Electronic Notes in Theoretical Computer Science 76 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume76.html
17 pages

Set-based Nondeterministic Declarative Programming in Singleton

Gianfranco Rossi [1,2]

Dipartimento di Matematica
Università di Parma
43100 Parma, Italy

Abstract

In this paper we present a declarative language that aims at combining valuable features of CLP languages—namely, nondeterminism, unification, constraint solving, dynamic data structures—with features of conventional programming languages we are accustomed to and that we do not want to give up, such as the syntactic form of programs, the deterministic control structures—in particular the iterative ones—, the notion of procedure and parameter passing. A key role to gain these objectives is played by the notion of sets: sets serve not only as a powerful data abstraction, but also as the (only) source of nondeterminism and as the main support for declarative (constraint) programming. Furthermore, semantics of the whole language can be described in terms of a CLP language with sets, which is used also as a straightforward implementation of the proposed language.

1 Introduction

Many of the efforts in designing declarative programming languages have been based at some extent on (Constraint) Logic Programming ((C)LP) languages. However, it is known that these languages (and Prolog in particular) present peculiarities that in practice make them not as well appreciated as they should be. This "negative reputation" is often inherited by those declarative languages that, though different in name and possibly more powerful than conventional LP languages, still clearly show their Prolog-based nature. Among the aspects of LP languages that are often criticized we can enumerate:

- the syntactic form of programs which is quite far from the usual syntactic form of conventional languages we are usually accustomed to

[1] This work is partially supported by MURST project *Ragionamento su aggregati e numeri a supporto della programmazione e relative verifiche.*
[2] Email: gianfranco.rossi@unipr.it

- the difficulty to understand the LP computational model, in particular the notion of nondeterministic computation
- the absence of the usual iterative control structures (e.g., for, while) and consequently the recursion based nature of many LP programs.

Conversely, there are a number of other features of LP languages that, though sometimes criticized, surely represent valuable features to support a real declarative programming style. Among them, the use of "logical" variables (instead of modifiable "programming language" variables), the uniform use of unification both to test and to assign values, the dynamic nature of its data structures and the capability to perform computation with partially specified data (i.e., those containing uninitialized variables). Moving form LP to CLP [5], we can add the capability to decide satisfiability of formulae on a specific domain, disregarding the order in which they are encountered and the instantiation of variables occurring in them. Last but not least, the availability of a precise formal semantics is also a feature of (C)LP languages which surely deserves to be preserved.

In this paper we describe a language—called Singleton—which tries to preserve as much as possible the valuable features of CLP languages, while avoiding as much as possible their controversial aspects. The proposed language superficially resembles conventional languages (e.g., Pascal, C) but in the depth it is akin to CLP languages.

Specifically, some notable features of Singleton are:

- a Pascal-like syntax, with Pascal-like control structures, in particular, iterative control structures
- nondeterminism, but confined to set operations, which are inherently nondeterministic and provide therefore a more natural and easier explanation of nondeterminism (see, e.g., [7])
- a powerful set constraint solver that allows to deal with sets and set operations in a real declarative way.

Furthermore, Singleton still preserves various features of LP languages, such as the availability of (only) "logical" variables (without assignment), the ability to compute with partial information, the use of unification, and the absence of any static type structure.

Other remarkable features are the possibility to use expressions as statements and also, vice versa, statements as expressions, and the presence of intensional sets (i.e., sets defined by property).

Sets play a key role in Singleton. In fact, not only sets can be used as a powerful data abstraction, but also they provide the (only) source of nondeterminism in the language and the main support for declarative (constraint) programming. Actually, the whole language semantics can be described in terms of a CLP language with sets, namely CLP($\mathcal{SET}$) [3]. CLP($\mathcal{SET}$) is also used to provide a straightforward implementation of Singleton.

The paper is organized as follows. In Section 2 we give an informal presentation of Singleton by showing a simple Singleton program. In Section 3 we introduce the fundamental data structures of Singleton, namely sets and lists. In Section 4 we define a few key notions concerning program computation and the use of expressions and statements. In Section 5 we introduce uninitialized variables and then, in Section 6, we briefly describe the (set) constraint handling facilities supported by the language. The fundamental notion of nondeterminism and its relationships with sets are addressed in Section 7. In Section 8 we present the constructs used to simulate the usual iterative control structures, while procedure definition and parameter passing are briefly discussed in Section 9. In Section 10 we provide a few hints on the technique used for dealing with intensional sets possibly denoting infinite sets. Next, in Section 11 we show the core part of the translation function which maps Singleton constructs to CLP($\mathcal{SET}$) clauses and terms. Finally, in Section 12 we briefly discuss related and future work.

