Domain Theories

A **domain** is the collection of simple objects which are of (mathematical) interest. Generally the labels of objects in a domain refer, or point, to concrete objects in reality. A **domain theory** consists of a collection of objects, together with a particular set of functions and relations which define and constrain the generic behavior of both simple and complex objects in the domain.

Domain theories have a specific mathematical form which not only identifies how the objects in that domain behave, but also provides *all the information needed to write processing algorithms* for the domain objects. The prototypical components of a domain theory are

- a **representation** of the elementary unit or constants, the **base** of the structure (also called the *carrier set*)
- **recognizer** predicates which identify the particular types of structure
- a **constructor** function which builds compound structures from simple units
- an **accessor** function which gets parts of a compound structure
- a collection of **functions** which transfer between domain objects
- a collection of **invariants,** or equations, which define the structure's behavior
- an **induction principle** which specifies how to verify correct manipulations

In an algebraic theory, you usually also have **variables**, names which are *generic*, standing in place of an *arbitrary member* of the domain base.

Propositional Calculus as an Example

Propositional calculus consists of a collection of simple objects, called **propositions**, and logical connectives which join propositions into more complex forms. A proposition is an object with a binary value from the set {True,False} and no internal structure. Propositional calculus is the domain theory for propositions. It consists of

a collection of *value labels*	`{T,F}`
a collection of *object labels*	`{a,b,c,...},`
the *logical connectives*	`{and,or,not,if,...},`
the *rules of inference* (deduction)	`{modus ponens,...}`

Recognizer predicates differentiate these components:

`isTrue[X]`	identifies the ground value True
`proposition[X]`	identifies when `X` is a proposition
`compound[X]`	identifies when `X` contains any logical connective.

Constructors tell us how to build compound objects. These are always defined inductively. This definition is called a **generation axiom**.

> Ground values are objects.
> Propositions are objects.
> If `X` and `Y` are objects, then so is `(X -> Y)`.
> There are no other valid objects.

Since all other Boolean functions can be constructed from the basis `{if,F}`, they do not need to be mentioned as part of the definition of compound objects. Rather they can be treated as syntactic variants. For example, define

```
not X       =     X -> F

X or Y      =     (not X) -> Y

X and Y     =     (not ((not X) or (not Y)))

X iff Y     =     (X -> Y) and (Y -> X)
```

Accessors are constructors used in reverse, to decompose compound objects. More importantly, accessors define how value is maintained across compound forms. Value is asserted by the predicate `isTrue`. Accessors is expressed mathematically as a **uniqueness axiom**. Here, "uniqueness" means that a compound form can be uniquely decomposed while maintaining its value. The uniqueness axiom for our only Boolean operator is:

`(X -> Y) isTrue`, when either `X isnotTrue` or `Y isTrue`.

We can substitute the syntactic variants to obtain the other decomposition rules. For example, substitute the definition of `not X` into the uniqueness axiom for implication, giving:

```
not X = X -> F
```

`(X -> F) isTrue` when either `X isnotTrue` or `F isTrue`.

Since `F` is never `True`, this simplifies to:

`(X -> F) isTrue` when `X isnotTrue`.

`(not X) isTrue` when `X isnotTrue`.

Other semantic definitions are derived in a similar fashion:

`(X or Y) isTrue` when either `X isTrue` or `Y isTrue`.

`(X and Y) isTrue` when both `X isTrue` and `Y isTrue`.

`(X iff Y) isTrue` when the value of `X` and the value of `Y` are the same.

Notice that the expression of these relationships between value and structure in English appears to sound rather obvious. This is because the logical connectives are so deeply built into the language that we cannot define them without using them in the definition.

In propositional calculus, there are ***no functions*** which compute properties, since propositions have no properties other than their value (they have no internal structure).

