Functional programming with Sets†

Bharat Jayaraman
David A. Plaisted

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract—Set abstraction, originally introduced in functional languages by Turner, is an appealing construct because it leads to concise definitions of many interesting operations. However, existing approaches treat sets as lists for the sake of efficiency, and thereby sacrifice a simple declarative semantics. In this paper, we present a novel language based on sets and equations, where sets are treated as sets, consistent with their semantics. The language is called SEL, for Set-Equation Language. Equations are assumed to define a confluent rewriting system when oriented left to right. Sets are defined in terms of their subsets; these rules define a nonconfluent rewriting system when oriented left to right. We show examples of programs in this language, and provide an operational semantics for such programs. Programs are executed by innermost reduction, which may be nondeterministic or deterministic. Nondeterministic reduction is used when one of the elements of a set is desired. Deterministic reduction is used to simplify a term via an equation or to obtain all the elements of a set. The correctness of the operational semantics is also established.

1. Introduction

Functional programming languages ideally have clean mathematical semantics, so that programs are easy to understand and verify. However, in practice it is sometimes found "necessary" to sacrifice clean semantics for sake of efficiency. One problem area is *set abstraction*, a construct originally introduced in the functional language KRC by David Turner [T81]. Although Turner's set expressions lead to concise and elegant formulations of a number of interesting operations, set expressions actually construct lists (not sets), and therefore cannot be formalized within a simple functional framework.

In this paper we propose a novel approach to handling sets in a functional language, without compromising their semantics. Our approach is based on ideas from associative-commutative term rewriting systems [BKN85, P86]. We present a language called SEL (for Set-Equation Language) as a vehicle for illustrating our approach. A SEL program is a collection of assertions, where each assertion is one of two kinds: an equation assertion or a subset assertion. We present a number of examples illustrating the language, and show that associative-commutative matching can aid in

† This research is supported by grants DCR-8516243 and DCR-8603609 from the National Science Foundation.

avoiding recursive definitions for many interesting operations, such as set difference, intersection, cross product, etc.

The use of recursive equations for functional programming has been argued by a number of people [T81, K82, HO82]. Our equational sublanguage bears a close relation with these various proposals; it is, in fact, an equational term rewriting system, as described, for example in [HO80, B83], except that our matching is a restricted form of associative-commutative matching.

The formal semantics of our language uses classical notions of equality and set membership. We require that equational assertions define a confluent rewriting system when oriented left to right for rewriting, but the subset assertions need not. In both cases, however, we require termination. In order to ensure that all argument sets have distinct elements, we propose innermost reduction as the basis of our operational semantics. We define the operational semantics of our language in terms of two rewriting relations, $\Rightarrow_d$ and $\Rightarrow_n$, for deterministic and nondeterministic rewriting respectively. The operational semantics of a SEL term is its $\Rightarrow_d$ *normal form.* We show the correctness of our operational semantics by proving that any computed term is a logical consequence of a suitable axiom system obtained from the program rules.

It turns out that our formulation of sets can also serve as a clean way to obtain "all solutions" to a predicate in a logic programming language such as Prolog. As with Turner's set abstraction, the all-solutions predicate is not a first class operation in a logic programming language, and suffers from not having a clean semantics; for a detailed discussion of this problem, see [N85].

The rest of this paper consists of the following sections: section 2 presents the language features of SEL and several examples to illustrate its methodology; section 3 formalizes the operational semantics by defining the $\Rightarrow_d$ and $\Rightarrow_n$ relations, and shows its correctness; section 4 mentions ways of proving the confluence and termination of SEL programs; and section 5 presents conclusions and directions for further work.

2. Language Features

2.1 Syntax

The grammar defining the syntax of SEL programs is given below. Note that the symbols $\longrightarrow$, $|$, and ϵ are meta symbols, and do not belong to SEL. The symbols **?**, **(**, **)**, **{**, **}**, **=**, $\supseteq$, $\cup$, and **,** are language-defined tokens. The symbols *opname*, *atom*, and *variable* are user-defined tokens.

program $\longrightarrow$ *rules goal*

rules $\longrightarrow \epsilon \mid$ *rule rules*

goal $\longrightarrow$ *?term*

rule $\longrightarrow$ *equation* $\mid$ *subset*

equation $\longrightarrow$ *opname*(*terms*) = *term*

subset $\longrightarrow$ *opname*(*terms*) $\supseteq$ *term*

terms $\longrightarrow$ *term* | *term* , *terms*

term $\longrightarrow$ *atom* | *variable* | { } | {*term*} | *term* $\cup$ *term* | *opname*(*terms*)

As in other equational languages [HO80], we do not make any distinction between constructors and function symbols. Thus, a *term* in SEL corresponds to an expression in functional languages. A *ground term* is a term without any variables. The operator $\cup$, which stands for set union, is an associative, commutative operator with the properties, $x \cup x = x$ and $x \cup \{\ \} = x$, where { } stands for the empty set. In the examples in this paper, it would be helpful to construe the operators $\cup$, { }, as well as cons and scons (explained below) as constructors. Thus, SEL programs would not have these operations as the outermost name on the left-side of any rule. Note that *opname* does not include these operators.

We use the operator cons for constructing binary trees. As in LISP, lists are a special form of binary trees, and we write them using the [...] notation, e.g. [1,2,3]. [ ] stands for the empty list, and is regarded as an atom. Thus the list [1,2,3] is represented as cons(1, cons(2, cons(3, [ ]))). We use the notation [h | t], similar to Prolog [CM81], to refer to a non-empty list, with head h and tail t. Thus, [h | t] $\equiv$ cons(h, t).

