alpha) \rightarrow \delta}$$

Expanding a term γ using a rewrite rule $\alpha \rightarrow \beta$ is roughly equivalent to reducing γ using the rule $\beta \rightarrow \alpha$. The difference lies in the fact that expansion uses unification (γ with β) whereas reduction uses matching (γ with β). Note that whenever $\beta \Leftarrow \delta$, δ necessarily reduces to $\sigma(\beta)$ for some substitution σ. The new rewrite rule obtained from the expansion rule of inference is also in the equational theory as the following diagram illustrates.

$$\begin{tikzcd}
 & \sigma(\beta) & \\
\delta \arrow[ur] & = & \sigma(\alpha) \arrow[ul]
\end{tikzcd}$$

The difference between expansion and folding [BuD77] is that the former uses unification while the latter uses matching. To see the advantage of expansion over folding it would be instructive to consider the purpose an expansion/folding inference is serving in the synthesis process: To obtain an arbitrary term δ that is reducible to a given term β using the rewrite rules in the S. Folding (used repeatedly) is adequate for the purpose only if every rewrite rule $lhs \rightarrow rhs$ in S is *variable preserving* (i.e., is such that every variable in lhs also appears in rhs). However, if S has non-variable-preserving rewrite rules then folding alone is not sufficient, and we need expansion as illustrated by the following example. Let us suppose we wish to obtain from **cons(x, nil)** the term **Reml(Insertall(cons(x, nil)))** using the following set of rewrite rules. The following sequence of expansion steps achieves the desired result while no sequence folds does. Specifically, the last step in the sequence (in which rewrite rule (4) is used) cannot be performed if folding were being used.

(1) Reml(cons(x, nil)) → nil
(2) Reml(cons(x, cons(y, L))) → cons(x, Reml(cons(y, L)))
(3) Insertall(nil) → nil
(4) Insertall(cons(x, L)) → cons(x, cons(1, Insertall(L)))

$$\begin{aligned}\textbf{cons(x, nil)} &\overset{*}{\Leftarrow} \textbf{Reml(cons(x, cons(x*, Insertall(nil))))}\\ &\Leftarrow \textbf{Reml(Insertall(cons(x, nil))}\end{aligned}$$

Note that every expansion step can in general be replaced by an arbitrary substitution for the variables followed by a folding. Mixing substitutions and folding, however, complicates the strategy for invocation of the inference rules since there are potenially infinite substitutions possible. The use of unification in expansion determines the productive substitutions automatically.

Further, while expanding a term β (with the hope of determining a term δ that is reducible to β) it is necessary to consider for unification only those variables that are newly introduced during expansion, but not the ones in β. New variables are introduced whenever a term is expanded using a rewrite rule that is not variable-preserving. We refer to such variables as *free* variables. For instance, the asterisked variables in the expansion sequence shown above are free variables. It can be shown that every term δ reducible to β is an instance of some term δ^* (for some substitution of the free variables in δ^*) that is obtained after performing a finite number of expansions on β. Intutively, the free variables are placeholders for an abitrary term. In our synthesis strategy the binding of the free variables is delayed until a decision is either automatically made by the unification performed during an expansion step (as in the above example), or expanding the term any further makes it bigger than a term even if the free variable is replaced by a least term in the ordering on the terms being used.

Hypothesis Testing

$$\frac{\{\alpha \rightarrow \beta\} \cup S \ \textit{is-KB-completable}}{\alpha \rightarrow \beta}$$

where is-KB-completable is a predicate that acts on a rewriting system **S**.

The above predicate, which is a partial function, characterizes the outcome of running the Indictive Knuth-Bendix completion proecedure ([HuH80]) on **S** which is a semidecision procedure for checking the *confluence* (i.e., consistency [KaM82]) of **S**. The predicate *is-KB-completable* returns true if the inductive Knuth-Bendix completion procedure terminates successfully; otherwise, it is undefined if the completion procedure does not terminate; in the other two cases, the preducate is false. This rule is powerful as it provides a way to check whether a hypothesis made based on derivations, or by generalizing a definition of a function on a class of examples (this technique is further discussed in the next section, and illustrated by the examples in section 5.) is indeed consistent with the rest of the specification. The new rewrite rule derived using this inference rule is in the inductive theory of the data types being considered.

