Multiworld Augmented Term Rewriting

Masami Takikawa Lawrence A. Crowl

Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 94-60-06

November 1994

Abstract

Augmented term rewriting (ATR) is a simple, uniform, and extensible computational model for constraint programming. Unfortunately, ATR cannot solve combinatorial constraint satisfaction problems (CCSPs). To enable solution of CCSPs, we introduce (don't know) nondeterminism into ATR via the *choice expression*, which identifies a set of possible values that may satisfy the constraints. The selection of a possible value from the expression represents one of many "possible worlds" in which the constraints may be satisfied. We show that our extended ATR, *multiworld augmented term rewriting* (MATR), is capable of expressing CCSPs concisely and readably via examples and via our experience with a significant application. We also show that an implementation of MATR can use the efficient constrain-and-generate technique for solving CCSPs, and describe our prototype implementation.

1 Introduction and Related Work

Augmented term rewriting (ATR) is a computational model for constraint programming developed by Leler [1988]. ATR extends term rewriting [Huet and Oppen, 1980; Dershowitz and Jouannaud, 1990] with constraints, bindable variables, and typed parameters and variables. In languages based on ATR, such as Bertrand [Leler, 1988] and Siri [Horn, 1992], simple term rewrite rules uniformly express how to (algebraically) simplify expressions, how to compose both functions and constraints (relations), and how to extend the computational domain with new data types and operations. This simplicity, uniformity, and extensibility are important advantages of ATR languages. The extensibility is particularly significant because most other constraint languages suffer from a lack of extensibility. For example, constraint logic programming (CLP) languages [Jaffar and Lassez, 1987; Jaffar and Maher, 1994] have fixed computational domains and extending their domains is difficult. In contrast, extending the computational domain of ATR languages is relatively easy, so ATR languages are better able to support large-scale program development.

Constraints and bindable variables enable ATR languages to express and solve slightly non-linear simultaneous equations over the real numbers. However, ATR cannot express or solve combinatorial constraint satisfaction problems (CCSPs). This limitation is significant because CCSPs are an important application area of constraint programming. Because of this limitation, ATR is regarded as a tool for implementing new constraint satisfaction mechanisms rather than as a constraint language itself [Van Hentenryck, 1989].

In this paper we show how to extend ATR's applicability to CCSPs while preserving its advantages. The result is a simple, uniform, and extensible constraint language for solving both numerical (continuous) and combinatorial constraint satisfaction problems.

In procedural and functional languages, programmers solve CCSPs by writing code to explicitly search the space of possible solutions. This places a high burden on the programmer. In existing constraint and logic languages, however, programmers specify a problem as a set of solution constraints, and the language system solves the problem by nondeterministic search. For example, two representative instances of constraint logic programming languages, Prolog [Bowen, 1981] and CHIP [Dincbas *et al.*, 1988; Van Hentenryck, 1989], both use a (don't know) nondeterministic choice of Horn clauses to solve CCSPs.

The introduction of nondeterminism to term rewriting (or functional programming) in general is not new. In fact, it has produced a group of languages called functional logic languages [DeGroot and Lindstrom, 1986; Hanus, 1994] where the major goal is to integrate functional and logic languages. For example, Fresh [Smolka, 1986] extends functional programming by introducing unification and nondeterminism.

Our approach to supporting CCSPs in ATR languages is to introduce nondeterminism via a new linguistic construct called *choice expressions*. Choice expressions describe multiple possible values in a solution of the constraints. Choice expressions are a generalization of mathematical expressions such as $\pm\sqrt{5}/3$, which is written as $\{+\sqrt{5}, -\sqrt{5}\}/3$. Choice expressions may also describe possible values among an arbitrary set, e.g. {red, white, blue, green}. Programmers can use any expression as a member of a choice expression, not just constants.

When interpreting a choice expression, the language system expands the expression to create multiple worlds, with each world substituting a different possible value for the expression, and

yielding a different configuration in the search space. We call our extended ATR model the *multi-world augmented term rewriting* (MATR) model. Section 2 describes the model and our prototype language in more detail. We describe our experience with a natural-language driven application for graphical design in section 3.

CSPs are often NP-hard, and hence solving them is likely to be time-consuming in the general case. To avoid excessive cost, MATR systems should pursue efficient execution, by efficient implementation of primitive operations, by employing efficient search algorithms, and by exploiting parallelism. We discuss these issues in section 4.

