The {sets} Language – White Paper

Haim Cohen – haimico@gmail.com

Background (non-detailed and simplistic)

In programming, algorithms are used to solve problems. Based on the nature of the problem an efficient algorithm (usually one that was already developed before hand by other computer scientists) is used to efficiently solve the problem.

This is not the case for a class of problems known as NP-Complete, where an efficient (non-exponential) is not known to exist.
As an exhaustive search is not acceptable with most of the problem's inputs, programmers usually employ different heuristics in order to come up with sub-optimal (yet acceptable) algorithm.

Theoretically, if we had an efficient implementation of sets, we would be able to efficiently solve virtually any problem.
I will explain the above statement.

We can think about specifying every problem as:

- The finite set of all its possible inputs, where each input is represented as integer vector of fixed length. (Inputs set)
- A set of vectors (subset of all possible inputs) which are valid solutions to the problem. (Solutions set)

For example, if we take an instance of the subset-set problem, where the set is **{ 3, 4, 100, 101 }** and the target is **104**, we can say that the following are all possible inputs of the problem (represented as vectors of integers) :

v0	*v1*	*v2*	*v3*	*Sum*	*A Solution?*
0	0	0	0	0	No
0	0	0	101	101	No
0	0	100	0	100	No
0	0	100	101	201	No
0	4	0	0	4	No
0	4	0	101	105	No
0	4	100	0	104	Yes
0	4	100	101	205	No
3	0	0	0	3	No
3	0	0	101	104	Yes
3	0	100	0	103	No
3	0	100	101	204	No
3	4	0	0	7	No
3	4	0	101	108	No
3	4	100	0	107	No

v0	*v1*	*v2*	*v3*	*Sum*	*A Solution?*
3	4	100	101	208	No

0 means the integer is not present in the subset.

All inputs which are valid solution (adds up to 104) are marked 'Yes'; the rest are marked 'No'.
If we had an efficient implementation of such set, we could easily solve this problem by defining the set of all possible solutions, applying different constrains over this set to build the set of vectors which are valid solution and then just print that set out.
These constrains we would apply in this example would be “all vectors with integers adds up to 104”.

It can easily be seen that this programming approach (set of integers-vectors) is equivalent to specifying a Boolean expression which gives true i.f.f the input is a valid solution to the problem.
These approaches are equivalent, as a set of (bounded) integers and a Boolean expression are interchangeable. (The Boolean expression would return true for an encoded integer i.f.f it is in the set).
I personally found that it easier for engineers to think about a problem using the “Set approach” rather than the “Boolean expression” approach.

Once we try represent a problem by a Boolean expression that returns “1” i.f.f an input is a valid solution, it seems like all hopes are almost gone.
Finding if such solution exists is the Circuit-Sat problem which is NP-Complete.
Fortunately, there are efficient implementations for Boolean expression (most know is Binary Decision Diagram – BDD) (By Bryant) which – although exponential in the worst case – are usually do good job with reasonable performance.

Introduction

The suggested language – {sets} - will allow to describe such problems in terms of constrained sets, with language elements and constructs which make it easy to define these constrains.

Here is a possible program in {sets} that solves the above problem: (identifiers are in **bold**)

```
# Let's try to solve an instance of “Subset-Sum” where the
# super set is {3,4,100,101} and the target is 104.

# each “dimension” in the vectors has 2 possible values.
# we define 4 dimensions for the vectors.
dim v0 = {0,3} ;
dim v1 = {0,4} ;
dim v2 = {0,100} ;
dim v3 = {0,101} ;

# The definition of dimensions implies a universal set which contains
# 2^4 vectors : { v0 x v1 x v2 x v3 }
# In this particular case, the implied universal set describes all
# valid inputs (not valid solutions!) to the problem, because
```

```
# the way we encoded the vectors.
# Had we encoded the vectors this way:
# dim v0, v1, v2, v3 = {0,3,4,100,101}
# We would have found it more challenging to define the set of valid
# inputs.

# Define the set of valid solutions.
set solutions = ( v0 + v1 + v2 + v3 == 104 ) ;

solutions.print() ;
# should print:
# (0,4,100,0)
# (3,0,0,101)
```

This approach also makes it easy to change the problem (adding more constrains) for example - looking only for solutions where the integer 100 is part of the subset. This can be achieved by refining the solutions set like this:

```
solutions = solutions && ( v2 == 100 ) ;
solutions.print() ;
# should print:
# (0,4,100,0)
```