The operation scons(x,y)†, for set-cons, builds a set, and is merely an abbreviation for {x} $\cup$ y. We write a set using the {...} notation, e.g. {1,2,3}. Analogous to the list notation, the set {1,2,3} may be represented as scons(1, scons(2, scons(3, { }))). Other permutations, such as scons(2, scons(1, scons(3, { }))), scons(2, scons(3, scons(1, { }))), etc., represent the same set. We use the notation {h | t} to refer a non-empty set, one of whose elements is h and the remainder of the set is t. Thus, {h | t} $\equiv$ scons(h, t) $\equiv$ {h} $\cup$ t.

2.2 Informal Semantics

Operationally, equations are treated as rewrite rules oriented left to right, and a term is rewritten, by innermost reduction, by matching it with the left-side of some rewrite rule using *one-way* matching. SEL programs with equations alone, i.e., without subset rules, are intended to define *canonical* term-rewriting systems, i.e., confluent and terminating programs. Like other equational languages [HO82], every variable on the right-side of an equation must be present on its left-side.

Operationally, subset rules are also oriented from left to right for rewriting, and a term is reduced by innermost reduction by first matching it with the left-side of the rule. The matching here is *associative-commutative* (or a-c) matching, where $\cup$ is the a-c operator. For example, when matching a term f({1,2,3}) with the left-side of a rule

f({h | t}) $\supseteq$ ...

† This operator may be contrasted with Friedman and Wise's frons [FW80], which defines a multi-set that turns into a list as elements are accessed.

all three matches for h and t must be considered, namely, $\{h \leftarrow 1, t \leftarrow \{2,3\}\}$, $\{h \leftarrow 2, t \leftarrow \{1,3\}\}$, and $\{h \leftarrow 3, t \leftarrow \{1,2\}\}$—recall that $\{1,2,3\}$ is represented as scons(1, scons(2, scons(3, { }))), or equivalently as $\{1\} \cup \{2\} \cup \{3\}$. The right-side of the rule for f is then reduced for each of these matches, and the union of the result for each match is defined as the value for $f(\{1,2,3\})$. Note that in taking this union duplicate elements would be eliminated (we mention in section 3.4.1 when this test for duplication can be avoided). As with equations, we require for subset assertions that every variable on the right-side also appear on the left-side. However, the rewrite rules derived from subset assertions need not be confluent.

Note that we use a-c matching even with the rewriting rules derived from equations. Because we assume confluence, the result of rewriting is assumed to be independent of which match is considered when using equations. Hence, only one successful match is considered, and the others are ignored. Note also that our a-c matching does not use the idempotent property of $\cup$. Hence, the matching of $\{1,2,3\}$ with $\{h \mid t\}$ cannot yield, for example, $\{h \leftarrow 1, t \leftarrow \{1,2,3\}\}$. However, $\{1\}$ can match $\{h \mid t\}$, yielding $\{h \leftarrow 1, t \leftarrow \{\ \}\}$. Finally, note that there are 2^n a-c matches of an n-element set s with a term $x \cup y$ (corresponding to the different possible subsets of s), whereas there are only n matches of s with scons(x,y), or equivalently with $\{x \mid y\}$. Both scons and $\cup$ have their uses, albeit with substantially different costs: the former is usually used for iterating over all *elements* of a set, whereas the latter is usually used for iterating over all *subsets* of a set. Although the complexity of a-c matching in general is NP-complete [BKN85], most SEL patterns in practice have very simple structure, with non-repeating variables, and hence can be matched reasonably fast.

2.3 Examples

We now present examples to show the capabilities of the foregoing constructs for functional programming.

Append:

Below is the SEL definition for the familiar LISP function append, and illustrates use of equations.

append([ ], y) = y
append([h | t], y) = [h | append(t, y)]
?append([1,2], [3,4])

The result of evaluating the goal is the list [1,2,3,4].

List-to-Set Conversion:

The operation list-to-set stands for list-to-set conversion.

list-to-set([ ]) = { }
list-to-set([h | t]) = {h | list-to-set(t)}

Note that a similar operation `set-to-list` for set-to-list conversion, defined below, is not a valid SEL program because the definition is not confluent.

```
set-to-list({ }) = [ ]
set-to-list({h | t}) = [h | set-to-list(t)]
```

One might suspect that all equational rules that use sets as arguments on their left-sides are not confluent. This is, however, not the case, as illustrated by the next example.

Or:

The following function `or` returns `true` if there is a `true` element in the set; it returns `false` if both elements of the set are `false`.

```
or({ }) = false
or({true | t}) = true
or({false | t}) = or(t)
```

Notice that the second rule above does not use the variable `t` on its right-side. Such variables may be replaced by a special "anonymous" variable (_), which can match any term, similar to Prolog [CM81]. Using this anonymous variable, the above rule becomes

```
or({true | _ }) = true
```

A more useful function `or` is one that returns `false` as long as there are no `true` elements, e.g., `or({1,2,3}) = false`. Defining such an operation requires a primitive conditional operation

```
cond(p,t,e)
```

which returns the result of reducing `t` if `p` reduces to `true`, otherwise (if `p` reduces to some term that is other than `true`) it returns the result of reducing `e`. Note that such a conditional cannot be expressed in term-rewriting systems, because they cannot in general express inequality. Henceforth, we assume that SEL has cond, which we also write using a more familiar syntax:

```
if p then t else e.
```

We can define the revised `or` function in terms of the following membership function `mem`:

```
mem(h,{h | _ }) = true.
```

Note that terms such as `mem(1,{ })` and `mem(1,{2,3,4})` are irreducible. The desired `or` function is:

```
or(s) = if mem(true, s) then true else false.
```

The primitive conditional is also useful in defining other set operations, as illustrated in the next example.