The above inference rule is more powerful and provides a more effective way of introducing new definitions into the system than the *redefinition* mechanism [BuD77] of Burstall and Darlington. In the redifinition mechanism, to hypothesize a new definition for a function one adds it to the system, and one tries to generate the rewrite rules constituting the existing definition for the function using fold/unfold transformations. This, however, is only a sufficient condition for the new definition to be consistent. For instance, one might have to introduce definitions besides the one being hypothesized in order to obtain the original definition. The inductive KB-completion procedure does this automatically in a significant number of cases. Also, inductive KB-completion procedure uses only reductions (not expansions), and hence is more effective.

The h-dropping Rule

This inference rule can be used to obtain a new rewrite rule $\alpha \rightarrow \beta$ from a rewrite rule of the form $h(\alpha) \rightarrow h(\beta)$. It is not always sound to drop h, since h may be many-to-one and dropping it may cause inconsistencies. It would be sound to drop the symbol provided $\alpha \rightarrow \beta$ does not introduce any inconsistency, i.e., does not introduce any relationships among values that are distinct in the system. This condition is usually satisfied when α involves a function which is unimplemented in the system.

$$\frac{h(\alpha) \rightarrow h(\beta) \in S \ ,\ \alpha \rightarrow \beta \cup S \ \textit{is-KB-completable}}{\alpha \rightarrow \beta}$$

The new rule derived this way is also in the inductive theory of data types being considered.

4.2. Strategies for Synthesis

A common mode in which the inference rules of our system is used to synthesize an implementation is to set up a *goal* that specifies the approximate form a program to be synthesized is supposed to take, and then make a judicious selection of the inference rules that achieves the goal. A *synthesis strategy* is a procedure which determines a sequence of invocations of the inference rules that will generate a new rewrite rule satisfying the desired goal. For instance, the goal in synthesizing an implementation for an operation **f** of a data type is to derive a well-spanned set of rewrite rules of the form $\mathbf{F}(\mathbf{g}_i)\rightarrow\mathbf{t}_i$, where **g** is a generator term and **t** is an arbitrary term that does not involve any operations of the type being implemented. Each of these rewrite rules is derived by deriving theorems of the form $h(\mathbf{F}(\mathbf{g}_i))\rightarrow h(\mathbf{t}_i)$, and then dropping the symbol h on either side using the h-dropping inference rule.

In the following we present two general strategies for using the inference rules of our system to derive theorems of the form $h(\mathbf{F}(\mathbf{g}_i))\rightarrow h(\mathbf{t}_i)$. The two strategies differ in the theory to which the new rewrite rules being derived belong. The first one derives rewrite rules in the equational theory, while the second can also derive rewrite rules in the inductive theory.

Different rewrite rules in the specification play different roles during the synthesis process. Based on their role we have categorized the rewrite rules into the following groups. This categorization facilitates our synthesis strategies greatly.

(1) Rules specifying the functions to be synthesized form the *specification* set.

(2) Rules used for expanding terms form the *expansion*-set.

(3) Rules that are used only for simplification and/or computing derived pairs form the *reduction*-set.

In the case of data type synthesis, the specification set initially consists of the rewrite rules expressing the homomorphism property between the abstract operations and their implementing function. For example, in the informal derivation shown in section 2, the specification set initially consists of only the rewrite rule (S1). The rewrite rules that go into the expansion set will, in general, depend on the desired form of the new rewrite rule to be derived. In addition to the specification rule and the rules specifying the abstraction function and its auxiliary functions, the expansion set includes the rules determined as follows. Suppose F is the set of function symbols that are permitted to appear on the right hand side of the rewrite rule to be derived. (This information has to be furnished by the user, in general.) Let us suppose that we have a *uses* relationship defined on the function symbols that holds if the definition of a function uses another function. Let F* be the reflexive, transitive closure of *uses* applied to F. The expansion set will include the rewrite rules that define the functions in F*. The reduction set will normally include the entire system.