For reference, the following is a hand-crafted C++ program to solve the above subset-sum problem, using the BuDDy BDD library:

```
#include "fdd.h"
#include "bdd.h"
#include "bvec.h"

#include <iostream>


int main() {
          using namespace std ;

          // {3,4,100,101}
          const std::size_t DOMS_NUM = 4 ;
          int doms[] = { 4, 5, 101, 102 } ;

          bdd_init(10000,10000);
          fdd_extdomain( doms, DOMS_NUM ) ;

          // for convenience, we create bdds vectors for the finite domains
          bvec v0 = bvec_varfdd( 0 ) ;
          bvec v1 = bvec_varfdd( 1 ) ;
          bvec v2 = bvec_varfdd( 2 ) ;
          bvec v3 = bvec_varfdd( 3 ) ;

          // useful constants ( since we can only perform operation with
          // bvecs of the same width
          bvec const_v0_0( bvec_con( v0.bitnum(), 0 ) ) ;
          bvec const_v1_0( bvec_con( v1.bitnum(), 0 ) ) ;
          bvec const_v2_0( bvec_con( v2.bitnum(), 0 ) ) ;
          bvec const_v3_0( bvec_con( v3.bitnum(), 0 ) ) ;

          bvec const_v0_3( bvec_con( v0.bitnum(), 3 ) ) ;
```

```
        bvec const_v1_4( bvec_con( v1.bitnum(), 4 ) ) ;
        bvec const_v2_100( bvec_con( v2.bitnum(), 100 ) ) ;
        bvec const_v3_101( bvec_con( v3.bitnum(), 101 ) ) ;

        bvec const_104( bvec_con( 7, 104 ) ) ;

        // creating the set {0,3}x{0,4}x{0,100}x{0,101}
        bdd valid_inputs =
                ( v0 == const_v0_0 | v0 == const_v0_3 ) &
                ( v1 == const_v1_0 | v1 == const_v1_4 ) &
                ( v2 == const_v2_0 | v2 == const_v2_100 ) &
                ( v3 == const_v3_0 | v3 == const_v3_101 ) ;

        cout << "all inputs:" << bdd_satcount( valid_inputs ) << endl ;

        // finding the solutions ( BuDDy will wire the implementations of '+'
        // and '==' for us )
        bdd solutions =
                ( bvec_coerce( const_104.bitnum(), v0 ) +
                  bvec_coerce( const_104.bitnum(), v1 ) +
                  bvec_coerce( const_104.bitnum(), v2 ) +
                  bvec_coerce( const_104.bitnum(), v3 ) == const_104 ) & valid_inputs ;

        cout << "solutions:" << endl ;
        cout << fddset << solutions << endl ;

        return 0 ;
}
```

The (quite expected) output of this program is :

```
all inputs:16
solutions:
<0:0, 1:4, 2:100, 3:0><0:3, 1:0, 2:0, 3:101>
```

From the output, we can see that the vectors in the solution set are (0,4,100,0) and (3,0,0,101) .
Based on the encoding we defined, it means that only the subsets {4,100} and {3,101} meet the target of 104.

Compiled or Interpreted ? (TBD)

I would really like to create an interpreter in Java, but I didn't find any good Java BDD library.
That leaves me with 2 possible alternatives:

- Implement a compier in Java, that generates C++ code to be linked with buddy
- Implement an interpreter in C++ and link it with a C++ BDD library.

I tend toward the 2nd option .

Object Oriented

Every data entity in the language is an object of one of the following possible types: integer, dimension, vector, set .These types are primitive types in the language, with their own built-in methods.

Objects of certain types, can be combined to create an object of another type. Examples:

- Combining 2 sets to create a new set:

```
set went_to_park = sunny_days && days_temp_less_than_60
```

- Creating a set from a vector (a set which contains only that vector)

```
set set_of_size_1 = ( 1, 3, 0 1 ) ;
```

- Creating a set from dimension and dimension, creating a set from dimension and integer

```
set pipe_stall =
        inst0_lhs_reg == inst1_rhs_reg &&
        inst0_opcode == 231 ; # LOAD
```

The key point in the language is the ability to easily create complicated sets (solutions sets) combining the above building blocks with expressive operations.

Each such object (integer, dimension, set and vector) has its own operations.

Future Enhancements

- Using more than one Universal set in a program (depends on support in the BDDs library)
- Adding support for user-defined functions
- Adding control structures (foreach, if)
- Adding I/O facilities for primitives objects. (Actually, only input is required)
- Using enumerated types as dimension values, in addition to integers

References and The Inspiring Work of Others

Randal E. Bryant – Graph Based Algorithms for Boolean Function Manipulation, 1986
http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf

Cormen, Leiserson, Rivest – Introduction to Algorithms, MIT Press, 1989

Jørn Lind-Nielsen (the father of BuDDy)
http://www.itu.dk/people/jln/

BuDDy's new project page in SF
https://sourceforge.net/projects/buddy

Gerd Behrman, IBEN – A Shell like environment for BDDs
http://sourceforge.net/projects/iben

TCL extension I developed during my career which did similar things (and much more), but was little bit awkward due to the constrains of the TCL language, line-by-line interpretation, and semantic error checking which had to be performed in runtime.