Chapter 3

Domain Theory I: Semantic Algebras

Before we can study the semantics of programming languages, we must establish a suitable collection of meanings for programs. We employ a framework called *domain theory:* the study of ''structured sets'' and their operations. A programmer might view domain theory as ''data structures for semantics.'' Nonetheless, domain theory is a formal branch of (computing-related) mathematics and can be studied on its own.

The fundamental concept in domain theory is a *semantic domain,* a set of elements grouped together because they share some common property or use. The set of natural numbers is a useful semantic domain; its elements are structurally similar and share common use in arithmetic. Other examples are the Greek alphabet and the diatonic (musical) scale. Domains may be nothing more than sets, but there are situations in which other structures such as lattices or topologies are used instead. We can use domains without worrying too much about the underlying mathematics. Sets make good domains, and you may safely assume that the structures defined in this chapter are nothing more than the sets discussed in Chapter 2. Chapter 6 presents reasons why domains other than sets might be necessary.

Accompanying a domain is a set of *operations.* The operations are functions that need arguments from the domain to produce answers. Operations are defined in two parts. First, the operation's domain and codomain are given by an expression called the operation's *functionality.* For an operation f, its functionality $f: D_1 \times D_2 \times \cdots \times D_n \rightarrow A$ says that f needs an argument from domain D_1 and one from D_2, . . ., and one from D_n to produce an answer in domain A. Second, a description of the operation's mapping is specified. The description is usually an equational definition, but a set graph, table, or diagram may also be used.

A domain plus its operations constitutes a *semantic algebra.* Many examples of semantic algebras are found in the following sections.

3.1 PRIMITIVE DOMAINS

A *primitive domain* is a set that is fundamental to the application being studied. Its elements are atomic and they are used as answers or ''semantic outputs.'' For example, the real numbers are a primitive domain for a mathematician, as are the notes in the key of C for a musician, as are the words of a dictionary for a copyeditor, and so on. Here is the most commonly used primitive domain:

3.1 Example: *The natural numbers*

Domain $Nat = \mathbb{N}$
Operations
$zero : Nat$

$one : Nat$
$two : Nat$
$\ldots$
$plus : Nat \times Nat \rightarrow Nat$
$minus : Nat \times Nat \rightarrow Nat$
$times : Nat \times Nat \rightarrow Nat$

The operations *zero, one, two, . . .* are *constants.* Each of the members of *Nat* is named by a constant. We list the constants for completeness' sake and to make the point that a constant is sometimes treated as an operation that takes zero arguments to produce a value. The other operations are natural number addition, subtraction, and multiplication, respectively. The *plus* and *times* operations are the usual functions, and you should have no trouble constructing the graphs of these operations. Natural number subtraction must be clarified: if the second argument is larger than the first, the result is the constant *zero*; otherwise a normal subtraction occurs.

Using the algebra, we can construct expressions that represent members of *Nat.* Here is an example: *plus* (*times* (*three*, *two*), *minus* (*one*, *zero*)). After consulting the definitions of the operations, we determine that the expression represents that member of *Nat* that has the name *seven*. The easiest way to determine this fact, though, is by simplification:

plus (*times*(*three*, *two*)), *minus* (*one*, *zero*))
= *plus* (*times*(*three*, *two*)), *one*)
= *plus* (*six*, *one*)
= *seven*

Each step of the simplification sequence preserved the underlying meaning of the expression. The simplification stopped at the constant *seven* (rather than continuing to, say, *times* (*one*, *seven*)), because we seek the simplest representation of the value. The simplification process makes the underlying meaning of an expression easier to comprehend.

From here on, we will use the arithmetic operations in infix format rather than prefix format; that is, we will write *six plus one* rather than *plus* (*six*, *one*).

To complete the definition of natural number arithmetic, let us add the operation $div : Nat \times Nat \rightarrow Nat$ to the algebra. The operation represents natural number (nonfractional) division; for example, *seven div three* is *two*. But the operation presents a technical problem: what is the answer when a number is divided by *zero*? A computer implementation of *div* might well consider this an error situation and produce an error value as the answer. We model this situation by adding an extra element to *Nat*. For any $n \in \mathbb{N}$, *n div zero* has the value *error*. All the other operations upon the domain must be extended to handle *error* arguments, since the new value is a member of *Nat.* The obvious extensions to *plus, minus, times,* and *div* make them produce an answer of *error* if either of their arguments is *error.* Note that the *error* element is not always included in a primitive domain, and we will always make it clear when it is.

The truth values algebra is also widely used, as shown in Example 3.2.

3.2 Example: *The truth values*

Domain $Tr = \mathbb{B}$
Operations
$true : Tr$
$false : Tr$
$not : Tr \to Tr$
$or : Tr \times Tr \to Tr$
$(\_ \to \_ \,[]\, \_) : Tr \times D \times D \to D$, for a previously defined domain D

The truth values algebra has two constants— *true* and *false*. Operation *not* is logical negation, and *or* is logical disjunction. The last operation is the choice function. It uses elements from another domain in its definition. For values $m, n \in D$, it is defined as:

$(true \to m \,[]\, n) = m$
$(false \to m \,[]\, n) = n$

Read the expression $(x \to y \,[]\, z)$ as saying ''if x then y else z.''

Here are some expressions using numbers and truth values:

1. $((not(false))\ or\ false$
   $= true\ or\ false$
   $= true$

2. $(true\ or\ false) \to (seven\ div\ three) \,[]\, zero$
   $= true \to (seven\ div\ three) \,[]\, zero$
   $= seven\ div\ three = two$

3. $not(not\ true) \to false \,[]\, false\ or\ true$
   $= not(not\ true) \to false \,[]\, true$
   $= not\ false \to false \,[]\, true$
   $= true \to false \,[]\, true$
   $= false$

The utility of the choice function increases if relational operations are added to the *Nat* algebra. Here are some useful ones:

$equals : Nat \times Nat \to Tr$
$lessthan : Nat \times Nat \to Tr$
$greaterthan : Nat \times Nat \to Tr$

All have their usual definitions. As an example, the expression $not(four\ equals\ (one\ plus\ three)) \to (one\ greaterthan\ zero) \,[]\, ((five\ times\ two)\ lessthan\ zero)$ simplifies to the constant *false.*

3.3 Example: *Character strings*

Domain *String* = the character strings formed from the elements of $\mathbb{C}$

(including an ‘‘error’’ string)
Operations
A, B, C, . . . , Z: *String*
empty : *String*
error : *String*
concat : *String* × *String* → *String*
length : *String* → *Nat*
substr : *String* × *Nat* × *Nat* → *String*