Our conclusions, in section 5, are that choice expressions provide a simple and effective mechanism for extending the applicability of ATR languages into CCSPs. The resulting MATR is both expressive and efficient, suggesting that all ATR systems should support choice expressions.

2 MATR and ConPi

In this section, we describe the primary mechanisms of term rewriting (TR), augmented term rewriting (ATR), and multiworld augmented term rewriting (MATR). Our description also provides an introduction to ConPi[1], a prototype constraint programming system based on MATR. ConPi is implemented in Lisp, so the syntax is Lisp-like and differs from that of other ATR languages, which use term notations. However, our approach to extending ATR can be applied to other ATR languages as well. Once the necessary primary mechanisms are described, we then use the N-queens problem to show how MATR can solve combinatorial constraint satisfaction problems.

2.1 TR: Rewrite Rules, Patterns, and Parameters

Term rewriting programs consist of a set of rewrite rules for transforming appropriate (sub)expressions into other (sub)expressions. Rules consist of a pattern with which to match source expressions and a replacement expression. For example, the ConPi rule

```
(=> (square ^X) (* ^X ^X))
```

specifies a pattern, `(square ^X)`, and a replacement, `(* ^X ^X)`. The pattern contains a *parameter*, `^X`, which may match any expression and will appear in the replacement. For example, the above rule applied to the expression `(square 2)` yields `(* 2 2)`. Rules may match subexpressions as well as entire expressions, so the rule applied to `(* (square 2) 5)` yields `(* (* 2 2) 5)`. As in standard term rewriting, the pattern in ConPi may not use the same parameter name twice, e.g. `(=> (+ ^X ^X) (* 2 ^X))` is not allowed.

2.2 ATR: Constraints, Bindable Variables, and Typed Variables

Augmented term rewriting, introduced by Leler [1988], extends term rewriting by introducing constraints, bindable variables, and typed parameters and variables into the rewrite rules.

[1] ConPi reads "con–π" whose sound resembles a Japanese word for a toast in sentences such as "Here's a toast for you, Achilles." and "We drank a toast for the glorious future of Mr. Tortoise."

Constraints are expressions that are required to reduce to `true`. Zero or more of them may appear between the pattern and its replacement. For example, the rule

```
(=> (/ 0 ^X) (<> ^X 0) 0)
```

has the constraint `(<> ^X 0)` and states that the division of 0 by anything rewrites to 0 if the divisor is not 0. Otherwise, this rule fails and this failure causes the ATR system stop. Note that this failure is in contrast to conditional term rewriting [Dershowitz and Jouannaud, 1990] in which conditions are applied before application and hence the rule would not apply.

Variables are scope limited to a single rule. In ConPi, variables must be declared using a `var` expression, which is a special form of constraint that always reduces to `true`. For example,

```
(var ^pi number 3.1415)
```

declares a variable whose name is `^pi` and whose type is `number`. (As with parameters, variables begin with `^`.) The third argument is optional and initializes the variable. Variables can be assigned at most once. This single assignment semantics preserves the declarativeness and referential transparency of programs.

When an expression gets assigned to a variable, it replaces all occurrences of the variable in the subject expression. This mechanism allows slightly-nonlinear, simultaneous equations to be solved. For example, with the rule in Figure 1, the expression `(aSampleEqu)` rewrites to `4` as shown in Figure 2.

```
(=> (aSampleEqu)                          — pattern
    (var ^X number) (var ^Y number)       — constraints (variables)
    (= (+ ^X ^Y) 5) (= (- ^X ^Y) -3)      — constraints (algebraic)
    (* ^X ^Y))                            — replacement
```

Figure 1: Example Simultaneous Equation

```
(aSampleEqu)
⇓ — substitute replacement for pattern, remembering constraints
(* ^X ^Y) subject to (= (+ ^X ^Y) 5) (= (- ^X ^Y) -3)
⇓ — (= (+ ^X ^Y) 5) ⇒ (= ^X (- 5 ^Y)) ⇒ ^X ↦ (- 5 ^Y)
(* (- 5 ^Y) ^Y) subject to (= (- (- 5 ^Y) ^Y) -3)
⇓ — (= (- (- 5 ^Y) ^Y) -3) ⇒ (= (* -2 ^Y) -8) ⇒ ^Y ↦ 4
(* (- 5 4) 4)
⇓ — algebraic simplification
4
```

