FunLog

Example problems

- Combinatorial auction
  - sell all the items
- Towers of Hanoi
- Rectangle packing
- Shortest route.
- 8 queens
- Soduko
- Maximizing (minimizing) costs

Finding a solution with given property

- The property relates known entities with parts of the solution.
- The property ensures that the solution is useable
- The property can be expressed as a small higher order function.
- The problem combines computation and search.

Computational Modality

- Evaluate (reduction)
  - Modality of languages like: C, Haskell, Datalog
- Find (Existential search)
  - Modality of languages like: Prolog, Alloy, IDP
- Combined
  - Curry: both reduction and search via Narrowing.

Modality v.s. Expressivity via Language

	Evaluate	**Find**
Tuple	Haskell, C, …	Prolog
FiniteSet	Datalog	IDP, Alloy
Algebraic	Haskell, ML, Curry	Curry
Array	C, Fortran,	

Language via Algorithm

Language	Computational Algorithms
Prolog	Backtracking, unification
Haskell, C, ML	Reduction
Datalog	SemiNaive fixpoint evaluation
Curry	Narrowing
Alloy	SAT, symmetry
IDP	SAT, grounding

FunLog

- FunLog is a language designed for a mixed modal language
- Data
  - Int, Bool   (eval & find)
  - Pressburger Arithmetic   (eval & find)
  - Tuples   (eval & find)
  - FiniteSets   (eval & find)
  - Algebraic Data   (eval only)
- Succinctness - $\lambda-$calculus expressions and datalog formula (denotes SPJ operations on sets)
- Abstraction – lexically scoped lambda calculus can abstract over anything.
- Computation modality is overloaded and determined by context.

Evaluate

```
dim i4    = [0,1,2,3]
input     = set (i4,i4,i4) [(0,3,3),(1,1,1),(1,2,0),(2,1,0),(2,2,3),(3,3,0)]
quadrantL =
  [(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,2),(1,0,3),(1,1,2),(1,1,3)
  ,(2,2,0),(2,2,1),(2,3,0),(2,3,1),(3,2,2),(3,2,3),(3,3,2),(3,3,3)]
quadrant  = set (i4,i4,i4) quadrantL
```

			3
	1	0	
	0	3	
			0

Find: grid(i,j,n).
Where: input(i,j,n) <= grid(i,j,n).
Such That:

```
   full grid(0,j,k) & full grid(1,j,k) & full grid(2,j,k) & full grid(3,j,k)
 & full grid(i,0,k) & full grid(i,1,k) & full grid(i,2,k) & full grid(i,3,k)
 & full quadrant(0,i,j),grid(i,j,k) & full quadrant(1,i,j),grid(i,j,k)
 & full quadrant(2,i,j),grid(i,j,k) & full quadrant(3,i,j),grid(i,j,k)
 & grid(i,j,n) | (i,j) -> k .
```

Syntax

- FunLog is a declarative language
- Declarations introduce new named objects
  - name(x,y) <- formula
    - Rules introduce Finite Sets (Relations)
  - name = expression
    - Equations introduce values
  - Exists name Where: _ SuchThat: _
    - Introduces Search, name is a lazy list
- Functions (lambda abstractions) can abstract over any named object.

Notation

- Funlog uses two different notations
  - Functions (like Haskell) Expressions
  - Relations (like Prolog and Datalog) Formulas
- The two notations use different conventions to determine the scope of a variable.
- One switches from one notation to the other by the use of the escape ($) operator,

Functional - Expressions

- Expressions denote a value
- A value can be many things
  - A primitive Int, Float, Char, String, Boolean,
  - A tuple of values (4,True,even)
  - A function
  - An algebraic data type.
  - A finite set

Example expressions

- Literals - 5, 2.3, "abc“
- Variable – x, date, tail
- Function calls – (f x 5)
- Lambda abstraction (\ x -> x + 3)
- Tuples – (2,3)
- Sets – set #(dim,width) [(2,”a”)]
- Comprehensions [x + 4 | x <- [2..6]]