2 An informal introduction to Singleton

First of all we show a simple example of a program written in Singleton which allows us to give the flavor of the programming style supported by the language.

Problem: Read a sequence of integers (ended by end-of-file) from the standard input, compute and print its maximum.

We present first the subprogram that computes the maximum of a set S of integers. Then we show the main program that implements the required input/output facilities and calls the subprogram. Observe that the proposed implementation does not take care of execution efficiency. Indeed, Singleton is mainly conceived as a tool for rapid software prototyping, where easiness of program development and program understanding prevail over efficiency.

```
procedure max(in S; inout x)
begin
    x in S;
    for y in S do x>=y end
end
```

The subprogram is implemented as a procedure, named `max`. The overall syntax is that of conventional block structured languages (e.g., Pascal). Procedures are the only abstraction available in Singleton for defining subprograms. Formal parameters in a procedure can be either input parameters (**in**), or output parameters (**out**) or both (**inout**). The first statement uses a possibly *uninitialized* variable `x`. The statement is a boolean expression (actually a constraint expression) used as a statement. If the expression is evaluated to true the statement succeeds; if it is evaluated to false the statement (hence the computation of `max`) fails. `x in S` is evaluated to true if `S` is a set and `x` belongs to `S`. If `x` is uninitialized when the expression is evaluated this amounts to *nondeterministically* assign an element of `S` to `x`.

The **for** statement allows to test if the condition `x >= y` holds for all `y` belonging to `S`, with `x` initialized to a specific element of `S`. `y` is implicitly declared as a local variable in the **for** construct. A new instance of `y` is created for each element of `S`, each time the statement part of the **for** is executed. Again, an expression (`x >= y`) is used as a statement. If the evaluation of this expression gets a false result, the whole statement **for** fails. If the **for** terminates with success (i.e., `x >= y` is evaluated to true for each `y`) then the whole procedure terminates with success. The value of `x` represents the integer we are looking for. If, on the contrary, `x >= y` is evaluated to false for some `y`, backtracking takes place and the computation goes back till the nearest choice point. In this case, the nearest and only choice point is the one created by the `in` operator. Its execution will bind nondeterministically `x` to each element of `S`, one after the other. If all values of `S` have been attempted, there is no further alternative to explore and the whole computation of the procedure fails.

The use of the **inout** mode for the second parameter of `max` allows the procedure to be used both to check whether a specified value for `x` is the maximum of the set `S` and to generate the maximum of `S`.

The main program that uses `max` can be defined as follows:

```
procedure main()
begin
    var L in list;
    var S,m;
    L = [x | read(x) & x in integer];
    ListToSet(L,S);
    max(S,m);
    writel("The maximum is " <+ m)
end
```

The main program is declared as a procedure with the special name **main**. The **var** keyword introduces variable declarations. `L`, `S`, and `m` are three local variables. There are no type declarations. However, one can constraint the value of a variable to a set of possible values, using set constraints (hence, checked at run-time). For example, `L` is constrained to be a list, being `list` a predefined set (namely, the set of all possible lists).

The value of `L` is computed by using an *intensional list definition*. In general, intensional formers are used to define sets/lists by properties rather than by enumerating all their elements. In this example the intensional definition is used to collect into a list all values of `x` which can be read from the standard input and which satisfy the constraint to be integer numbers. The collection operation terminates as soon as the end-of-file marker is encountered. The `ListToSet` operation is a library procedure that transforms a list into a set.

This transformation allows the program to subsequently pass the collection of numbers that have been read to the `max` procedure as a set. `max` requires to work on a set (rather than on a list) in order to be able to exploit the nondeterminism offered by set operations (specifically, by set membership).

Finally, the `writel` instruction prints on the standard output all elements of the list which is passed to it. The string literal (in double quotes) is just a syntactic notation for the list of its component characters, whereas the `<+` is one of the built-in operators for list management: the result is the list obtained by adding the value of `m` as the tail element to the list represented by the string literal.

Most features of the language (e.g., atomic data objects, expressions) are very much like those provided by conventional programming languages and will be skipped. Hereafter, we shall concentrate, instead, on those features that differentiate Singleton from conventional languages (and, on the other hand, that make it closer to a CLP language).

3 Atomic and composite data objects

Singleton provides two kinds of composite data objects: *sets* and *lists*. In sets the order of elements and the repetitions do not matter. We shall use the term *set/list aggregate expression* to refer to a data expression which denotes a set or a list data object. There are three kinds of set/list aggregate expressions, namely extensional, intensional and compound set/list aggregates.

