Concurrent Distinct Choices*

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. An injective finite mapping is an abstraction common to many programs. We describe the design of an injective finite mapping and its implementation in Curry, a functional logic language. Functional logic programming supports the concurrent asynchronous execution of distinct portions of a program—a condition that prevents passing from one portion to another the structure containing a partially constructed mapping to ensure that a new choice does not violate the injectivity condition. We present some motivating problems and we show fragments of programs that solve these problems using our design and implementation. The complete programs are available on-line.

1 Introduction

A finite mapping is one of the most common abstractions in computer programs. Finite mappings are so ubiquitous that many programming languages offer a builtin type, the *array*, with a special notation to ease the implementation and use of finite mappings. In some situations, e.g., when a programming language does provide builtin arrays or when a mapping has particular requirements, dynamic structures such as linked lists, trees or hash tables are suitable representations of a mapping.

Regardless of the underlying representation, a *mapping* is a (total) function μ from a set I of *indices* to a set V of *values*, i.e., $\mu : I \to V$. The type of both the indices and the values is arbitrary. A mapping is *injective* when distinct indices are mapped to distinct values, i.e., if $i_1, i_2 \in I$ and $i_1 \neq i_2$, then $\mu(i_1) \neq \mu(i_2)$. A mapping is *finite* when the set of indices is finite.

We make either one of two additional assumptions on the set V of values. The first assumption requires the *a priori* knowledge of a finite subset V' of V containing $\mu(I)$. If μ is a injective finite mapping, V' necessarily exists, since $\mu(I)$ has the same cardinality as I, and hence is finite. However, V' must be known

* This research has been partially supported by the DAAD/NSF grant INT-9981317, the German Research Council (DFG) grant Ha 2457/1-2 and the NSF grant CCR-0110496.

before computing μ. This assumption trivially holds when V itself is finite. The second assumption, much weaker, requires the existence of an *enumeration* function of the values, i.e., a bijection $\nu : \mathbb{N} \to V$. Since in a program V is represented by either a primitive type or an algebraic type, this second assumption is easily satisfied for most problems.

In this paper we describe the design and implementation of an injective finite mapping with two particular characteristics. Our programming language is declarative, thus neither state updates nor side effects are allowed. Our programming language is concurrent, thus different portions of the mapping are computed concurrently and asynchronously by different portions of a program. This second characteristic has some non-trivial consequences that will be discussed later.

A class of puzzles knows as cryptarithms is an ideal problem to discuss our design and implementation of an injective finite mapping: the mapping itself is the solution of the problem and it is convenient to compute index-value pairs of this mapping concurrently. This second condition will be explained and motivated in Section 3.

The Merriam-Webster OnLine dictionary defines a *cryptarithm* as "an arithmetic problem in which letters have been substituted for numbers and which is solved by finding all possible pairings of digits with letters that produce a numerically correct answer." A well-known example of cryptarithm is:

```
SEND + MORE = MONEY                (1)
```

Customarily, in a cryptarithm distinct letters stand for distinct digits and leading zeros are not allowed.

The solution of a cryptarithm is an injective finite mapping. The indices are the letters occurring in the cryptarithm. The values are the digits. The solution of (1), graphically represented as a mapping, is shown below in a form that eases verifying its correctness:

```
S E N D + M O R E = M O N E Y
↓ ↓ ↓ ↓   ↓ ↓ ↓ ↓   ↓ ↓ ↓ ↓ ↓
9 5 6 7 + 1 0 8 5 = 1 0 6 5 2
```

A cryptarithm such as (1) in which letters form meaningful words, often in meaningful phrases, is referred to as an *alphametic*. There exist a large number of witty alphametics. Alphametics with a unique solution, such as (1), are more elegant, but a unique solution is not required. The following alphametic has 130 solutions:

```
TOO + MUCH = BEER
```

Solving a cryptarithm by brute force, i.e., by generating and testing every plausible mapping, is inefficient. Finite domain constraint solvers find solutions efficiently. Our program for cryptarithms is not as efficient, although it finds solutions in milliseconds. Our focus is not on the program itself. Rather, the program is a concrete environment for discussing the design and implementation of an injective finite mapping. Our solution contributes the simplicity and efficiency of

this program. Our solution is also general and we will sketch other applications in which it can be applied.

This paper is structured as follows. Section 2 briefly recalls some principles of functional logic programming and the programming language Curry which we use to present the examples. Section 3 presents the design of an injective finite mapping in a functional logic program and its implementation in Curry. Section 4 concludes the paper.