Figure 2: ATR Rewrite Steps

In ATR, parameters can have a type, which restricts the range of matchable expressions. For example, suppose we have the following rules which define the factorial:

```
(=> (factorial 0) 1)
(=> (factorial ^N'positive) (* ^N (factorial (- ^N 1))))
```

Here, the typed parameter `^N'positive` can only match a positive constant number. For example, the subject expression `(factorial 0)` matches the first rule, `(factorial 3)` the second one, while `(factorial true)` does not match any of them. The presence of variables in the subject expression creates an interesting case; because ATR uses one-way matching (not two-way unification), the subject expression `(factorial ^X)` does not match any of the above rules. This expression will only reduce after `^X` becomes a constant.

The one-way matching used in ATR defines the direction of data flow and realizes the synchronization needed for constraint propagation, in contrast to unification used by logic and functional logic languages where an additional mechanism (e.g., `freeze` in Prolog-II [Colmerauer, 1984]) is needed for such data driven computation.

2.3 MATR: Choice Expressions

Multiworld augmented term rewriting extends augmented term rewriting with *choice expressions*, which introduce nondeterminism to ATR. In ConPi, choice expressions are written as `(alt` x_1 x_2 ... x_n`)` where x_1 to x_n identify a set of possible expressions that may satisfy the constraints. Programmers can use any expression as a member of a choice expression.

When interpreting a choice expression, the language system copies the subject expression (of which the choice expression is a part) to create multiple worlds, with each world substituting a different possible value for the choice expression. Each world represents a different configuration in the search space. For instance, the expression `(= ^X (+ (alt 1 2 3) 4))` creates three worlds, `(= ^X (+ 1 4))`, `(= ^X (+ 2 4))`, and `(= ^X (+ 3 4))`. All worlds have an independent name space, which entables multiple assignments to the variable `^X`. Each worlds will be further reduced independently of other worlds.

Unlike ATR, the failure of a constraint does not make the system stop, but it makes a world in which the failure occurs disappear. Only successful worlds survive to become answers. When all worlds disappear, there is no solution to the problem.

One use of choice expressions is to handle multiple solutions in numerical constraint satisfaction problems. For example, the rule

```
(=> (= (square ^X) ^N'positive) (= ^X (alt (sqrt ^N) (- (sqrt ^N)))))
```

states that if the square of `^X` is a positive constant `^N` then `^X` is either $\sqrt{\text{\^{}N}}$ or $-\sqrt{\text{\^{}N}}$. This rule can be used to transform a non-linear equation into a linear (thus easily solvable) equation.

Note that the order in which constraints are written is not important in MATR; changing that order does not affect the meaning of the program or the efficiency of the execution. This property of MATR helps the programmers read their programs declaratively and sharply contrasts with other constraint languages such as CS-Prolog [Kawamura *et al.*, 1987] and CHIP [Dincbas *et al.*, 1988; Van Hentenryck, 1989]. In these languages, the programmer uses the predicate `indomain` to explicitly assign values to domain variables. The order of `indomain` predicates affects the meaning and efficiency of programs greatly.

2.4 4-Queens Problem

The important use of choice expressions is for expressing combinatorial constraint satisfaction problems (CCSPs). We illustrate how to solve CCSPs with the 4-queens problem, which consists of placing 4 queens on a (4×4) chess board so that no two queens attack each other. (We generalize to the N-Queens problem later.)

Figure 3 shows a ConPi program to solve the 4-queens problem. The first rule, `fourQueens`, solves the problem. The second rule, `noattack`, is a subroutine that prevents two queens from attacking each other. Each queen is presumed to be on a separate row. The first four constraints in `fourQueens` define the domain of variables, each of which specifies the column for a given queen. The six `noattack` expressions constrain each pair of queens not to attack each other. Within `noattack`, `^X` and `^Y` are queen column numbers and `^N` is the difference in row numbers. Finally, the list of queen columns is the replacement for `fourQueens`. Given this program, ConPi finds an assignment to variables (`^Q1` to `^Q4`) such that the assignment satisfies all the constraints. An example assignment is `^Q1` $\mapsto 2$, `^Q2` $\mapsto 4$, `^Q3` $\mapsto 1$, and `^Q4` $\mapsto 3$.

```
(=> fourQueens
    (var ^Q1 number (alt 1 2 3 4)) (var ^Q2 number (alt 1 2 3 4))
    (var ^Q3 number (alt 1 2 3 4)) (var ^Q4 number (alt 1 2 3 4))
    (noattack ^Q1 ^Q2 1) (noattack ^Q1 ^Q3 2) (noattack ^Q1 ^Q4 3)
    (noattack ^Q2 ^Q3 1) (noattack ^Q2 ^Q4 2)
    (noattack ^Q3 ^Q4 1)
    (list ^Q1 ^Q2 ^Q3 ^Q4))
(=> (noattack ^X ^Y ^N) (<> ^X ^Y) (<> ^X (- ^Y ^N)) (<> ^X (+ ^Y ^N)))
```