Relational - Formulas

- A formula denotes a finite set of tuples that range over primitive data.
- An Atomic formula (atom) is a relation symbol followed by a parenthesized list of patterns.
  - R(p1,p2,p3) the largest subset of R where each element of a tuple (a,b,c) matches the patterns.
  - I.e. a matches p1, b matches p2, and c matches p3.
- Compound formulas

Compound formulas

- Conjunction
  – son(y) <- father(x,y), male (y)
- Disjunction
  – parent(x) <- father(x,y); mother(x,z)
- Negation
  – !father(x,y)
- Projection
  – {(y,x) <- r(x), z(x,y,z)}

Lexical Scoping

- The normal rules of lexical scoping apply to the expression part of the language.
- Rules and formula use implicit conventions to determine scoping.
- f (xi ..) <- rhs
  - f is introduced by the rule, and is in scope in rhs
  - Free variables in the xi are universally quantified and are bound in rhs
  - Free variables in rhs are existentially scoped and are bound in rhs.
  - So how do we “import” variables bound in an outer scope?

The Escape ($) annotation

```
transClosure f =
     let anc(x,y)
          <- $f(x,y);
             $f(x,z),anc(x,y). in anc
```

```
row n x = let f(k) <- $x($n,j,k). in f
col n x = let f(k) <- $x(i,$n,k). in f
```

Dimensions

- Dimensions a finite sets over scalar data
  - Int, float, char, string, Bool, and enumerations
  - dim small#Int [0,1,2,3]
  - data week = Sun | Mon | Tue | Wed | Thu | Fri | Sat
- Dimensions can be multidimensional
  - #(small,week)
- Dimensions are used to limit the elements in finite sets
  - Set #(small,week) [(0,Mon), (1,Tue)]

Materializing functions in small domains

```
dim i6 = [0,1,2,3,4,5]

lift1 d f =
  set (d,d) [ (x, f x) | x <- d ]
lift2 d f =
 set (d,d,d)
     [ (x,y,f x y) | x <- d, y <- d ]

plus = lift2 i4 (+)
minus = lift2 i6 (-)

f(x,y) <- g(x,i),h(y,j),plus(i,j,7).
```

Language Adjectives

- Expressive
  - What can the language compute
- Succinct
  - How many key-strikes does it take to write it
- Abstract
  - Finding patterns, naming them and re-using them
  - Functional abstraction is one example
  - Modality abstraction is another

Datalog v.s. Relational Algebra

- Datalog and Relational Algebra are equally expressive.
- Datalog is more succinct.