Set Difference:

```
diff({ },s) = { }
```

```
diff(s,{ }) = s
diff({h | _ }, t) ⊇ if mem(h,t) then { } else {h}
```

This example illustrates use of the subset rule. The different a-c matches of the actual parameter set with the term {h | _ } will effectively cause h to be bound to different elements of the set.

Set intersection can be defined similarly. However, set intersection can be defined without primitive conditional as illustrated below.

Set Intersection:

```
intersect({ },s) = { }
intersect(s,{ }) = { }
intersect({h | _ },{h | _ }) ⊇ {h}
?intersect({1,3,5},{3,4})
```

The result of evaluating the goal is the set {3}.

Cross Product:

```
prod({ },s) = { }
prod(s,{ }) = { }
prod({x | _ },{u | _ }) ⊇ {[x | u]}
?prod({1,2},{3,4})
```

The result of evaluating the goal is the set {[1 | 3], [1 | 4], [2 | 3], [2 | 4]}.

Relative Set Abstraction:

We can define Turner's relative set abstraction construct, $\{f(x) \mid x \in S \wedge p(x)\}$, as follows:

```
all-fp({ }) = { }
all-fp({x | _ }) ⊇ if p(x) then {f(x)} else { }
```

The function `all-fp` produces the set of all `f(x)` where `x` is drawn from some given set `S`, such that `p(x)` is true.

Join:

As can be seen from the foregoing definitions, SEL can be used to define the operators in the relational algebra. The following rule shows how a simple form of relational join can be expressed.

```
join({[x,y] | _ },{[y,z] | _ }) ⊇ {[x,z]}
```

A more general join can also be expressed similarly.

All Solutions:

The all-solutions predicate `collect` of some Prolog systems (see [N85] for details) may be expressed with unusual brevity in SEL, as follows:

```
collect(s) = {s}.
```

The result produced by `collect` is a singleton set whose element is the set of all solutions. For example, the result of the term

$$?\texttt{collect}(\texttt{prod}(\{1,2\},\{3,4\}))$$

is the set $\{\{[1 \mid 3],[1 \mid 4],[2 \mid 3],[2 \mid 4]\}\}$.

Permutations:

$$\texttt{perms}(\{\ \}) = \{[\ ]\}$$
$$\texttt{perms}(\{\texttt{x} \mid \texttt{t}\}) \supseteq \texttt{distr}(\texttt{x},\texttt{perms}(\texttt{t}))$$
$$\texttt{distr}(\texttt{x},\{\ \}) = \{\ \}$$
$$\texttt{distr}(\texttt{x},\{\texttt{y} \mid \_\ \}) \supseteq \{[\texttt{x} \mid \texttt{y}]\}$$
$$?\texttt{perms}(\{1,2,3,4\})$$

The function `distr` expects a set of lists as its second argument. Its result is a set whose elements are constructed by "consing" its first argument to each list in its second-argument set. The result of evaluating the goal is the set of permutations $\{[1,2,3,4],[1,2,4,3],\ldots,[4,3,2,1]\}$.

Four Queens Problem:

$$\texttt{queens}(\texttt{col},\texttt{safeset}) = \texttt{if eq}(\texttt{col},5)\ \texttt{then safeset}$$
$$\texttt{else placequeen}(\texttt{col},\{1,2,3,4\},\texttt{safeset})$$
$$\texttt{placequeen}(\texttt{col},\{\texttt{row} \mid \_\ \},\texttt{safeset}) \supseteq$$
$$\texttt{if safe}([\texttt{col} \mid \texttt{row}],\ \texttt{safeset})$$
$$\texttt{then queens}(\texttt{col}+1,\ \{[\texttt{col} \mid \texttt{row}] \mid \texttt{safeset}\})$$
$$\texttt{else}\ \{\ \}$$
$$\texttt{safe}([\texttt{c1} \mid \texttt{r1}],\{\ \}) = \texttt{true}$$
$$\texttt{safe}([\texttt{c1} \mid \texttt{r1}],\{[\texttt{c2} \mid \texttt{r2}] \mid \texttt{s}\}) = (\texttt{r1} \neq \texttt{r2})\ \texttt{and}\ (\texttt{abs}(\texttt{c1}-\texttt{c2}) \neq \texttt{abs}(\texttt{r1}-\texttt{r2}))$$
$$\texttt{and safe}([\texttt{c1} \mid \texttt{r1}],\texttt{s})$$
$$?\texttt{queens}(1,\{\ \})$$

The above example illustrates how a search may be specified. The algorithm places a queen on each successiye column, beginning from column 1, as long as each new queen placed is safe with respect to all queens in the preceding columns. A solution is found if a queen can be thus be placed on all columns. The second argument to `placequeen`, viz., the set $\{1,2,3,4\}$, enumerates the row positions in each column. If a particular row-column position is not safe, `placequeen` returns the empty set $\{\ \}$, thereby pruning this line of search. The function `safe` specifies the safety condition—note that SEL has the usual complement of arithmetic primitives.