Figure 3: The 4-queens Problem in ConPi

This program embodies a basic three-part form of CCSP programs, the definition of variables and their domains, the constraints that limit the allowable values of variables, and the expression composing the answer.

2.5 ConPi: Classes and Higher-Order Constraints

In addition to the simple rewrite rules, ConPi provides programming constructs useful for expressing problems concisely. Those constructs are expanded by ConPi into the simple rewrite rules and choice expressions described so far.

ConPi provides object-oriented data abstraction via classes. Classes may have class parameters, a parent class, typed instance variables, class constraints, and methods. Class parameters provide the data for instance variables upon creating an object. ConPi provides single inheritance of instance variables, class constraints, and methods. Instance variables are single-assignment and preserve referential transparency. Class constraints must be satisfied by all instances of the class, and are often the primary source of computation for an object. Methods are functions or constraints on the instance variables, the method's arguments, and the pseudo variable `^self`, which refers to the receiver of the method. (An example of a class appears later in Figure 4.)

In addition to the data abstraction facility provided by classes, ConPi has facilities for higher-order constraint constructs such as `if`, `and`, and `for-all`. The `for-all` provides iteration over a numeric range and has the form: (`for-all` ^*index lower upper constraint*) This expression means that for all integers from *lower* to *upper* inclusive, the *constraint* holds. The variable ^*index* is the iteration index. For example, the expression `(for-all ^i -1 1 (> ^i 0))` is a concise notation for `(and (> -1 0) (> 0 0) (> 1 0))` which obviously reduces to `false`.

2.6 N-Queens Problem

We now generalize the example to the N-queens problem to show how concisely a general CCSP can be solved in ConPi using a class and higher-order constraints.

Figure 4 shows an N-queens program. First, `define-class` defines a new data type `nqueens` which is parameterized by the number of queens `^n`. Instances of this class correspond to various sizes of N-queens problems. The only instance variable in `nqueens` is `^Q`, which contains the column number for each queen (on separate rows). The next expression defines a class constraint, which must be satisfied by all instances of the class `nqueens`, in this case that no queen may attack another. That is, the class constraint restricts the value of the instance variable to a solution to the problem. In the class constraint, `for-all` is used for iteration and `field` is used for retrieving a member from the array. Finally, this class has a method called `answer` which simply returns the solution as an array.

```
(define-class                          — defining a class
  (nqueens ^n'constant)                — class name and typed parameter
  ()                                   — parent class (none)
  ( ^Q'(array 1 ^n (range 1 ^n))       — instance variable Q, an array 1..n of 1..n
    (for-all ^i 2 ^n                   — class constraint: for each queen except the first
      (for-all ^j 1 (- ^i 1)           — for each queen previous to her
        (noattack                      — ensure not attacking
          (field ^Q ^i)                — column of first queen
          (field ^Q ^j)                — column of second queen
          (- ^i ^j))))                 — difference in rows
  )
  (answer                              — method definition for answer
    ()                                 — method parameters (none)
    ^Q)                                — result of method, just the value of variable Q
)
```

Figure 4: The N-Queens Problem in ConPi

Using this class, the following program solves a 12-Queens problem:

```
(=> (12Queens) (var ^12Queens (nqueens 12)) (answer ^12Queens))
```

Our solution to the N-Queens problems depends on two factors, the extensibility of ATR and MATR systems to define higher-order constructs, and most importantly, the choice expressions of MATR to permit search among possible answers.

3 Application

In this section, we report on our experience using ConPi in a non-trivial application. The application is a prototype markup system for graphical designers, called PiMark, developed at BSJ Technologies, Inc., California. In addition to a graphical interface using a pen-device, this system provides a natural language interface. (Note that though our examples are in English, the application interprets Japanese sentences.) The designer can specify requirements on the page model using traditional markup sentences. Given a set of markup sentences, PiMark must first represent the meanings of the sentences, and then infer a page model that satisfies those meanings.