Text processing systems use this domain. Single characters are represented by constants. The constant *empty* represents the string with no characters. Words are built using *concat,* which concatenates two strings to build a new one. We will be lazy and write a string built with *concat* in double quotes, e.g., "ABC" abbreviates A *concat* (B *concat* C). Operation *length* takes a string as an argument and returns its length; *substr* is a substring extraction operator: given a string *s* and two numbers n_1 and n_2, *substr* (*s*, n_1, n_2) extracts that part of *s* that begins at character position number n_1 and is n_2 characters long. (The leading character is at position *zero*.) Some combinations of (*s*, n_1, n_2) suggest impossible tasks. For example, what is the value represented by *substr* (*"ABC"*, *one*, *four*) or *substr* (*"ABC"*, *six*, *two*)? Use *error* as the answer for such combinations. If *concat* receives an *error* argument, its result is *error;* the length of the *error* string is *zero;* and any attempt to use *substr* on an *error* string also leads to *error*.

3.4 Example: *The one element domain*

Domain *Unit*, the domain containing only one element
Operations
() : *Unit*

This degenerate algebra is useful for theoretical reasons; we will also make use of it as an alternative form of error value. The domain contains exactly one element, (). *Unit* is used whenever an operation needs a dummy argument. Here is an example: let *f*: *Unit* → *Nat* be *f*(*x*) = *one*; thus, *f*(()) = *one*. We will discard some of the extra symbols and just write *f*: *Unit* → *Nat* as *f*() = *one*; thus, *f*() = *one*.

3.5 Example: *Computer store locations*

Domain *Location*, the address space in a computer store
Operations
first-locn : *Location*
next-locn : *Location* → *Location*
equal-locn : *Location* × *Location* → *Tr*
lessthan-locn : *Location* × *Location* → *Tr*

The domain of computer store addresses is fundamental to the semantics of programming

languages. The members of *Location* are often treated as numbers, but they are just as likely to be electrical impulses. The constant *first-locn* gives the ''lowest address'' in the store, and the other locations are accessed in order via the *next-locn* operation. (Think of *next-locn*(*l*) as *l*+1.) The other two operations compare locations for equality and lessthan.

This algebra would be inadequate for defining the semantics of an assembly language, for an assembly language allows random access of the locations in a store and treats locations as numbers. Nonetheless, the algebra works well for programming languages whose storage is allocated in static or stack-like fashion.

3.2 COMPOUND DOMAINS

Just as programming languages provide data structure builders for constructing new data objects from existing ones, domain theory possesses a number of domain building constructions for creating new domains from existing ones. Each domain builder carries with it a set of operation builders for assembling and disassembling elements of the compound domain. We cover in detail the four domain constructions listed in Section 2.3 of Chapter 2.

3.2.1 Product

The *product* construction takes two or more component domains and builds a domain of tuples from the components. The case of binary products is considered first.

The product domain builder × builds the domain $A \times B$, a collection whose members are ordered pairs of the form (a, b), for $a \in A$ and $b \in B$. The operation builders for the product domain include the two disassembly operations:

fst: $A \times B \rightarrow A$
: which takes an argument (a,b) in $A \times B$ and produces its first component $a \in A$, that is, $fst(a,b) = a$

snd: $A \times B \rightarrow B$
: which takes an argument (a,b) in $A \times B$ and produces its second component $b \in B$, that is, $snd\,(a,b) = b$

The assembly operation is the ordered pair builder:

> if *a* is an element of *A*, and *b* is an element of *B*, then (a, b) is an element of $A \times B$

The product domain raises a question about the functionalities of operations. Does an operation such as $or : Tr \times Tr \rightarrow Tr$ receive two elements from *Tr* as arguments or a pair argument from $Tr \times Tr$? In domain theory, as in set theory, the two views coincide: the two elements form one pair.

The product construction can be generalized to work with any collection of domains

$A_1, A_2, \cdots, A_n$, for any $n > 0$. We write $(x_1, x_2, \cdots, x_n)$ to represent an element of $A_1 \times A_2 \times \cdots \times A_n$. The subscripting operations *fst* and *snd* generalize to a family of *n* operations: for each *i* from 1 to *n*, $\downarrow i$ denotes the operation such that $(a_1, a_2, \cdots, a_n)\downarrow i = a_i$. Theoretically, it is possible to construct products from an infinite number of component domains, but infinite products raise some technical problems which are considered in exercise 16 in Chapter 6.

Example 3.6 shows a semantic algebra built with the product construction.

3.6 Example: *Payroll information: a person's name, payrate, and hours worked*

Domain *Payroll-record* = *String* × *Rat* × *Rat*

(Note: *Rat* is the domain defined in Example 2.1 in Chapter 2)

Operations

new-employee : *String* → *Payroll-record*
new-employee(*name*) = (*name*, *minimum-wage*, **0**),
where *minimum-wage* ∈ *Rat* is some fixed value from *Rat*
and **0** is the *Rat* value (*makerat* (0) (1))

update-payrate : *Rat* × *Payroll-record* → *Payroll-record*
update-payrate (*pay*, *employee*) = (*employee*↓1, *pay*, *employee*↓3)

update-hours : *Rat* × *Payroll-record* → *Payroll-record*
update-hours (*hours*, *employee*) = (*employee*↓1, *employee*↓2, *hours*
addrat employee↓3)

compute-pay : *Payroll-record* → *Rat*
compute-pay (*employee*) = (*employee*↓2) *multrat* (*employee*↓3)

This semantic algebra is useful for the semantics of a payroll program. The components of the domain represent an employee's name, hourly wage, and the cumulative hours worked for the week. Here is an expression built with the algebra's operations:

compute-pay(*update-hours*(**35**, *new-employee*(*"J.Doe"*)))
= *compute-pay*(*update-hours*(**35**, (*"J.Doe"*, *minimum-wage*, **0**)))
= *compute-pay*((*"J.Doe"*, *minimum-wage*, **0**)↓1, (*"J.Doe"*, *minimum-wage*, **0**)↓2,
35 *addrat* (*"J.Doe"*, *minimum-wage*, **0**)↓3)
= *compute-pay*(*"J.Doe"*, *minimum-wage*, **35** *addrat* **0**)
= *minimum-wage multrat* **35**

3.2.2 Disjoint Union

The construction for unioning two or more domains into one domain is *disjoint union* (or *sum*).