Definition 3.1 An *extensional set aggregate* is a data expression of the form $\{e_1, \ldots, e_n\}$ $(n \geq 0)$, where $e_1, \ldots, e_n$ are data expressions. The set denoted by this expression is the set $\{val(e_1)\} \cup \cdots \cup \{val(e_n)\}$. In particular, $\{\}$ is used to denote the *empty set*.

Definition 3.2 An *intensional set aggregate* is a data expression of the form

$$\{\mathsf{DExpr} \mid \mathbf{var}\, \mathsf{VarList}; \mathsf{BoolExpr}\}$$

where DExpr is a data expression containing a not empty set of variables C, $\mathsf{VarList}$ is a (possibly empty) list of variables with $C \cap \mathsf{VarList} = \emptyset$, and $\mathsf{BoolExpr}$ is a boolean expression involving all the variables in $C \cup \mathsf{VarList}$. The set denoted is the collection of all the values obtained by evaluating DExpr over all possible values of the variables occurring in it such that the boolean expression $\mathsf{BoolExpr}$ is true. Variables occurring in DExpr and those in $\mathsf{VarList}$ are *local* variables, i.e., their scope is the intensional aggregate. When the list $\mathsf{VarList}$ is empty the whole variable declaration can be omitted.

From a logical point of view the meaning of $S = \{T \mid \mathbf{var}\, V_1, \ldots, V_n; E\}$, where T contains variables $Y_1, \ldots, Y_m$, is

$$\forall x(x \in S \leftrightarrow \exists V_1, \ldots, V_n, Y_1, \ldots, Y_m(x = T \wedge E))$$

Example 3.3 Given a set `S` build a new set `R` whose elements are those of `S` incremented by 1.

- `R = {z + 1 | z in S}`
  (or, equivalently, `R =` {`x` | **var** `z; x = z + 1` **and** `z in S`}).

A special form of an intensional set aggregate is the interval aggregate.

Definition 3.4 An *interval aggregate* is a data expression of the form $\{e_i..e_f\}$, where e_i and e_f are integer expressions and $val(e_i) \leq val(e_f)$. $\{e_i..e_f\}$ denotes the set $\{val(e_i), val(e_i)+1, \ldots, val(e_f)-1, val(e_f)\}$.

A set can be also obtained as the result of evaluating a compound set aggregate expression.

Definition 3.5 Let e be a data expression and s be a set aggregate. A *compound set aggregate* is a data expression of one of the forms:

(i) e `>>` s (element insertion) (ii) e `<<` s (element removal)

Expression (i) denotes the set obtained by removing from s the element, if it exists, whose value equals $val(e)$ (i.e., $val(s) \setminus \{val(e)\}$). Expression ($ii$) denotes the set obtained by adding $val(e)$ to s (i.e., $s \cup \{val(e)\}$).

Example 3.6 (Compound set aggregates)

- `3 + 2 >> {1,3,7}` and `5 >> 1 >> {1,3,7}` denote the set $\{1,3,7,5\}$
- `0 << 1 << {1,2}` denotes the set $\{2\}$.

List aggregates have almost the same syntactic form as set aggregates, apart using square instead of curly brackets. The element insertion and removal operators, however, are different from those of sets. In fact, when dealing with lists it is common to have to apply insertion and removal to the first (the *head*) or to the last (the *tail*) element of a list, while the order of elements is immaterial in sets.

Definition 3.7 Let e be a data expression, l a list aggregate, and x an uninitialized variable. A *compound list aggregate* is a data expression of the form:

(i) e `+>` l (head element insertion) (iii) x `<-` l (head element removal)

(ii) l `<+` e (tail element insertion) (iv) l `->` x (tail element removal)

Expressions (i) and (ii) denote the list obtained by adding $val(e)$ as the first and the last element of the list l, respectively, whereas expressions (iii) and (iv) denote the list obtained by removing from l the first and the last element, respectively. Evaluation of expressions (iii) and (iv) also causes the value of the removed element to become the value of x.

Character list aggregates can be used to denote *strings*. A more convenient syntactic notation is also introduced: the character list aggregate

$['c'_1,'c'_2,\ldots,'c'_n]$, $n \geq 0$, can be equivalently written as $"c_1c_2\ldots c_n''$.

The language provides also some *predefined sets*: **char**, **integer**, **real** denote the set of characters, integer and real numbers, respectively; **list** and **set** denote the (infinite) sets of all possible lists and sets, respectively. Operations on predefined sets, as well as on interval aggregates, however, are limited to the membership and not membership relations.