PiMark represents the meaning of sentences with ConPi programs. This approach is possible because markup sentences can be naturally regarded as constraints on the properties and relationships for objects in the page model. For example, "this title uses 24-point font" specifies a property (the size of the font) of an object (this title); "this picture needs to fit into that rectangle" states a relationship (fit into) between two objects (this picture and that rectangle). Our representation of sentences as constraints has two desirable properties. First, the translation of sentences to constraint programs is relatively straightforward. Second, we do not need to build a separate inference engine; the meaning of sentences represented in ConPi can now be directly interpreted by the ConPi system to produce the specified page model.

Any natural language processing system must share a vocabulary with the system's users. We map the vocabulary of sentences into the vocabulary of constraint programs: common nouns, such as rectangle, map to data types, as defined by classes; proper nouns, such as rectangle (1), map to class instances; attributes of an object, such as the width of a rectangle, map to method functions; and relationships, such as wider, map to constraints. This simple mapping from sentence elements to program elements makes the translation from sentence to program easy. Furthermore, we rely on the extensibility of MATR (inherited from ATR) in the computational domain to extend the vocabulary of the application domain.

The basic entities in the application domain are *frames*, e.g. text frames and picture frames. PiMark represents these frames as objects, and represents their properties as method functions that return the desired value of the property. Since the application entities are all frames, their corresponding objects share many common properties. All types of frames, including text frames, inherit their common interface from a single base class. This common interface simplifies the parser by delegating all interpretation of properties to ConPi's dynamic method dispatch. The translation from natural language to constraint program does not interpret properties; the constraint program does so.

Noun phrases such as "title (1)", "black rectangle", and "the title on the picture (3)" refer to specific frame instances. Note that unlike identifiers or proper nouns, these references are restrictive and must be interpreted within a context. In PiMark, that context is the page model, which is represented as a list of frame instances. Identifying a referenced frame requires searching the page model for an instance that meets the reference conditions. The search for referents is done by the higher-order construct `(find-all ^`*Frame* `^Page` *condition constraint...*) which finds all instance frames `^`*Frame* in the page model `^Page` such that *condition* holds. The system is then in a position to apply any *constraints* to the referents. Noun phrases are translated into a *condition*, and verb phrases into *constraints*.

Figures 5 and 6 show two examples, each a sentence and its ConPi program. These examples

illustrate how classes, methods, and constraints can be combined to capture the great degree of freedom found in natural language sentences. As an example of specifying properties for referents, consider the sentence of Figure 5. This sentence specifies properties (font color and font size) of the title.

"The font color of the title is either white or blue and its size is either 24 points or 36 points."

```
(find-all ^V0 ^Page
 (and (= (type ^V0) "text") (= (style ^V0) "title"))  — the title
 (= (fontColor ^V0)                                   — its font color
   (alt (white) (blue)))                              — is white or blue
 (= (fontSize ^V0)                                    — its font size
   (alt (pt 24) (pt 36))))                            — is 24pt or 36pt
```

Figure 5: Example Sentence Asserting Properties of a Referent

Figure 6 exemplifies a relation. Two nested `find-all`s are used to specify two objects. The relationship is then represented as a constraint within the innermost `find-all`.

"The width of text frame (1) is 3 or 4 cm larger than the height of the black rectangle."

```
(find-all ^V0 ^Page
    (and (= (type ^V0) "text"))              — the text frame
         (= (name ^V0) "(1)"))               — named "(1)"
  (find-all ^V1 ^Page
      (and (= (color ^V1) (black))           — the black
           (= (type ^V1) "rectangle"))       — rectangle
    (= (width ^V0)                           — the width of the text
       (+ (cm (alt 3 4)) (height ^V1)))))    — = height of rectangle + 3 or 4 cm
```

Figure 6: Example Sentence Asserting a Relationship

PiMark takes advantage of the constraints and extensibility of ATR and MATR systems to simplify the implementation of its natural-language interface. More importantly, choice expressions enable PiMark to provide its users with the capability to specify alternative solutions to a graphical layout, which avoids the need to manually explore alternative designs.

4 Implementation Issues

This section describes implementation issues. We concentrate on three areas: efficiency of primitive operations, efficiency of search, and parallelism.