For domains A and B, the disjoint union builder + builds the domain $A + B$, a collection whose members are the elements of A and the elements of B, labeled to mark their origins. The classic representation of this labeling is the ordered pair (*zero*, a) for an $a \in A$ and (*one*, b) for a $b \in B$.

The associated operation builders include two assembly operations:

in$A$: $A \to A + B$
which takes an $a \in A$ and labels it as originating from A; that is, in$A(a) = (zero, a)$, using the pair representation described above.

in$B$: $B \to A + B$
which takes a $b \in B$ and labels it as originating from B, that is, in$B(b) = (one, b)$.

The ''type tags'' that the assembly operations place onto their arguments are put to good use by the disassembly operation, the *cases* operation, which combines an operation on A with one on B to produce a disassembly operation on the sum domain. If d is a value from $A + B$ and $f(x) = e_1$ and $g(y) = e_2$ are the definitions of $f: A \to C$ and $g: B \to C$, then:

(cases d of is$A(x) \to e_1$ [] is$B(y) \to e_2$ end)

represents a value in C. The following properties hold:

(cases in$A(a)$ of is$A(x) \to e_1$ [] is$B(y) \to e_2$ end) = $[a/x]e_1 = f(a)$

and

(cases in$B(b)$ of is$A(x) \to e_1$ [] is$B(y) \to e_2$ end) = $[b/y]e_2 = g(b)$

The *cases* operation checks the tag of its argument, removes it, and gives the argument to the proper operation.

Sums of an arbitrary number of domains can be built. We write $A_1 + A_2 + \cdots + A_n$ to stand for the disjoint union of domains $A_1, A_2, \cdots, A_n$. The operation builders generalize in the obvious way.

As a first example, we alter the *Payroll-record* domain of Example 3.6 to handle workers who work either the day shift or the night shift. Since the night shift is less desirable, employees who work at night receive a bonus in pay. These concepts are represented with a disjoint union construction in combination with a product construction.

3.7 Example: *Revised payroll information*

Domain $Payroll\text{-}rec = String \times (Day + Night) \times Rat$
where $Day = Rat$ and $Night = Rat$
(The names *Day* and *Night* are aliases for two occurrences of *Rat*. We use $dwage \in Day$ and $nwage \in Night$ in the operations that follow.)

Operations

$newemp : String \to Payroll\text{-}rec$
$newemp(name) = (name,$ in$Day(minimum\text{-}wage), \mathbf{0})$

$move\text{-}to\text{-}dayshift : Payroll\text{-}rec \to Payroll\text{-}rec$

```
  move-to-dayshift(employee)=( employee↓1,
                (cases (employee↓2) of isDay(dwage)→ inDay(dwage)
                          [] isNight(nwage)→ inDay(nwage)  end),
                employee↓3 )
move-to-nightshift : Payroll-rec → Payroll-rec
  move-to-nightshift(employee)= ( employee↓1,
                (cases (employee↓2) of isDay(dwage)→ inNight(dwage)
                          [] isNight(nwage)→ inNight(nwage)  end),
                employee↓3 )
. . .

compute-pay : Payroll-rec → Rat
  compute-pay(employee)= (cases (employee↓2) of
                          isDay(dwage)→ dwage multrat (employee↓3)
                         [] isNight(nwage)→ (nwage
                              multrat 1.5) multrat (employee↓3)
                         end)
```

A person's wage is labeled as being either a day wage or a night wage. A new employee is started on the day shift, signified by the use of in*Day* in the operation *newemp*. The operations *move-to-dayshift* and *move-to-nightshift* adjust the label on an employee's wage. Operation *compute-pay* computes a time-and-a-half bonus for a night shift employee. Here is an example: if *jdoe* is the expression *newemp*(*"J.Doe"*) = (*"J.Doe"*, in*Day*(*minimum-wage*), **0**), and *jdoe-thirty* is *update-hours*(**30**, *jdoe*), then:

```
compute-pay(jdoe-thirty)
= (cases jdoe-thirty↓2 of
     isDay(wage)→ wage multrat (jdoe-thirty↓3)
     [] isNight(wage)→ (wage multrat 1.5) multrat (jdoe-thirty↓3)
     end)
= (cases inDay(minimum-wage) of
     isDay(wage)→ wage multrat 30
     [] isNight(wage)→ wage multrat 1.5 multrat 30
     end)
= minimum-wage multrat 30
```

The tag on the component in*Day*(*minimum-wage*) of *jdoe-thirty*'s record helps select the proper pay calculation.

The primitive domain *Tr* can be nicely modelled using the *Unit* domain and the disjoint union construction.

3.8 Example: *The truth values as a disjoint union*

Domain $Tr = TT + FF$
  where $TT = Unit$ and $FF = Unit$
Operations

$true : Tr$
  $true = \text{in}TT()$

$false : Tr$
  $false = \text{in}FF()$

$not : Tr \to Tr$
  $not(t) = \text{cases } t \text{ of is}TT() \to \text{in}FF() \text{ [] is}FF() \to \text{in}TT() \text{ end}$

$or : Tr \times Tr \to Tr$
  $or(t, u) = \text{cases } t \text{ of}$
    $\text{is}TT() \to \text{in}TT()$
    $\text{[] is}FF() \to (\text{cases } u \text{ of is}TT() \to \text{in}TT() \text{ [] is}FF() \to \text{in}FF() \text{ end})$
    end

The dummy argument () isn't actually used in the operations— the tag attached to it is the important information. For this reason, no identifier names are used in the clauses of the *cases* statements; () is used there as well. We can also define the choice function:

$(t \to e_1 \text{ [] } e_2) = (\text{cases } t \text{ of is}TT() \to e_1 \text{ [] is}FF() \to e_2 \text{ end})$

As a third example, for a domain D with an *error* element, the collection of finite lists of elements from D can be defined as a disjoint union. The domain

$$D^* = Unit + D + (D \times D) + (D \times (D \times D)) + \cdots$$

captures the idea: *Unit* represents those lists of length zero (namely the empty list), D contains those lists containing one element, $D \times D$ contains those lists of two elements, and so on.