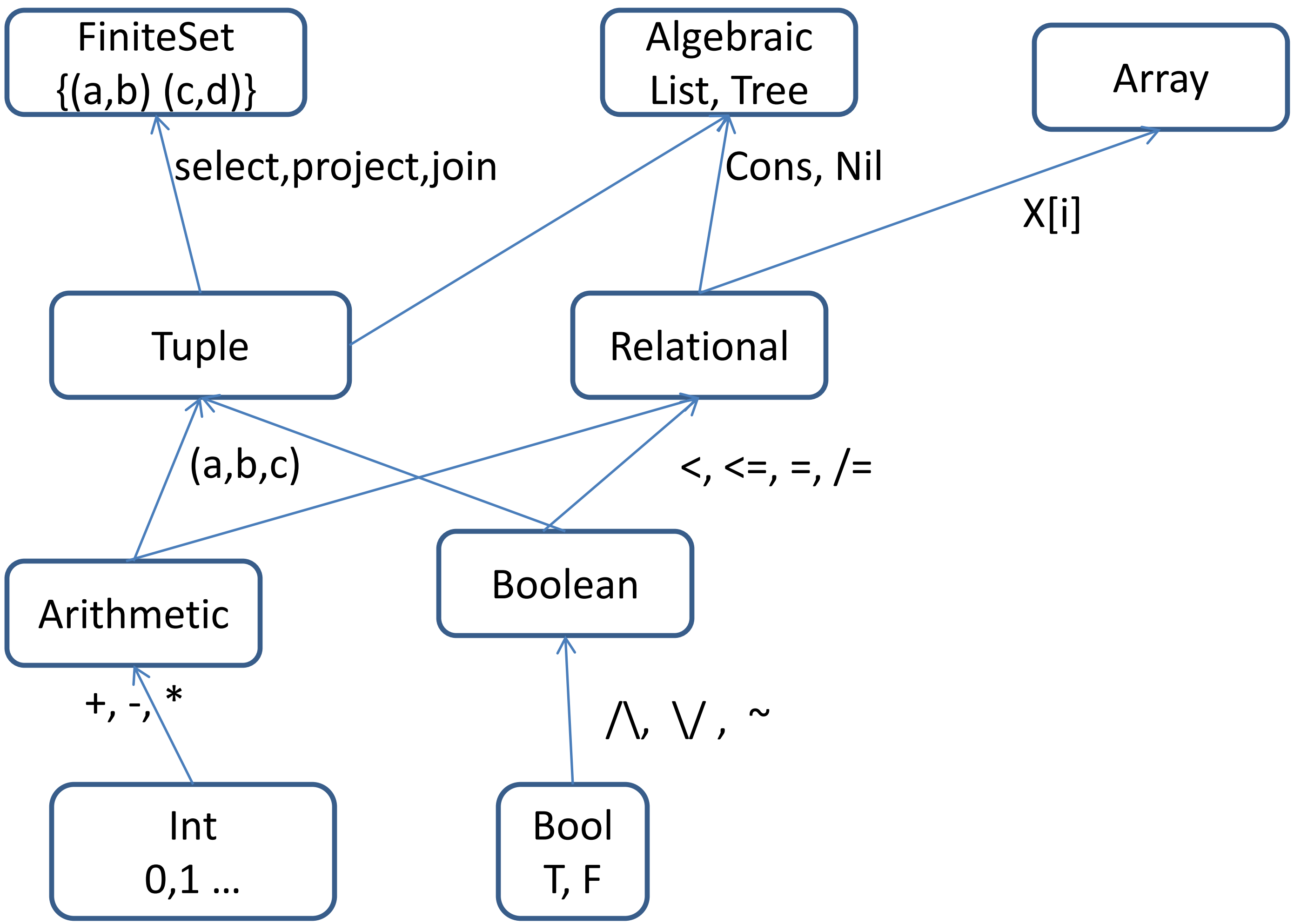
```
parent(x,y),parent(y, ``Tom”)

vs

select ((x,y)->y==``Tom”)
       (Join (project ((y,z)->(z,y)) parent)
             parent)
```

- Neither is abstract over transitive closure

An Expressivity Hierarchy



Points to note

- Its is a real hierarchy
- Any point lower in the hierarchy can be lifted to a point higher in the hierarchy
- Computations lower in the hierarchy always have translations into richer computations higher in the hierarchy

Functional Abstraction & the λ–calculus

- Find a pattern, name it, and reuse it

```
inRange x lo hi =  lo <= x  &&  x <= hi
inRange 5 2 6 → T
inRange 7 2 6 → F
```

- Not all languages have this kind of abstraction

```
anc(x,y) <- parent(x,y); parent(x,z),anc(z,y).

reach(x,y) <- path(x,y); path(x,z),reach(z,y).
```

Modality abstraction

- A term of type Bool can be interpreted as
  - A set of reduction steps to get T or F
  - A specification for a search based tool like minisat

- By using constrained types, its is possible to overload a term to do both things.
- The context of the term determines its modality.
- A value in the Evaluate modality is a value in the Find modality (the search is trivial)