4 Expressions, statements and program computation

All user defined subprograms in Singleton take the form of *procedures* with an implicitly associated boolean result which can be exploited whenever the procedure call is used as an expression (see below). The computation of a procedure can terminate with either *success* or a (run-time) *error* or *failure*, according to the following definition.

Definition 4.1 (Failure) A statement fails if it is a boolean expression used as a statement, or it contains a boolean expression used as a statement, and the expression evaluates to false, or it is a procedure call and the sequence of statements in the body of the procedure fails. A sequence of statements fails if one of its statements fails. The program computation fails if the sequence of statements in the main procedure fails. Otherwise, the computation succeeds.

The computation in Singleton is *nondeterministic*. Nondeterminism introduces choice points and backtracking. Once a computation branch terminates with failure, the computation backtracks to the most recently created choice point. If no choice point is left open the whole computation fails.

There is no assignment statement. Control structures are similar to those of conventional programming languages (at least superficially) and will be described more precisely in Section 8. Differently from most conventional languages (but similarly to, e.g., Alma-0 [1]), Singleton allows statements to be used as boolean expressions and also, vice versa, boolean expressions to be used as statements.

Definition 4.2 (Expression as statements) Let B be a boolean expression used as a statement. If B evaluates to false the computation of the statement fails. If B evaluates to true the computation continues.

Definition 4.3 (Statement as expressions) Let S be a statement used as an expression. If execution of S succeeds, then the value returned by the statement is true; otherwise, the value is false. In particular, a procedure call succeeds (hence, returns a true value) if the called procedure terminates with success. Furthermore, a compound statement **begin** `S1; S2; ...; Sn` **end** is true if and only if all statements `S1; S2; ...; Sn` are true.

Any statement execution returns a boolean result. Statements, therefore, can be used everywhere usual boolean expressions can occur. In particular, statements can be used as conditions in the test part of an **if** or of a **while** statement, and as part of a larger boolean expression.

Example 4.4 Check whether all members of the collection `S` are negative integer numbers.

```
if for x in S do x in integer and x < 0 end
then writel("all negative numbers")
end
```

The **for** statement is used as a boolean expression, while the boolean expression `x in` **integer and** `x < 0` is used as a statement.

Of particular relevance is the use of *equality* both as a statement and as a boolean expression. Equality is always and uniformly dealt with as *unification* (specifically, *set unification* [4]). We will come back to equalities and set unification in Section 6.

5 Uninitialized variables

A variable upon its declaration has no value associated with it, that is the variable is *uninitialized*. A variable remains uninitialized until a value t is assigned to it. After a variable x has got a value then no other assignment to x is feasible. That is, all variables are dealt with as real "logical" variables.

Uninitialized variables can occur in: (*i*) set/list extensional aggregates and compound set/list aggregates (with some restrictions in the case of list aggregates); (*ii*) equality, disequalities and other constraint expressions (see next section); (*iii*) in procedure calls, in correspondence with **out** and **inout** parameters (see Section 9). No other expression can contain uninitialized variables, and a run-time error is detected if this is not respected.[3]

Set/List aggregates containing uninitialized variables represent *partially specified sets/lists*, i.e., sets/lists where either some of the elements or part of the sets/lists themselves are unknown. Actually, each partially specified set/list denotes a possibly infinite collection of different objects, that is all sets/lists which can be obtained by assigning values to the uninitialized variables.

Example 5.1 (Partially specified sets/lists)

- The extensional set aggregate {`3+2,x,y`}, where `x` and `y` are uninitialized variables, represents a partially specified set which contains two unknown elements, denoted `x` and `y`; note that the cardinality of the sets denoted by

[3] The fact that uninitialized variables can occur only in a quite limited number of different expressions, possibly with restrictions, depends primarily on the capabilities of the constraint solver provided by the language. If, for instance, also arithmetic operators would be dealt with as constraints then we could evaluate arithmetic expressions containing uninitialized variables as well. Including other constraint domains and the relevant solvers will be a possible future extension of our language.

this aggregate can vary from 3 to 1 depending on the values assigned to `x` and `y` (being 1 if both `x` and `y` get the value `5`).

- The compound set aggregate `x+1 >> x+2 >> S`, where `S` is an uninitialized variable and `x` has value `3`, represents a partially specified set containing two elements, `4` and `5`, and an unknown part `S`; in this case, the cardinality of the denoted sets has no upper bound (the lower being 2).

Uninitialized variables can occur also in constraint expressions. Although containing uninitialized variables such expressions can be always evaluated, yielding a true or a false result. Constraints are addressed in more details the next section.

