The Structure of Domain Theories

A **domain theory** (or abstract knowledge structure) consists of a domain of objects, and axioms and rules which define the symbolic interaction between the symbolic form of these objects. In particular, a domain theory consists of:

1. A collection of *symbols*, including
   constants
   variables naming arbitrary forms
   functions
   relations

2. *Generation axioms*
   These define the typing hierarchy of forms

3. *Uniqueness axioms*
   These define how forms stay the same when they are manipulated, and how forms are composed of atomic units.

4. *Special axioms*
   These define the characteristics of special types.

5. A*n Induction Principle*
   This rule template is the mechanism which allows construction and deconstruction of arbitrary forms, and provides an algebraic (abstract) approach to domain forms.

For proof and for programming, several **composition** tools are then proved/provided for construction and deconstruction.

6. D*ecomposition*
   Permission to take apart an arbitrary form into atomic components and functions to do the construction/deconstruction.

7. E*quality under Decomposition*
   Equal forms don't change if you do equivalent things to them. Generally, forms are **mappable**, you can map a function across the atomic parts.

8. *Special functions as theorems*
   With the above basis (1-7), we now begin to build specialized functions (macros) which make it easier to take large steps while manipulating forms. A recursive definition axiom says what we mean by the new function in terms of the basis functions. Then other theorems relate all the other mechanisms to the new function. Generally each new function has analogous axioms for each item above.

Abstract Domain Theory: STRINGS

Here is the **Theory of Strings** as an example. Note that the **Theory of Sequences** and the **Theory of Non-Embedded Lists** are almost identical.

Constants: `{E}` the Empty string

Variables (typed): `{u,v,...}` characters
`{x,y,z,...}` strings

Functions: `{·, head, tail, *, rev, rev-accum, butlast, last}`

· is prefix, attach a character to the front of a string
* is concatenate, attach a string to the front of another string
[the rest are defined below as special functions]

Relations: `{isString, isChar, isEmpty, =}`

`isEmpty[x]`	test for the empty string
`isChar[x]`	test for valid character
`isString[x]`	test for valid string

GeneratorFacts:

```
isString[E]
isString[u]
isString[u·x]
```

Uniqueness:

```
not(u·x = E)
if (u·x = v·y) then u=v and x=y
```

Special char axiom:

```
u·E = u
E·u = u
```

Decomposition:

```
if not(x=E) then (x = u·y)
head[u·x] = u
tail[u·x] = x
if not(x=E) then (x = head[x]·tail[x])
```

Decompose equality:

```
if (u=v) then (u·x = v·x)
if (x=y) then (u·x = u·y)
```

Mapping:

```
F[u·x] = F[u]·F[x]
```

The String ***Induction*** *Principle:*

```
        if F[E] and
              forall x:   if not[x=E],
                          then if F[tail[x]] then F[x]
        then  forall x:   F[x]
```

Recursion, mapping:

`F[E]`	base
`F[u·x] = F[u]·F[x]`	general1
`F[x] = F[head[x]]·F[tail[x]]`	general2

Pseudo-code for testing *string equality*, using the Induction and Recursion templates for binary relations

```
if =[E,E] and
   forall x,y:
      if (not[x=E] and not[y=E]),
        then if (=[head[x],head[y]] and =[tail[x],tail[y]])
               then =[x,y]
  then forall x,y:      =[x,y]
```

`=[E,E]`	base
`=[x,y] = =[head[x],head[y]] and =[tail[x],tail[y]]`	general1

```
=[a,b]  =def=
      (a=E and b=E)
   or (=[head[a],head[b]] and =[tail[a],tail[b])
```

Some *axioms and theorems* for specialized functions

Concatenate, `*`, for joining strings together:

`E*x = x,    x*E = x`	base definition
`(u·x)*y  =  u·(x*y)`	recursive definition
`isString[x*y]`	type
`u*x = u·x`	character special
`x*(y*z) = (x*y)*z`	associativity
`if x*y = E, then x=E and y=E`	empty string
`if not(x=E) then head[x*y] = head[x]`	head
`if not(x=E) then tail[x*y] = tail[x]*y`	tail

Reverse, rev, for turning strings around:

`rev[E] = E`	base definition
`rev[u·x] = rev[x]*u`	recursive definition
`isString[rev[x]]`	type
`rev[u] = u`	character special
`rev[x*y] = rev[y]*rev[x]`	concatenation
`rev[rev[x]] = x`	double reverse
`rev[x*u] = u·rev[x]`	suffix

Reverse-accumulate, reverse the tail and prefix the head onto the accumulator:

`rev-acc[x,E] = rev[x]`	identicality
`rev-acc[E,x] = x`	base definition
`rev-acc[u·x,y] = rev-acc[x,u·y]`	recursive definition

Last and *Butlast*, for symmetrical processing of the end of a string:

`butlast[x*u] = x`	definition
`last[x*u] = u`	definition
`if not(x=E) then isString[butlast[x]]`	type
`if not(x=E) then char[last[x]]`	type
`if not(x=E) then x = butlast[x]*last[x]`	decomposition
`if not(x=E) then butlast[x] = rev[tail[rev[x]]]`	tail reverse
`if not(x=E) then last[x] = head[rev[x]]`	head reverse

Here is a function which mixes two domains, Strings and Integers:

Length, for counting the number of characters in a string

```
length[E] = 0

length[u·x] = length[x] + 1

length[x*y] = length[x] + length[y]
```

A symbolic proof by induction

To prove: `rev[rev[x]] = x` x is of type STRING

Base case:

	Rule applied:
`rev[rev[E]] =?= E`	1. problem
`rev[E] =?= E`	2. rev[E] = E
`E =?= E`	3. rev[E] = E
`QED`	4. identity

Inductive case:

`rev[rev[x]] =?= x`	1. problem
`rev[rev[u·x]] = u·x`	2. assume by induction rule
`rev[rev[x]*u] = u·x`	3. rev[a•b] = rev[b]*a
`rev[u]*rev[rev[x]] = u·x`	4. rev[a*b] = rev[b]*rev[a]
`u*rev[rev[x]] = u·x`	5. rev[a] = a a is a char
`u·rev[rev[x]] = u·x`	6. lemma a*b=a•b a is a char
`rev[rev[x]] = x`	7. a•b = a•c iff b=c
`QED`	

Lemma:

`u*x =?= u·x`	1. problem
`(u·x)*y = u·(x*y)`	2. prefix/concatenate distribution
`(u·E)*y = u·(E*y)`	3. let x=E
`u*y = u·(E*y)`	4. a•E = a
`u*y = u·y`	5. E*a = a
`QED`	

Abstract Domain Theory: TREES

Predicates
```
atom[x]
tree[x]
```

Constructor
```
+[x,y]
```

Uniqueness
```
not[atom[+[x,y]]]
if (+[x1,x2] = +[y1,y2]) then (x1=y1 and x2=y2)
```

Left and Right
```
left[+[x,y]] = x
right[+[x,y]] = y
```

Decomposition
```
if not[atom[x]] then x = +[left[x],right[x]]
```

Induction
```
if F[atom] and
   (if F[x1] and F[x2] then F[+[x1,x2]])
then F[x]
```

Some recursive *tree functions*

```
size[x] =def=     size[atom[x]] = 1;
                  size[+[x,y]] = size[x] + size[y] + 1

leaves[x] =def=   leaves[atom[x]] = 1;
                  leaves[+[x,y]] = leaves[x] + leaves[y]

depth[x] =def=    depth[atom[x]] = 1;
                  depth[+[x,y]] = max[depth[x],depth[y]] + 1
```

(pseudocode for leaves)
```
leaves[x] =def=   if empty[x] then 0
                    else if atom[x] then 1
                    else leaves[left[x]] + leaves[right[x]]
```

(pseudocode for leaves-accumulate)
```
leaves-acc[x,res] =def=
      if empty[x] then res
        else if atom[x] then leaves-acc[(), res + 1]
        else leaves-acc[right[x], res + leaves-acc[left[x]]]
```

Abstract Domain Theory: SETS

An **set implementation** with the functions Insert, Delete, and Member is called a *dictionary*.

