{SETS} — A LIGHTWEIGHT CONSTRAINT PROGRAMMING LANGUAGE BASED ON ROBDDS

Haim Cohen
Columbia University
City of New York, NY, The United States
hc2311@columbia.edu

Stephen A. Edwards†
Columbia University
City of New York, NY, The United States
sedwards@cs.columbia.edu

ABSTRACT

Constraint programming is a step toward ideal programming: you merely define the problem domain and the constraints the solution must meet and let the computer do the rest. Many constraint programming languages have been developed; the majority of them employ iterative constraint propagation over the problem variables. While such an approach solves many problems and can handle very rich data types, it is often too inefficient to be practical.

To address this problem, we developed a constraint programming language called {sets} that uses reduced ordered binary decision diagrams (ROBDDs) as the solution engine. Providing a minimal syntax, the language can be used to solve many finite problems that fit the constraint programming paradigm. The minimal syntax and simple semantics of the language enable the user to create libraries customized for a specific problem domain. {sets} is particularly useful in problems where an efficient search algorithm yet know to exist or can not be developed due to time constraints. As long as the solution domain is finite and discrete, {sets} should be able to describe the problem and search for a solution.

We describe the {sets} language through a series of examples, show how it is compiled into C++ code that uses a public-domain ROBDD library, and compare the performance of this language with other constraint languages.

KEYWORDS

Constraint Programming, Constraint Satisfaction Problem, Programming Languages, Binary Decision Diagrams, BDDs

1. INTRODUCTION

Constraint programming [2] is an approach to programming in which the programmer states a problem by specifying properties of a solution and lets the computer find them. Constraint programming is especially practical for problems where efficient algorithms are not known.

Constraint programming addresses the solution of the constraint satisfaction problem (CSP). A CSP consists of a set of variables, each of which take a value from a domain, and constraints that specify relationships among them. A solution is an assignment of values to variables that satisfies the constraints.

Many interesting problems can be expressed as CSPs, such as a variant of the *SUBSET-SUM* problem. In this problem, we are given a set of integers S and a target number t. A solution to the problem is a subset of S that sums to exactly t. This problem is NP-complete [12]; no efficient algorithm is known.

In this paper, we describe the {sets} language: an efficient constraint programming language for finite-domain constraint problems that uses reduced ordered binary decision diagrams [4][1] (ROBDDs) as a solution engine. We explain the language with examples, show how we translate it into C++ that calls a ROBDD library, and describe some experiments that illustrate the efficiency of our approach.

† Edwards and his group are supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

2. {SETS} THROUGH EXAMPLES

2.1 The SUBSET-SUM problem

Consider solving an instance of the SUBSET-SUM problem using {sets}. Specifically, consider trying to pick a subset of numbers from the set {1, 3, 7, 15, 21, 22, 26, 27, 40, 41} that sums to 187. Using {sets}, we first need to define all the potential solutions to the problem, which in this case is a 10-dimensional tuple whose entries are each either zero or the corresponding value from the set of integers. The set of valid solutions are simply those tuples whose elements add up to 187. In {sets}, this problem can be coded as follows.

```
dim e0 = [ 0, 1  ]; dim e1 = [ 0, 3  ]; dim e2 = [ 0, 7  ]; dim e3 = [ 0, 15 ]; dim e4 = [ 0, 21 ];
dim e5 = [ 0, 22 ]; dim e6 = [ 0, 26 ]; dim e7 = [ 0, 27 ]; dim e8 = [ 0, 40 ]; dim e9 = [ 0, 41 ];
print( e0 + e1 + e2 + e3 + e4 + e5 + e6 + e7 + e8 + e9 == 187 );
```

Compiling and running this program prints `<0:0, 1:3, 2:7, 3:0, 4:21, 5:22, 6:26, 7:27, 8:40, 9:41>`. Each entry in this tuple is of the form *index*:*value*. A value of 0 means the corresponding element is not in the solution subset. This result corresponds to the solution 3 + 7 + 21 + 22 + 26 + 27 + 40 + 41 = 187.

Here is another, more efficient way to write this, which uses a binary encoding for set membership.

```
dim e0 = [ 0, 1 ]; dim e1 = e0; dim e2 = e0; dim e3 = e0; dim e4 = e0;
dim e5 = e0;       dim e6 = e0; dim e7 = e0; dim e8 = e0; dim e9 = e0;
print( e0*1 + e1*3 + e2*7 + e3*15 + e4*21 + e5*22 + e6*26 + e7*27 + e8*40 + e9*41 );
```

This is more efficient because ROBDDs are more compact when representing smaller number ranges. Internally, ROBDDs manipulate Boolean variables; integers are represented as groups of such variables. Thus, the second example only required ten Boolean variables; far fewer than the first.