6 (Set) Constraints

Basic set-theoretical operations, as well as equalities and disequalities, are dealt with as *constraints* in Singleton. The evaluation of expressions containing such operations is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the global *constraint store*) using domain specific constraint solvers. Those parts of these expressions, usually involving one or more uninitialized variables, which cannot be completely solved are added to the constraint store and will be used to narrow the set of possible values that can be assigned to the uninitialized variables.

The approach adopted for constraint solving in Singleton is the one developed for CLP($\mathcal{SET}$)[3]. Basically, the constraint store is a conjunction of atomic formulae built using basic set-theoretic operators, along with equality and disequality. Satisfiability is checked in a set-theoretic domain, using a suitable constraint solver which tries to reduce any conjunction of constraints to a simplified form—the *solved form*—which can be easily tested for satisfiability. The success of this reduction process allows one to conclude the satisfiability of the original collection of constraints. The reduction to false, on the contrary, implies the unsatisfiability of the original constraints. Solved form constraints are left in the current constraint store and passed ahead to the new state. A successful computation, therefore, may terminate with a not empty collection of solved form constraints in the final computation state.

Constraints in Singleton are basically the *set constraints* of CLP($\mathcal{SET}$), that is conjunctions of atomic constraint expressions based on: equality, membership, (strict) inclusion, union, disjunction, intersection, set difference, and, for most of them, also their negative counterparts.

Example 6.1 Let `x`, `y`, `R`, `S`, and `T` be uninitialized variables, and let the global constraint store be initially empty.

- `y in S` **and** `!subset({x},S)`: true, with the solved form constraint

$$\texttt{S = y>>A} \wedge \texttt{set(A)} \wedge \texttt{x != y} \wedge \texttt{x !in A}$$

added to the constraint store, where `A` is a new uninitialized variable and `set(A)` a solved form constraint (`set(t)` is `true` whenever `t` denotes a set);

- `S = y >> R` **and** `un({x},S,T)` **and** `disj({x},S)`: `true`, with the constraint in solved form

$$\texttt{T = x>>y>>R} \wedge \texttt{x != y} \wedge \texttt{x !in R}$$

added to the constraint store.

The constraint solver is always able to decide whether a constraint is false or true, even if all its arguments are uninitialized variables. Moreover, the order of atomic constraints is completely immaterial. All these features strongly contribute to support a highly declarative programming style.

An implicit *delay* mechanism is also provided for dynamically postponing the evaluation of some constraints. Specifically, membership constraints of the form `x` **in** `T`, where `T` is a predefined set, or membership constraints of the form `x` **in** `A` where `A` is any aggregate but the constraint occurs in a variable or in a formal parameter declaration (e.g., **var** `x` **in** `{1..10000}` are always automatically delayed until `x` becomes initialized to some value. A delayed constraint is not evaluated until the blocking condition does not hold. At the end of the computation, the delayed constraints that still remain blocked (if any), are anyway evaluated disregarding their blocking conditions.

Finally, observe that the possibility to associate membership constraints with variable declarations provides a sort of run-time typing for variables in a program.

7 Nondeterminism

A computation in SINGLETON can be nondeterministic. Nondeterminism is another key feature of a programming language to support declarative programming. One distinguishing feature of SINGLETON, however, is that nondeterminism is confined to set operations. The notion of nondeterminism fits into that of set very naturally (see for instance [7]). Set unification and many other set operations are inherently and naturally nondeterministic. For example, the evaluation of `x in {1,2,3}` with `x` an uninitialized variable, nondeterministically returns one among `x = 1`, `x = 2`, `x = 3`. Since the semantics of set operations is usually well understood and quite "intuitive", making nondeterministic programming the same as programming with sets can contribute to make the (not trivial) notion of nondeterminism easier to understand and to use. Furthermore, restricting the creation and handling of choice points to set operations is likely to make nondeterminism run-time support simpler to implement.

On the other hand, other kinds of nondeterministic constructs can be easily "simulated" using the nondeterministic facilities provided by set operations. For example the Alma-0 [1] nondeterministic statement `EITHER S ORELSE T`

`END`, is simulated in Singleton by:

```
x in {1,2};
if x = 1 then S else T end
```

Similarly, the Alma-0 nondeterministic statement `SOME i:=s TO t DO S END`, whose logical meaning is the bounded existential quantification, $\exists$`i` $\in$ [`s`..`t`] `S`, is simulated by the Singleton statements:

```
i in {s..t}; S
```

A simple way to exploit nondeterminism in Singleton is through the use of intensional sets. This powerful abstraction allows one to explore the whole search space of a nondeterministic computation and to collect into a set all the computed solutions. Then the collected set can be processed, e.g., by iterating over all its elements using the `for` statement. In this way, for example, one can easily simulate the behaviour of the rather complicated `FORALL S DO T END` statement of Alma-0, whose purpose is to iterate over all choice points created by `S`.