4.1 Efficiency of Primitive Operations

The primary primitive operation is term rewriting. For fast term rewriting, our system translates ConPi programs into Lisp programs through the following five steps:

1. Translate abstraction facilities such as `define-class` into equivalent rewrite rules.

2. Convert patterns into a table-driven automaton for fast pattern matching.

3. Simplify the body of each rule as much as possible by applying other applicable rewrite rules. For example, given the definitions:

   (a) `(=> (pi) 3.1416)`

   (b) `(=> (square ^X'constant) (* ^X ^X)`

   (c) `(=> (area ^C'circle) (* (square (radius ^C)) (pi)))`

   the simplified `area` rule is

   ```
   (=> (area ^C'circle) (* (square (radius ^C)) 3.1416))
   ```

   Note that `(square (radius ^C))` is not further simplified because `(radius ^C)` is not known *a priori* to be a constant.

4. Convert the body of each rule into a Lisp function that creates a copy of the body. (The pattern matching is done by the automata.) For example, the above examples become the following Lisp functions after simplification and conversion:

   (a) `(lambda () 3.1416)`

   (b) `(lambda (X) (* X X))`

   (c) `(lambda (C) (list '* (list 'square (list 'radius C)) 3.1416))`

   Note that most rules are converted into functions such as the third one above that creates a copy of the body in which parameters are substituted by the actual arguments. The second function above is an exception in which the converter knows that the actual argument `X` is always a constant number, so it outputs a function which directly executes Lisp's multiplication.

5. Compile each Lisp function with the Lisp compiler to yield more efficient code.

Because the technology of pattern matching and rule rewrites in ConPi is the same as that of standard ATR systems, those parts of an ConPi (MATR) program that do not use choice expressions can as efficient as standard ATR systems.

4.2 Efficiency of Search

The overall MATR reduction strategy is to reduce all non-choice expressions, make a choice, and repeat.

Reduction is done using the standard (efficient) ATR methods while treating choice expressions as unevaluable terms. Through this reduction, all evaluable constraints are simplified and verified until no more reduction can be done.

During reduction, some choice expressions can be simplified. For example, if a choice expression contains only one value, it can be reduced to the value without the need for making a choice.

Another simplification removes a value from the domain of a variable when it is obvious that the value cannot satisfy a constraint. For example, inequality constraints (<>) in the N-queens problem in Figure 4 can reduce the domain of queens' column variables.

When the subject expression becomes irreducible, the system expands a choice expression to create multiple worlds, each of which corresponds to a value within the choice expression and represents a different configuration in a search space. This expansion creates opportunities for further rewriting, so the standard ATR again reduces each world independently.

Because all evaluable constraints are tested to prune the search space before making a choice, this strategy subsumes the *constrain-and-generate* approach which is an efficient computational scheme for combinatorial constraint satisfaction problems [Kumar, 1992], and has been used by some constraint logic programming languages, e.g. CS-Prolog [Kawamura *et al.*, 1987] and CHIP [Dincbas *et al.*, 1988; Van Hentenryck, 1989].

There are many variations of our algorithm depending on:

- which expression is chosen from the list of multiple expressions,
- which choice expression is chosen to expand to multiple worlds,
- how many branches are created after expanding choice expressions,
- whether ATR reduction is done sequentially or in parallel,
- whether multiple words are reduced sequentially or in parallel,
- and so on.

For example, if most the recently created expression is chosen from the multiple expression list, the search process becomes depth-first. These issues greatly affect the performance of the algorithm. Further research is necessary to determine the best strategy, which may depend on applications and machines.

The current implementation of ConPi supports both sequential and parallel depth-first search. The sequential version tries to minimize the space by creating only one branch at a time, while the parallel version tries to minimize the time by creating more branches (tasks) at a time.

4.3 Parallel Evaluation

Each world has an independent name space, so those worlds can be rewritten simultaneously without any conflicts. This is a source of coarse-grained parallelism; multiple processors can work as standard augmented term rewriters on different independent expressions. This coarse-grained parallelism can lead to an efficient parallel implementation on both shared-memory multiprocessors and distributed-memory multicomputers.

This coarse-grained parallelism corresponds to OR-parallelism [Tick, 1991] in logic programming languages, but the grain size of OR-parallelism is usually finer than MATR's because in logic programming languages most predicates are non-deterministic, and because unification is basically simpler and does less work than ATR.