We built the semantics of our language around mathematical sets. A program begins by defining of the set of potential solutions then states constraints the solution must satisfy. Running the program amounts to applying these constraints to the universal set and reporting the members of the result. An over-constrained problem produces the empty set. In {sets}, the universal set is an *n*-dimensional vector.

2.2 SEND+MORE=MONEY

Although constraint programming languages are fundamentally declarative, we added a few classical imperative constructs to {sets} to make it easier to code certain problems. The following example illustrates how loops and functions make it easier to code the classic alphametic "SEND+MORE=MONEY" puzzle [5][7]. In this puzzle, each of the characters S, E, N, D, M, O, R, Y need to be assigned to a digit from 0 and 9 such that the equation "SEND+MORE=MONEY" is satisfied if each digit string is treated as a decimal number. Each digit may be assigned to at most one letter and each number may not start with a 0.

This prints `<0:9, 1:5, 2:6, 3:7, 4:1, 5:0, 6:8, 7:2>`, corresponding to 9567 + 1085 = 10652. This shows how user-defined functions and *foreach* can provide the *all_different* constraint and *range* function.

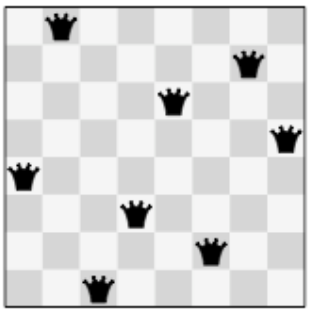
```
function set all_different( dim[] vars ) {
    set rslt = all; int ind = 0;
    foreach v0 vars[0 .. |vars|-2] {
        foreach v1 vars[ ind+1 .. |vars|-1 ] {
          rslt = rslt & v0 != v1;
        }
        ind = ind + 1;
    }
    return rslt;
}
function int[] range( int from, int to ) {
  return from >= to ? [from] :
                 [from] + range( from+1, to );
}
int[] digits = range(0,9);
dim S = digits; dim E = digits; dim N = digits;
dim D = digits; dim M = digits; dim O = digits;
dim R = digits; dim Y = digits;
set equation =
          S*1000 + E*100 + N*10 + D +
          M*1000 + O*100 + R*10 + E  ==
M*10000 + O*1000 + N*100 + E*10 + Y;
set no_leading_zeros = S != 0 & M != 0;
print( no_leading_zeros &
  all_different([S,E,N,D,M,O,R,Y]) & equation );
```

2.3 The Eight Queens Puzzle

The objective in the eight queens puzzle [6][7] is to place eight queens on a chess board such that none of them is able to capture any other, i.e., each column, row, and diagonal may have at most one queen. This well-studied problem, proposed by Max Bezzel in 1848, is easily posed and solved by {sets}. In the following program, we have encoded each solution as a 8-tuple, where each location in the vector represents

the row for the queen in that column. Such an encoding guarantees each column has exactly one queen. We first define the solution space: eight variables that may each take on values from 0 to 7. Here, we use the *range* function from SEND+MORE=MONEY and create two sets: *different_row* contains all solutions in which each queen is in a different row (we use the *all_different* function we defined earlier), and *not_in_diag* are those solutions with pairs of queens on different rows. We use two *foreach* loops to consider each pair.

To produce the answer, we intersect the two sets and print the result: all ninety-two solutions, including `<0:3, 1:7, 2:0, 3:2, 4:5, 5:1, 6:6, 7:4>`.



```
dim q0 = range(0,7); dim q1 = q0; dim q2 = q0; dim q3 = q0;
dim q4 = q0;         dim q5 = q0; dim q6 = q0; dim q7 = q0;
dim[] queens = [q0, q1, q2, q3, q4, q5, q6, q7];
set different_row = all_different( queens );
set not_in_diag = all;
foreach i range(0,|queens|-2) {
  foreach j range(i+1, |queens|-1) {
    not_in_diag = not_in_diag &
      j-i != (queens[j] - queens[i]) & j-i != (queens[i] - queens[j]);
    }
}
print( different_row & not_in_diag );
```

3. IMPLEMENTING {SETS}

Our {sets} compiler generates C++ code that calls the open source BuDDy ROBDD library[10]. Running the generated program, which we also link with a {sets} runtime library, constructs ROBDDs that represent the sets defined in the {sets} program and usually pretty-prints their contents to produce the result.

