Painless Programming Combining Reduction and Search

Design Principles for Embedding Decision Procedures in High-Level Languages

Tim Sheard
Portland State University
sheard@cs.pdx.edu

Abstract

We describe the Funlogic system which extends a functional language with existentially quantified declarations. An existential declaration introduces a variable and a set of constraints that its value should meet. Existential variables are bound to conforming values by a decision procedure. Funlogic embeds multiple external decision procedures using a common framework. Design principles for embedding decision procedures are developed and illustrated for three different decision procedures from widely varying domains.

Categories and Subject Descriptors D.1.0 [*Software.*]: Programming Techniques.General.

General Terms Applicative Programming, Logic Programming, Functional Programming

Keywords Search, first order logic, decision procedures

1. Introduction

There are many styles of declarative programming – functional programming (FP), logic programming (LP), and constraint based programming (CLP), to name a few. Most systems that implement a particular style fall exclusively into one of two broad computational modalities that I call *reduction* and *search*. A declarative program, in the reduction modality, consists of a set of instructions for transforming the input into the output. A declarative program, in the search modality, consists of a description of the properties of a solution using some sort of logic, and then searching a solution space for an answer that has those properties. It is important to emphasize that both kinds of systems can emulate the other (since most are Turing complete). In fact, there are some systems that combine both modalities, such as Curry[4], Ciao![10], Flora[20], AMPL[8], and Oz[24]; The most common approach is to build a general purpose language around a single search modality mechanism. These languages often call specialized external tool, such as a SAT solver (Alloy[12]), Linear-Programming libraries (AMPL), an SMT solver (DMinor[3]), or a CLP solver library (Ciao!).

Another approach is to embed a search modality into a functional language through the use of libraries or specialized data structure design. Such systems often do a competent job. But in general both kinds of systems suffer from one or more of the following problems.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

- **Loss of generality.** Systems built around a single search based mechanisms can solve a one class of problems well, but break down on other classes.
- **Impedance mismatch.** The use of embedded libraries to broaden the class of problems amenable to solution often results in a certain unnaturalness in their use. For example they may require the user to script a bunch of library calls using a given API.
- **Sub-optimal notation.** Some problems are naturally and succinctly encoded using specialized notation. Systems built around libraries often require the user to encode their problem description in an embedded specification language using the host language's notation.
- **Poor abstraction.** Specialized systems are often very expressive, but not succinct. It matters not, that a problem can be expressed, if it requires thousands of lines to do so.
- **Loss of incremental improvement.** Through contests such as SMT-COMP [2], and the CADE ATP System Competition [25] the competition amongst implementers of specialized solvers is immense. The winning systems often make tremendous improvements over previous year's winners. Capturing these gains is important.

The author, interested in alternate ways to specify programs declaratively, tried many systems, and ran into all of the problems above. He noticed that every problem is avoided in some system. Could all the problems be avoided in a single system? He decided to try by following the design principles below.

- A good system combines both computational modalities. Search naturally describes some problems, and reduction others. The two systems complement each other in several important ways. Logic based programs are often both concise and easy to understand; while functional programs make great scripting languages, for combining things together.
- A good system should have multiple external solvers. This allows the system to be general over several classes of problems, yet benefit from specialized implementations and incremental improvements.
- A good system naturally supports alternate notations where necessary, but reuses notation where possible.
- A good abstraction that bridges between the functional and logic worlds is necessary. The abstraction should be general – it should apply to all logics. It should be both easy to use and understand.

Funlogic is a new language (not an embedded domain specific language). It has its own compiler. It reuses key ideas from many systems (set notation from Datalog, escape from one syntax to an-

other from MetaML, relational algebra as first order logic from Alloy and KodKod, narrowing from Curry, and overloaded types from Haskell). In building Funlogic, the author learned many engineering lessons and much about combining multiple solvers. He made a number of research discoveries worth reporting.

- Search problems are best described by multiple dimensions. These include description of the search space, special cases that either shrink the search space (such as symmetries) or make its description smaller, the properties that should hold of the solution, the strategy used to search the space, and a description of the target of the search – are we looking for any solution, no solution, multiple solutions, a solution that maximizes an objective function, an approximate solution, or a solution as a probability distribution? We have found these dimensions to describe every kind of search we have studied. Additional dimensions, to add to our understanding of existing systems, are sought.
- Overloading of terms allows the same language to be used in both the reduction and search modality. A term that describes the property of a solution specifies input to an external solver. The meaning of a term in a constraint, is not the same as an identical term in another part of the program. This difference in meaning can be explained by two techniques: overloading and staging. Precise semantics can be given for the language in terms of these two techniques.

The rest of this paper is as follows: In Section 2 we describe our system. In Section 3 we solve several small problems (each with a different solver). In Section 4 we discuss the eight steps necessary to add a new solver. Then, in three large Figures (4, 5, 6), we step through the eight steps for each solver, discussing similarities and differences using real data extracted from the examples introduced in Section 3. In Section 5 we discuss a few of the many possible extensions that might make our system even more useful. In Section 6 we discuss how it is related to other systems.

2. Language description

Funlogic combines functional programming with first-order logic. A program in Funlogic consists of a sequence of *declaration*s. A declaration introduces into scope one or more *names*, and each name is bound to a *value*. Values bound to newly introduced names are either primitive (like data constructors) or they are computed by either reduction or search. There are six kinds of declarations.

1. **Value.** `(twoPi,x) = (3.14159 * 2.0, not True)`
   A value declaration introduces one or more names by use of a pattern on the left-hand-side of an equation. The term on the right-hand-side is reduced to a value, and that value is matched against the pattern, binding the names in the pattern. In the example above `twoPi` is bound to `6.28318` and `x` is bound to `False`.
2. **Dimension.** `dim width#Int = [1 .. 3]`
   A dimension declaration introduces a finite subset of a base type. Base types include `Int`, `Real`, `Bool`, `String`, `Char`. Dimensions are finite sets and play a key role in describing search spaces.
3. **Data.** `data Tree a = Tip |Fork(Tree a) a (Tree a)`
   A data declaration introduces one or more *constructor*s which are either functions or constants of a newly introduced type. In this example `Tree` is a new type, and `Tip` is a constant of type `Tree`, and `Fork` is a ternary function that returns a Tree. An enumeration (a data type consisting of only constants), also introduces a **Dimension** with the same name.
4. **Function.**
   ```
   len [] = 0
   len (x:xs) = 1 + len xs
   ```
   A function declaration introduces a function defined by pattern matching over one or more clauses. Functions may be recursive.
5. **Formula.**
   ```
   anc(x,y) -> person(x),person(y).
   anc(x,y) <- parent(x,y);
               parent(x,z),anc(z,y).
   ```
   A formula declaration is an alternate syntax for introducing a name that binds to a finite set. A finite set describes a relation, and the formula syntax is reminiscent of a Prolog or Datalog program. The formula is a concise way of describing complicated sets. A formula declaration has two parts: a constraint `anc(x,y) -> person(x),person(y)` and a computation: `anc(x,y) <- parent(x,y);parent(x,z),anc(z,y)`. The constraints can place arbitrary limits on what the computation can add to the set. In the example above `person` is a previously introduced dimension (playing the role of a unary predicate), and `parent` a previously introduced relation. Formula are an example of an alternate notation and are discussed in detail in Section 3.3.
6. **Search.**
   ```
   exists  ys : List 4 Int
     where  sum ys == 9
      find  First
        by  SMT
   ```
   A search declaration introduces one or more names whose values are computed by search. It contains a number of components: the name(s) being introduced (`ys`), a description of the search space (`List 4 Int`), a set of constraints (`sum ys == 9`), a strategy (`First`) and a technique (or solver) used to perform the search (`SMT`).

In the paragraphs that follow we will introduce additional features of Funlogic by introducing a number declarations, then we will explore the consequences of the declarations by showing an interaction with the read-eval-print loop of Funlogic. An interaction starts with the prompt `exp>` , which is followed on the same line with the user's input, followed on the next line with the system's response. For example

```
exp> 4+4
8:: Int
```

Here we see that the system responds with `8:: Int` when the user types `4+4` after the prompt.