3.9 Example: *Finite lists*

Domain D^*
Operations

$nil : D^*$
  $nil = \text{in}Unit()$

$cons : D \times D^* \to D^*$
  $cons(d, l) = \text{cases } l \text{ of}$
    $\text{is}Unit() \to \text{in}D(d)$
    $\text{[] is}D(y) \to \text{in}D \times D(d,y)$
    $\text{[] is}D \times D(y) \to \text{in}D \times (D \times D)(d,y)$
    $\text{[] } \cdots \text{ end}$

$hd : D^* \to D$

$hd(l)$= cases l of
is$Unit()\rightarrow error$
[] is$D(y)\rightarrow y$
[] is$D\times D(y)\rightarrow fst(y)$
[] is$D\times(D\times D)(y)\rightarrow fst(y)$
[] $\cdots$ end

$tl : D^* \rightarrow D^*$
$tl(l)$= cases l of
is$Unit()\rightarrow$ in$Unit()$
is$D(y)\rightarrow$ in$Unit()$
[] is$D\times D(y)\rightarrow$ in$D(snd(y))$
[] is$D\times(D\times D)(y)\rightarrow$ in$D\times D(snd(y))$
[] $\cdots$ end)

$null : D^* \rightarrow Tr$
$null(l)$= cases l of
is$Unit()\rightarrow true$
[] is$D(y)\rightarrow false$
[] is$D\times D(y)\rightarrow false$
[] $\cdots$ end

Even though this domain has an infinite number of components and the *cases* expressions have an infinite number of choices, the domain and codomain operations are still mathematically well defined. To implement the algebra on a machine, representations for the domain elements and operations must be found. Since each domain element is a tagged tuple of finite length, a list can be represented as a tuple. The tuple representations lead to simple implementations of the operations. The implementations are left as an exercise.

3.2.3 Function Space

The next domain construction is the one most removed from computer data structures, yet it is fundamental to all semantic definitions. It is the *function space builder*, which collects the functions from a domain A to a codomain B.

For domains A and B, the function space builder $\rightarrow$ creates the domain $A\rightarrow B$, a collection of functions from domain A to codomain B. The associated disassembly operation is just function application:

$\_(\_) : (A\rightarrow B)\times A\rightarrow B$
which takes an $f\in A\rightarrow B$ and an $a\in A$ and produces $f(a)\in B$

An important property of function domains is the principle of *extensionality:* for any f and g in $A\rightarrow B$, if for all $a\in A$, $f(a)=g(a)$, then $f=g$. Functions are understood in terms of their

argument-answer behavior, and an extensional function domain never contains two distinct elements representing the same function.

The assembly principle for functions is:

> if e is an expression containing occurrences of an identifier x, such that whenever a value $a \in A$ replaces the occurrences of x in e, the value $[a/x]e \in B$ results, then $(\lambda x.e)$ is an element in $A \to B$.

The form $(\lambda x.e)$ is called an *abstraction.* We often give names to abstractions, say $f = (\lambda x.e)$, or $f(x) = e$, where f is some name *not* used in e. For example, the function $plustwo(n) = n\ plus\ two$ is a member of $Nat \to Nat$ because $n\ plus\ two$ is an expression that has a unique value in *Nat* when n is replaced by an element of *Nat.* All of the operations built in Examples 3.6 through 3.9 are justified by the assembly principle.

We will usually abbreviate a nested abstraction $(\lambda x.(\lambda y.\ e))$ to $(\lambda x.\lambda y.\ e)$.

The binding of argument to binding identifier works the expected way with abstractions: $(\lambda n.\ n\ plus\ two)one = [one/n]n\ plus\ two = one\ plus\ two$. Here are other examples:

1. $(\lambda m.\ (\lambda n.\ n\ times\ n)(m\ plus\ two))(one)$
   $= (\lambda n.\ n\ times\ n)(one\ plus\ two)$
   $= (one\ plus\ two)\ times\ (one\ plus\ two)$
   $= three\ times\ (one\ plus\ two) = three\ times\ three = nine$

2. $(\lambda m.\ \lambda n.\ (m\ plus\ m)\ times\ n)(one)(three)$
   $= (\lambda n.\ (one\ plus\ one)\ times\ n)(three)$
   $= (\lambda n.\ two\ times\ n)(three)$
   $= two\ times\ three = six$

3. $(\lambda m.\ (\lambda n.\ n\ plus\ n)(m)) = (\lambda m.\ m\ plus\ m)$

4. $(\lambda p.\lambda q.\ p\ plus\ q)(r\ plus\ one) = (\lambda q.\ (r\ plus\ one)\ plus\ q)$

Here is a bit of terminology: an identifier x is *bound* if it appears in an expression e in $(\lambda x.e)$. An identifier is *free* if it is not bound. In example 4, the occurrences of p and q are bound (to λp and λq, respectively), but r is free. In an expression such as $(\lambda x.\lambda x.\ x)$, the occurrence of x is bound to the innermost occurrence of λx, hence $(\lambda x.\lambda x.\ x)(zero)(one) = ([zero/x](\lambda x.\ x))(one) = (\lambda x.\ x)(one) = one$.

Care must be taken when simplifying nested abstractions; free identifiers in substituted arguments may clash with inner binding identifiers. For example, the proper simplification of $(\lambda x.\ (\lambda y.\lambda x.\ y)x)$ is $(\lambda x.\ (\lambda x_1.\ x))$ and *not* $(\lambda x.(\lambda x.\ x))$. The problem lies in the re-use of x in two different abstractions. The solution is to rename the occurrences of identifiers x in $(\lambda x.\ M)$ if x clashes with a free occurrence of x in the argument that must be substituted into M. For safety's sake, avoid re-using binding identifiers.

Finally, we mention again the abbreviation introduced in Section 2.2.2 of Chapter 2 for function creation:

> $[n \mapsto v]r$ abbreviates $(\lambda m.\ m\ equals\ n \to v\ []\ r(m))$

That is, $([n \mapsto v]r)(n) = v$, and $([n \mapsto v]r)(m) = r(m)$ when $m \neq n$.

Let's look at some algebras. Example 3.10 is simple but significant, for it illustrates operations that will appear again and again.

3.10 Example: *Dynamic arrays*

Domain $Array = Nat \to A$
where A is a domain with an *error* element
Operations

$newarray : Array$
$newarray = \lambda n.\ error$

$access : Nat \times Array \to A$
$access(n, r) = r(n)$

$update : Nat \times A \times Array \to Array$
$update(n, v, r) = [n \mapsto v]r$

A *dynamic array* is an array whose bounds are not restricted, so elements may be inserted into any position of the array. The array uses natural number indexes to access its contents, which are values from A. An empty array is represented by the constant *newarray*. It is a function and it maps all of its index arguments to *error*. The *access* operation indexes its array argument r at position n. Operation *update* creates a new array that behaves just like r when indexed at any position but n. When indexed at position n, the new array produces the value v. Here is the proof:

1. for any $m_0, n_0 \in Nat$ such that $m_0 \neq n_0$,
   $access(m_0, update(n_0, v, r))$
   $= (update(n_0, v, r))(m_0)$ by definition of *access*
   $= ([n_0 \mapsto v]r)(m_0)$ by definition of *update*
   $= (\lambda m.\ m\ equals\ n_0 \to v\ []\ r(m))(m_0)$ by definition of function updating
   $= m_0\ equals\ n_0 \to v\ []\ r(m_0)$ by function application
   $= false \to v\ []\ r(m_0)$
   $= r(m_0)$

2. $access(n_0, update(n_0, v, r))$
   $= (update(n_0, v, r))(n_0)$
   $= ([n_0 \mapsto v]r)(n_0)$
   $= (\lambda m.\ m\ equals\ n_0 \to v\ []\ r(m))(n_0)$
   $= n_0\ equals\ n_0 \to v\ []\ r(n_0)$
   $= true \to v\ []\ r(n_0)$
   $= v$

The insight that an array is a function from its index set to its contents set provides interesting new views of many computer data structures.