ROBDD packages manipulate finite Boolean functions, a very flexible representation. A characteristic function can represent the members of a set: for each possible member, the characteristic function indicates whether the set includes it. Set intersection and union on a characteristic function are Boolean AND and OR.

Boolean functions may also be used to represent arithmetic. The functions resemble those used in computer hardware, e.g., addition is performed with an adder function that treats bit vectors as binary.

An ROBDD is a directed acyclic graph that canonically represents a Boolean function of a certain number of variables. Since it is canonical, checking whether an ROBDD represents a constant 0 or constant 1 function (equivalent to the empty or universal set) is a constant-time operation. Furthermore, logical operations such as the logical AND of two functions, can be done in polynomial time. But there is no free lunch: checking for 0 is cheap but building an ROBDD from a sum-of-products can be exponentially costly.

Multiplication is a costly function for ROBDDs. In particular, building a Boolean function that indicates whether the product of two numbers is a third is always exponentially costly. {sets} permits such product functions to be built, but doing do so for large numbers will certainly exhaust memory. However, multiplication by a constant, such as we did in the SEND+MORE=MONEY problem, is polynomial-cost.

The cost of ROBDD operations depends on the number of variables and the complexity of the function. This is why the second implementation of SUBSET-SUM is more efficient: it uses fewer Boolean variables.

To illustrate ROBDDs, consider solving X + Y = 6 over the domain [0,7]. This is easily expressed:

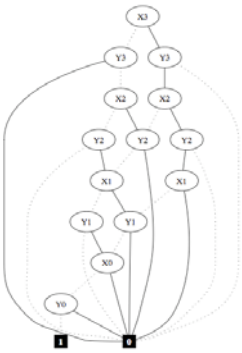
```
dim X = [0,1,2,3,4,5,6,7]; dim Y = X; print( X + Y == 6 );
```

As expected, this gives all seven solutions to this equation in the domain:

```
<0:0, 1:6> <0:4, 1:2> <0:2, 1:4><0:6, 1:0> <0:1, 1:5> <0:5, 1:1> <0:3, 1:3>
```

The variables X and Y are encoded by the Boolean variables X3,..., X0, Y3,...,Y0 (two sign bits are unnecessary). A simple heuristic orders these variables as X3,Y3, X2, Y2, X1, Y1, X0, Y0. In the figure on the right, every path from the root to the square "1" represents a solution to this equation. One such path follows dotted arcs (representing zeros) except at the leftmost X2 and X1 nodes. This corresponds to the assignment X3 = 0, Y3 = 0, X2 = 1, Y2 = 0, X1 = 1, Y1 = 0, X0 = 0, Y0 = 0, a binary representation of the solution X=6, Y=0.

The {sets} compiler generates C++ code that calls the BuDDy ROBDD library. We use a syntax-directed translation that also calls our runtime library.



ROBDD of X+Y=6

For example, the *all_different* function becomes.

```
bdd all_different( std::vector< Dim > vars ) {
    bdd rslt = RtEnv::instance().all();
    int ind = 0;
    { vector<Dim> __iterated_6808485(range(vars, 0, operator_size(vars) - 2));
        for( unsigned __i_6808485 = 0 ;  __i_6808485 < __iterated_6808485.size() ; ++__i_6808485 ) {
           Dim v0 = __iterated_6808485[__i_6808485];
           { vector<Dim> __iterated_9596560(range(vars, ind + 1, (operator_size(vars) - 1 )));
               for(unsigned __i_9596560 = 0 ; __i_9596560 < __iterated_9596560.size() ; ++__i_9596560) {
                   Dim v1 = __iterated_9596560[__i_9596560];
                   rslt = rslt & ( v0 != v1 );
           }   }
        ind = ind + 1;
    }   }
    return rslt;
}
```

Our compiler translates `print(no_leading_zeros & all_different([S,E,N,D,M,O,R,Y]) & equation);` into this code: `print(((no_leading_zeros & all_different((vector<Dim>(), S, E, N, D, M, O, R, Y ))) & equation ));`

Our runtime library simplified code generation, e.g., we overloaded the C++ comma operator for tuples:

```
template<typename T> vector<T> operator, (vector<T> vec, T elem) { vec.push_back( elem ); return vec; }
```

4. EXPERIMENTS

We ran three problems—SEND+MORE=MONEY, SUBSET-SUM and N-Queens—under {sets} and under the Oz [11] constraint language. For N-Queens, we ran the programs with N ranging from 8 to 15 and measured the increase on run time as a function of N. In the SUBSET-SUM problem, we used the same target value with different set sizes. For each problem except SEND+MORE=MONEY, we calculated the number of solutions rather than enumerate them to eliminate printing overhead. For SEND+MORE=MONEY and N-Queens, Oz performed better, but {sets} performed better on large instances of SUBSET-SUM. We ran these experiments on an Apple MacBook‡ with a 2 GHz Intel Core Duo with 1 Gb of memory.