A language with modality abstraction

```
Evaluate
dim i10 = [0,1,2,3,4,5,6,7,8,9]
dim colors = ["Red","Blue","Green","Yellow"]

graph = [(1,2),(2,3),(3,4),(4,5),
         (5,1),(1,6),(2,7),(3,8),
         (4,9),(5,0),(6,8),(7,9),
         (8,0),(9,6),(0,7)]

edges = set (i10,i10) graph
color = toSet colors
twoHop(x,y) <- edges(x,z),edges(z,y).

Find coloring(n,c)
Where  same(x,y,c) <- color(c)coloring(x,c),
                      edges(x,y),coloring(y,c).
Such That: none same(x,y,c) & full ( w(n) <- coloring(n,c) ).
```

This section evaluated

This section uses search

Mixed modal expressions

Mixed Modality

- Operators for each point in the Expressivity hierarchy are given over loaded types.
- Mode of use determines how they are interpreted.
- Automatic conversion from Evaluate -> Find
- Conversion from Find to Evaluate is non deterministic. I.e. a search may find many results. Answers are encapsulated in a lazy list. New answers are computed only on demand.

Overloaded Boolean

```
class Boolean b where
  true :: b
  false :: b
  isTrue :: b -> Bool
  isFalse :: b -> Bool
  conj:: b -> b -> b       -- conjunction
  disj:: b -> b -> b       -- disjunction
  neg:: b -> b             -- negation
  imply:: b -> b -> b      -- implication
```

Pressburger Arithmetic

```
class (Num n) => Arithmetic n where
  lit:: Int -> n
  (+):: n -> n -> n
  (-):: n -> n -> n
  (*):: Int -> n -> n

class (Arithmetic n,Boolean b) =>
      (Relational f n b) where
  (<) :: f n -> f n -> f b
  (<=):: f n -> f n -> f b
  (=) :: f n -> f n -> f b
  (/=) :: f n -> f n -> f b
```

FiniteSet Examples

```
select::
   (Boolean b) =>
   ([Int] -> Bool) -> FiniteSet b -> FiniteSet b
project ::
   (Boolean b) =>
   [Int] -> FiniteSet b -> FiniteSet b
join::
   (Boolean b) =>
   Int -> FiniteSet b -> FiniteSet b -> FiniteSet b
none:: (Boolean b) => FiniteSet b -> b
some:: (Boolean b) => FiniteSet b -> b
funDep::
   (Boolean b) =>
   [Int] -> [Int] -> FiniteSet b -> b
```

Using the hierarchy

- Every term has an overloaded type.
- Every instance of the overloaded type determines a computation strategy.

```
range e lo hi =
    conj (lo <= e) (e <= hi)

range::
 (Relational f n b, Boolean (f b)) =>
  f n -> f n -> f n -> f b
```

```
instance Boolean(Value Bool)
instance Relational Value Int Bool
```

Given the overloaded type

```
range::
 (Relational f n b, Boolean (f b)) =>
  f n -> f n -> f n -> f b
```

Used at the instances above

```
range 6 4 10  ->    True
```

```
instance Boolean(SMT Bool)
instance Relational SMT Int Bool
```

Given the overloaded type

```
range::
 (Relational f n b, Boolean (f b)) => f
  n -> f n -> f n -> f b
```

Used at the instances above

```
range x1 x2 x3    ->
(x2 <= x1) /\ (x1 <= x3)
```