4.2.1. Equational Strategy

The equational strategy is based on the property that a rewrite rule $lhs \rightarrow rhs$ in an implementation is in the equational theory of **R** if lhs and rhs have the same normal form in **R** . Thus, if *lhs* of the desired rewrite rule can be determined somehow, then *rhs* has to be a term that has the same normal form as *lhs* . The *lhs* 's of the rewrite rules are fixed based on the requirement that they have to be of the form $h(\mathbf{F}(\mathbf{g}_i))$ such that $\{\mathbf{g}_i\}$ forms a well-spanned set of generator terms.

This strategy synthesizes an implementation by repeatedly performing the following steps in sequence: The *Instantiation Step*, the *Simplification Step*, and the *Expansion Step*. Every iteration of the loop generates at most one rewrite rule $lhs_i \rightarrow rhs_i$ which is inserted into the set I (initially empty). The loop is terminated when a well-spanned set of rewrite rules are collected in I. The instantiation step consists of setting up lhs_i. The instantiation step is perfomed in such a way that every possible well-spanned set of lhs_i is generated after a finite number of iterations. The simplification step consists of simplifying the right hand side of the rewrite rule obtained in the first step to its normal form. The expansion step consists of repeatedly expanding the right hand side of the rewrite rule obtained in the simplification step. The rewrite rule returned by the expansion step is inserted into I. The expansion step, which is guaranteed to terminate (see below), may not yield an appropriate rhs_i for the lhs_i set up in the instantiation step. In such a case nothing is inserted into I during that iteration.

The instantiation step essentially consists of invoking the *Instantiation* -inference rule between a rewrite rule in the specification set and a rewrite rule outside the specification set. Although the exact rewrite rules which participate in the instantiation step are left unspecified, we assume that all the rewrite rules in the specifictaion set are treated *fairly*. This essentially means no rewrite rule in the specification set is ignored infinitely often. In other words the instantiation step has to ensure that the *Instantiation* -inference rule is invoked on every rewrite rule in the specification set and every other possible rewrite rules in the system after a finite number of iterations. This ensures every possible well-spanned set is generated after finite number of execution of the instaniation step. To see this is why, note that the specification set initially consists of the rewrite rule $h(F(x)) \rightarrow f(h(x))$. Computing derived pairs between this rewrite rule and every other possible rule generates the first well-spanned set. Each of the resulting rewrite rule (after simplification) is inserted into the specification set. Computation of derived pairs using each of these rules will generate the next well-spanned set, and so on.

```
repeat
(1) Apply Instantiation-inference between a rewrite rule in the specification set, and any
    other rewrite rule in the program.
(2) Repeatedly apply Simplification-inference to the new rewrite rule generated in step 1
    until it is no longer applicable. Let the resulting rewrite rule be $\alpha \rightarrow \beta$. Add the
    simplifed rewrite rule to the specification set.
(3) Carry out the Expansion step (described below) on $\alpha \rightarrow \beta$ to obtain the rewrite rule
    $\alpha \rightarrow \gamma$.
(4) Replace any free variable in $\gamma$ by an appropriate least element, and insert into the
    output set $I$.
until a well-spanned set of rewrite rules is obtained in $I$
```