2 Functional Logic Programming and Curry

This section introduces both the basic ideas of functional logic programming and the elements of the programming language Curry that are necessary to understand the subsequent examples.

Functional logic programming integrates in a single programming model the most important features of functional and logic programming (see [6] for a detailed survey). Thus, functional logic languages declare algebraic datatypes, define functions by pattern matching and evaluate expressions containing logical variables. Supporting the latter requires some built-in search principle to guess the appropriate instantiations of logical variables. There exist many languages that are functional logic in this broad sense, e.g., Curry [9], Escher [10], Le Fun [2], Life [1], Mercury [16], NUE-Prolog [12], Oz [15], Toy [11], among others.

One of the most characterizing features of functional logic programming is the evaluation—particularly the *lazy* evaluation—of expressions containing logical variables. Both *narrowing* and *residuation* serve this purpose.

When an expression e cannot be evaluated due to the presence of an uninstantiated logical variable X, narrowing non-deterministically instantiates X to keep the evaluation of e from halting. By contrast, residuation suspends the evaluation of e, transfers control to another portion of the program, and resumes the evaluation of e if and when X becomes sufficiently instantiated.

Residuation is conceptually simple and relatively efficient, but incomplete, i.e., not always able to obtain the result of a computation. By contrast, narrowing is complete if an appropriate strategy [3,4] is chosen, but it is potentially less efficient than residuation for its propensity to generate a larger search space in some situations. Functional logic languages can be effective with either mechanism. Curry offers both residuation and narrowing to the programmer in a unified computation model [7].

Curry has a Haskell-like syntax [13], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry supports logic programming by means of free (logical) variables in both conditions and right-hand sides of defining rules. Thus, a Curry *program* consists of the definition of functions and the declaration of data types on which the functions operate. Functions are evalu-

ated lazily and can be called with partially instantiated arguments. In general, functions are defined by conditional equations, or *rules*, of the form:

$f\ t_1 \ldots t_n$ `|` c `=` e `where` vs `free`

where $t_1, \ldots, t_n$ are *data terms* (i.e., terms without defined function symbols), the *condition* c is either a Boolean function or constraint, e is an expression and the `where` clause introduces a set of free variables. The condition c and the `where` clause are optional. Curry predefines *equational constraints* of the form e_1 `=:=` e_2 which are evaluated by narrowing and are satisfiable if both sides e_1 and e_2 are narrowed to unifiable data terms. Furthermore, "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2 which is evaluated by solving both c_1 and c_2 concurrently.

The `where` clause introduces the free variables vs occurring in c and/or e but not in the left-hand side. Similarly to Haskell, the `where` clause can also contain other local function or pattern definitions. In contrast to Haskell, where the first matching function rule is applied, in Curry all matching (to be more precise, unifiable) rules are non-deterministically applied to support logic programming. This enables the definition of non-deterministic functions which may have more than one result for a given input. An example follows:

```
insert :: a -> [a] -> [a]
insert e []     = [e]
insert e (x:xs) = e : x : xs
insert e (x:xs) = x : insert e xs
```

As in Haskell, `[]` (empty list) and `:` (non-empty list) are the constructors of the polymorphic type *list*. The symbol `a` is a type variable ranging over all types. The first line of the code declares the type of the function `insert`. This declaration is optional, since the compiler can infer it, and it is stated only for checkable redundancy. The type expression α `->` β denotes the type of all functions from type α to type β. Since the application of a function is curried, `insert` takes an element of type `a`, a list of elements of type `a` and returns a list of elements of type `a`, where `a` is any type. The function `insert` inserts an element into a list at some non-deterministically chosen position.

The second and third rule defining `insert` overlap. As a consequence, the expression `(insert 1 [3,5])` has three values: `[1,3,5]`, `[3,1,5]`, and `[3,5,1]`. Using `insert`, we define a permutation of a list by:

```
perm []     = []
perm (x:xs) = insert x (perm xs)
```

As an example of solving constraints, we define a function that checks whether some list starts with a permutation of another list and delivers the list of the remaining elements. For this purpose we use the concatenation of two lists which we define as:

```
(++) eval flex
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

The first line declares that the operator "++" is *flexible*. The application of "++" may instantiate variables in the arguments, if this is necessary to execute a computation step. By default, only constraints are flexible.

Now we define the required function by a single conditional rule:

```
pprefix xs ys | perm ys ++ zs =:= xs
              = zs
              where zs free
```

The operational semantics of Curry, precisely described in [7,9], is a conservative extension of both lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Since computations are based on an optimal evaluation strategy [3,4], Curry can be considered a generalization of concurrent constraint programming [14] with a lazy (optimal) evaluation strategy. Furthermore, Curry also offers features for application programming like modules, monadic I/O, ports for distributed programming, and specialized libraries. We do not discuss these aspects since they are unnecessary to understand our ideas.

There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS [8], a compiler/interpreter for a large subset of Curry.

3 Design and Implementation

As outlined in the introduction, an injective finite mapping is a component of the solution of many problems. A plausible implementation of a finite mapping is any structure defining index-value pairs, e.g., an array, a list of pairs, etc. Index-value pairs are computed during the execution of a program. To ensure injectivity, when a new index-value pair is computed the program must check whether a pair with the same index was previously computed and, if so, whether the value in the previous and the new pairs are the same.

A problem with this approach arises if a functional logic program computes index-value pairs concurrently, e.g., due to residuation. This condition prevents sequentially passing a partially constructed mapping through the portions of a program computing index-value pairs to ensure that a newly computed pair does not violate the injectivity condition. Concurrency is quite common in declarative programming (see also Erlang [5] or Oz [15]) and not uncommon in modern imperative languages. The declarative coordination of concurrent activities demands for specific techniques. In the following, we will show one such technique for the case of injective finite mappings. We make this point more concrete by discussing the architecture of a simple program to solve a cryptarithm.

A program to solve (1) declares one variable for each letter. Initially, these variables are uninstantiated:

```
vs,ve,vn,vd,vm,vo,vr,vy free
```

The solution of the problem is a suitable instantiation of these variables, which implicitly defines the mapping which is the subject of this paper. The instanti-

ation of each variable is determined by the equation of the problem. This equation can be processed as a single unit or it can be broken into a set of "smaller" equations. These smaller equations establish the conditions that the letters must satisfy for the column of the units, the tens, the hundreds, etc., exactly as one would perform the addition by hand. The following display depicts the situation:

$$\begin{array}{cccccc} c_3 & c_2 & c_1 & c_0 & & \\ & \mathtt{S} & \mathtt{E} & \mathtt{N} & \mathtt{D} & + \\ & \mathtt{M} & \mathtt{O} & \mathtt{R} & \mathtt{E} & = \\ \hline \mathtt{M} & \mathtt{O} & \mathtt{N} & \mathtt{E} & \mathtt{Y} & \end{array} \tag{2}$$

where c_i, for $i = 0, 1, 2, 3$, is a carry. For example, the equations of the units and the tens are:

$$\begin{aligned} \mathtt{D} + \mathtt{E} &= 10 * c_0 + \mathtt{Y} \\ c_0 + \mathtt{N} + \mathtt{R} &= 10 * c_1 + \mathtt{E} \end{aligned}$$

Splitting the problem's equation into a set of smaller equations is a slight complication, since it requires the introduction of additional variables for the carries. However, a set of smaller equations has a significant advantage. With appropriate control, the program detects a wrong instantiation, i.e., an instantiation of a variable that does not satisfy some equation, when fewer variables are instantiated. This considerably improves the efficiency of the program execution.

Thus, the program encodes as follows the set of equations that the variables must satisfy:

```
vd+ve    =:= c0*10+vy  &
vn+vr+c0 =:= c1*10+ve  &
ve+vo+c1 =:= c2*10+vn  &
vs+vm+c2 =:= c3*10+vo  &
      c3 =:= vm
```

where `c0` is the carry of the units, `c1` of the tens, etc. Each carry must be either 0 or 1 and consequently it is (non-deterministically) initialized as follows:[3]

`c`$_i$ `= 0!1` $\qquad\qquad i = 0, \ldots, 3$

It follows from the conventions of the problem that `vm` is not zero and consequently `c3` is equal to one. Our simple program ignores this precise inference. However, this equation together with the equation defining the carry constrain the possible values of `vm` to zero and one only.

As we mentioned, splitting the equation of the problem into a set of smaller equations considerably improves the efficiency of the execution, but it introduces a substantial complication. The solutions of the equations are computed concurrently. The order in which the solution of each equation is computed is undetermined. Since the variables, `vs`, `ve`, ... that stand for the letters of the

[3] The infix operator `!` returns one of its arguments. It is defined by the two rules:

```
x ! y = x
x ! y = y
```

puzzle are initially unbound and the addition and multiplication operators residuate, the execution of the equations that the variables must satisfy is suspended until both the operands of an operator become bound. Each variable is non-deterministically bound to a digit, similarly to the carries. In this case, though, the choice ranges over every digit, or every positive digit for `vm` and `vs`. It is inappropriate to non-deterministically instantiate the variables as it is done for the carries, e.g.:

```
vd =:= 0!1!2!3!4!5!6!7!8!9
...                                                    (3)
vm =:= 1!2!3!4!5!6!7!8!9
```

since some of these instantiations do not ensure that distinct variables are bound to distinct digits. It is inappropriate as well to pass around a structure containing the current binding of the variables, since the order in which the variables will be instantiated cannot be easily determined in advance. Here is where our ideas make the difference.

We represent the mapping as a list referred to as the *store*. The store is indexed by the *values* of the problem. In the particular case of cryptarithms, this indexing is natural and straightforward since the values are the digits `0,1,...,9`. Initially, the elements of the store are free variables. The elements in the store are referred to as *tokens*. Putting a token into the store represents the action of choosing a value that must be different from the value of any other choice. The type of the tokens is arbitrary. Often, it is convenient to represent the tokens with the *indices* of the problem. In the particular case of the alphametic (1), we choose the characters `S`, `E`, `N`, ... as the tokens.

Thus, the indexes and values of a problem are used as values and indexes respectively, in the store, i.e., the roles they have in the problem is reversed in the store. We will shortly explain why this reversal of roles is a natural and necessary aspect of the design. In the particular case of the alphametic (1), the set of values is finite. This enables us to create the store when the execution of the program begins. The store is a list of length 10. At the end of the only successful computation for our example (note that in general there can be more than one successful computation), the content of the store is shown below, where • represents an uninstantiated variable:

0	M	Y	•	•	E	N	D	R	S

Thus, in the program, the initial store is a list of 10 free variables:

```
store = [s0,s1,s2,s3,s4,s5,s6,s7,s8,s9]
      where s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 free         (4)
```

A letter of the cryptarithm is paired to a digit by the function `digit` defined as follows:

```
digit token | store !! x =:= token
            = x
            where x = 0!1!2!3!4!5!6!7!8!9
```

Although the associated digit is non-deterministically selected, the condition on the store ensures the injectivity of the mapping. The argument `token` must be unique for each letter, hence, it is natural and convenient to represent it with the letter itself—a character in the program. The store, identified by the variable `store`, is defined in the scope of the function `digit`, hence it does not appear as an argument. The operator `!!` applied to arguments l and i returns the i-th (counting from zero) element of the list l.

Thus, the letters of the cryptarithm are nondeterministically instantiated as follows:

```
vs =:= nzdigit 'S'
ve =:= digit   'E'
vn =:= digit   'N'
...
```

where `nzdigit` is a variant of `digit` that returns only non-zero digits. For example, `(digit 'Y')` returns 2 if and only if the second (counting from zero) element of the store is bound to `'Y'`. The entire program for this problem is available on-line[4].

The reversal of the roles of indices and values in the store may be confusing at first, but it has a natural explanation. The injectivity requirement is intended to *prevent* the condition in which two distinct indices, say l and m, satisfy $\mu(l) = v = \mu(m)$, for some value v. In the store, the value v is associated to some value of the problem, i.e., a digit i. Specifically, the variable v is the i-th element of the store. Every time an index of the problem, i.e., some letter L, is mapped to i, the program attempts to unify, hence instantiate, the variable v to L. The attempt succeeds if and only if either v was uninstantiated, or v was instantiated to L already. Thus, no two distinct indices of the problem can be mapped to the same value of the problem.

In the program that we are discussing, the association between a variable v of the store and a value i of the problem is positional. The store is a list and the variable v is the i-th element of the list. The store is constructed by (4), since the set of values of the problem is finite and known in advance. There are many variations of this design. We discuss two of these variations below. The first variation is useful when no finite set of values is known in advance. The second variation allows more efficient access to the data.

The first variation constructs the list lazily. The `!!` operator, defined in the prelude, is *rigid*. We define an analogous operator *flexible*—the rewrite rules are unchanged:

```
(!!!)           eval flex
(x:xs) !!! n    = if n==0 then x else xs !!! (n-1)
```

If we replace the standard index operator "`!!`" with "`!!!`" in the definition of `digit`, we can replace (4) with the following:

```
store = x where x free
```

[4] `http://www.cs.pdx.edu/~antoy/flp/patterns/distinct-choices-dir/`

This variation is interesting when the set of values of the problem is infinite. The store is indexed by the values of the problem, which in general will not be natural numbers. In this case, we use the *enumeration* function, ν, discussed in the introduction. In this case, a value v is indexed in the store by $\nu^{-1}(v)$.

The second variation represents the store with a tree rather than a list. In this case, a positional association between the variables of the program and the values of the problem is unfeasible or inconvenient. Thus, each node of the tree representing the store is decorated by both indices and values. The tree is filled with all the values of the problem to establish the association between a value and a variable which for lists is implicitly established by the position of the variable in the list. Initially, a node contains a value i of the problem and an uninstantiated variable v. As the program's execution progresses, and an index l of the problem is mapped to the value i, the node of the store containing i is retrieved and the index l is unified, if possible, with v. This variation requires the *a priori* knowledge of a finite set containing the index mapped by the problem.

The crucial feature of the injective finite mapping that we discussed is the possibility of concurrently computing index-value pairs. However, the proposed design can be employed also in problems where concurrency is not an issue. For example, the n-queens puzzle can be implemented in this way. The proposed program, similar to many others for this problem, computes a permutation of the integers $0, 1, \ldots, n-1$, where the i-th element of the permutation is the row in which the queen in the i-th column is placed. A permutation can be seen as an injective mapping of the values $0, 1, \ldots, n-1$ into themselves. In this case, both the indexes and the values of this problem's mapping are most naturally represented by the integer numbers in the range 0 through $n-1$.

The only difference between an implementation using an the injective finite mapping and a more traditional implementation is how a permutation is computed. Using our design, a function to compute a permutation of the integers $0, 1, \ldots, n-1$ is:

```
permute n = result n
   where result n = if n==0 then [] else pick n : result (n-1)
         pick i | store !!! k =:= i = k  where k = range n
         range n | n > 0 = range (n-1)
         range n | n > 0 = n-1
         store free
```

This implementation computes the store lazily. As discussed earlier, this requires the flexible version of the index operator, "`!!!`", which was defined earlier.

We compared the execution time of a program computing all the solutions of the n-queens puzzle using the above function with a similar program using the following function:

```
permute n = result [0..n-1]
   where result []       = []
         result (x:xs)   = insert x (result xs)
         insert x y      = x:y
         insert x (y:ys) = y:insert x ys
```

Using the PAKCS implementation of Curry, the first program takes about 50% more time, regardless of n, than the second program. The memory allocated is the same in each program. This simple experiment suggests that using our implementation of an injective finite mapping imposes no severe overhead w.r.t. more traditional implementations.

As we mentioned above, the splitting of a problem into smaller parts that are solved concurrently has the advantage that wrong instantiations of some variables (i.e., those cannot lead to a solution) are detected earlier. This can lead to a considerable reduction of the search space. For instance, a naive functional solution to our cryptarithm (i.e., enumerating all the digits and testing equation (1)) has an inacceptable execution time. This solution can be improved by merging partial tests with the enumeration of values in a sophisticated way. However, the resulting code is less concise and more difficult to generalize than our concurrent implementation of injective finite mappings.

4 Conclusion

Functional logic programs, in addition to ordinary functional computations, provide both concurrency and logic variables. Concurrency supports a powerful and expressive programming style, but it complicates some tasks, in particular the computation of an injective finite mapping. We have presented the design and implementation of one such mapping for a functional logic language.

The design relies on a representation of index-value pairs where the values of the problem play the role of indices in the representation and the indices of the problem are initially unbound variables. During the computation, the variables of the representation are non-deterministically bound to the indices of the problem. This design ensures the injectivity of the mapping even when index-value pairs are computed concurrently and independently.

We have shown the implementation of our design in PAKCS, a popular compiler/interpreter of Curry. We have discussed a few implementations with different characteristics—in particular, linear and tree-based implementations when the set of the values of the problem is finite. We have compared our implementation of an injective finite mapping with more traditional implementations. We have found that the overhead of supporting the concurrent asynchronous computation of index-value pairs is an increase in computing time by a very small factor.

References

1. H. Aït-Kaci. An overview of LIFE. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.

3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
6. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
7. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
8. M. Hanus, S. Antoy, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2002.
9. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
10. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, (3):1–49, 1999.
11. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
12. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
13. S. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.
14. V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
15. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
16. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.