Likewise, there is a source of parallelism that corresponds to AND-parallelism [Tick, 1991] in logic programming languages — parallelism derived from multiple constraints within a world. This parallelism comes from the fact that ATR has no prescribed order of evaluation, so expressions can be rewritten eagerly, lazily, or concurrently. This parallelism is fine-grained.

Currently, ConPi exploits the coarse-grained parallelism. Our experimental implementation runs on the Sequent Symmetry, a shared-memory multiprocessor. Our benchmark results show that good speed-up (14.63 when using 20 processors for all solutions to the 12-queens problem) can be achieved even at the current early experimental stage.

5 Conclusion

In this paper, we demonstrated that choice expressions provide a simple and effective mechanism for extending the applicability of Augmented Term Rewriting (ATR) into combinatorial constraint satisfaction problems. The resulting programming model, Multiworld Augmented Term Rewriting (MATR), is simple, extensible, expressive, and efficient.

MATR inherits simplicity and extensibility from ATR systems. The simplicity of MATR was illustrated by examples. The extensibility of MATR was demonstrated by our natural language application experience where high-level vocabularies, which correspond to natural language words, can be defined so that the task of the parser is simplified. The use of data abstraction and rewrite rules facilitates the easy extension of computational domain.

MATR is expressive because it allows the programmer to express and solve both numerical and combinatorial constraint satisfaction problems concisely and readably. Data and higher-order constraint abstraction provided by our MATR language ConPi further facilitates concise programs.

Finally, we showed that the execution of MATR can be efficient because (1) its primitive operations (pattern matching and determinate rewriting) can be efficiently implemented; (2) its search mechanism subsumes the efficient constrain-and-generate technique; and (3) it has many opportunities for parallelization.

Given the large benefits and relatively small cost, we believe all ATR systems should adopt choice expressions.

Acknowledgements

We thank Margaret Burnett and Timothy Budd for their comments on this paper.

References

[Bowen, 1981] D. L. Bowen, "DECsystem-10 Prolog User's Manual," Occasional Paper 27, University of Edinburgh, Department of Artificial Intelligence, December 1981.

[Colmerauer, 1984] A. Colmerauer, "Equations and Inequations on Finite and Infinite Trees," In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, 1984.

[DeGroot and Lindstrom, 1986] D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1986.

[Dershowitz and Jouannaud, 1990] N. Dershowitz and J.P. Jouannaud, "Rewrite Systems," In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier Science Publishers B.V., 1990.

[Dincbas *et al.*, 1988] P. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, "The Constraint Logic Programming Language CHIP," In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 693–702, November 1988.

[Hanus, 1994] M. Hanus, "The Integration of Functions Into Logic Programming: From Theory To Practice," *The Journal of Logic Programming*, 19,20:583–628, 1994.

[Horn, 1992] B. Horn, "Properties of User Interface Systems and The Siri Programming Language," In B. A. Myers, editor, *Languages for developing user interfaces*, chapter 13, pages 211–236. Jones and Bartlett Publishers International, London, England, 1992.

[Huet and Oppen, 1980] G. Huet and D. C. Oppen, "Equations and Rewrite Rules: A Survey," In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[Jaffar and Lassez, 1987] J. Jaffar and J.-L. Lassez, "From Unification to Constraints," In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the Sixth Logic Programming Conference*, Lecture Notes in Computer Science, 315, pages 1–18. Springer-Verlag, 1987.

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

[Kawamura *et al.*, 1987] T. Kawamura, H. Ohwada, and F. Mizoguchi, "CS-Prolog: A Generalized Unification Based Constraint Solver," In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the Sixth Logic Programming Conference*, Lecture Notes in Computer Science, 315, pages 19–39. Springer-Verlag, 1987.

[Kumar, 1992] V. Kumar, "Algorithms for Constraint Satisfaction Problems: A Survey," *AI Magazine*, 13(1):32–44, September 1992.

[Leler, 1988] W. Leler, *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley Publishing Company, Reading, MA, 1988.

[Smolka, 1986] G. Smolka, "Fresh: A Higher-order Language With Unification And Multiple Results," In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pages 469–524. Prentice-Hall Inc., Englewood Cliffs, NJ, 1986.

[Tick, 1991] E. Tick, *Parallel Logic Programming*, MIT Press, Cambridge, MA, 1991.

[Van Hentenryck, 1989] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.