The reader may have noted that several useful set operations, such as difference, intersection, membership, etc., are defined *without* recursion. This is one of the strengths of a-c rewriting. We assume that a SEL program defines a *closed world* [C78]. That is, a set is completely defined by its subsets; there are no other elements in the set than the ones specified.

3. Operational Semantics

We now formalize the operational semantics of SEL programs that we informally sketched in section 2.2. This is expressed using two rewriting relations $\Rightarrow_d$ and $\Rightarrow_n$, the subscripts d and n stand for 'deterministic' and 'nondeterministic' respectively. These two relations are defined over the set of terms, and are mutually dependent.

3.1 The $\Rightarrow_d$ and $\Rightarrow_n$ relations

Suppose $\mathcal{R}$ is a set of SEL program rules. In the following definition, program rules are enclosed within $[\![\ ]\!]$. The substitution θ binds variables to terms. The relation $x =_{ac} y$ means that x and y are equal using associative-commutative equations; for example, $a \cup (b \cup c) =_{ac} (c \cup b) \cup a$.

Below, rules 1-4 pertain to the $\Rightarrow_d$ relation and rules 5-7 pertain to the $\Rightarrow_n$ relation. The relationship between these two relations is expressed by rules 8 and 9. Note that, in the following rules, the symbol $\supset$ stands for "logically implies," and should not be confused with the SEL operation $\supseteq$.

1. $[\![s = t]\!] \in \mathcal{R} \quad \supset \quad s \Rightarrow_d t$
2. $s \Rightarrow_d t \wedge u =_{ac} s \quad \supset \quad u \Rightarrow_d t$
3. $s \Rightarrow_d t \quad \supset \quad s\theta \Rightarrow_d t\theta$
4. $s \Rightarrow_d t \quad \supset \quad f(\ldots, s, \ldots) \Rightarrow_d f(\ldots, t, \ldots)$
5. $[\![s \supseteq t]\!] \in \mathcal{R} \quad \supset \quad s \Rightarrow_n t$
6. $s \Rightarrow_n t \wedge s =_{ac} u \quad \supset \quad u \Rightarrow_n t$
7. $s \Rightarrow_n t \quad \supset \quad s\theta \Rightarrow_n t\theta$
8. $s \Rightarrow_d t \wedge u \in t \quad \supset \quad s \Rightarrow_n \{u\}$
9. $s \Rightarrow_n t \quad \supset \quad s \Rightarrow_d \cup\{w : s \Rightarrow_n w\}$

In addition, we have the following two rules for the $\cup$ operator:

10. $x \cup x \Rightarrow_d x$
11. $x \cup \{\ \} \Rightarrow_d x.$

Definition 3.1: We say x is *$\mathcal{R}$-reducible*, or simply *reducible* (if $\mathcal{R}$ is understood), if there exists a y such that $x \Rightarrow_d y$ or $x \Rightarrow_n y$; otherwise we say x is *irreducible.*

Definition 3.2: We say that

$$s \Rightarrow_d! t$$

if term t is irreducible and related to s in the transitive closure of $\Rightarrow_d$, and obtained by using *innermost reduction* and giving *priority* to the $\Rightarrow_d$ reductions involving the $\cup$ operator.

Note that the termination requirement guarantees such a t for any s. Also, prioritizing reductions involving the $\cup$ operator ensures that all argument sets will be free of duplicate elements.

Definition 3.3: If $s \Rightarrow_d! t$, then t is referred to as the $\Rightarrow_d$ *normal form* of s.

We intend that the $\Rightarrow_d$ relation is confluent, i.e., if $s \Rightarrow_d! t$, then t is intended to be unique (for a given s). Similar to $\Rightarrow_d$ *normal form*, we also have the $\Rightarrow_n$ *normal form* of a term. The $\Rightarrow_n$ relation, however, need not be confluent, hence there can be several $\Rightarrow_n$ normal forms of a term. The primitive conditional cond can now be defined as follows:

12. $p \Rightarrow_d! \texttt{true} \wedge t \Rightarrow_d t' \quad \supset \quad \texttt{cond}(p,t,e) \Rightarrow_d t'.$

13. $p \Rightarrow_d! \texttt{true} \wedge t \Rightarrow_n t' \quad \supset \quad \texttt{cond}(p,t,e) \Rightarrow_n t'.$

14. $p \Rightarrow_d! p' \wedge p' \not\equiv \texttt{true} \wedge e \Rightarrow_d e' \quad \supset \quad \texttt{cond}(p,t,e) \Rightarrow_d e'.$

15. $p \Rightarrow_d! p' \wedge p' \not\equiv \texttt{true} \wedge e \Rightarrow_n e' \quad \supset \texttt{cond}(p,t,e) \Rightarrow_n e'.$

Definition 3.4: The operational semantics of a SEL term s is its $\Rightarrow_d$ *normal form.*

Thus the value computed for a goal

? s

is its $\Rightarrow_d$ *normal form.* Sometimes one in interested in obtaining any one element of the set. This can be stated in SEL using a special syntax at the top-level, such as

any? s

which computes a t such that $\{t\}$ is a $\Rightarrow_n$ *normal form* of s.

Definition 3.5: We say that an operation $\mathtt{f}$ *distributes over nondeterminism in the i-th argument* iff there is a program rule

$$\mathtt{f}(\ldots,\mathtt{x} \cup \mathtt{y},\ldots) = \mathtt{f}(\ldots,\mathtt{x},\ldots) \cup \mathtt{f}(\ldots,\mathtt{y},\ldots)$$

where the i-th argument of $\mathtt{f}$ is the one shown above.