Example 7.1 Compute and print the number of occurrences of a string `p` in a string `s`.
Let `StringMatch(p,s,k)` be a procedure which is able to check if `p` is a substring of `s` and to return the position `k` where the substring `p` starts (see Section 9).

```
write(#{k | StringMatch(p,s,k));
```

where `#A` yields the number of elements of the set/list denoted by `A`.

Again we stress the fact that in Singleton nondeterminism is completely confined to set operations. The user needs only to understand the semantics of these operations: nondeterminism is naturally embedded in them.

8 Control structures

Singleton provides a few constructs for implementing the usual control structures, namely the **if**, **for**, and **while** statements. The **if** statement is defined in the very standard way. Conversely, the **for** and **while** statements, though superficially similar to the usual ones, are substantially different from them. Indeed, the absence of the assignment statement prevents one to use modifiable loop control variables. We give an intuition of the syntactic form and the semantics of the **for** and **while** statements through a couple of simple examples.

Example 8.1 Write the squares of the first ten natural numbers, one for each output line.

```
for i in {1..10} do
```

```
    write(i*i); nl
end
```

The **for** statement is used to specify that a certain action is to be performed for each element of a given set/list (in particular, of an interval). At each iteration, a new instance of the loop variable (`i` in the above example) is obtained and the value of a new element from the set/list is assigned to it. If the statement part fails for some value of the loop variable then the whole statement **for** fails. Otherwise, it terminates with success.

Example 8.2 Check whether a sequence of characters `L` is symmetrical or not.

```
while #(L) > 1 do
    var first, last, NewL;
    NewL = first <- L -> last;
    first = last;
end(NewL => L)
```

The **while** statement is repeatedly executed until the number of elements in `L`, `#(L)`, is less or equal to 1. All variables declared in the body of the **while** statement—as well as those of a **for** statement—are *local* to the statement itself. At each iteration, new fresh copies of all local variables are allocated, like local variables in block structured languages. In Example 8.2, a new list `NewL` is obtained from `L` at each iteration by removing from `L` its first and last elements. Unification is used as an assignment to initialize `NewL` and as a test for checking whether `first` equals `last`. If this test fails, the **while** statement fails. Otherwise, the **while** statement terminates with success after #(L)/2 iterations.

Since the value of a variable `x` involved in a **for**/**while** loop can not be changed, a mechanism is provided that allows to store the new value computed from `x` in a new (local) variable `y` and to state that the latter is to be used in place of the former at the next iteration. Syntactically, this is specified by a clause `y => x` following the **end** keyword of the loop statement. In general, a loop statement can be ended by a sequence of pairs `Vi => Vj`, where `Vi` and `Vj` are variable identifiers, whose meaning is that `Vi` literally replaces `Vj` at the next iteration (note that this is not an assignment: the value possibly bound to `Vj` remains unaltered). In Example 8.2, for instance, the clause `(NewL => L)` requires that the current instance of `NewL` is used in place of `L` at the next iteration.

9 Procedure definition and parameter passing

Procedure declarations and procedure calls in Singleton have basically the usual Pascal-like form. When used as a boolean expression, a procedure call always (implicitly) returns a boolean value: namely, if execution of the procedure terminates with success the procedure call returns a true result; otherwise it returns false.

Formal parameters must be variables, possibly constrained by `in` constraints (e.g., **inout** `X in` **integer**). The parameter passing modes can be: (*i*) **in** (the default mode), the actual parameter must be a completely specified value; (*ii*) **out**, the actual parameter must be an uninitialized variable; **inout**, the actual parameter can be a partially specified value. The flexible parameter passing mechanism, along with the ability to compute with uninitialized variables, allow in general to use procedures in a quite flexible way, e.g., using the same procedure both for testing and computing solutions (see for example the procedure `max` in Section 2).

Procedure definitions can be also recursive. The next example shows a fully nondeterministic recursive definition of the classical list concatenation procedure.