Feature List. Haskell-like list syntax is supported. Lists can be constructed by enumeration, the use of constructors (`[]` for nil, and the infix `(:)` for cons), and list comprehensions. Lists also support pattern matching.

```
exp> [True,False]
[True,False]:: List Bool

exp 1:2:[]
[1,2]:: List Int

exp> [2..6]
[2,3,4,5,6]:: List Int

exp> [(i,i-j) | i <- [8,9], j <- [3,4]]
[(8,5),(8,4),(9,6),(9,5)]:: List (Int,Int)

exp> case [3,4] of { [] -> 99; (x:xs) -> x}
3:: Int
```

Feature Dimension. Finiteness plays an important role in Funlogic. Search spaces are finite n-dimensional spaces, or can be de-

scribed by a finite number of logical variables. Funlogic uses the notion of *dimension* to describe this phenomena. A single dimension is introduced by the `dim` declaration, or by an enumeration.

```
dim int11#Int = [0..10]
dim int2#Int = [4,5]
data Name = Tom | Hal | Jon
```

These declarations introduce names for three dimensions.

```
exp> int11
Int#11:: Dim Int

exp> int2
Int#2:: Dim Int

exp> Name
Name#3:: Dim Name
```

In general, Funlogic uses the symbol `#` in the syntax that manipulates dimensions. Multi-dimensions are constructed from other dimensions by use of the dimension aggregate operator that consists of the `#` operator followed by a tuple of dimensions. For example.

```
pair = #(Name,int11)
triple = #(pair,int11)
```

Dimensions are flattened when they are aggregated. Note how `triple` has been flattened into sequence of 3 simpler dimensions even though it was constructed by aggregating two dimensions.

```
exp> pair
#(Name#3,Int#11):: Dim (Name,Int)

exp> triple
#(Name#3,Int#11,Int#11):: Dim (Name,Int,Int)
```

Operations on dimensions include iteration using a list comprehension, and the function `elem:: Dim a -> a -> Bool`.

```
exp> [ i | i <- #(Name,int2) ]
[(Tom,4),(Tom,5),(Hal,4),(Hal,5),(Jon,4),(Jon,5)]
:: List (Name,Int)
```

```
exp> elem pair (Tom,5)
True:: Bool
```

```
exp> elem pair (Hal,55)
False:: Bool
```

Feature Array. Array are finite aggregates with constant time access functions. An array with type: `Array d i` is indexed by values in the dimension `d` and contain elements of type `i`. Arrays are constructed by `array:: Dim d -> List i -> Array d i`.

```
twoD = array #(Name,int2) ['a','b','c','d','e','f']
oneD = array width ["red","blue","green"]
```

For 2-D arrays the elements in the initialization list appear in row-major order. It is assumed that there is an element in the list for every point in the domain `d`.

```
exp> oneD
 1      2       3
+-----+-------+--------+
|"red"| "blue"| "green"|
+-----+-------+--------+
:: Array Int String
```

```
exp> twoD
    4   5
   +---+---+
Tom|'a'|'b'|
   +---+---+
Hal|'c'|'d'|
   +---+---+
Jon|'e'|'f'|
   +---+---+
:: Array (Name,Int)
          Char
```

Arrays are accessed using `index:: Array d i -> d -> i`. The infix operator `(.)` is also bound to the same function. Dimensions of an array are accessed by `arrayDim:: Array d i -> Dim d`.

```
exp> index oneD 3
"green":: String

exp> twoD.(Jon,4)
'e':: Char

exp> arrayDim twoD
#(Name#3,Int#2):: Dim (Name,Int)
```

Feature Set. A value of type `Set(A,B,C)` stores a set of tuples of type `(A,B,C)`. A set is constructed with the function `set:: Dim d -> List d -> Set d`. Elements of a set are constrained by the domain of the set.

```
dim int3#Int = [1,2,3]
s1 = set #(int3,int3) [(1,2),(1,2),(2,3),(0,4)]
```

Set construction removes elements from the list that are outside the domain. Set construction also ignores duplicates. Note the "missing" tuples in `s1` below.

```
exp> s1
{(1,2) (2,3)}:: Set(Int,Int)
```

3. Several small problems

In Figure 1 are several small problem solutions written in Funlogic. Each illustrates a different search based paradigm. All are remarkably similar. A problem is defined. A solution is phrased in terms of an existentially declared data structure with first order constraints, and a solver is chosen. The majority of the code comprising a solution consists of a few small functions that manipulate data and express relevant boolean valued functions used to constrain which solutions are acceptable.

3.1 A production problem

In the lower-left quadrant of Figure 1 is a Funlogic program that solves a production problem using an linear-programming solver. The problem involves choosing the production level at several factories to meet estimated sales demand while minimizing transportation costs, subject to some global constraints. The existentially declared array `prod` holds production information. The value stored in `prod.(f,s)` holds the number of units produced at factory `f` destined for store `s`. Constraints include:

- Factory `A` is smaller than the others, and its total production cannot exceed 150 units.
- every `prod.(f,s)` value is positive or zero.
- The sum of production for each store is equal to estimated sales at that store.

Shipping costs vary between each factory and store, and are stored in the array `ship`. The store owners wish to minimize total shipping costs. Note that the specification of the problem is expressed in terms of ordinary user level functions: `sum`, and `and`. These and several other "library" functions are found in the lower-right quadrant of Figure 1.

3.2 An N-queens solver

In the upper-right quadrant of Figure 1 is a Funlogic program that solves the N-queens problem using an SMT solver. The problem involves placing n-queens on a $n \times n$ chessboard in a manner such that no queen can take another queen using the moves of chess.

```
rank = 2    -- Rank 2 (4x4) Soduko solver

dim size#Int = [0 .. rank*rank - 1 ]
dim digit#Int = [1.. rank*rank ]

input = set #(size,size,digit)
            [(0,3,4),(1,1,2),(1,2,1)
            ,(2,1,1),(2,2,4),(3,3,1)]

-- (i,j,n) is in the set if  "ij"= n (ij in base-rank)
square = set #(size,size,size)
         [ (i,j,(div i rank) * rank + (div j rank))
         | i <- size, j <- size ]

exists grid : set #(size,size,digit) input .. universe
 where     -- every row(n) has 1-4
       and[$(full {k<-grid($n,j,k)}) | n <- size] &&
           -- every column(n) has 1-4
       and[$(full {k<-grid(i,$n,k)}) | n <- size] &&
           -- every box(n) has 1-4
       and[$(full {k<-grid(i,j,k),square(i,j,$n)})
           | n <- size ] &&
           -- each coordinate has only one digit
       $(grid(i,j,n),grid(i,j,m) -> eq#digit(n,m) )
  find First
    by SAT

ans = setToArray grid
```

```
size = 4           -- An N-Queens Solver

dim width#Int = [1 .. size]

dim i2#Int = [0,1]

rowPts i = [(i,j) | j <- width]
colPts j = [(i,j) | i <- width]
nwEdges = append (rowPts 1) (colPts 1)
swEdges = append (rowPts size) (colPts 1)

add m pts = sum [m.p | p <- pts]

downDiag(x,y) =
  (x,y):[(x+i,y+i)| i <- width, x+i <= size, y+i <= size]
upDiag  (x,y) =
  (x,y):[(x-i,y+i)| i <- width, x-i >= 1, y+i <= size]

exists  bd : Array #(width,width) i2
 where      -- every row(i) adds to 1
       and [add bd (rowPts i)   == 1 | i <- width] &&
           -- every column(i) adds to 1
       and [add bd (colPts i)   == 1 | i <- width] &&
           -- every diagonal adds to 0 or 1
       and [add bd (downDiag p) <= 1 | p <- nwEdges ] &&
       and [add bd (upDiag p)   <= 1 | p <- swEdges ]
  find  First
    by  SMT
```

```
-- Production minimization problem

data Factory = A | B | C
data Store = NYC | ATL | LA

pairs = #(Factory,Store)

ship = array pairs [2,3,5,3,2,1,3,4,2]
sales = array Store [230,140,300]

exists prod: Array #(Factory,Store) Int
 where sum[ prod.(A,s) | s <- Store ] <= 150 &&
       and [ prod.(f,s) >= 0 | (f,s) <- pairs ] &&
       and [ sales.s == sum [prod.(f,s) | f <- Factory]
           | s <- Store ]
  find Min sum[ prod.(f,s) * ship.(f,s)
              | (f,s) <- pairs ]
    by LP
```

```
-- Library functions

append :: List a -> List a -> List a
append [] ys = ys
append (x:xs) ys = x :(append xs ys)

and :: BoolLike b => [b] -> b
and [] = true
and [x] = x
and (x:xs) =  x && (and xs)

sum :: NumLike t => [t] -> t
sum [] = liftI 0
sum [x] = x
sum (x: xs) = x + (sum xs)
```

Figure 1. Solutions for several small problems

We assume the reader is familiar with this problem[1]. The program works as follows. It represents a solution by an array of integers in the range `[0..1]` A 4-queens solution looks like:

solution

	Q		
			Q
Q			
		Q	

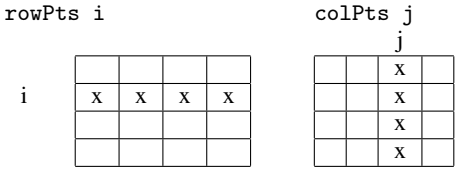
representation

0	1	0	0
0	0	0	1
1	0	0	0
0	0	1	0

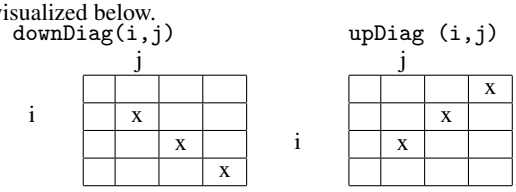
The invariant (on the representation) is that every row and column sums to exactly 1, and that the sum of every diagonal is at most 1. To compute these sums, we proceed in two steps. First we define simple functions that return a list of points, representing coordinates in the array, for rows, columns, and diagonals. It is best to visualize the points that a function returns by using a graphical representation. An `X` in a square means that coordinate is an element of the list returned. Rows (`rowPts`) and columns (`colPts`) are par-

[1] See **wikipedia.org/wiki/Eight_queens_puzzle** for a good discussion.