3.11 Example: *Dynamic array with curried operations*

Domain $Array = Nat \to A$
Operations

$newarray : Array$
$newarray = \lambda n.\ error$

$access : Nat \to Array \to A$
$access = \lambda n.\lambda r.\ r(n)$

$update : Nat \to A \to Array \to Array$
$update = \lambda n.\lambda v.\lambda r.\ [n \mapsto v]r$

This is just Example 3.10 rewritten so that its operations accept their arguments in *curried form,* that is, one argument at a time. The operation $access : Nat \to Array \to A$ has a functionality that is more precisely stated as $access : Nat \to (Array \to A)$; that is, the default precedence on the arrow is to the right. We can read *access*'s functionality as saying that *access* takes a *Nat* argument and then takes an *Array* argument to produce an *A*-value. But *access*(*k*), for some number *k,* is itself a well-defined operation of functionality $Array \to A$. When applied to an argument *r,* operation *access*(*k*) looks into position *k* within *r* to produce the answer (*access*(*k*))(*r*), which is *r*(*k*). The heavily parenthesized expression is hard to read, so we usually write *access*(*k*)(*r*) or (*access k r*) instead, assuming that the default precedence of function application is to the left.

Similar conventions apply to *update.* Note that $update : Nat \to A \to Array \to Array$ is an operation that needs a number, a value, and an array to build a new array; $(update\ n_1) : A \to Array \to Array$ is an operation that builds an array updated at index n_1; $(update\ n_1\ v_1) : Array \to Array$ is an operation that updates an array at position n_1 with value v_1; $(update\ n_1\ v_1\ r_1) \in Array$ is an array that behaves just like array r_1 except at position n_1, where it has stored the value v_1. Curried operations like *access* and *update* are useful for situations where the data values for the operations might be supplied one at a time rather than as a group.

3.2.4 Lifted Domains and Strictness

In Section 2.3 of Chapter 2 the element $\perp$ (read ''bottom'') was introduced. Its purpose was to represent undefinedness or nontermination. The addition of $\perp$ to a domain can itself be formalized as a domain-building operation.

For domain *A,* the *lifting* domain builder $(\ )_\perp$ creates the domain $A_\perp$, a collection of the members of *A* plus an additional distinguished element $\perp$. The elements of *A* in $A_\perp$ are called *proper elements;* $\perp$ is the *improper element.*

The disassembly operation builder converts an operation on *A* to one on $A_\perp$; for $(\lambda x.e) : A \to B_\perp$:

$(\underline{\lambda} x.e) : A_\perp \to B_\perp$ is defined as

$(\underline{\lambda}x.e)\lfloor_\perp = \lfloor_\perp$
$(\underline{\lambda}x.e)a = [a/x]e \quad \text{for } a \neq \lfloor_\perp$

An operation that maps a $\lfloor_\perp$ argument to a $\lfloor_\perp$ answer is called *strict*. Operations that map $\lfloor_\perp$ to a proper element are called *nonstrict*. Let's do an example.

$(\underline{\lambda}m.\, zero)((\underline{\lambda}n.\, one)\lfloor_\perp)$
$= (\underline{\lambda}m.\, zero)\lfloor_\perp$, by strictness
$= \lfloor_\perp$

On the other hand, $(\underline{\lambda}p.\, zero) : Nat_\perp \to Nat_\perp$ is nonstrict, and:

$(\underline{\lambda}p.\, zero)((\underline{\lambda}n.\, one)\lfloor_\perp)$
$= [(\underline{\lambda}n.\, one)\lfloor_\perp /p]zero$, by the definition of application
$= zero$

In the first example, we must determine whether the argument to $(\underline{\lambda}m.zero)$ is proper or improper before binding it to *m*. We make the determination by simplifying the argument. If it simplifies to a proper value, we bind it to *m*; if it simplifies to $\lfloor_\perp$, we take the result of the application to be $\lfloor_\perp$. This style of ''argument first'' simplification is known as a *call-by-value* evaluation. It is the safe way of simplifying strict abstractions and their arguments. In the second example, the argument $((\underline{\lambda}n.one)\lfloor_\perp)$ need not be simplified before binding it to *p*.

We use the following abbreviation:

$(\text{let } x = e_1 \text{ in } e_2) \quad \text{for} \quad (\underline{\lambda}x.\, e_2)e_1$

Call this a *let* expression. It makes strict applications more readable because its ''argument first'' appearance matches the ''argument first'' simplification strategy that must be used. For example:

1. let $m = (\underline{\lambda}x.\, zero)\lfloor_\perp$ in $m\ plus\ one$
   $=$ let $m = zero$ in $m\ plus\ one$
   $= zero\ plus\ one = one$

2. let $m = one\ plus\ two$ in let $n = (\underline{\lambda}p.\, m)\lfloor_\perp$ in $m\ plus\ n$
   $=$ let $m = three$ in let $n = (\underline{\lambda}p.\, m)\lfloor_\perp$ in $m\ plus\ n$
   $=$ let $n = (\underline{\lambda}p.\, three)\lfloor_\perp$ in $three\ plus\ n$
   $=$ let $n = \lfloor_\perp$ in $three\ plus\ n$
   $= \lfloor_\perp$

Here is an example using the lifting construction; it uses the algebra of Example 3.11:

3.12 Example: *Unsafe arrays of unsafe values*

Domain $Unsafe = Array_\perp$,
where $Array = Nat \to Tr_1$ is from Example 3.11
(A in Example 3.11 becomes Tr_1)
and $Tr_1 = (\mathbb{B} \cup \{error\})_\perp$