The ***invariants*** which describe and constrain the structure and behavior of compound propositional forms are the **axioms of deduction**. The choice of an axiom set depends on the goals of transformation. One single (algebraic) axiom from which all others can be derived is:

`(A -> B) and (not A -> B) = B`	Single axiom basis

Another set of propositional axioms is that of Boolean algebra. These axioms use three Boolean connectives and are therefore highly redundant:

`X and (Y and Z) = (X and Y) and Z`	Associativity
`X or (Y or Z) = (X or Y) or Z`	Associativity
`X and Y = Y and X`	Commutativity
`X or Y = Y or X`	Commutativity
`X and (Y or Z) = (X and Y) or (X and Z)`	Distributivity
`X or (Y and Z) = (X or Y) and (X or Z)`	Distributivity
`X or F = X`	Zero element
`X and T = X`	One element
`X and (not X) = F`	Complement
`X or (not X) = T`	Complement

Yet another set of axioms are the rules of natural deduction. Again these are highly redundant, the usual list includes about 35 rules. This system is typified by the axiom of *modus ponens*:

`(X and (X -> Y)) -> Y`	Modus Ponens

A fourth example of a possible axiomatic basis is the erasure axioms of boundary logic:

`X or T = T`	Dominion
`not (not X) = X`	Involution
`(X or Y) -> X = Y -> X`	Pervasion

Finally, the **induction principle** for propositional calculus might be loosely stated as:

Base case: `T isTrue`

Inductive case: If an arbitrary form `X` is assumed to be true, `X isTrue`, and if we can demonstrate that `(X -> Y) isTrue`, then for any `Y`, `Y isTrue`.

This of course is just a restatement of *modus ponens*. This induction principle is rather degenerate, since the *logic of induction* itself is expressed using the same logical connectives that define the operations in propositional calculus. The well-founded ordering is not a countable structure like the natural integers, rather it is the potential nesting of implications. When a premise implies a conclusion, then the premise, in a very general sense, is *less-than* the conclusion. This ordering is clearly demonstrated both in the Boolean lattice and in the deep Pervasion rule of boundary logic.

Domains with Internal Structure

To add descriptive complexity, we add internal structure to propositions. There are two general classes of structure: **relations** and **functions**. Relations are connections, or structures, holding together pairs of simple objects. Functions are a restricted type of relation, one that permits functions to stand in place of object names. Functions are relations which name objects in a domain by using other object names.

In computer science, we refer to complex objects as *data structures*, and the set of relational constraints on these objects as *abstract data types*.

The important idea is that all data structures, all domains, have the same organizational structure. All domains and data types consist of a collection of these axiomatic principles:

Labels
Recognizers
Constructors
Accessors
Functions
Invariants (relations)
Induction Principle

In *object-oriented approaches*, the abstract data type includes all algorithmic functionality. That is, using oo-techniques, the above principles define the entirety of an object, and thus the entirety of a program.

Three examples of domain theories follow. These examples are schematic outlines, intended to suggest both mathematical approach and coding technique. Each domain has additional functions and relations which are not included here.

The Domain Theory of Non-negative Integers

Base

```
0
```

Objects

```
{positive integers}
```

Recognizer

```
integer[n]
```

Constructor

```
+1[n], inc[n]
```

Accessor

```
-1[n], dec[n]
```

Decomposition axioms

```
(integer[n] and not[n=0]) -> (+1[-1[n]] = n)
integer[n] -> (-1[+1[n]] = n)
```

Uniqueness axiom

```
(+1[n1] = +1[n2]) iff n1=n2
```

Functions

```
+:    (associative, commutative, identity=0)
    n+0 = n
    n1 + +1[n2] = +1[n1+n2]
    (n1=n2) -> (n1+n3) = (n2+n3)

-:
    n-0 = n
    +1[n1] - +1[n2] = n1-n2

*:    (associative, commutative, identity=1)
    n*0 = 0
    n1*(n2+1) = n1*n2 + n1

^:
    n^0 = 1
    n1^(n2+1) = (n1^n2)*n1
```

Some invariants

```
integer[n] or not[integer[n]]
integer[+1[n]]
integer[0]
not[+1[n] = 0]
integer[n1+n2]
+1[0] = 1
n+1 = +1[n]
n^1 = n
if not[n=0] then 0^n = 0
```

Induction

```
if (F[0] and (F[n] -> F[+1[n]])) then F[n]
```

Decomposition Induction

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

The Domain Theory of Lists

Note that nested lists are *trees*.