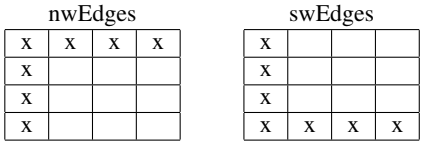
ticularly easy and are implemented by simple comprehensions over the size of the problem.



Diagonals are more complex. Given a point, (i,j), we compute the list of points on the up and down diagonals starting at that point as visualized below.



To compute all the diagonals, we see that every complete diagonal is rooted at a point on the edge of the array. Down-diagonals are rooted on points on the north and west edges, and up diagonals are rooted on points on the south and west edges.



To sum the elements in an array we define the function `add`. It is given an array and a list of coordinates, and sums the elements of the array at those coordinates. Here we make use of the library function `sum` which adds all the elements in a list.

The answer we are searching for will be stored in the existentially declared array `bd`. It is a 2-D array with dimensions `#(width,width)` . It stores elements in the range of the dimension `i2 ([0..1])`.

The constraint on `bd` is a large conjunction consisting of 4 parts: rows, columns, up-diagonals, and down-diagonals. each part has the form: `and [ add bd (`f `i) ◇ 1 | i <-` *alphas* `]` where *alphas* is a set of elements (here, either a positive integer less than the puzzle size or a coordiate of an edge), f is a function from an element to a list of points, and $\diamond$ is a boolean relation. For each coordinate in $(f\ i)$ we sum the element at that coordinate, and compare that sum to 1.

3.3 An exercise in alternate notation

In the upper-left quadrant of Figure 1 is a Funlogic program for solving soduko problems. It uses a SAT solver, and much of its elegance and simplicity relies on using an alternate notation for describing and manipulating sets. A set of n-tuples is an n-ary relation.

FunLog has two alternate notations for manipulating relations – *formula* and *constraints*. The simplest formula is called an atom. It is comprised of a name followed by a parenthesized list of patterns. For example $R(p_1, .., p_n)$. To be well formed R must be a set of n-tuples of type $(t_1, .., t_n)$ and each pattern p_i must have type t_i. The atom $R(p_1, .., p_n)$ denotes the largest subset of R in which every tuple matches the patterns $(p_1, .., p_n)$. Atoms are the basic building blocks of formula and constraints.

In Figure 2 are rules for constructing formula and constraints. These notations are modelled after both Prolog terms and Datalog formula (that express the relational algebra). Because I assume that most readers are familiar with at least one of these notations[2], I only give suggestive examples for each supported construction in both notations.

Formula syntax. let `x::Set(A,B)`, `y::Set(A,B)`, `g::Set(B,B)`, `h::Set(C,D)`, and `f::Set(C,B)`. Also let `a`, `b`, `c`, and `d` be variables, and `p` and `q` be patterns. Then we can denote operations on these sets by the following formula:

formula	*meaning*	*type*
`x(p,q)`	atomic formula, filter	`Set (A,B)`
`x(a,b); y(c,d)`	union of x and y	`Set (A,B)`
`x(a,b), y(a,b)`	intersection of x and y	`Set (A,B)`
`x(a,b), h(c,d)`	product of x and h	`Set (A,B,C,D)`
`h(c,d),f(c,b)`	join of h and f	`Set (C,D,B)`
`{ a <- y(a,b) }`	the 1st projection of y	`Set A`
`eq#n(a,b)`	equality on n	`Set (n,n)`

Constraint syntax. A constraint denotes a boolean valued function over a set of tuples. Let `r::Set d`, `s::Set d`, and `f::Set(d,e)` where `d` and `e` are finite domains as explained in Section 2, then:

formula	*meaning*
`none r(x)`	`r` is the emptyset
`full r(x)`	`r` contains every tuple in domain `d`
`some r(x)`	`r` has at least one element
`one r(x)`	`r` has exactly one element
`r(x) <= s(y)`	`r` is a subset of `s`
`f(x,y) -> r(x)`	`r` is a subset of `{ x <- f(x,y)}`
`f(x,y) | x -> y`	in `f`, `y` functionally depends on `x`

Example translation. A formula denotes a set, and a constraint denotes a boolean value. Each translates to ordinary function calls over sets. Formula translate into calls to functions that implement the relational algebra over sets of tuples. An example translation follows.

```
{ c <- x(a,"Tom"),y(a,c) }
```

First we select only those tuples of `x` whose second component is `"Tom"`, then join the resulting relation with `y` on the common component `a`. This results in a ternary relation, `z(a,"tom",c)`, which is then projected on its third component.

```
project3of3 (join (select (\(a,b)->b=="Tom") x) y)
```

Figure 2. The meaning of formula and constraints by example.

Embedding alternate syntax. The default syntax of Funlogic is the expression. Any place an expression is expected a formula or a constraint can be used by *escaping* into one of the alternate syntaxes by use of the `$( ... )` operator. Consider the declarations

```
dim people#String =
   ["Anita","Barbara","Caleb","Frank","Tim"]
parent = set #(people,people)
   [("Frank","Tim"),("Tim" , "Caleb")
   ,("Anita","Tim"),("Barbara","Caleb")]
```

To create a set consisting of those tuples that include valid (child, parent, grandparent) triples, one may declare:

```
threeGen = $(parent(x,y),parent(y,z))
```

which escapes to the alternate *formula* notation to express a self-join on the `parent` relation. One can test if that set has exactly one element by escaping into the alternate *constraint* notation.

[2] if you are not, see **wikipedia.org/wiki/Datalog** for an introduction

```
exp> threeGen
{("Caleb","Tim","Anita") ("Caleb","Tim","Frank")}
: Set(String,String,String)

exp>
exp> $(one threeGen(x,y,z))
False: Bool
```

It is also possible to escape from the formula notation into the expression notation. Let the variable `n` have value `"Tim"`, then in a formula `parent(x,$n)` the `$n` indicates an escape into the expression notation, and is equivalent to the formula `parent(x,"Tim")`. This makes it possible to parameterize sets specified using the formula notation.

3.4 A Soduko solver

Soduko puzzle of rank n consists of a square matrix with edge size equal to $n \times n$ where some of the squares have been filled in with digits in the range `[1.. n*n]`. A sample puzzle of rank = 2 is given below.

	0	1	2	3	
0				4	(0,3,4)
1		2	1		(1,1,2) (1,2,1)
2		1	4		(2,1,1) (2,2,4)
3				1	(3,3,1)

To solve this problem, a number between 1 and 4 must be inserted into each empty coordinate. The invariant of a successful solution is that all the digits 1-4 must appear (in any order) in every row, in every column, and in every 2×2 box. We make the term *box* precise in our code, but note, in a rank n problem, the boxes are $n \times n$. Each coordinate in a row, column, or box is given an index (between 0 and $n-1$) as illustrated below. For example the coordinates `(1,3,i)` (in **bold** font) are in row `1`, column `3`, and box `1`.

row

0	0	0	0
1	1	1	**1**
2	2	2	2
3	3	3	3

column

0	1	2	3
0	1	2	**3**
0	1	2	3
0	1	2	3

box

0	0	1	1
0	0	1	**1**
2	2	3	3
2	2	3	3

We represent a problem as a finite set of triples. One for each filled in square in the problem description. The triples for the sample puzzle are listed to the right of the puzzle above (they are called `input` in the solution).

The interpretation of a triple `(row,col,k)` is that the value k is stored at coordinate `(row,col)`. Because both `(row,col,3)` and `(row,col,5)` could be in the set, it is possible for many numbers to be stored at each coordinate. Thus an additional invariant is that exactly one number is stored at each coordinate.

