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. An 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. Decomposition
   Permission to take apart an arbitrary form into atomic components and functions to do the construction/deconstruction.

7. Equality 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, =}

isString[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**: 

\[
\begin{align*}
\text{if } F[E] \text{ and } \\
\quad \text{forall } x: \\
\quad \quad \text{if not}[x=E], \\
\quad \quad \quad \text{then if } F[tail[x]] \text{ then } F[x] \\
\quad \text{then forall } x: \\
\quad \quad F[x]
\end{align*}
\]

Recursion, mapping:

\[
\begin{align*}
F[E] & \quad \text{base} \\
F[u \cdot x] = F[u] \cdot F[x] & \quad \text{general1} \\
F[x] = F[head(x)] \cdot F[tail(x)] & \quad \text{general2}
\end{align*}
\]

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

\[
\begin{align*}
\text{if } =\![E,E] \text{ and } \\
\quad \text{forall } x,y: \\
\quad \quad \text{if (not}[x=E] \text{ and not}[y=E]), \\
\quad \quad \quad \text{then if (=}\![\text{head}[x],\text{head}[y]] \text{ and } =\![\text{tail}[x],\text{tail}[y]]) \\
\quad \quad \quad \quad \text{then } =\![x,y] \\
\quad \text{then forall } x,y: \quad =\![x,y]
\end{align*}
\]

\[
\begin{align*}
=\![E,E] & \quad \text{base} \\
=\![x,y] = =\![\text{head}[x],\text{head}[y]] \text{ and } =\![\text{tail}[x],\text{tail}[y]] & \quad \text{general1}
\end{align*}
\]

\[
\begin{align*}
=\![a,b] = & =\! \text{def=} \\
& (a=E \text{ and } b=E) \\
& \text{or } (=\!\![\text{head}[a],\text{head}[b]] \text{ and } =\!\![\text{tail}[a],\text{tail}[b])
\end{align*}
\]

Some *axioms and theorems* for specialized functions

**Concatenate, ***, for joining strings together:

\[
\begin{align*}
E*x = x, \quad x*E = x & \quad \text{base definition} \\
(u \cdot x)*y = u \cdot (x*y) & \quad \text{recursive definition} \\
isString[x*y] & \quad \text{type} \\
u*x = u \cdot x & \quad \text{character special} \\
x*(y*z) = (x*y)*z & \quad \text{associativity} \\
\text{if } x*y = E, \text{ then } x=E \text{ and } y=E & \quad \text{empty string} \\
\text{if not}(x=E) \text{ then } head[x*y] = head[x] & \quad \text{head} \\
\text{if not}(x=E) \text{ then tail}[x*y] = tail[x]*y & \quad \text{tail}
\end{align*}
\]
Reverse, rev, for turning strings around:

\[ \text{rev}[E] = E \]  
base definition

\[ \text{rev}[u \cdot x] = \text{rev}[x] \cdot u \]  
recursive definition

\[ \text{isString}[^{\text{rev}[x]}] \]  
type

\[ \text{rev}[u] = u \]  
character special

\[ \text{rev}[x \cdot y] = \text{rev}[y] \cdot \text{rev}[x] \]  
concatenation

\[ \text{rev}[\text{rev}[x]] = x \]  
double reverse

\[ \text{rev}[x \cdot u] = u \cdot \text{rev}[x] \]  
suffix

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

\[ \text{rev-acc}[x,E] = \text{rev}[x] \]  
identicality

\[ \text{rev-acc}[E,x] = x \]  
base definition

\[ \text{rev-acc}[u \cdot x,y] = \text{rev-acc}[x,u \cdot y] \]  
recursive definition

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

\[ \text{butlast}[x \cdot u] = x \]  
definition

\[ \text{last}[x \cdot u] = u \]  
definition

if not \((x=E)\) then \text{isString}[^{\text{butlast}[x]}]  
type

if not \((x=E)\) then \text{char}[^{\text{last}[x]}]  
type

if not \((x=E)\) then \(x = \text{butlast}[x] \cdot \text{last}[x]\)  
decomposition

if not \((x=E)\) then \text{butlast}[x] = \text{rev}[\text{tail}[\text{rev}[x]]]  
tail reverse

if not \((x=E)\) then \text{last}[x] = \text{head}[\text{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

\[ \text{length}[E] = 0 \]

\[ \text{length}[u \cdot x] = \text{length}[x] + 1 \]

\[ \text{length}[x \cdot y] = \text{length}[x] + \text{length}[y] \]
A symbolic proof by induction

To prove: \( \text{rev[rev[x]]} = x \) \( x \) is of type STRING

Base case:

\[
\begin{align*}
\text{rev[rev[E]]} &= E \\
\text{rev[E]} &= E \\
E &= E \\
\hline
\text{QED}
\end{align*}
\]

Rule applied:

1. problem
2. \( \text{rev[E]} = E \)
3. \( \text{rev[E]} = E \)
4. identity

Inductive case:

\[
\begin{align*}
\text{rev[rev[x]]} &= x \\
\text{rev[rev[u•x]]} &= u•x \\
\text{rev[rev[x]*u]} &= u•x \\
\text{rev[u]*rev[rev[x]]} &= u•x \\
u*\text{rev[rev[x]]} &= u•x \\
u•\text{rev[rev[x]]} &= u•x \\
\text{rev[rev[x]]} &= x \\
\hline
\text{QED}
\end{align*}
\]

Lemma:

\[
\begin{align*}
\text{u*x} &= u•x \\
(u•x)*y &= u•(x*y) \\
(u•E)*y &= u•(E*y) \\
u*y &= u•(E*y) \\
u*y &= u•y \\
\hline
\text{QED}
\end{align*}
\]
Abstract Domain Theory: TREES

Predicates

atom[x]

Predicate

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:

\[ S = \{x| <\text{statement about } x>\} \text{ extensional, collection defined by common property} \]
\[ S = \{a, b, c, \ldots\} \text{ intensional, collection defined by naming the members} \]

- empty set: \( \text{not } (x \in S) \) forall x
- membership: \( x \in S \overset{\text{def}}{=} x = s_1 \text{ or } x = s_2 \text{ or } x = s_3 \text{ or } \ldots \)
- subset: if \( x \in S_1 \) then \( x \in S_2 \)
- union: \( (x \in S_1) \text{ or } (x \in S_2) \)
- intersection: \( (x \in S_1) \text{ and } (x \in S_2) \)
- difference: \( (x \in S_1) \text{ and not}(x \in S_2) \)

recursive set membership:

\[ x \in S \overset{\text{def}}{=} \]
\[ \text{not}[x=\text{empty-set}] \]
\[ \text{and} \]
\[ x = \text{get-one}\{S\} \text{ or } (x \in \text{rest}\{S\}) \]

Implementation functions:

Make-empty-set
Make-set\{elements\}
Insert\{element, set\}
Delete\{element, set\}
Equal\{set1, set2\}

Cardinality\{set\} = count of members

Characteristic function \( F \):

\( (F[x] = 1 \text{ iff } x \in S) \text{ and } (F[x] = 0 \text{ iff } \text{not}(x \in S)) \)
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*X)/Y

endmodule BASICRAT