Mixed Computation

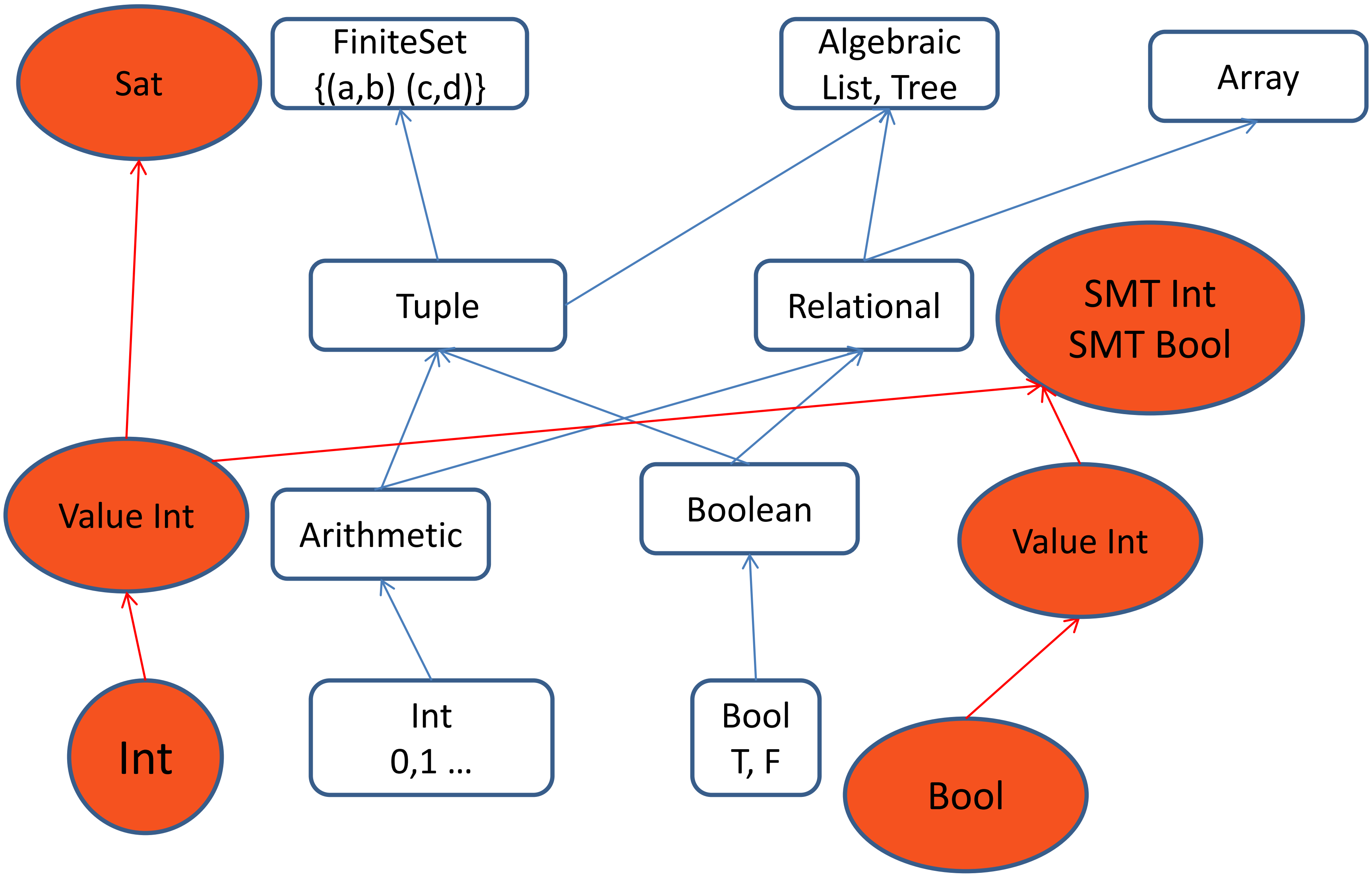
Overloaded `xs, conj, true, /=`

Not overloaded `less`

```
distinct xs =
    foldr conj true
    [i /= j | i<-xs, j<-xs, less i j]

distinct [x1,x2,x3] :: SMT Bool
(x1 /= x2) /\ (x1 /= x3) /\ (x2 /= x3)
```

Current points in the hierarchy



Conclusions

- Abstracting over computational modality is a good thing
- Eval modality can always be lifted to Find
- Find can be lifted to Eval using lazy lists
- Constrained types isolate exactly the expressivity needed to state the problem
- Use the lowest tool (known instance of the constrained type) to solve the problem
- Functional abstraction is a great glue to tie together many different approaches.
- Materializing functions in small domain lets us add arithmetic to the FiniteSet expressivity point for free.