We solve the problem by computing sets of triples, which are subsets of the 3- dimensional search space. Each subset, s, contains the tuples comprising a single row, column, or box. If we project ({ `k <-` s`(i,j,k)` }) a set of tuples, like `s`, on the digit column, we obtain a set of digits, where each element is in the range 1-4 (see the `digit` dimension declaration). An invariant is met if this set is the `full` set {`1,2,3,4`}. Computing the projection over row n ({`k<-grid($n,j,k)`}) and column n ({`k<-grid(i,$n,k)`}) is trivial. The n^{th} box takes some care. The set `square` assigns every coordinate to a single box index. Thus the tuple `(i,j,n)` is in the set, `square`, if coordinate `(i,j)` is assigned to box `n`. This is easy to compute by noting that $n = ij$ if we read ij as a 2 digit number in base *rank*. When rank is 2, `square` is the set:

```
{(0,0,0),(0,1,0),(0,2,1),(0,3,1),(1,0,0),(1,1,0)
,(1,2,1),(1,3,1),(2,0,2),(2,1,2),(2,2,3),(2,3,3)
,(3,0,2),(3,1,2),(3,2,3),(3,3,3)}
```

See the graphic labeled ***box*** above for a visual representation of `square` at rank 2.

4. Seven steps to adding a new solver

The steps we discuss in this section constitute a prescription of how to incorporate an external solver into a high-level language. We consider these steps to be the research results of this paper. We discuss these steps in two passes. First we introduce the steps in the abstract. Then in Figures 4, 5, and 6 we illustrate each of the steps, on each of the solvers, using actual data from the problems introduced in Figure 1.

Every solver accepts problems in a given form, the compiler must capture this form, but hide its details from the programmer. The programmer thinks in terms of data and functions supported by Funlogic. The key to "painless programming" is maintaining the programmer's view. This is done by the use of overloading. In Figure 3 are four abstract classes of operations. Three of these classes are familiar to most programmers – arithmetic, booleans, comparisons. The fourth class captures operations in the relational algebra, and will be familiar to any one who has studied data bases.

In the same figure we supply *concrete instances*. These are the functions programmers normally associate with these operations. Each solver will associate a different set of functions with these operators, and supply a mechanism to lift a concrete value to its representation type(s). This process is described in the next few paragraphs. As we look closely at each solver, keep in mind how widely their structure varies, yet they will all yield to this same process.

1. Representation types. The first step to incorporating a solver is to choose a data structure to represent problems solvable by that solver. Actual representation types appear as the first step in each of the Figures 4, 5, and 6. These come in several flavors.

- **Term representations.** The type `SAT` (figure 5) is an abstract representation of the booleans. The type `SMT` (figure 4) is an abstract representation of operations and comparisons over numeric types. These types are essentially term representations of expressions over of the concrete type they represent.
- **Structural representations.** The type `MExp` (figure 6) captures the domain of constraints over linear arithmetic expressions as used in linear-programming problems. Here the representation captures structural properties of the problem domain – that a term is a polynomial.
- **Propositional representations.** The type (`BitVector SAT t`) (figure 5) is an abstract representation of finite sets of elements of type `t`. A propositional representation stores "bits" and represents different values depending on the truth or falsity of the bits stored. Some users may be familiar with a bit-blasting propositional representation of arithmetic, where integers in the range `[0 .. n]` are represented as log_2 bits. A propositional representation "compiles" to a SAT problem.
- **Search tree.** A fourth kind of representation type is that of an explicit search tree. Overloaded operations "prune" paths in the tree that do not lead to a solution that meets the constraints. For space reasons, an example of this type of representation is not given in the paper.

2. Overloading. The second step in the process of maintaining an abstract view of solver representations is the use of overloading. Programmers write constraints using the computational mechanisms of FunLog – functions and data structures. User defined

Overloaded operators

```
class BoolLike b where
  true :: b
  false :: b
  (&&) :: b -> b -> b
  (||) :: b -> b -> b
  liftB :: Bool -> b
```

```
class NumLike t where
  liftI :: Int -> t
  liftR:: Rational -> t
  (+) :: t -> t -> t
  (*) :: t -> t -> t
```

```
class(NumLike t,BoolLike b)
     => Compare t b where
  (<=):: t -> t -> b
  (==):: t -> t -> b
```

```
class SetLike s where
  create:: Dim a -> [a] -> s a
  select:: (a -> Bool) ->
           s a -> s a
  proj3of3:: s (a,b,c) -> s c
```

Concrete instances

```
instance BoolLike Bool where
  true = P.True
  false = P.False
  x && y = x P.&& y
  x || y = x P.|| y
  liftB x = x
```

```
instance NumLike Int
 where
  liftI x = x
  liftR x =
    error "UnSupported"
  (+) x y = x P.+ y
  (*) x y = x P.* y
```

```
instance Compare Int Bool
 where
  (<=) x y = x P.<= y
  (==) x y = x P.== y
```

```
instance SetLike Set.Set where
  create dom xs =
     Set.fromList
        [ x | x <- tuples dom
            , elem x xs]
  select p xs = Set.filter p xs
  proj3of3 xs = Set.map third xs
    where third(x,y,z) = z
```

We use a Haskell-like notation to describe classes and instances. The use of the notation "P.x" indicates the concrete un-overloaded function or value. Both classes and concrete instances are abbreviated, we show only a few member functions, enough to explain the examples in Figure 1, in the implementation there are many more member functions. Note that all solvers, no matter what their representation types, will support the same abstract interfaces. For space reasons, concrete instances for `Float`, `Double`, and `Rational` are not shown.

Figure 3. Overloading with abstract and concrete instances.

functions manipulate both real data and abstract data representations through the magic of overloading. Overloading in Funlogic is similar to overloading in Haskell. Every primitive (numeric operators, boolean operators, set operations, etc.) has a standard concrete implementation and one or more overloaded abstract implementations. One for each solver that might use that operator. Abstract implementations of operators manipulate abstract representations. User written functions, through overloading, inherit multiple implementations through a library passing mechanism. Which library is passed depends upon the context. Whether the user is manipulating real data or solver representations, he uses the same functions in the same way.

3. Initialization. The third step in the process of maintaining an abstract view of solver problems is initialization. An existentially declared variable must be translated (or initialized) into the internal representation of the appropriate solver. The programmer views this internal representation as if it was an ordinary concrete value when he writes constraints. Ordinary functions and data, defined by the programmer, are used to manipulate it. Initialization chooses an abstract representation and constructs a view consistent with the programmers view of the data. An initializer looks like a type. Depending upon the solver, this type will be expanded into some abstract representation, different for each solver.

4. Staging and Resolving Overloading. The fifth step in the process of maintaining an abstract view of solver problems is handling mixed concrete and abstract data in the constraints associated with existential declarations. Data is concrete if it is a literal constant, or declared outside the existential declaration. Consider a constraint for an SMT existential declaration.

```
exists x::Int, z::Int   where  ((x + (2 + y)) == z)
```

Where `(+)` and `(==)` are overloaded, `2` is concrete, `y` is concrete because it is declared outside the existential, and `x` and `z` are abstract (existentially bound). We type check the program in the following environment.

```
(+):: forall n . NumLike n => n -> n -> n
(==):: forall n b . Compare n b => n -> n -> b
x:: t1    -- existentially bound x's type is unconstrained
y:: Int   -- y's type is concrete
z:: t2    -- existentially bound z's type is unconstrained
```

Type checking infers a type for a term, and reconstructs the term where overloading is made explicit, and unconstrained types my become constrained by context, and concrete sub terms are made as large as possible. The term is reconstructed with the following type, and the types of `x` and `z` are further constrained.

```
((x (+)#A (liftI#B ((liftI#C 2) (+)#D y))) (==)#E z):: t4
x:: t3;    y:: Int;     z:: t3
```

The reconstructed overloaded operators (`(+)`, `(==)`, `liftI`) are tagged with constraints (`A`, `B`, etc). We separate the constraints from the reconstructed term to make the term easier to read.

```
#A = (NumLike t3);  #B = (NumLike t3);  #C = (NumLike I)
#D = (NumLike I);   #E = (Compare t3 t4)
```

Note that the term `((liftI#C 2) (+)#D y)` is completely static, since the constraints `#C` and `#D` are completely static. Note further, that some of the others are unconstrained. This is because we make few assumptions about the variables `x` and `z`. In the next step, we use the solver context to remove this uncertainty. First, a where clause represents a boolean value, so the whole term must have the type representing SMT's version of `Bool`, which is SMT. Second, the existential variables have type `Int` and SMT's version of `Int` is also `SMT`. See Figure 4 for the details. So under the variable assignment {`x:: SMT, y:: Int, z:: SMT` } we check the reconstructed term.

```
((x (+)#A (liftI#B ((liftI#C 2) (+)#D y))) (==)#E z):: SMT
```

This completely fixes the types in each of the constraints

```
A# = (NumLike SMT);  B# = (NumLike SMT);  C# = (NumLike I)
D# = (NumLike I);    E# = (Compare SMT SMT)
```

This specifies an exact function for each overloaded call.

```
((x :+: LitI (id 2 P.+ y)) :=: z)
```

What we have described is a type based binding time analysis where concrete terms are static, and abstract terms are dynamic. We have used two binding time analyses, and have found them both to work well. The first is embedded in an on-line partial evaluator that uses a lazy (just in time) lifting. We have also used a static (off-line) analysis, based upon some previous work [17, 22], appropriate for a compiled semantics. See Appendix B for details of this step.