Mathematical model:

`S = {x| <statement about x>}` extensional, collection defined by common property
`S = {a,b,c,...}` intensional, collection defined by naming the members

empty set:	`not (x in S)`	`forall x`
membership:	`x in S =def= x=s1 or x=s2 or x=s3 or ...`	
subset:	`if (x in S1) then (x in S2)`	
union:	`(x in S1) or (x in S2)`	
intersection:	`(x in S1) and (x in S2)`	
difference:	`(x in S1) and not(x in S2)`	

recursive set membership:

```
x in S  =def=
      not[x=empty-set]
      and
      x = get-one[S] or (x in rest[S])
```

Implementation functions:

```
Make-empty-set
Make-set[elements]
Insert[element,set]
Delete[element,set]
Equal[set1,set2]
```

`Cardinality[set]` = count of members

Characteristic function F:

```
(F[x] = 1 iff x in S)  and  (F[x] = 0 iff not(x in S))
```

Algebraic Specification of Sets:

This algebraic specification is also a functional implementation (ie code) in a programming language designed for formal verification.

```
theory TRIVIAL is

    sorts Elt

endtheory TRIVIAL

module BASICSET [ELT :: TRIVIAL] is

    sorts       Set

    functions
        Phi, Universe :         Set
        {_}:                    Elt -> Set
        _ symmetric-diff _ :    Set, Set -> Set
                                (assoc comm ident: 0)
        _ intersect _ :         Set, Set -> Set
                                (assoc comm idem ident: Universe)

    variables
        S,S',S'':               Set
        Elt,Elt':               Elt

    axioms
        (S sym-diff S) = Phi

        {Elt} intersect {Elt'} = Phi  :-  not(Elt = Elt')

        S intersect Phi = Phi

        S intersect (S' sym-diff S'')
             = (S intersect S') sym-diff (S intersect S'')

endmodule BASICSET


module SET [X :: TRIVIAL] using NAT, BASICSET[X] is

    functions
        _ union _ :             Set, Set -> Set
        _ - _ :                 Set, Set -> Set
        #_ :                    Set -> Nat

    predicates
        _ member _ :            Elt, Set
        _ subset _ :            Set, Set
        empty :                 Set
        _ not-member _ :        Elt, Set
```

```
    variables
          X:                          Elt
          S,S',S'':                   Set

    axioms
          S union S' = ((S intersect S') sym-diff S) sym-diff S'
          S - S' = S intersect (S sym-diff S')
          empty(S) :- S = Phi
          X member S :- {X} union S = S
          X not-member S :- {X} intersect S = Phi
          S subset S' :- S union S' = S'
          # Phi = 0
          #({X} sym-diff S) = #(S) - 1 :- X member S
          #({X} sym-diff S) = #(S) + 1 :- X not-member S

endmodule SET
```

Abstract Domain Theory: RATIONAL NUMBERS

base	`0`
recognizer	`is-number[n]`
constructor	`+1[n]`
accessor	`-1[n]`

some invariants	`is-number[n] or not[is-number[n]]`
	`is-number[+1[n]]`
	`is-number[0]`
	`+1[n] =/= 0`
	`(is-number[n] and n =/= 0) implies (+1[-1[n]] = n)`
	`is-number[n] implies (-1[+1[n]] = n)`

induction	`if F[0] and (F[n] implies F[-1[n]]) then F[n]`

```
module BASICRAT using INT is

    sorts                       Rat
    subsorts                    Int =< Rat

    functions
          _ / _ :               Int, NzInt -> Rat
          _ * _ :               Rat, Rat -> Rat  (assoc commut ident: 1)
          _ + _ :               Rat, Rat -> Rat  (assoc comm ident: 0)

    variables
          N,X,Z:                Int
          Y,W:                  NzInt
          A:                    NzNat
```

```
    axioms
          nzint(Y*W)

          N/1 = N

          0/Y = 0

          N/(-A) = (-N)/A

          X/Y = (X/gcd(X,Y))/(Y/gcd(X,Y)) :- not(gcd(X,Y)=1)

          (X/Y)+(Z/W) = ((X*W)+(Z*Y))/(Y*W)

          N+(X/Y) = ((N*Y)+X)/Y

          (X/Y)*(Z/W) = (X*Z)/(Y*W)

          N*(X/Y) = (N*X)/Y

endmodule BASICRAT
```