Runtimes on Various Examples

Example	{sets}	Oz
SEND+MORE+MONEY	32s	0.2s
SUBSET-SUM 10	1s	<1s
SUBSET-SUM 15	1.1s	<1s
SUBSET-SUM 20	1.3s	<1s
SUBSET-SUM 25	2.4s	<1s
SUBSET-SUM 30	6.5s	>600s
SUBSET-SUM 35	7.1s	>600s
8 Queens	1.2s	0.2s
9 Queens	14.6s	0.4s
10 Queens	>640s	1.2s
11 Queens	-	1.6s
12 Queens	-	3.7s
13 Queens	-	15s
14 Queens	-	74s
15 Queens	-	>718s

5. RELATED WORK

Hawkins, Lagoon and Stuckey [9] also use ROBDDs to solve constraint problems, but did not propose a language. We believe, {sets} is the first constraint language to rely on ROBDDs. Other non-constraint languages have been built atop ROBDDs, such as Behrmann's IBEN [3]. Developed as an educational tool to teach ROBDDs, it is an interpreter that wraps the BuDDy library. However, instead of our constraint semantics, it provides ways to manipulate BDDs.

The Oz constraint programming language [11] attempts to solve problems much like {sets}, but we believe the {sets} syntax is superior: {sets} consists of a small set of features that can be combined. To the right is SEND+MORE=MONE written in Oz. Compare this with the {sets} version of the puzzle presented earlier.

```
declare Money
proc {Money Root}
    S E N D M O R Y
in
    Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
    Root ::: 0#9
    {FD.distinct Root}
    S \=: 0
    M \=: 0
          1000*S + 100*E + 10*N + D +
          1000*M + 100*O + 10*R + E  =:
10000*M + 1000*O + 100*N + 10*E + Y
    {FD.distribute ff Root}
end
{Browse {SearchAll Money}}
```

‡ MacBook and Mac OS are trademarks of Apple Computer, Inc., registered in the U.S. and other countries.

6. CONCLUSIONS

Our {sets} language provides a simple syntax for constraints programming, which can be extended by the user to create a powerful constraint programming library. Using ROBDDs to implement a constraint programming language, we demonstrated that this approach is feasible and can be easily adopted into other constraint programming implementations. No one size fits all; while ROBDDs gave superior performance for large instances of the SUBSET-SUM problem, they were inferior for other problems. The programming language we have developed is the first constraint programming language to use ROBDDs.

REFERENCES

[1] Henrik Andersen. An introduction to binary decision diagrams. http://www.itu.dk/people/hra/notes-index.html, 1998.

[2] Roman Barták. on-line guide to constraint programming. http://ktiml.mff.cuni.cz/bartak/constraints/, 1998.

[3] Gerd Behrmann. Iben: The interactive BDD environment. http://iben.sourceforge.net/, 2003. accessed October 2006.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, 1986.

[5] Henry E. Dudeney. Send more money. *Strand Magazine*, 68:97, 214, July 1924.

[6] C.F. Gauss. Briefwechsel zwischen C. F. Gauss und H. C. Schumacher. herausgeg, yon Peters, 6:105–122, 1865.

[7] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. http://www.csplib.org/, 1999.

[8] Peter Hawkins. Solving Set Constraint Satisfaction Problems using ROBDDs. PhD thesis, Department of Computer Science and Software Engineering , The University of Melbourne, Melbourne, Australia, November 2004.

[9] Peter J. Stuckey Peter Hawkins and Vitaly Lagoon. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[10] Jørn Lind-Nielsen. Buddy: A binary decision diagram package. http://sourceforge.net/projects/buddy, http://vlsicad.eecs.umedu/BK/Slots/cache/www.itu.dk/research/buddy/index.html,, 1996. accessed October 2006.

[11] C. Schulte. Finite domain constraint programming in oz: A tutorial, 1998.

[12] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1997.