5. Constraint generation. The fourth step in the process of maintaining an abstract view of solver problems is constraint specifi-

cation. The user writes a boolean valued expression involving the existentially declared variables. His constraints may also mention any other concrete data in scope. This constraint is executed using the overloading associated with the particular solver, as described above. Evaluation under the overloaded functions associated with the solver produces abstract-input appropriate for that solver.

6. Input formatting. While the representation type is meant to capture the structure of the input to a solver, there will always be some reformatting necessary to accommodate the input format of individual solvers.

7. Instantiation. Once a problem has been solved by an external solver, the solution must be used to instantiate the abstract structure of existential variables into concrete data.

4.1 The N-Queens problem

The N-Queens problem is solved by a SMT solver. The 7-step process is illustrated in Figure 4. It uses a term representation we call `SMT`. This is an untyped term algebra that builds data structures representing expressions over arithmetic, booleans, and comparisons.

Its abstract instances just build larger terms from smaller terms, by using the constructor functions from `SMT`.

The n-queens problem initializes a small vector of values, each in the range `[0..1]`. In the SMT solver, an array is initialized to a real array of abstract variables (elements of type `SMT`). The types of these abstract variables is taken from the initializer (the range `[0..1]`) and passed as input to the solver (see step 7). Functions that manipulate these variables will be overloaded and build SMT data.

To illustrate constraint generation in the queens example study one of the constraints from Figure 1.

```
and [add bd (rowPts i) == 1 | i <- width]
```

Binding time analysis, lifts the constant `1`, and the expression is evaluated in a context where the functions `add`, `and`, and `(==)`, are bound to their abstract instances.

Abstract variables from each row `i` are added and their sum is equated with `1`. The effect is to build `SMT` data. Inspect the abstract initialization to see that the correct variables are indeed added. The `SMT` data is then formatted to meet the input specifications of the solver.

4.2 The Soduko problem

The Soduko problem is solved by a SAT solver. The 7-step process is illustrated in Figure 5. It uses a term representation (we call `SAT`) to represent the booleans, and a propositional representation we call `BitVector` to represent sets.

The type (`BitVector SAT t`) is an abstract representation of finite sets of elements of type `t`. A `BitVector` value (`BV d xs`) stores a list of pairs, `xs`. There is one pair, `(t,b)`, in the list for each possible tuple element, `t`, of the dimension `d`. The second element, `b`, of a pair, is an abstract boolean. If that abstract boolean represents `True` then the tuple element, `t`, is in the set, otherwise it is not. When concrete booleans are used (i.e. `BitVector Bool t`), a set is a concrete bit-vector (one bit for each possible tuple). When abstract booleans are used (i.e. `BitVector SAT t`), elements can be conditionally present in a set, depending upon the assignment of truth values to logical variables (i.e. values of the form `(VarP n)`) in the abstract boolean expression.

In addition to operations over booleans, the abstract functions `create`, `select`, `proj3of3`, and `join`, that manipulate abstract sets, are defined as instances. The missing definitions appear in Appendix A, along with the functions `combine` and `mergeL`, that play important roles in explaining how abstract sets are manipulated. The function call (`combine f (x,p) pairs`) finds the pair

Step 1. Problem Representation.

```
data SMT
  = VarE String
  | LitB Bool       -- True or False
  | LitI Int        -- 23
  | SMT :&&: SMT    -- x && y
  | SMT :+: SMT     -- x + y
  | SMT :==: SMT    -- x == y
  | SMT :<=: SMT    -- x <= y
```

Step 2. Overloading.

```
instance NumLike SMT where
  liftI = LitI
  (+) x y = x :+: y
instance BoolLike SMT where
  true = LitB P.True
  false = LitB P.False
  (&&) = (:&&:)
  liftB = LitB
instance Compare SMT SMT where
  (<=) x y = x :<=: y
  (==) x y = x :==: y
```

Step 3. Initialization.
Produces an array where each element is an SMT variable.

```
bd : Array #(width,width) i2

  1     2     3     4
 +-----+-----+-----+-----+
1|‘bd1 |‘bd2 |‘bd3 |‘bd4 |
 +-----+-----+-----+-----+
2|‘bd5 |‘bd6 |‘bd7 |‘bd8 |
 +-----+-----+-----+-----+
3|‘bd9 |‘bd10|‘bd11|‘bd12|
 +-----+-----+-----+-----+
4|‘bd13|‘bd14|‘bd15|‘bd16|
 +-----+-----+-----+-----+
```

Steps 4. Binding time analysis.

```
-- user function
add m pts = sum [m.p | p <- pts]

-- one part (for brevity) of queens constraint
and [add bd (rowPts i) == 1 | i <- width]
--->
and [add bd (rowPts i) == liftI 1 | i <- width]
```

Step 5. Constraint generation. Constraint evaluates using overloaded functions `and`, `add`, and `(==)`.

```
 (and (= (+ bd1  (+ bd2  (+ bd3 bd4))) 1)
      (= (+ bd5  (+ bd6  (+ bd7 bd8))) 1)
      (= (+ bd9  (+ bd10 (+ bd11 bd12))) 1)
      (= (+ bd13 (+ bd14 (+ bd15 bd16))) 1))
```

Step 6. Input formatting leads to SMT input file

```
(define bd1::(subtype (x::int) (or (= x 0)
                                   (= x 1))))
(define bd2::(subtype (x::int) (or (= x 0)
                                   (= x 1))))
...
(assert (and (= (+ bd1 (+ bd2 (+ bd3 bd4))) 1)
             (= (+ bd5 (+ bd6 (+ bd7 bd8))) 1)
             ...
```

Figure 4. N-Queens problem pipeline.

Step 1. Problem Representation.

```
data SAT =
     VarP Int
   | FalseP
   | TruthP
   | AndP SAT SAT

data BitVector b a = BV (Domain a) [(a,b)]
```

Step 2. Overloading. See Appendix A for full definitions.

```
instance BoolLike SAT where
  true = TruthP
  false = FalseP
  (&&) = AndP
  liftB True = TruthP
  liftB False = FalseP

instance BoolLike b => SetLike (BitVector b) where
  create d xs = ...
  select p (BV d xs) =
     BV d [(x, liftB (p x)) | (x,b) <- xs]
  proj3of3 (BV (D3 _ _ d) xs) = ...
  join (BV (D2 a b) xs) (BV (D2 _ c) ys) = ...
```

Step 3. Initialization.
Produces `BitVector SAT (Int,Int,Int)`

```
grid: set #(size,size,digit) input .. full

[(0,0,1)=p1  (0,0,2)=p2  (0,0,3)=p3  (0,0,4)=p4
 (0,1,1)=p5  (0,1,2)=p6  (0,1,3)=p7  (0,1,4)=p8
 (0,2,1)=p9  (0,2,2)=p10 (0,2,3)=p11 (0,2,4)=p12
 (0,3,1)=p13 (0,3,2)=p14 (0,3,3)=p15 (0,3,4)=T
 (1,0,1)=p16 (1,0,2)=p17 (1,0,3)=p18 (1,0,4)=p19
 (1,1,1)=p20 (1,1,2)=T   (1,1,3)=p21 (1,1,4)=p22
 (1,2,1)=T   (1,2,2)=p23 (1,2,3)=p24 (1,2,4)=p25
 (1,3,1)=p26 (1,3,2)=p27 (1,3,3)=p28 (1,3,4)=p29
 (2,0,1)=p30 (2,0,2)=p31 (2,0,3)=p32 (2,0,4)=p33
 (2,1,1)=T   (2,1,2)=p34 (2,1,3)=p35 (2,1,4)=p36
 (2,2,1)=p37 (2,2,2)=p38 (2,2,3)=p39 (2,2,4)=T
 (2,3,1)=p40 (2,3,2)=p41 (2,3,3)=p42 (2,3,4)=p43
 (3,0,1)=p44 (3,0,2)=p45 (3,0,3)=p46 (3,0,4)=p47
 (3,1,1)=p48 (3,1,2)=p49 (3,1,3)=p50 (3,1,4)=p51
 (3,2,1)=p52 (3,2,2)=p53 (3,2,3)=p54 (3,2,4)=p55
 (3,3,1)=T   (3,3,2)=p56 (3,3,3)=p57 (3,3,4)=p58]
```

Steps 4. Binding time analysis and alternate syntax expansion produces constraint.

```
full {k <- grid(3,j,k)}
-->
full (proj3of3 (select (\ (i,j,k)->i==3) grid))
```

Step 5. Constraint generation. Constraint evaluates using overloaded functions, `full`, `proj3of3`, and `select`.

```
(p45 \/ p49 \/ p53 \/ p56) /\
(p46 \/ p50 \/ p54 \/ p57) /\
(p47 \/ p51 \/ p55 \/ p58)
```

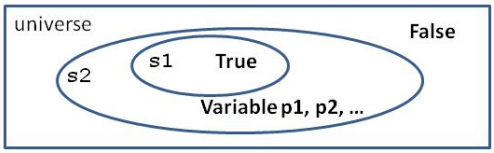
Step 6. Input formatting leads to .cnf file

```
p cnf 58 3
45 49 53 56 0
46 50 54 57 0
47 51 55 58 0
```

Figure 5. Soduko problem pipeline.