When an operation distributes over nondeterminism in a particular argument, it is permissible to compute with the *individual* elements of the set corresponding to this argument, rather than the *entire* set.

3.2 Correctness

The correctness of the operational semantics defined in the previous subsection is established by showing that the normal forms computed using the $\Rightarrow_d$ and $\Rightarrow_n$ are correct with respect to the classical equality and subset relations. Note that these two relations define only one step of the reduction process. Suppose $\mathcal{R}$ is a set of SEL program rules. We assume that the axiom system Γ reflects our closed world assumption about the rules of $\mathcal{R}$, as follows:

(i) $s \Rightarrow_d t$ implies $\Gamma \models s = t$, and

(ii) $s \Rightarrow_n t$ implies $\Gamma \models s \supseteq t$.

The expression $\Gamma \models s = t$ means that the universally quantified formula $s = t$ is a *logical consequence* of Γ. Similarly, $\Gamma \models s \supseteq t$ means that the universally quantified formula $s \supseteq t$ is a *logical consequence* of Γ.

Theorem 3.1: Suppose $\mathcal{R}$ is a set of SEL program rules, and the $\Rightarrow_d$ and $\Rightarrow_n$ relations are as defined in the previous subsection, and Γ is an axiom system as defined above. Then

(i) $s \Rightarrow_d! \ t$ implies $\Gamma \models s = t$, and

(ii) $s \Rightarrow_n! \ t$ implies $\Gamma \models t \supseteq s$.

The proof is straightforward, and follows from the transitivity of $=$ and $\supseteq$. The above is a *soundness* theorem as it says that every computed solution is a correct solution according to the logical semantics. *Completeness* follows from the fact that programs terminate and the $\Rightarrow_d$ *normal form* is unique.

Before we proceed with examples, the following theorems may be noted:

Theorem 3.2: $\Rightarrow_d$ is terminating if and only if $\Rightarrow_n$ is terminating.

Theorem 3.3: $\Rightarrow_d$ is terminating if there is well-founded ordering $\succ$ such that $s \Rightarrow_d t$ implies $s \succ t$.

Theorem 3.4: $s \Rightarrow_d! \ \cup \{t : s \Rightarrow_n! \ t\}$.

In section 4.2, we discuss how the well-founded ordering $\succ$ referred to in theorem 3.3 can be obtained.

3.3 Examples

We present two examples illustrating the operational semantics. We assume that the reader is familiar with computation using equations; we therefore concentrate on the novel aspects of the language, namely, sets and a-c matching.

Example 1:

Consider the cross product example from section 2.3, which we reproduce here for convenience.

$$\begin{aligned}
&\mathtt{prod}(\{\ \}, \mathtt{s}) = \{\ \} \\
&\mathtt{prod}(\mathtt{s}, \{\ \}) = \{\ \} \\
&\mathtt{prod}(\{\mathtt{x} \mid \mathtt{y}\}, \{\mathtt{u} \mid \mathtt{v}\}) \supseteq \{[\mathtt{x} \mid \mathtt{u}]\} \\
&?\mathtt{prod}(\{1,2\}, \{3,4\})
\end{aligned}$$

Suppose the top-level goal were

$$\mathtt{any?}\ \mathtt{prod}(\{1,2\}, \{3,4\})$$

where we are interested in any one member of the cross product set. The above goal might be represented as

$$\mathtt{prod}(\mathtt{scons}(1, \mathtt{scons}(2, \{\ \})), \mathtt{scons}(3, \mathtt{scons}(4, \{\ \}))).$$

There are two possible matches of $\mathtt{scons}(1, \mathtt{scons}(2, \{\ \}))$ with the pattern $\{\mathtt{x} \mid \mathtt{y}\}$, namely $\{\mathtt{x} \leftarrow 1, \mathtt{y} \leftarrow \mathtt{scons}(2, \{\ \})\}$, and $\{\mathtt{x} \leftarrow 2, \mathtt{y} \leftarrow \mathtt{scons}(1, \{\ \})\}$. Similarly there are two possible

matches of scons(3, scons(4, { })) with the pattern {u | v}. Because we specify a $\Rightarrow_n$ reduction in the top-level goal, any one match for the variables x, y, u and v is acceptable. Thus the next step in the reduction, following rules 5 and 7 of the operational semantics, might be

$\Rightarrow_n$ {[1 | 3]},

which is not further reducible. Other $\Rightarrow_n$ *normal forms* are {[1 | 4]}, {[2 | 3]}, and {[2 | 4]}. Hence, the response to the top-level goal

?prod({1,2},{3,4})

would be the entire set {[1 | 3], [1 | 4], [2 | 3], [2 | 4]}, which corresponds to the $\Rightarrow_d$ reduction of the above goal, following rule 9 of the operational semantics.

Example 2:

Consider the permutations example from section 2.3:

perms({ }) = {[ ]}
perms({x | t}) $\supseteq$ distr(x, perms(t))
distr(x, { }) = { }
distr(x, {y | _ }) $\supseteq$ {[x | y]}
?perms({1,2,3,4})

Suppose once again we are interested in any one answer, and specify this through a top-level goal

any? perms({1,2,3}).