Example 9.1 `L3` is the list obtained by concatenating the two lists `L1` and `L2`.

```
procedure concat(inout L1,L2,L3)
begin
    var x, R, NewL3;
    L1 in {[],x +> R};
    if L1 = [] then L3 = L2
    else L3 = x +> NewL3
         concat(R,L2,NewL3);
    end
end
```

When `L1` is initialized, the statement `L1 in {[],x +> R}` is just a test that `L1` is a list. If, in contrast, `L1` is uninitialized (or it is initialized to a partially specified list), the statement nondeterministically unifies `L1` with one of the two possible values, `[]` or `x +> R`. Without this statement, `concat` could be used only to test or to generate `L3` provided either `L1` or `L2` are initialized to a list; with this statement (and the **inout** mode), `concat` can be used both to check if a given concatenation of lists holds and to build any of the three lists, starting from any of the other two (like in the usual well-known definition of the `append` predicate in Prolog). Using this nondeterministic version of the procedure `concat`, it is easy to define the `StringMatch` procedure of Example 7.1 in the very same way as in Prolog.

10 Dealing with infinite sets/lists

The value of an intensional aggregate expression is the set/list of all elements satisfying the property stated by the intensional definition. However, not always the evaluation of an intensional aggregate necessarily requires the explicit construction of this set/list. For instance, `c in {x | P(x)}` can be equivalently evaluated as `P(c)` without having to generate and collect all possible values of `x` for which `P(x)` is true. This is particularly convenient if the set/list denoted by the intensional aggregate is an infinite one. Dealing with `c in {x | P(x)}` as `P(c)` allows to get an answer (true of false) even if `{x | P(x)}` denotes an infinite set.

Following [2] we assume that intensional aggregates are dealt with so as to reduce the need to perform set/list collection. Specifically, we assume that *membership* predicates involving intensional aggregates are always transformed as in the above example, thus completely avoiding the need to enumerate all the constituting elements. Moreover, equalities of the form $x = e$ where x is an uninitialized variable and e is an intensional aggregate do not force evaluation of e: the aggregate is instead passed on through the computation unaltered. As soon as the intensional aggregate needs to be evaluated, e.g. when it occurs in a `write` statement, then the appropriate set collection operation is performed.

Example 10.1 Consider the sequence of statements

```
var nested_lists = {z | (z = [x] and x in nested_lists) or z = []};
[[]] in nested_lists;
```

The intensional set aggregate denotes the sets of nested lists of the form `[]`, `[[]]`, `[[[]]]`, ..., with arbitrary nesting depth, which is clearly an infinite set. This set however is not explicitly generated but is kept in an intensional form. Thus when executing the second statement the expression actually evaluated is

```
([[]] = [x] and in nested_lists) or [[]] = [].
```

Evaluation of the left-hand part of the **or** expression gets `[] in nested_lists` which in turn is evaluated as

```
([] = [x] and x in nested_lists) or [] = []
```

which succeeds (specifically, the left-hand part of the **or** fails whereas the right-hand part succeeds).

The same technique could be applied to other set/list predicates and operators as suggested in [2]. Since this kind of general intensional set constraint management has not been explored in depth yet, we prefer here to restrict the current version of our language to membership predicates, leaving the other cases for future work.

11 Semantics

Singleton programs can be translated into CLP($\mathcal{SET}$) programs in a relatively straightforward way. This translation can serve as a precise (logical) semantics for Singleton. Moreover, since CLP($\mathcal{SET}$) is an executable language, this translation provides also a quick implementation of the Singleton language. In fact, the current available implementation of our language is based on a translator written in Prolog (using DCG) that generates CLP($\mathcal{SET}$) code.

The translation from Singleton to CLP($\mathcal{SET}$) is defined by considering first the translation of data expressions, then of boolean expressions, and finally of statements and procedures. Hereafter we show the core part of the definition of a function ϕ that translates any Singleton data expression t into the corresponding CLP($\mathcal{SET}$) definition. We assume ϕ has access to a global collection of CLP($\mathcal{SET}$) atoms C_ϕ and that the function $\mathsf{add}(a)$ is used to add a new atom a to C_ϕ. $\phi(t)$ will return the CLP($\mathcal{SET}$) term corresponding to t and possibly will modify C_ϕ as a side-effect.