`(x,q)` in `pairs` (if any) and replaces its abstract boolean `q` with `(f p q)`. This can be used to effectively insert or delete elements depending upon the values of `p` and `f`, for example `(combine (||) (x,True) xs)` adds `x` and `(combine (&&) (x,False) xs)` removes `x`. The function `mergeL` iterates `combine`.

The Soduko problem initializes a finite set. An initializer for a finite set has the form: `Set #(d1,d2) s1 .. s2`. It includes a pair of concrete sets, `s1` and `s2`. The set `s1` must be a subset of the set `s2` and both must have the same dimensions as the set being initialized. Like all abstract sets, the initialized set consists of a list of pairs. The intuition for initialization can be seen in the picture below



Tuples in `s1` have their `BoolLike` values set to `True`, they are definitely in the set. Tuples not in `s2` have their `BoolLike` values set to `False`, they are definitely not in the set. The others are assigned a boolean valued propositional variable. Different assignments of `True` or `False` to the propositional variables will change what is in the set.

Note that the tuples given as input to the Soduko puzzle (`(0,3,4)`, `(1,1,2)`, `(1,2,1)`, `(2,1,1)`, `(2,2,4)`, and `(3,3,1)`) all have their `BoolLike` value set to `True`, and all the others are assigned a propositional variable. This is because the set `s2` is the full set of tuples (the universe). It is not unusual for an initializer to be: `Set #(d1,d2) none .. universe`, which is completely unconstrained (i.e. every tuple is assigned a propositional variable). But, choosing appropriate bounds can dramatically decrease the size of the problem sent to the solver, and thus effect its efficiency. Other initializers (not shown) allow users to describe symmetries[5, 23], which also can make the search process more efficient.

In the Soduko example, one part of the constraint is: `full {k <- grid(3,j,k)}`. The alternate notation expands to a term involving the overloaded functions `full`, `proj3of3` and `select`. No binding time annotations are needed. This compares a one column projection to the full set `{1,2,3,4}`.

The term is evaluated in an overloaded context to get a `SAT` term. To see how the answer arises, consider its first conjunct `(p45 \/ p49 \/ p53 \/ p56)`. A `2` is in the set if and only if this conditions holds. The only tuples in row `3` with a `2` in the `k` position are:
`(3,0,2)=p45 (3,1,2)=p49 (3,2,2)=p53 (3,3,2)=p56`.
So, one of the variables `p45`, `p49`, `p53`, or `p56` must be true. In a similar fashion, the second and third conjuncts assure `3` and `4` are also in the set (work it out for yourself). What about `1`? Why no conjunct for `1`? Because the tuple `(3,3,1)=T` is already fixed by the input, so `1` will always in the set `{k <- grid(3,j,k)}`.

This representation is then formatted into a standard .cnf file and passed to the sat solver.

4.3 The Production problem

The Production problem is solved by a Linear Programming solver. The 7-step process is illustrated in Figure 6. It uses a structural representation (we call `Mexp`, for *M*athematical programming) to represent polynomials over several variables. For example: `3x + 2y + 1`. Operations, like `(+)`, combine polynomials. For example: `(3x + 2y + 1) + (2x + 2z + 3)` results in `(5x + 2y + 2z + 4)`.

Step 1. Problem Representation.

```
type PolyNom n = [(String,n)]
data MExp n = Term (PolyNom n) n
data Rel n
  = RANGE (PolyNom n) (Range n)
  | TAUT   -- True
  | UNSAT  -- False
```

Step 2. Overloading. See Appendix A for full definitions.

```
instance NumLike (MExp Int) where
  liftI n = Term [] n
  (+) (Term [] a) (Term [] b) = ...
instance BoolLike [Rel Int] where
  liftB True = [TAUT]
  (&&) xs ys = ...
instance Compare (MExp Int) [Rel Int]  where
  (<=) (Term [] a) (Term [] b) = ...
```

Step 3. Initialization. Array filled with unit polynomials.

```
prod: Array #(Factory,Store) Int

    NYC       ATL       LA
 +---------+---------+---------+
A|`(1a + 0)|`(1b + 0)|`(1c + 0)|
 +---------+---------+---------+
B|`(1d + 0)|`(1e + 0)|`(1f + 0)|
 +---------+---------+---------+
C|`(1g + 0)|`(1h + 0)|`(1i + 0)|
 +---------+---------+---------+
: Array (Factory,Store) (MExp Int)
```

Steps 4. Binding time analysis. Constraint is annotated.

```
and [ sales.s == sum [prod.(f,s) | f <- Factory]
    | s <- Store ]
-->
and [liftI(sales.s)==sum[prod.(f,s) | f<-Factory]
    | s <- Store ]
```

Step 5. Constraint generation. Constraint evaluates using overloaded functions. Transformed to coefficient matrix.

```
230 <= (1a + 1d + 1g) <= 230
140 <= (1b + 1e + 1h) <= 140
300 <= (1c + 1f + 1i) <= 300

         a b c d e f g h i
+----+--+-+-+-+-+-+-+-+-+---+---+
|230 |<=|1|0|0|1|0|0|1|0|0|<= |230|
+----+--+-+-+-+-+-+-+-+-+---+---+
|140 |<=|0|1|0|0|1|0|0|1|0|<= |140|
+----+--+-+-+-+-+-+-+-+-+---+---+
|300 |<=|0|0|1|0|0|1|0|0|1|<= |300|
+----+--+-+-+-+-+-+-+-+-+---+---+
```

Step 6. Input formatting leads to .mps file

```
NAME          prod
ROWS
 N  COST
 E  R1
 E  R2
 E  R3
COLUMNS
    a         COST                 2
    a         R1                   1
...
ENDATA
```

Figure 6. Production Problem pipeline.

The type `[Rel n]` represents a boolean term. An element of such a list is a ternary relation, bounding a polynomial from above and below. For example: $-\infty$ `<= (3x + 2y) <= -3`.

Comparisons build these relations. E.g. a constant comparison `(3x + 2y + 1) <= 7`, produces: $-\infty$ `<= (3x + 2y) <= 6`.

More complex, comparing two polynomials
`(3x + 2y + 5) <=  (y + 5z + 2)` becomes
$-\infty$ `<= (3x + y - 5z) <= -3`

Using this representation of terms, an abstract boolean is a list of ternary relations. To overload the operator `(&&)`, the two sets are unioned, by combining elements with a common polynomial, by "squeezing" the lower and upper bounds.

{$-\infty$ `<= (3x) <= 6`} `&&` { `4 <= (3x) <= 9`} becomes
{ `4 <= (3x) <= 6` }

The full definitions for the overloaded operations in the abstract instance step are found in Appendix A.

The production problem, like the Soduko problem, initializes to a real array storing abstract data. Here each array cell holds a unit polynomial over a different variable. A unit polynomial has only one variable (with a coefficient of 1) and an additive constant of 0.

In the production problem, we illustrate constraint generation using the last of the three constraints. Binding time analysis recognizes that the sub term `sales.s` is completely static (it will reduce to a constant, like `230`) so it is annotated with `liftI`.

Each of the sums leads to a bounded polynomial. From these bounded polynomials an array of coefficients is constructed. Note that for each polynomial, only some of the coefficients are set to `1`, these correspond to the entries in `prod` for the same store (i.e. the same column in the abstract array).

The array of coefficients is formatted as a standard .mps file.

5. Possible extensions

Additional solvers. The first and most natural extension is to find and incorporate additional solvers. The linear-programming solver framework can be generalized to solve problems over real or rational numbers, or to allow non-linear constraints. We are looking for suggestions for new kinds of solvers.