The following is one possible sequence of $\Rightarrow_n$ reductions, where one a-c match is considered at each recursive call of perms:

perms(scons(1, scons(2, scons(3, { }))))
$\Rightarrow_n$ distr(1, perms(scons(2, scons(3, { }))))
$\Rightarrow_n$ distr(1, distr(2, perms(scons(3, { }))))
$\Rightarrow_n$ distr(1, distr(2, distr(3, perms({ }))))
$\Rightarrow_n$ distr(1, distr(2, distr(3, {[ ]})))
$\Rightarrow_n$ distr(1, distr(2, {[3]}))
$\Rightarrow_n$ distr(1, {[2,3]})
$\Rightarrow_n$ {[1,2,3]}

3.4 Discussion

3.4.1 Nondeterminism

Note that, in the second step of the above derivation, we did not compute the entire set for perms(scons(2, scons(3, { }))). The reason that it is valid to compute one element from this set

is because the `distr` definition *distributes over nondeterminism.* Although we did not explicitly include an equation stating that `distr` distributes over nondeterminism, we assume that this is the case for all definitions unless otherwise specified. There are, however, cases where we do not wish an operation to distribute over nondeterminism. The functions `perms`, `safe`, `or`, and `mem` are examples. The following function `size` which determines the number of elements in a set is another example.

$$\texttt{size}(\{\ \}) = 0$$
$$\texttt{size}(\{\texttt{h} \mid \texttt{t}\}) = 1 + \texttt{size}(\texttt{t})$$

Functions that compute some aggregate property of a set usually do not distribute over nondeterminism. Functions, such as prod, `intersect`, etc., that are defined in terms of the elements of the set, do distribute over nondeterminism.

We assume that SEL is extended with suitable syntax so that a programmer can specify which operations do not distribute over nondeterminism. This information could then be used by the SEL interpreter to decide when it is permissible to compute with the individual elements of an argument set rather than with the entire set. Because of the termination requirement, we can compute all elements of a set by depth-first search and backtracking. When an operation distributes over nondeterminism in a particular argument, it is not necessary to check for duplicate elements in the set computed for this argument. Instead, the operation can be applied to each element of the multi-set, and the resulting multi-set can be propagated upward. In many practical cases, duplications do not occur, hence the absence of a check can lead to faster execution.

3.4.2 Outermost Reduction

When computing with sets in SEL, although *innermost reduction* is a sound reduction strategy, *outermost reduction* can lead to early termination in some cases. Suppose that, in evaluating some term $\texttt{f}(\texttt{r})$, we have found $\texttt{r}_1, \texttt{r}_2, \ldots, \texttt{r}_n$ such that $\texttt{r} \Rightarrow_n! \texttt{r}_i$. Suppose further that there are more such possibilities, due to more a-c matches or more rules that have not been looked at. We can then represent the current state of the value of r by

$$\{\texttt{r}_1, \texttt{r}_2, \ldots, \texttt{r}_n\} \cup x,$$

where x is a variable. Now, if there is a rule, $\texttt{f}(\ldots) = \ldots$, that permits

$$\texttt{f}(\{\texttt{r}_1, \texttt{r}_2, \ldots, \texttt{r}_n\} \cup x)$$

to be reduced to some term in which x does not appear, it is then not necessary to continue to look for more $\Rightarrow_n$ reductions for r. This gives a way to stop a nondeterministic search, and is analogous to Prolog's "cut" [CM81], but with clean semantics. An example of this kind of early termination arises when evaluating the or function (of section 2.3), when one of the elements of its input set is `true`. A more direct example is the function ge-k, shown below, which checks if a set has at least k elements:

$$\texttt{ge-k}(\{\texttt{r1}, \texttt{r2}, \ldots, \texttt{rk}\} \cup \texttt{x}) = \texttt{true}.$$

Outermost reduction, however, is not always a sound strategy when computing with sets. For example, in the case of the function `size`, which does not distribute over nondeterminism, outermost reduction is sound only if the set corresponding to this argument has distinct elements. This requirement of distinct elements is a sufficient but not necessary condition for the soundness of outermost reduction. Note that the above equation is valid even if x had duplicates. It may be convenient to use such equations in a general context, and not just for early termination of a nondeterministic search.

4. Confluence and Termination

We make some comments about how confluence and termination can be proven for programs in SEL. The methods are similar to those for standard term rewriting systems. However, since SEL programs are not conventional term rewriting systems, some modifications are necessary. The differences may be illustrated by an example. Consider the following program:

`pairs({x | {y | z}})` $\supseteq$ `{[x | y]}`.

The query `?pairs({a,b,c,d})` reduces to

`{[a | b], [a | c], [a | d], [b | a], [b | c], [b | d], ...}`.

That is, the rule

`pairs({x | {y | z}})` $\supseteq$ `{[x | y]}`

is applied many different times in different ways in a single rewriting step—this is not possible in an ordinary term rewriting system. Another problem is the conditional operator, which cannot be expressed in an ordinary term rewriting system because it uses a kind of *negation by failure* [C78].

4.1 Confluence

There may be methods of showing confluence of SEL programs based on critical pairs, as in the Knuth-Bendix method [KB70], but we are not aware of any as yet. We propose a method based on the *semantic confluence* ideas of Plaisted [P85]. The idea is as follows: Suppose we define a sorting program in SEL or some other rewriting language, and we can show that the program is correct, i.e., that the output is always equal to a sorted form of the input. (This can often be done using program verification methods.) Suppose also that we can show by syntactic means that the output is a list of elements. Since there is only one list of elements that is equal to a sorted form of the input, the output must be unique. Therefore the program is confluent. Thus, the general idea is largely based on semantics (correctness) rather than syntax. In addition, the method is largely insensitive to the particular method of rewriting being used, and so seems applicable to SEL programs as well as to ordinary term rewriting systems.