The Expansion Step

The expansion step takes a rewrite rule $\alpha \rightarrow \beta$, and produces another rewrite rule $\alpha \rightarrow \gamma$ using the *Expansion*-inference rule repeatedly. The rewrite rules used for the expansion of β are picked from the expansion set. At each step there can be several different rewrite rules may be used for expanding. We assume that the algorithm uses a "dove tailing" technique in which all possible expansions are considered one step at a time. Also while checking if $\alpha<\beta$ below we assume that a free variable in β are treated as a least element in the ordering. It is important to note that the expansion step is guaranteed to terminate assuming there is a termination ordering on the terms. This is because the right hand side of a rewrite rule is expanded only as long as it is less than its left hand side.

```
while $\beta < \alpha$ and $\beta$ is not of the desired form do
    (1) Apply Expansion-inference rule between $\alpha \rightarrow \beta$ and a rule from the
    expansion set to obtain a new rewrite rule $\alpha \rightarrow \gamma$.

    (2) Replace $\beta$ by $\gamma$.

endwhile
```

The equational strategy will find an implementation as long there is an implementation $\{lhs_i \rightarrow rhs_i\}$ in the equational theory such that $lhs_i > rhs_i$. This is because the instantiation step is guranteed to generate the desired lhs_i. For such an lhs_i repeated expansion is guaranteed to find an rhs_i as long there is one that is less than (in the ordering $>$) lhs_i. If there is no such implementation then the strategy may not terminate. In such a case the user would have to interrupt the strategy himself.

4.2.2. Strategies for the Inductive Theory

When the equational strategy is unsuccessful, we end up in a partial implementation that defines the function being implemented on a subset of the domain values. Even when successful one might wish to derive a better implementation that is not in the equational theory. Under such circumstances we switch over to an inductive strategy. The KB-Completion inference rule is the one that gives our system the ability to derive rewrite rules in the inductive theory. Inference by KB-completion, unlike the rest of the inferences, does not derive a new rewrite rule by directly modifying an existing rewrite rule in the program. The KB-completion rule only gives a condition under which a candidate rewrite rule hypothesized to be consistent with the program can be added to the program. Thus, developing inductive strategies involves finding ways to systematically hypothesize candidate rewrite rules. The h-dropping inference rule provides one way of generating a candidate rewrite rule: $\alpha \rightarrow \beta$ is obtained from an existing rule of the form $h(\alpha) \rightarrow h(\beta)$.

Another inductive strategy that is more generally applicable uses the technique of *generalization*. This strategy is related to the technique of synthesis by example [Sum75]. This strategy is based on the following fact: If $\alpha \rightarrow \beta$ is a rewrite rule in the inductive theory then every ground instance of it is in the equational theory. We pick an instance of one of the rewrite rules that belongs to the partial implementation derived by the equational strategy. The rewrite rule so chosen is then "massaged" by applying a few expansion steps until the rewrite rule obtained appears to be an instance of the desired rewrite rule. The "massaged" rewrite rule is then ***generalized*** by replacing selected constants on either side of the rewrite rule by appropriate variables. The generalized rule is used as the candidate rule. Steps (3) and (4) need some human intervention in this strategy. In step (3) the user has to check if the rewrite rule derived has the suitable form; if not, the expansion step has to be continued further. In step (4) the user has to help the synthesis process in deciding which terms to generalize.

(1) Pick a rewrite rule $\alpha \rightarrow \beta$ (from a partial implementation derived by other means) that defines the function being synthesized on a subset of the domain values.

(2) Simplify β to its irreducible form (say γ).

(3) Apply the expansion step (described above) starting with the rewrite rule $\alpha \rightarrow \gamma$. Let the outcome of this step be $\alpha \rightarrow \delta$.

(4) *Generalize*: The candidate rewrite rule is $\alpha_1 \rightarrow \delta_1$ such that $\sigma(\alpha_1) = \alpha$ and $\sigma(\delta_1) = \delta$, where σ is an appropriate substitution.

5. Examples

In the following we present two examples. The first is a data type synthesis example; the second one is a short example presented mainly to illustrate the advantage of expansion over folding. To keep the presentation simple, we have not shown all the steps of the strategy, but only the interesting ones.

Example 1: Synthesis of Queue in terms of Sequence

We synthesize an implementation for **queue** (specified in section 2) using **sequence** as the representation type. The representation scheme used is the same as the one described in section 4. Here we show a complete derivation of two different implementations for the operation **enqueue** only. Derivation of the first implementation uses the equational strategy while the second employs the inductive strategy. Implementations for the remaining operations of **queue** can also be derived similarly.