Allowing programmers to add solvers. A more useful extension would be to add language features that allow programmers to add their own solvers. Currently the Funlogic compiler must be hacked to add a new solver. The design principles specify the steps necessary. To allow programmers to add solvers, each step would have to be internalized. Currently the language is a simple call by value language. Users write no type information at all, and every term is given, what appears to be a simple Hindley-Milner type.

In reality, 2-stage overloaded types are actually used. To internalize the 7 steps the language would have to be much more sophisticated. It would need overloading and classes, staging annotations, and more. All this would have to accessible to the programmer. While I believe this is possible. I have not built it, yet.

What should we search for? New solvers expand the design principles to accommodate new ideas. The linear programming solver is a case in point. The other solvers can return any solution that meets the constraints. The LP solver must find a solution that maximizes the objective function. When we first embedded the LP solver we had to generalize the notion of what we were solving for. This led us to add the `find` clause to an existential declaration. Now one can `find` the `First` (SAT,SMT,Narrowing), or the `Max` or `Min` (LP) solution. This led us to think about other possible ways of describing what we should search for. We have currently added two

other `find` modes: `Many` and `Abstract`. We envision these being used as design exploration tools by the programmer. They help the programmer design his program by exploring the design space of possible constraints.

The `Abstract` mode allows the user to visualize the results of *initialization*. Rather than generate and find a solution that meets the constraint. It prints out the constraint and binds the existentially quantified variables to their initializations. This lets the programmer use the read-eval-print loop to visualize the results of potential constraints (much the way we did in Figures 4, 5, and 6, which were in fact created using the `Abstract` mode).

The (`Many x`) mode solves the constraint, and then instantiates and prints out `x` under the solution. It then pauses and goes into an interactive loop that allows the programmer to type in additional constraints. These constraints are then added to the original ones and the process is repeated. This allows the programmer to incrementally develop what constraints are necessary. Over or under constraining problem specifications is common, and using this mechanism allows exploration of the possibilities. Other possibilities of design exploration include

- **Populating large constrained data structures.** Given a data structure and a bunch of constraints one can explore questions like: Is there a value which meets all the constraints? What does a value look like? What if I add an additional constraint?
- **Model Checking.** Given a data structure and a bunch of constraints is it true that every solution (or model) has additional properties.
- **Test generation.** Find some input data that forces a computation to go down a certain path for testing purposes.

The existential declaration allows programmers to explore these design parameters from within the language. No need for an external tool or analysis.

6. Related work

Funlogic embeds multiple solvers in a general purpose language. This combination is rare. Several systems, self described as modelling languages, such as AMPL [8] and GAMS, support multiple solvers. But, they are not programming languages, just a convenient notation to write down the math that describes a problem. The solvers they support all concern minimizing or maximizing an objective under various kinds of constraints. I know of two systems that embed solvers in database languages, based upon Datalog, by using the notion of "plugin" [14, 19]. But neither provides strong scripting capability.

Recently, Kuncak et al.[16] introduced the idea of a software synthesis procedure, where code is synthesized at compile time by the use of a decision procedure. The synthesized code, when executed at runtime, will provide values for existentially defined variables. They introduce the notion of a formula, a syntactic subset of boolean valued terms, for which the decision procedure knows how to synthesize code. In this paper we demonstrate how overloading can both replace the syntactic restriction with constrained types, and broaden the class of decision procedures applicable. In more recent work[15] they have strengthened their decision procedures by extending them with a notion of symbolic evaluation of user defined functions.

While Funlogic is a unique collection of ideas, I would be remiss if I did not acknowledge many fine papers which strongly influenced my thinking in its design.

The work of Daniel Jackson[11, 29] and his student Emina Torlak[28] first opened my eyes to the fact that non-trivial specifications (all of relational algebra) could be expressed in first order logic. The thesis by Toni Mancini[18], strongly reinforced this fact.

The Curry language[9], developed by Michael Hanus and Sergio Antoy was also influential. The recent paper[4] *A New Compiler from Curry to Haskell*, explained to me exactly how Curry fits within the design principles developed in this work (though I didn't realize it at the time I first implemented a narrowing solver).

I first encountered the use of overloading to generate constraints in the paper *Logical Abstractions in Haskell*[7] by Nancy A. Day, John Launchbury, and Jeff Lewis. It took a while for me to realize that this could be generalized from boolean constraints to constraints over other domains, such as numeric domains, or even algebraic data structures. My knowledge of how to type and implement overloading comes mainly from the fine paper *Typing Haskell in Haskell*[13] by Mark Jones.

One key element necessary to use overloading as a constraint generation mechanism is effective initialization of existentially introduced variables. Good initialization abstracts over aggregates (allowing the user to declare one array, rather than many individual variables), and chooses good representations that minimize the problems to be solved. My approach to initialization was strongly influenced by the small check system[21] (which uses a type-based system to generate all "small" values of a given type) , and by conversations with Emina Torlak. Emina taught me the bounding trick for finite set initialization, and the importance of using symmetry in initialization.

Binding time analysis was the last piece of the puzzle. Experience with MetaML[26, 27] made it possible to recognize a binding time problem when I saw one. The two binding analyses I have experimented with include an interpreted approach based upon normalization by evaluation[1, 6], and an approach based on some work by a former student, Nathan Linger[22], which is outlined in Appendix B. Interesting enough, a third approach[17], outlined by Linger, performs the analysis, not by abstract interpretation, but by reducing the problem to a boolean SMT problem. So we have come full circle.

7. Performance

I built the system as a proof of concept, not to optimize performance. There are lots of possibilities. Here are a few baseline timings for the programs in the paper: 4 queens a few hundredths of a second; 8 queens in a second; 10 queens time out (the limit is 60 seconds). Rank 2 Soduko in a few hundredths of a second; Rank 3 Soduko (the normal 9x9) in about 3 seconds; Rank 4 Soduko times out.

But, by changing the initializer for SAT, and swapping in a specialized formula to CNF pass, a student solved rank 7 (49x49) puzzles. Using specialized representations makes a difference, and giving users programmers access to these is important. We leave this to future work

8. Conclusion

The existential declaration is an expressive abstraction, bridging the functional and logic worlds, for many different kinds of problems, solvable by a wide variety of decision procedures. Implementing existential declarations over a wide variety of domains requires embedding multiple decision procedures. Fortunately, the steps involved can be can be precisely described. Overloading and staging are the key ingrediants to giving precise semantics to the embedding process.

Acknowledgments

There are many people who helped me in my research. First I would like to thank Molham Aref and Emir Pasalic who got me started thinking about other ways to think about declarative programming, and LogicBlox (a great place to work in Atlanta Ga.)

which partially supported this research. I would also like to thank Jim Hook (my co-teacher) and all the class members of the Winter 2011 class *Mathematical Logic via Foundational Algorithms* at PSU that helped refine my thinking about how to combine logic and functional programming. This work was also supported in part by NSF grant 0910500.

References

[1] V. Balat and O. Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. *Lecture Notes in Computer Science*, 1473:240–252, 1998. ISSN 0302-9743.

[2] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 3rd annual satisfiability modulo theories competition (SMT-comp 2007). *International Journal on Artificial Intelligence Tools*, 17(4): 569–606, 2008.

[3] G. M. Bierman, A. D. Gordon, C. Hritc, and D. Langworthy. Semantic Subtyping with an SMT Solver. TechReport MSR-TR-2010-99, Microsoft Research, Dec. 2010. URL `http://research.microsoft.com/en-us/projects/dminor/`.

[4] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from curry to haskell. In H. Kuchen, editor, *WFLP*, volume 6816 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011. ISBN 978-3-642-22530-7. URL `http://dx.doi.org/10.1007/978-3-642-22531-4`.

[5] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.

[6] O. Danvy, M. Rhiger, and K. H. Rose. Normalization by evaluation with typed abstract syntax. *J. Funct. Program*, 11(6):673–680, 2001.

[7] N. A. Day, J. Launchbury, and J. Lewis. Logical abstractions in haskell. In *Proceedings of the 1999 Haskell Workshop*. Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, October 1999.

[8] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL – A Modeling Language for Mathematical Programming*. The Scientific Press, South San Francisco, 1993.

[9] M. Hanus. *Report on Curry (ver.0.8.2)*. Inst. fur Informatik, Christian-Albrechts Universitat, .de, 2006.

[10] M. Hermenegildo and T. CLIP Group. An Automatic Documentation Generator for (C)LP – Reference Manual. The Ciao System Documentation Series–TR CLIP5/97.3, Facultad de Informática, UPM, Aug. 1997. URL `http://clip.dia.fi.upm.es/Software/Ciao/`. Online at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[11] D. Jackson. An intermediate design language and its analysis. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 121–130, New York, Nov. 3–5 1998. ACM Press.

[12] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., 2006.

[13] M. P. Jones. Typing Haskell in Haskell. In *ACM Haskell Workshop* , informal proceedings, Oct. 1999.