Operations

new-unsafe: *Unsafe*

new-unsafe = *newarray*

access-unsafe: $Nat_\perp \to Unsafe \to Tr_\perp$

access-unsafe = $\underline{\lambda} n.\underline{\lambda} r.$ (*access n r*)

update-unsafe: $Nat_\perp \to Tr_\perp \to Unsafe \to Unsafe$

update-unsafe = $\underline{\lambda} n.\lambda t.\underline{\lambda} r.$ (*update n t r*)

The algebra models arrays that contain truth values that may be improper. The constant *new-unsafe* builds a proper array that maps all of its arguments to the *error* value. An array access becomes a tricky business, for either the index or the array argument may be improper. Operation *access-unsafe* must check the definedness of its arguments *n* and *r* before it passes them on to *access,* which performs the actual indexing. The operation *update-unsafe* is similarly paranoid, but an improper truth value may be stored into an array. Here is an evaluation of an expression (let $not_\perp = \underline{\lambda} t.\ not(t)$):

let *start-array* = *new-unsafe*
in *update-unsafe*(*one plus two*)($not_\perp(\lfloor\perp\rfloor)$)(*start-array*)

= let *start-array* = *newarray*
  in *update-unsafe*(*one plus two*)($not_\perp(\lfloor\perp\rfloor)$)(*start-array*)

= let *start-array* = ($\lambda n.$ *error*)
  in *update-unsafe*(*one plus two*)($not_\perp(\lfloor\perp\rfloor)$)(*start-array*)

= *update-unsafe*(*one plus two*)($not_\perp(\lfloor\perp\rfloor)$)($\lambda n.$ *error*)
= *update-unsafe*(*three*)($not_\perp(\lfloor\perp\rfloor)$)($\lambda n.$ *error*)
= *update* (*three*)($not_\perp(\lfloor\perp\rfloor)$)($\lambda n.$ *error*)
= [*three* $\mapsto$ $not_\perp(\lfloor\perp\rfloor)$] ($\lambda n.$ *error*)
= [*three* $\mapsto$ $\lfloor\perp\rfloor$] ($\lambda n.$ *error*)

You should study each step of this simplification sequence and determine where call-by-value simplifications were used.

3.3 RECURSIVE FUNCTION DEFINITIONS

If you read the description of the assembly principle for functions carefully, you will note that the definition $f(x_1, \cdots, x_n) = e$ does *not* permit *f* itself to appear in *e.* There is good reason: a recursive definition may not uniquely define a function. Here is an example:

$q(x) = x\ equals\ zero \to one\ [\!]\ q(x\ plus\ one)$

This specification apparently defines a function in $\mathbb{N} \to \mathbb{N}_\perp$. The following functions all satisfy *q*'s definition in the sense that they have exactly the behavior required by the equation:

$$f_1(x) = \begin{cases} one & \text{if } x = zero \\ \perp & \text{otherwise} \end{cases}$$

$$f_2(x) = \begin{cases} one & \text{if } x = zero \\ two & \text{otherwise} \end{cases}$$

$$f_3(x) = one$$

and there are infinitely many others. Routine substitution verifies that f_3 is a meaning of q:

for any $n \in Nat$, $n\ equals\ zero \rightarrow one\ [\!]\ f_3(n\ plus\ one)$
$= n\ equals\ zero \rightarrow one\ [\!]\ one$ by the definition of f_3
$= one$ by the definition of the choice function
$= f_3(n)$

Similar derivations also show that f_1 and f_2 are meanings of q. So which of these functions does q really stand for, if any? Unfortunately, the tools as currently developed are not sophisticated enough to answer this question. The problem will be dealt with in Chapter 6, because recursive function definitions are essential for defining the semantics of iterative and recursive constructs.

Perhaps when you were reading the above paragraph, you felt that much ado was made about nothing. After all, the specification of q could be typed into a computer, and surely the computer would compute function f_1. However, the computer gives an *operational semantics* to the specification, treating it as a program, and the function expressions in this chapter are mathematical values, not programs. It is clearly important that we use only those function expressions that stand for unique values. For this reason, recursive function specifications are suspect.

On the positive side, it is possible to show that functions defined recursively over abstract syntax arguments *do* denote unique functions. Structural induction comes to the rescue. We examine this specific subcase because denotational definitions utilize functions that are recursively defined over abstract syntax.

The following construction is somewhat technical and artificial, but it is sufficient for achieving the goal. Let a language L be defined by BNF equations:

B_1 ::= $Option_{11} \mid Option_{12} \mid \cdots \mid Option_{1m}$
B_2 ::= $Option_{21} \mid Option_{22} \mid \cdots \mid Option_{2m}$
$\cdots$
B_n ::= $Option_{n1} \mid Option_{n2} \mid \cdots \mid Option_{nm}$

and let $\mathbf{B}_i$ be a function symbol of type $\mathrm{B}_i \rightarrow D_i$ for all $1 \le i \le n$. For an $Option_{ij}$, let $\mathrm{S}_{ij1}, \mathrm{S}_{ij2}, \cdots, \mathrm{S}_{ijk}$ be the nonterminal symbols used in $Option_{ij}$, and let $\mathbf{B}_{ijl}$ represent the $\mathbf{B}_l$ appropriate for each S_{ijl} (for example, if $\mathrm{S}_{ijl}=\mathrm{B}_p$, then $\mathbf{B}_{ijl}=\mathbf{B}_p$).

3.13 Theorem:

If, for each B_i *in* L*'s definition and each* $Option_{ij}$ *of* B_i*'s rule, there exists an equation of form:*

$$\mathbf{B}_i(Option_{ij}) = f_{ij}(\mathbf{B}_{ij1}(\mathrm{S}_{ij1}), \mathbf{B}_{ij2}(\mathrm{S}_{ij2}), \cdots, \mathbf{B}_{ijk}(\mathrm{S}_{ijk}))$$

where f_{ij} is a function of functionality $D_{ij1} \times D_{ij2} \times \cdots \times D_{ijk} \to D_i$, then the set of equations uniquely defines a family of functions $\mathbf{B}_i$: $\mathrm{B}_i \to D_i$ for $1 \le i \le n$.