For the derivation of the first implementation of **ENQUEUE**, we categorize the rewrite rules of the various specifications as follows. Note that the goal here is to derive rewrite rules of the form h (**ENQUEUE(g)**) $\rightarrow h$ (**t**), where **g** is a generator term of type **sequence**, and **t** is an abitrary term. The specification set will initially consists of the homomorphism rewrite rule specifying **ENQUEUE**. We first wish to synthesize a recursive implementation that does not use any defined function symbols besides **ENQUEUE**. Hence the expansion set will consist of only the rewrite rules defining the abstraction function and the functions it is dependent on. The reduction set will include all the rewrite rules in the specification.

Derivation of a Recursive Implementation

Expansion Set

(H1) h**([ ]) → nullq**
(H2) h**(e + v) → add_at_head(**h**(e),** h**(v))**

(H3) add_at_head(e, nullq) → enqueue(e, nullq)
(H4) add_at_head(e1, enqueue(e2, q)) → enqueue(e2, add_at_head(e1, q))

(S1) h**(ENQUEUE(e, v)) → enqueue(**h**(e),** h**(v))**

Specification Set

(S1) h**(ENQUEUE(e, v)) → enqueue(**h**(e),** h**(v))**

Reduction Set = Expansion Set ∪ Specification Set

h**(ENQUEUE(e, [ ]))**	
→ **enqueue(**h**(e),** h**([ ]))**	*Derived pair of (S1) on (H1)*
→ **enqueue(**h**(e), nullq)**	*Simplify using (H1)*
→ **add_at_head(**h**(e), nullq)**	*Expansion using (H3)*
→ **add_at_head(**h**(e),** h**([ ]))**	*Expansion using (H1)*
→ h**(e + [ ])**	*Expansion using (H2)*

(I1) ENQUEUE(e, [ ]) → e + [ ] *h -dropping*

h**(ENQUEUE(e, e1 + v1))**	
→ **enqueue(**h**(e),** h**(e1 + v1))**	*Derived pair of (S1) on (H2)*
→ **enqueue(**h**(e), add_at_head(**h**(e1),** h**(v1)))**	*Simplify using (H2)*
→ **add_at_head(**h**(e1), enqueue(**h**(e),** h**(v1)))**	*Expansion using (H4)*
→ **add_at_head(**h**(e1),** h**(ENQUEUE(e, v1)))**	*Expansion using (S1)*
→ h**(e1 + ENQUEUE(e, v1))**	*Expansion using (H2)*

(I2) ENQUEUE(e, e1 + v1) → e1 + ENQUEUE(e, v1) *h -dropping*

Rewrite rules (I1) and (I2) form a well-formed implementation for **ENQUEUE**.

Derivation of a Nonrecursive Implementation

The second implementation of **ENQUEUE** is intended to be a nonrecursive implementation in terms of only the operations of **sequence**. Hence, the expansion set in this case will only include the rewrite rules in the specification of **sequence**. (Below, we only show the part of the expansion set that is used in the derivation.) The specification set consists of the rewrite rules (I1) and (I2), above, snce the nonrecursive implementation is derived from the recursive implementation. The reduction set consists of the rewrite rules in the specification of **sequence**. We employ the inductive strategy by picking the rewrite rule (I1) from the specifictaion set.

Specification Set

(I1) **ENQUEUE(e, [ ]) → e + [ ]**
(I2) **ENQUEUE(e, el + vl) → el + ENQUEUE(e, vl)**

Expansion Set

(7) **rotate([ ]) ≡ [ ]**
(8) **rotate(e + [ ]) ≡ e + [ ]**
(9) **rotate(el + (e2 + v)) ≡ e2 + rotate(el + v)**

Reduction Set ≡ Expansion Set ∪ Specification Set

(I1) **ENQUEUE(e, [ ]) → e + [ ]** *Pick (I1)*
ENQUEUE(e, [ ]) → rotate(e + [ ]) *Expansion using (8)*

(G) **ENQUEUE(e, s) → rotate(e + s)** *Generalize [ ] to get a candidate rewrite rule*

(NR) **ENQUEUE(e, s) → rotate(e + s)** *Hypothesis-Testing using (G)*
(NR) is the desired nonrecursive implementation.