[14] D. Klabjan, R. Fourer, and J. Ma. Algebraic modeling in a deductive database language. In *11th INFORMS Computing Society Conference*, 2009.

[15] A. S. Koksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL '12, Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 151–164. ACM, 2012.

[16] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Communications of the ACM*, 55(2):103–111, Feb. 2012. ISSN 0001-0782 (print), 1557-7317 (electronic). doi: http://dx.doi.org/10.1145/2076450.2076472.

[17] N. Linger and T. Sheard. Binding-time analysis for metaML via type inference and constraint solving. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 266–279. Springer, 2004. ISBN 3-540-21299-X.

[18] T. Mancini. *Declarative constraint modelling and specification-level reasoning*. Diploma thesis, Universita degli Studi di Roma 'La Sapienza', 2004.

[19] D. Z. Molham Aref and E. Pasalic. Using optimization services in datalog. In *11th INFORMS Computing Society Conference*, 2009.

[20] M. Novak, G. Gardarin, and P. Valduriez. Flora: A functional-style language for object and relational algebra. *Lecture Notes in Computer Science*, 856:37–46, 1994. ISSN 0302-9743.

[21] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices*, 44(2):37–48, Feb. 2009. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). doi: http://doi.acm.org/10.1145/1543134.1411292.

[22] T. Sheard and N. Linger. Search-based binding time analysis using type-directed pruning. In *ASIA-PEPM*, pages 20–31, 2002. URL `http://doi.acm.org/10.1145/568173.568176`.

[23] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007. URL `http://dx.doi.org/10.1016/j.dam.2005.10.018`.

[24] G. Smolka. The definition of kernel oz. Technical report, Saarländische Universitäts- und Landesbibliothek; Sonstige Einrichtungen. DFKI Deutsches Forschungszentrum für Künstliche Intelligenz, 1994. URL `urn:nbn:de:bsz:291-scidok-37290`.

[25] G. Sutcliffe and C. B. Suttner. The CADE ATP system competition. In D. A. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *Lecture Notes in Computer Science*, pages 490–491. Springer, 2004. ISBN 3-540-22345-2.

[26] Taha and Sheard. MetaML and multi-stage programming with explicit annotations. *TCS: Theoretical Computer Science*, 248, 2000.

[27] W. Taha and T. Sheard. MetaML and multi-stage programming with, Feb. 09 1999. URL `http://citeseer.ist.psu.edu/516106.html; http://cse.ogi.edu/ walidt/paper-2.ps`.

[28] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007. ISBN 978-3-540-71208-4.

[29] E. Torlak, A. Prof, and D. Jackson. Thesis: A constraint solver for software engineering: Finding models and cores of large relational specifications., Dec. 04 2008. URL `http://stuff.mit.edu/people/emina/papers/etorlak-cv.pdf`.

A. MP abstract instance functions

In this appendix are the missing functions for the abstract instance declarations in Figures 5 and 6.

```
mergeP f [] ys = ys
mergeP f xs [] = xs
mergeP f ((x,n):xs)((y,m):ys)=
 case compare x y of
  EQ -> case (f n m) of
         0 -> mergeP f xs ys
         i -> (x,i):mergeP f xs ys
  LT -> (x,n):
        mergeP f xs ((y,m):ys)
  GT -> (y,m):
        mergeP f ((x,n):xs) ys

combine oper(t,b)[] = []
combine oper(t,m)((s,n):xs)| t==s = (t,oper m n): xs
combine oper(t,m)((s,n):xs)=(s,n):combine oper(t,m)xs

mergeL oper [] ys = ys
mergeL oper ((z,m):xs) ys =
  mergeL oper xs (combine oper (z,m) ys)

instance NumLike (MExp Int) where
  liftI n = Term [] n
```

```
  (+) (Term [] a) (Term [] b) = Term [] (a+b)
  (+) (Term [] a) (Term ys b) = Term ys (a+b)
  (+) (Term xs a) (Term [] b) = Term xs (a+b)
  (+) (Term xs a) (Term ys b)
      = Term (mergeP (+) xs ys) (a+b)

instance Compare (MExp Int) [Rel Int]  where
  (<=) (Term [] a) (Term [] b) =
       if (a <= b) then [TAUT] else [UNSAT]
  (<=) (Term [] a) (Term xs b) =
       [RANGE xs (Range (LtEQ(a-b)) PlusInf)]
  (<=) (Term xs a) (Term [] b) =
       [RANGE xs (Range MinusInf (LtEQ (b-a)))]
  (<=) (Term xs a) (Term ys b) =
       [RANGE (mergeP (+) xs (negPoly ys))
              (Range MinusInf (LtEQ (b-a)))]

instance BoolLike [Rel Int] where
  liftB True = [TAUT]
  liftB False =[UNSAT]
  true = [TAUT]
  false = [UNSAT]
  (&&) xs ys = help (sort xs) (sort ys)
       where help (UNSAT:_) ys = [UNSAT]
             help xs (UNSAT:_) = [UNSAT]
             help (TAUT: xs) ys = help xs ys
             help xs (TAUT: ys) = help xs ys
             help [] ys = ys
             help xs [] = xs
             help (RANGE x a:xs) (RANGE y b:ys)
               | x P.== y
               = RANGE x (intersectRange a b):
                 help xs ys
             help (RANGE x a:xs)(ys@(RANGE y b:_))
               | x < y
               = RANGE x a:(help xs ys)
             help (xs@(RANGE x a:_))(RANGE y b:ys)
               | x > y
               = RANGE y b:(help xs ys)


instance BoolLike b =>
         SetLike (BitVector b) where
  create d xs =
     BV d [(t,liftB(elem t xs)) | t <- tuples d]
  select p (BV d xs) =
     BV d [(x, liftB (p x)) | (x,b) <- xs]
  proj3of3 (BV (D3 _ _ d) xs) =
     BV (D1 d)
        (mergeL (||)
                [(z,b) | ((x,y,z),b) <- xs]
                [(x,false) | x <- d])
  join (BV (D2 a b) xs)(BV (D2 _ c) ys)= BV d3 ans
    where d3 = D3 a b c
          ans = mergeL (||)
                       [ ((a,b,c),p&&q)
                       | ((a,b),p) <- xs
                       , ((x,c),q) <- ys
                       , x P.== a ]
                       [(t,false)| t <- tuples d3]
```

B. Staging type inference

In this section we provide further details of the staging type-inference process discussed in Paragraph 4 of Section 4. Lets start with a description of syntax. I keep everything very simple here. The real language has more, but this simple version extends naturally.

```
constants:     i ::= {..., -1,0 +1, ...}
               b ::= {True,False}

class library: A ::= NumLike t   | BoolLike t
                   | Compare t t | SetLike t

types:         T,t ::= Int | Bool | t -> t | x

schemes:       S ::= forall xs . A => T

terms:         E,f,e ::=  v | f e | i | v#A | b
```

Each class library has a number of overloaded methods. See Figure 3 for details. In the staging type-inference process, we will need to compute a type from an overloaded operator and a class library.

```
libType:: v -> A -> T
libType (+)   (NumLike t)   = t -> (t -> t)
libType liftI (NumLike t)   = Int -> t
libType (==)  (Compare t b) = t -> (t -> t)

instan (forall xs . A => T) = (A[ts/xs],T[ts/xs])
```

Where the `ts` in `instan` are fresh type variables. The staging type-inference process is a syntax directed walk over a term in the presence of an environment, `s:: v -> S`, that maps variable names to schemes. The judgment `s |- E --> (E',T)` means, under `s` the term `E` has type `T`, and reconstructs to `E'`. The process is strongly reminiscent of type inference in the presence of class constraints[13] and search based binding time analysis[22].

```
---------------------------------------- LIB
s |-  x#A --> (X,libType x A)


fresh t
---------------------------------------- INT
s |-  i   --> (liftI#(NumLike t) i, t)


fresh t
---------------------------------------- BOOL
s |-  b   --> (liftB#(BoolLike t) b, t)


instan(s v) --> (C,t)
---------------------------------------- VAR
s |- v  -->  (v#C v,t)


fresh w
unify dom xt
s |- x --> (x',xt)
s |- f --> (f',d -> r)
t = (f' x')
---------------------------------------- APP
s |- f x --> (cast xt d r)
 where cast I d r = (liftI#(NumLike w) t,w)
       cast B d r = (liftB#(BoolLike w) t,w)
       cast n d r = (f' x',r)
```

Three points are worth making. First, in the rules `INT` and `BOOL` every constant is lifted to an overloaded type. Second, in the rule `VAR` every overloaded variable is instantiated and lifted to a fresh type. Third, in the rule `APP` if an argument of an application has a concrete type (`I` for `Int` and `B` for `Bool`), then the reconstructed call is lifted to a fresh type.