Proof: The proof is by a simultaneous structural induction on the rules of L. We show that each $\mathbf{B}_i(Option_{ij})$ is uniquely defined for a syntax tree of form $Option_{ij}$. Let $\mathbf{B}_i(Option_{ij})$ be defined as above. By the inductive hypothesis, for $1 \le l \le k$, each $\mathbf{B}_{ijl}(\mathrm{S}_{ijl})$ is uniquely defined. Since f_{ij} is a function, its application to the $\mathbf{B}_{ijl}(\mathrm{S}_{ijl})$'s yields a unique answer, so $\mathbf{B}_i(Option_{ij})$ is uniquely defined. The equations for all the $Option_{ij}$'s of rule B_i taken together define a unique function $\mathbf{B}_i$: $\mathrm{B}_i \to D_i$. □

3.4 RECURSIVE DOMAIN DEFINITIONS

We have used an equation format for naming semantic domains. For example, $Payroll\text{-}record = String \times Rat \times Rat$ associates the name *Payroll-record* with a product domain. In later chapters, we will see that certain programming language features require domains whose structure is defined in terms of themselves. For example, $Alist = Unit + (A \times Alist)$ defines a domain of linear lists of *A*-elements. Like the recursively defined operations mentioned in the previous section, a domain may not be uniquely defined by a recursive definition.

What's more, equations such as $F = F \to Nat$, specifying the collection of functions that accept themselves as arguments to produce numeric answers, apparently have no solution at all! (It is not difficult to show that the cardinality of the collection of all functions from *F* to *Nat* is larger than *F*'s cardinality.) Chapter 11 provides a method for developing solutions to recursive domain definitions.

3.5 SUMMARY

Here is a summary of the domain constructions and their operations that were covered in this chapter.

1. *Domain construction*: primitive domain, e.g., natural numbers, truth values

   Operation builders: the operations and constants that are presented in the semantic algebra. For example, the choice function is presented with the *Tr* algebra; it is $(e_1 \to e_2 \,[]\, e_3) \in A$, for $e_1 \in \mathbb{B}$, $e_2, e_3 \in A$.

   Simplification properties: as dictated by the definition of the operations, e.g., *not*(*true*) simplifies to *false* because the pair (*true*, *false*) is found in the graph of the operation *not*. The simplification properties of the choice function are:
   $true \to e_2 \,[]\, e_3 = e_2$

$false \rightarrow e_2 \; [] \; e_3 = e_3$

2. Domain construction: product space $A \times B$

Operation builders:
$fst : A \times B \rightarrow A$
$snd : A \times B \rightarrow B$
$(a, b) \in A \times B$ for $a \in A$ and $b \in B$
$\downarrow i : A_1 \times A_2 \times \cdots \times A_i \times \cdots \times A_n \rightarrow A_i$, for $1 \leq i \leq n$

Simplification properties:
$fst(a, b) = a$
$snd(a, b) = b$
$(a_1, a_2, \cdots, a_i, \cdots, a_n) \downarrow i = a_i$, for $1 \leq i \leq n$

3. Domain construction: disjoint union (sum) space $A + B$

Operation builders:
$\text{in}A : A \rightarrow A + B$
$\text{in}B : B \rightarrow A + B$
$(\text{cases } d \text{ of is}A(x) \rightarrow e_1 \; [] \; \text{is}B(y) \rightarrow e_2 \text{ end}) \in C$
for $d \in A+B$, $(\lambda x.e_1) : A \rightarrow C$, and $(\lambda y.e_2) : B \rightarrow C$

Simplification properties:
$(\text{cases } \text{in}A(a) \text{ of is}A(x) \rightarrow e_1 \; [] \; \text{is}B(y) \rightarrow e_2 \text{ end}) = [a/x]e_1$
$(\text{cases } \text{in}B(b) \text{ of is}A(x) \rightarrow e_1 \; [] \; \text{is}B(y) \rightarrow e_2 \text{ end}) = [b/y]e_2$

3a. Domain construction: list space A^*

Operation builders:
$nil : A^*$
$cons : A \times A^* \rightarrow A^*$
$hd : A^* \rightarrow A$
$tl : A^* \rightarrow A^*$
$null : A^* \rightarrow Tr$

Simplification properties:
$hd(a \; cons \; l) = a$
$tl(a \; cons \; l) = l$
$null(nil) = true$
$null(a \; cons \; l) = false$

4. Domain construction: function space $A \rightarrow B$

Operation builders:
$(\lambda x.e) \in A \to B$ such that for all $a \in A$, $[a/x]e$ has a unique value in B.
$g(a) \in B$, for $g : A \to B$ and $a \in A$

$(g\ a)$ abbreviates $g(a)$
$[x \mapsto v]g$ abbreviates $(\lambda x'.\ x'\ equals\ x \to v\ []\ g(x'))$
$[a/x]e$ denotes the substitution of expression a for all free occurrences of identifier x in expression e

Simplification properties:
$g(a) = [a/x]e$, where g is defined equationally as $g(x) = e$
$(\lambda x.\ e)a = [a/x]e$
$([x \mapsto v]g)x = v$
$([x \mapsto v]g)y = g(y)$, where $y \neq x$

5. Domain construction: lifted space $A_\perp$

Operation builder:
$(\underline{\lambda} x.\ e) : A_\perp \to B_\perp$, for $(\lambda x.\ e) : A \to B_\perp$

(let $x = e_1$ in e_2) abbreviates $(\underline{\lambda} x.\ e_2)e_1$
(Note: the above expression occasionally abbreviates $(\lambda x.e_2)e_1$ when $e_1 \in A$ and A is an *unlifted* domain; that is, A has no $\perp$ element.)

Simplification properties:
$(\underline{\lambda} x.\ e_2)e_1 = [e_1/x]\,e_2$, when e_1 is a proper member of $A_\perp$, i.e., $e_1 \neq \perp$
$(\underline{\lambda} x.\ e)\perp = \perp$
(let $x = e_1$ in e_2) $= [e_1/x]e_2$, when e_1 is a proper member of $A_\perp$
(let $x = \perp$ in e) $= \perp$

SUGGESTED READINGS

Semantic domains: Gordon 1979; Scott 1976, 1982; Stoy 1977; Strachey 1973; Tennent 1981
Semantic algebras: Bauer & Wossner 1982; Burstall & Goguen 1977, 1981; Cohn 1981; Gratzer 1979; Mosses 1979a, 1983, 1984

EXERCISES

1. Given the algebras of natural numbers and truth values, simplify the following expressions. Show all the steps in your simplification.

a. ((*six equals* (*two plus one*)) → *one* [] (*three minus one*)) *plus two*
b. (*two equals* (*true* → *one* [] *two*)) *and true*
c. *not*(*false*) → *not*(*true*) [] *not*(*true*)

2. Define primitive semantic algebras for each of the following:

a. The musical notes playable on a piano.
b. The U.S. (or your favorite) monetary system.
c. The ''colors of the rainbow.''