4.2 Termination

We express SEL terms using the singleton set and binary set union operators, so that $\{a,b,c\}$ would be expressed as $\{a\} \cup (\{b\} \cup \{c\})$—note that any SEL term can be put in this form. Now, since $\cup$ is an associative-commutative operator, it is convenient to express terms in a "flattened" form; that is, $\{a\} \cup (\{b\} \cup \{c\})$ is expressed as $\cup(\{a\},\{b\},\{c\})$. In this way we introduce a union operator of a variable number of arguments. Methods of proving termination of term rewriting systems involving one or more associative-commutative (a-c) operator were given in Bachmair and Plaisted [BP85], and also in Bachmair and Dershowitz [BD86]. These methods can be adapted to SEL programs. The idea is to show that if $s \Rightarrow_d t$ then $s \succ t$ in some well founded ordering $\succ$. For this ordering we choose the *associative path ordering* of Bachmair and Plaisted [BP85]. To order two terms s and t, we flatten them to obtain s' and t'. Then we compare and s' and t' in the *recursive path ordering* of Dershowitz [D82]. If we use a precedence ordering on function symbols such that $\cup$ is minimal in this ordering, then this gives a termination ordering according to results in Bachmair and Plaisted [BP85]. Now, to show that if $s \Rightarrow_d t$ then $s \succ t$ in this ordering, it suffices to show that if $s \Rightarrow_n d$ then $s \succ t$ in this ordering, because of properties of the recursive path ordering. Without going into details, we have the following result:

Theorem 4.1. Suppose we are using a precedence ordering on function symbols in which the a-c operator $\cup$ is minimal. Let us define ordering $\succ$ so that $s \succ t$ if $s' \succ_{rpo} t'$ where s' and t' are the flattened forms of s and t and $\succ_{rpo}$ is the recursive path ordering using this precedence ordering on function symbols. Then a SEL program $\mathcal{R}$ (without conditionals) is terminating if the following conditions are true:

1. If $s = t$ is in $\mathcal{R}$ then $s \succ t$.
2. If $s \supseteq t$ is in $\mathcal{R}$ then $s \succ t$.

To deal with conditionals, we need to use another method, because conditionals are not terminating in the usual sense of the term. Consider the following program:

```
fact(x) = if (x = 0) then 1 else x * fact(x - 1).
```

The `if then else` connective presupposes that the `if` part is evaluated first. However, in usual term rewriting, any subterm may be reduced. Thus, if we always choose to evaluate the `else` part first, this definition of `fact` will not terminate. To avoid this problem, we reformulate the above program as follows:

```
fact(x) = g(x = 0, x)

g(true,x) = 1

g(false,x) = x * fact(x - 1)
```

Now, no matter what evaluation strategy is chosen, this program is terminating. This same method of reformulating conditionals can be used in SEL, and then it is at least theoretically possible to prove termination of SEL programs involving conditionals, using conventional methods.

5. Conclusions

We have proposed a language based on equations and sets for functional programming. We have shown several examples of operations in this language, and hope that the reader is convinced of its usefulness. The need for a set construct has been recognized by a number of researchers in the fields of functional and logic programming [T81, R84, N85, DFP86, JS86, P86]. Existing approaches fall short of an ideal solution in that sets are not treated as first class objects in the language. Our approach does not compromise the basic properties of sets, yet is amenable to a reasonably efficient execution. The heart of our approach lies in associative-commutative rewriting, which allows many useful definitions to be stated in a non-recursive manner. We now review some of our major assumptions, potential limitations, and possible extensions.

One major requirement we have made in our semantics is termination. Although this is a fairly common assumption in term-rewriting systems—O'Donnell's equational language is an exception [HO82]—it is less usual in a programming language. Nevertheless, the class of terminating programs is interesting because it includes many useful applications. In this paper, we have sketched an approach to proving the termination of SEL programs—for a thorough survey of methods, see [D85]. Another requirement we have placed on the language is that equational rules must be confluent. Syntactic checks for confluence have been considered fairly extensively in the literature [H80, HO80, HO82]. Our approach is based on semantic confluence [P85], extended suitably for handling the associative-commutative $\cup$ operator. We have not presented the details of this approach in this paper.

We can relax the termination requirement without causing any change in our programming paradigm. Thus, we can write definitions such as

$$\texttt{ints}(n) = \texttt{cons}(n, \texttt{ints}(n+1))$$

to define the infinite sequence of integers. However, the semantics of our language would become more complicated; we might have to give up a simple logical semantics in favor of a complex denotational semantics. Also, in a sequential implementation, the search for solutions by a depth-first search would not guarantee completeness because some reduction sequence might not terminate; a more space-consuming breadth-first search would be necessary.

As we wished to concentrate on semantic issues in this paper, we have not discussed the implementation of our proposed language. The reader might easily see that, in a sequential implementation, simplification using equations and backtracking to compute elements of a set can be implemented straightforwardly. The use of a-c matching in accessing elements of a set in SEL can also be made efficient because most definitions use the *scons* operator, rather than the more general $\cup$.