Base

```
[] nil
```

Objects

```
{x,y,z,...} lists
{u,v,...}   atoms
```

Recognizers

```
atom[x]
list[x]
atom-or-list[x]
```

Constructor

```
x·y, insert x into list y
```

Accessors

```
first[x], rest[x]
first[x·y] = x
rest[x·y] = y
```

Decomposition axiom

```
(not[x=nil]) -> (x = (first[x] · rest[x]))
```

Uniqueness axiom

```
((x1·x2) = (y1·y2)) -> (x1=y1 and x2=y2)
```

Functions

```
append:     (associative, identity=nil)
    append[nil,x] = x
    append[u·x,y] = u · append[x,y]

member:
    not[member[x,nil]]
    member[u,v·x] iff u=v or member[u,x]

flat:
    flat[nil] = []
    flat[u·y] = u · flat[y]
    flat[x·y] = append[flat[x],flat[y]]
```

Some invariants

```
not[x·y] = nil
not[atom[x·y]]
(atom[x] and list[x]) -> x=nil
list[append[x,y]]
member[u,append[x,y]] iff member[u,x] or member[u,y]
flat[append[x,y]] = append[flat[x],flat[y]]
```

Induction

```
if (F[nil] and (F[x] -> F[u·x])
           and ((F[y] and F[x]) -> F[y·x]))
      then F[x]
```

This Inductive Principle has three conditions: the base case, the atom case, and the list case. The Induction Principle for lists can also be stated as a decomposition rule:

Decomposition Induction

```
if (F[nil] and
   (when (not[x=nil] and atom[first[x]])
         F[rest[x]] -> F[x]) and
   (when (not[x=nil] and list[head[x]])
         (F[first[x]] and F[rest[x]]) -> F[x]))
  then F[x]
```

The Domain Theory of Sets

Set Base `{} Phi (the empty set)`

Element Base `{a,b,c,...} from some domain`

Objects `{S1,S2,...}` Universe = PowerSet[elements]

Recognizers `atom[a]` `set[S]`

Constructor `a·S, insert atom a into set S`

Accessor

```
member[a,S]
member[choice[S],S]
not[member[choice[S],rest[S]]]
```

Decomposition axiom `(not[S=Phi]) -> (S = (choice[S] · rest[S]))`

Uniqueness axiom `(member[a,b·S] iff (a=b) or member[a,S]`

Functions

Equality:

```
S1=S2 iff (choice[S1]=choice[S2] and
  member[choice[S1]] and member[choice[S2]])
```

Intersection: (associative, commutative, idempotent, identity=Universe)

```
intersect[Phi,S] = Phi
intersect[a·S1,S2] = if member[a,S2]
          then (a·intersect[S1,S2]) else intersect[S1,S2]
```

Symmetric-difference: (associative, commutative, identity=Phi)

```
sym-diff[S1,Phi] = Phi
sym-diff[S1,S2] = if (member[a,S1] and not[member[a,S2]])
            or (member[a,S2] and not[member[a,S1]])
            then member[a,sym-diff[S1,S2]]
```

Cardinality, #:

```
#[Phi] = 0
if not[member[a,S]] then #[a·S] = #[S] + 1
```

Some invariants

```
set[Phi]
set[a·S]
not[member[a,Phi]]
(intersect[{a},{b}] = Phi) -> (not[a=b])
intersect[S,Phi] = Phi
member[a,intersect[S1,S2]] iff
    member[a,S1] and member[a,S2]
S1 intersect (S2 sym-diff S3) =
    (S1 intersect S2) sym-diff (s1 intersect S3)
```

Induction

```
if F[Phi] and if not[member[a,S]] then (F[a] -> F[a·S])
   then F[S]
```