Definition 11.1 Let s, t, $t_1, \ldots, t_n$ be Singleton data expressions, $X_1, \ldots, X_n$ be new CLP($\mathcal{SET}$) variables not occurring in C_ϕ, and $V(v)$ a function that maps each Singleton variable v into a new distinct CLP($\mathcal{SET}$) variable X_i.

$$\phi(c),\ c \text{ numerical or character literal } \mapsto c$$

$$\phi(v),\ v \text{ variable } \mapsto V(v)$$

$$\phi(\{t_1, \ldots, t_n\}) \mapsto \{\phi(t_1), \ldots, \phi(t_1)\}$$

$$\phi(t >> s) \mapsto \{\phi(t) \mid \phi(s)\}$$

$$\phi(t << s) \mapsto X_1;\ \mathsf{add}(\mathsf{diff}(\phi(s), \{\phi(t)\}, X_1))$$

where diff is the CLP($\mathcal{SET}$) constraint for the set difference operation

$$\phi(\{t \mid \mathbf{var}\, x_1, \ldots, x_n; b)\} \mapsto \{X : exists([X_1, \ldots, X_m], X = \phi(t)\ \&\ \psi(b))\}$$

$$\text{where } \{X_1, \ldots, X_m\} = \{\phi(x_1), \ldots, \phi(x_n)\} \cup \mathsf{vars}(\phi(t))$$

$$\phi(a),\ a \text{ arithmetic expression} \mapsto X_1;\ \mathsf{add}(X_1 \,\mathsf{is}\, a)$$

12 Conclusions, related and future work

We have presented a language that aims at amalgamating features of imperative programming languages with features of CLP languages. The notion of set plays a fundamental role in this combination. In particular, nondeterminism is completely confined to set operations. Programs in this language exhibit a quite good declarative reading, while maintaining most of the structure of programs in conventional languages.

Two works influenced our work more than others: SETL [6] and Alma-0 [1]. Like SETL, Singleton is strongly based on the notion of sets. SETL, however, is much richer than Singleton as concerns primitive facilities for dealing with sets. As a consequence SETL is also "heavier" and more complex than Singleton. Moreover, a notable difference is that SETL is not a constraint language. The notion of constraint is completely lacking in SETL and computing with unspecified values (**om** in SETL terminology) is rather cumbersome. Alma-0, instead, is a quite small elegant imperative language, with nondeterministic constructs and logical variables. Constraints appeared in the last versions of Alma-0, but are not completely developed yet. The data structures are basically those of Pascal, hence static.

Both Alma-0 and SETL provide a number of constructs to support nondeterminism, whose semantics is not always easy to understand. Conversely, in Singleton nondeterminism is naturally supported by set operations. Both Alma-0 and SETL have assignment statement. While it is undeniable that assignment is very useful and natural in many programming situations, it is also quite clear that it strongly complicates the language definition and implementation when it has to coexist with logical variables and nondeterminism. For that, we preferred to restrict our language to logical variables, providing suitable variants of the usual loop constructs that allow to maintain an almost conventional programming style. As a future work, we plan to investigate the possibility to introduce also "programming language" variables as an "impure" facility which extends the base language.

Many other new facilities could be added on the top of the base language, as well, mostly by exploiting set abstractions. In particular, sets along with membership constraints (possibly over infinite sets) can be used to provide in a quite natural and flexible way run-time type information for variables. Static program analysis tools then could exploit this information to perform also type checking at compile-time. Arrays could be introduced, at least at the logical level, as sets of ordered pairs using the existing sets and list manipulation facilities. Also functions could be represented, at least at the logical level, through their graphs, that is sets of ordered tuples (possibly intensionally defined, possibly infinite). For example, the function $f(x) = 2x + 1$ can be equivalently defined as the set `f = {[x,y] | y=2x+1}`, and evaluating $z = f(2)$ amounts to solve the constraint `[2,z] in f` (which is actually already solvable in Singleton). Having functions as sets would allow to manipulate them as data and, for instance, to devise an object-oriented extension of the language, using (nested) sets as the construct where to encapsulate data and the related functions.

Acknowledgments

I wish to thank Barbara Baiocchi for her contribution to the development of the Singleton compiler.

References

[1] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5) 1998, 1014–1066.

[2] R. Carmona, A. Dovier, and G. Rossi. Dealing with Infinite Intensional Sets in CLP. In M. Falaschi, M. Navarro, and A. Policriti, eds, *AGP'97. Joint Conf. on Declarative Programming*, 467–477. (available at www.math.unipr.it/~gianfr/PAPERS/AGP.CDR97.ps).

[3] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5) 2000, 861–931.

[4] A. Dovier, E. Pontelli, and G. Rossi. Set unification. TR-CS-001/2001, Dept. of Computer Science, New Mexico State University, USA, January 2001 (available at www.cs.nmsu.edu/TechReports)

[5] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19–20*, 1994, 503–581.

[6] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets, an introduction to SETL.* Springer-Verlag, 1986.

[7] M. Walicki, S. Meldal. Sets and Nondeterminism. In O. Omodeo and G. Rossi, eds, ICLP93 Post-conference Workshop on Logic Programming with Sets, Budapest, 1993.