Perhaps the major source of inefficiency lies in ensuring that sets have distinct elements; but here too, hashing can be used to minimize the search for duplicate elements. It should be noted that multi-sets are a correct implementation for functions that distribute over nondeterminism,

and therefore the need to test for duplicate elements is avoided in many cases.

It is easy to extract parallelism in the language SEL because it is a pure declarative language [CK81]. Both *and* and *or* parallelism is possible; the former corresponds to evaluating multiple arguments to operators (except in the case of *cond*), and the latter corresponds to different $\Rightarrow_n$ reductions. Extracting *and* parallelism in SEL does not incur the problems of a logic programming language because the different *and* processes have fully instantiated arguments. Similarly, inheriting the parent environment while extracting *or* parallelism is also easier; the parent environment cannot be altered by a child *or* process through the binding of *logical variables* (there are no logical variables in SEL). With parallel execution, it becomes feasible to maintain completeness even when some reduction sequence does not terminate.

We have limited our language to being first-order, and also limited the sets to be defined by enumeration of elements according a recursive rule, i.e., we do not allow a set to be defined as the collection of solutions of a set of equations, as, for example, in [DFP86, R84, JS86]. In other words, we have not allowed *narrowing* [DP85, R85] in this language; we are nevertheless able to obtain much of the benefits of narrowing through our restricted form of nondeterministic evaluation. We are at present investigating ways of including these additional capabilities in our language. We are also planning an implementation of these ideas.

Acknowledgments

We thank the anonymous referees for their comments and suggestions.

References

[B83] A. Bundy, "The Computer Modelling of Mathematical Reasoning," Academic Press, New York, 1983.

[BD85] L. Bachmair and N. Dershowitz, "Commutation, transformation, and temination," In *Proc. of 8th Int'l CADE*, Oxford, Springer Lecture Notes in Computer Science, **230**, pp. 5-20.

[BKN85] D. Benanav, D. Kapur, and P. Narendran, "On the complexity of matching problems," In *Rewriting Techniques and Applications*, pp. 417-429, Dijon, France, May 1985.

[BP85] L. Bachmair and D.A. Plaisted, "Associative Path Ordering," *J. of Symbolic Computation*, **1**, pp. 329-349.

[C78] K. L. Clark, "Negation as Failure," In *Logic and Data Bases*, Ed. H. Gallaire and J. Minker, Plenum Press, New York, 1978, pp. 293–322.

[CK81] J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," In *Conf. Functional Prog. Lang. and Comp. Arch., ACM*, 1981, pp. 163–170.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog.* Springer-Verlag, New York, 1981.

[D82] N. Dershowitz, "Orderings for term-rewriting systems," *Theoretical Computer Science*, **17**, pp. 279-301.

[D85] N. Dershowitz, "Termination of Rewriting," Technical report UIUCDCS-R-85-1220, University of Illinois at Urbana-Champaign, August 1985.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, pp. 54–66.

[DFP86] J. Darlington, A.J. Field, and H. Pull, "Unification of Functional and Logic Languages," In DeGroot and Lindstrom (eds.), *Logic Programming, Relations, Functions and Equations*, pp. 37-70, Prentice-Hall, 1986.

[FW80] D.P. Friedman and D.S. Wise, "An Indeterminate Constructor for Applicative Programming," In *7th ACM POPL*, pp. 245-250, Las Vegas, January 1980.

[H80] G. Huet, "Confluent Reductions: abstract properties and applications to term rewriting systems," *J. ACM*, **27**, 1980, pp. 797-821.

[HO80] G. Huet and D. Oppen, "Equations and Rewrite Rules: a Survey," In *Formal Languages: Perspectives and Open Problems*, R. Book (ed.), Academic Press, New York 1980.

[HO82] C. M. Hoffman and M. J. O'Donnell, "Programming with Equations," *ACM TOPLAS* **4**, No. 1 (January 1982) pp. 83–112.

[JS86] B. Jayaraman and F.S.K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," *In 1986 ACM Symposium on LISP and Functional Programming*, pp. 320-331, Boston, 1986.

[KB70] D. E. Knuth and P. Bendix, "Simple Word Problems in Universal Algebras," In *Computational Problems in Abstract Algebra*, J. Leech (ed.), Pergamon Press, New York, pp. 263-297, 1970.

[K82] R. M. Keller, "FEL (Function Equation Language) Programmer's Guide," AMPS Technical Memo 7, Department of Computer Science, University of Utah, April 1982.

[N85] L. Naish, "All Solutions Predicates in Prolog," In *Symp. on Logic Programming*, Boston, 1985, pp. 73-77.

[O85] M. J. O'Donnell, "Equational logic as a programming language," M.I.T. Press, 1985.

[P85] D.A. Plaisted, "Semantic Confluence Tests and Completion Methods," *Information and Control*, **65**, pp. 182-215, 1985.

[P86] D.A. Plaisted, "Nondeterminism by Associative-Commutative Rewriting," Internal Report, Department of Computer Science, University of North Carolina, Chapel Hill, March 1986, 30 pages.

[R84] J. A. Robinson, "New Generation Knowledge Processing: Syracuse University Parallel Expression Reduction," First Annual Progress Report, December 1984.

[R85] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages," In *1985 Symp. on Logic Programming*, Boston, 1985, pp. 138–151.

[T81] D. A. Turner, "The semantic elegance of applicative languages," In *ACM Symp. on Func. Prog. and Comp. Arch.*, New Hampshire, October, 1981, pp. 85-92.