Example 2: Non-recursive Implementation

The goal of the synthesis in this example is to convert a recursive implementation of a function **Insert–rest** into a non-recursive one. **Reml** and **Insertall** are two functions (for which efficient implementations are assumed to exist) on **List** with generators **nil** and **cons**. **Reml** removes the last element of a list, and **Insertall** inserts the atom 1 before every element in a list. Specifications for **Insertall** (rules 4-5) and **Reml** (rules 1-3) are given below. We employ the inductive strategy to derive the new implementation.

(1) **Reml(nil) → Error**
(2) **Reml(cons(x, nil)) → nil**
(3) **Reml(cons(x, cons(y, L))) → cons(x, Reml(cons(y, L)))**

(4) **Insertall(nil) → nil**
(5) **Insertall(cons(x, L)) → cons(x, cons(1, Insertall(L)))**

Consider the function **Insert_rest** that inserts 1 before every element of a list except the first one. A recursive implementation for **Insert_rest** is:

(6) **Insert_rest(nil) → Error**
(7) **Insert_rest(cons(x, nil)) → cons(x, nil)**
(8) **Insert_rest(cons(x, cons(y, L))) → cons(x, cons(1, Insert_rest(cons(y, L))))**

We want to transform the above implementation of **Insert_rest** into one that uses **Reml** and **Insertall**. The method used is to "guess" an implementation for **Insert_rest** by trying out **Insert_rest** on a few concrete list objects. The concrete objects are determined by computing derived pairs. The guess is generated by generalizing, i.e., replacing subexpressions by variables, in the rewrite rules derived with concrete instances. The **Hypothesis–Testing** inference rule is then used to confirm that our guess is a correct implementation. In this example the expansion set consists of rewrite rules (1) through (5) (which define the functions **Reml** and **Insertall**) and nothing else because the intent is to synthesize an implementation for **Insert–rest** in terms **Reml** and **Insertall**.

Insert_rest(cons(x, nil)) $\rightarrow$ **cons(x, nil)**
$\rightarrow$ **cons(x, Reml(cons(z, nil)))** *Derived Pair of (7) on (4)*
$\rightarrow$ **Reml(Insertall(cons(x, nil)))** *Expansion using (2), (4), (5)*

(9) **Insert_rest(cons(x, nil))** $\rightarrow$ **Reml(Insertall(cons(x, nil)))**

(10) **Insert_rest(L)** $\rightarrow$ **Reml(Insertall(L))** *Hypothesis-Testing on the rule obtained by replacing* **cons(x, nil)***by* **L** *in (9)*

Note that although this implementation involves pipelining of **Reml** and **Insertall**, it could be more efficient than the recursive implementation since **Insertall** and **Reml** are assumed to be primitive operations.

6. References

[BuD77] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", *Journal of the Association for Computing Machinery,* **24**, 1 (January 1977), 44-67.

[Dar82] J. Darlington, "Program Transformation", in *Functional Programming and its Applications, An advanced course,* J. D. al, (ed.), Cambridge University Press, 1982, 193-209.

[Der82] N. Dershowitz, "Orderings for Term Rewriting Systems", *J.TCS,* **17**, 3 (1982), 279-301.

[DHJ83] N. Dershowitz, J. Hsiang, N. Josephson and D. Plaisted, "Associative-Commutative Rewriting", in *Proc. 8th IJCAI,* Karlsruhe, Germany, 1983.

[Fea82] M. S. Feather, "A System for Assisting Program Transformation", *Transactions on Programming Languages and Systems,* **4**, 1 (January 1982), .

[GoT79] J. A. Goguen and J. Tardo, "An Introduction to OBJ-T", in *Specification of Reliable Software,* IEEE, 1979.

[GTW8.] J. A. Goguen, J. W. Thatcher and E. G. Wagner, "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in *Current Trends in Programming Methodology,* vol. IV Data Structuring, R. T. Yeh, (ed.), Prentice Hall (Automatic Computation Series), Englewood Cliffs, NJ, 1978..

[GuH78] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types", *Acta Informatica,* **10**, 1 (1978), 27-52.

[GKM82] J. V. Guttag, D. Kapur and D. R. Musser, "On Proving Uniform Termination and Restricted Termination of Rewriting Systems", in *Proc. 9th ICALP,* Aarhus, Denmark, 1982.

[Hsi82] J. Hsiang, "Topics in Automated Theorem Proving and Program Generation", UIUCDCS-R-82-1113, U. of Illinois at Urbana Champaign, Urbana Illinios, Dec. 1982.

[HuH80] G. Huet and J. M. Hullot, "Proofs by Induction in Equational Theories with Constructors", in *21st IEEE Symposium on Foundations of Computer Science*, 1980, 96-107.

[Kap80] D. K. Kapur, "Towards a Theory for Abstract Data Types,", Tech. Rep.-237, Lab. for Computer Science, MIT, Cambridge, MA 02139, May 1980.

[KaM82] D. K. Kapur and D. R. Musser, "Rewrite Rule Theory and Abstract Data Type Analysis", in *Computer Algebra, EUROSAM 1982, Lecture Notes in Computer Science 144*, Calmet, (ed.), Springer Verlag, April 1982, 77-90.

[KaS84] D. Kapur and M. K. Srivas, "A Rewrite Rule Based Approach for Synthesizing Abstract Data Types", Tech. Rep. 84/080, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, July 1984.

[KnB70] D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebras", in *Computational Algebra*, J. Leach, (ed.), Pergamon Press, 1970, 263-297.

[Kot82] L. Kott, "Unfold/Fold Program Transformations", Research Report No. 155, INRIA, Le Chesnay, France, August 1982.

[LaB77] D. S. Lankford and A. M. Ballantyne, "Decision Procedure for Simple Equational Theories with Commutative-Associative Axioms", Report ATP-39, Univ. of TExas at Austin, 1977.

[LaB79] D. S. Lankford and A. M. Ballantyne, "The Refutation Completeness of Blocked Permutative Narrowing and Resolution", in *4th Conf. on Automated Deduction*, Austin, TX, 1979.

[Lan81] D. S. Lankford, "A Simple Explanation of Inductionless Induction", MTP-14, Louisiana Tech Univ., 1981.

[Les83] P. Lescanne, "Computer Experiments with the REVE Term Rewriting System Generator", in *10th Annual Symposium on Principles of Prgoramming Languages*, Austin, Texas, January 1983.

[MaW80] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis", *ACM Trans. Prog. Lang. and Systems*, **2**, 1 (January 1980), 90-121.

[Mus80a] D. R. Musser, "Abstract Data Types in the AFFIRM System", *Trans. on Software Eng.*, **1(6)**, (Jan. 1980), , IEEE.

[Mus80b] D. R. Musser, "On Proving Inductive Properties of Abstract Data Types", in *Conference record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, January 1980, 154-162.

[PeS81] G. E. Peterson and M. E. Stickel, "Complete Sets of Reductions for Some Equational Theories", *J. ACM*, **28**, (1981), 233-264.

[Sri82] M. K. Srivas, "Automatic Synthesis of Implementations for Abstract Data Types from Algebraic Specifications", MIT/LCS/Tech. Rep.-276, Laboratory for Computer Science, MIT, June 1982.

[Sti81] M. E. Stickel, "A Unification Algorithm for Associative-Commutative Functions", *J. ACM*, **28**, (1981), 233-264.