3. Using the operations defined in Section 3.5, simplify the following (note that we use identifiers $m,n \in Nat$, $t \in Tr$, $p \in Tr \times Tr$, $r \in Tr+Nat$, and $x,y \in Nat_\perp$):

a. *fst*((λ*m.zero*)*two*, (λ*n.n*))
b. (λ*p*. (*snd p*, *fst p*))(*true*, (*two equals one*))
c. ((λ*r*. cases *r* of
   is*Tr*(*t*) → (λ*m*. *zero*)
   [] is*Nat*(*n*) → (λ*m*. *n*)
   end)(in*Nat*(*two*)))(*one*)
d. cases (*false* → in*Nat*(*one*) [] in*Tr*(*false*)) of
   is*Tr*(*t*) → *true or t*
   [] is*Nat*(*n*) → *false* end
e. (λ*x*.λ*y*.*y*(*x*))(*one*)(λ*n*. *n plus two*)
f. ((λ_*n*. [*zero* ↦ *n*](λ*m*. *zero*))(*two*)) *zero*
g. (λ*x*. (λ*m*. *m equals zero* → *x* [] *one*)(*two*))(⊥)
h. (λ_*m*. *one*)(*true* → ⊥ [] *zero*)
i. (λ(*x*, *y*). (*y*, *x*))(⊥, (λ_*n*. *one*)⊥)
j. let *m* = ⊥ in *zero*
k. let *m* = *one plus two* in let *n* = *m plus one* in (λ*m*. *n*)
l. let *m* = (λ*x*. *x*)*zero* in let *n* = (*m equals zero* → *one* [] ⊥) in *m plus n*
m. let *m* = *one* in let *m* = *m plus two* in *m*

4. Let *Tr* = { *tt*, *ff* }. List all the elements in these domains:

a. $Unit + ((Tr \times Tr)_\perp)$
b. $(Unit + (Tr \times Tr))_\perp$
c. $(Unit_\perp + (Tr_\perp \times Tr_\perp))$
d. $(Unit + Tr) \times Tr$
e. $Unit \to Tr_\perp$
f. $(Unit \to Tr)_\perp$

5. a. Complete the definition of the algebra in Example 3.7 by defining these operations:

i. $update\text{-}payrate : Rat \times Payroll\text{-}rec \to Payroll\text{-}rec$
ii. $update\text{-}hours : Rat \times Payroll\text{-}rec \to Payroll\text{-}rec$

b. Use the completed algebra to define a payroll record stating that

i. ''Jane Doe'' has been assigned a payroll record.
ii. She is moved to the night shift.
iii. She works 38 hours that week.
iv. Her payrate goes to 9.00.

Next, write an expression denoting Jane Doe's pay for the week.

c. What other operations should this algebra possess to make it more useful for defining the semantics of a payroll system?

6. Using the algebra of Example 3.9, simplify these list-valued expressions:

a. $(hd(one\ cons\ nil)\ cons\ nil)$
b. $(\lambda l.\ (null\ l) \to (zero\ cons\ nil)\ [] (one\ cons\ nil))(tl(one\ cons\ nil))$
c. $((one\ cons\ (two\ cons\ nil))\ cons\ nil)$
d. $(\lambda l.\ tl\ l)(tl(zero\ cons\ nil))$

7. Design an algebra called *Set-of-A* (where *A* is any primitive domain) with operations:

$$empty\text{-}set : Set\text{-}of\text{-}A$$
$$make\text{-}singleton : A \to Set\text{-}of\text{-}A$$
$$member\text{-}of : A \times Set\text{-}of\text{-}A \to Tr$$
$$union : Set\text{-}of\text{-}A \times Set\text{-}of\text{-}A \to Set\text{-}of\text{-}A$$

The operations are to satisfy the expected set theoretic properties, e.g., for all $a \in A$, $member\text{-}of(a, make\text{-}singleton(a)) = true$. (Hint: use the domain $A \to Tr$ in the definition.)

8. Modify the dynamic array algebra of Example 3.11 so that arrays carry with them upper and lower bounds. The operations are altered so that:

a. $newarray : Nat \times Nat \to Array$ establishes an empty array with lower and upper bounds set to the values of the two arguments.
b. *access* and *update* both compare their index argument against the lower and upper bounds of their array argument. (Hint: use $Array = (Nat \to A) \times Nat \times Nat$.)

9. Use the algebra of payroll records in Example 3.6 and the array of Example 3.11 to derive an algebra describing data bases of payroll records. A data base indexes employees by identity numbers. Operations must include ones for:

a. Adding a new employee to a data base.
b. Updating an employee's statistics.
c. Producing a list of employee paychecks for all the employees of a data base.

10. Specify algebras that would be useful for defining the semantics of the grocery store inventory system that is mentioned in Exercise 6 of Chapter 1.

11. a. Describe the graphs of the following operations:

    i. $(\_ \to a \ [] \ b) : \mathbb{B} \to D$, for $a, b \in D$
    ii. $fst : A \times B \to A$ and $snd : A \times B \to B$
    iii. $\text{in}A : A \to A + B$, $\text{in}B : B \to A + B$, and (cases _ of is$A(a) \to f(a)$ [] is$B(b) \to g(b)$ end) : $A + B \to C$, for $f : A \to C$ and $g : B \to C$
    iv. $(\lambda x.\, E) : A \to B$, for expression E such that for all $a \in A$, $[a/x]E$ is a unique value in B
    v. $(\underline{\lambda} x.\, E) : A_\perp \to B_\perp$, for $(\lambda x.\, E) : A \to B_\perp$

    b. Using the definitions in part a, prove that the simplification rules in Section 3.5 are *sound*; that is, for each equality $L = R$, prove that the set-theoretic value of L equals the set-theoretic value of R. (Note: most of the proofs will be trivial, but they are still well worth doing, for they justify all of the derivations in the rest of the book!)

12. The assembly and disassembly operations of compound domains were chosen because they possess certain *universal properties* (the term is taken from *category theory*; see Herrlich and Strecker 1973). Prove the following universal properties:

    a. For arbitrary functions $g_1 : C \to A$ and $g_2 : C \to B$, there exists a unique function $f : C \to A \times B$ such that $fst \circ f = g_1$ and $snd \circ f = g_2$.
    b. For arbitrary functions $g_1 : A \to C$ and $g_2 : B \to C$, there exists a unique function $f : A + B \to C$ such that $f \circ \text{in}A = g_1$ and $f \circ \text{in}B = g_2$.
    c. For arbitrary function $g : A \times B \to C$, there exists a unique function $f : A \to B \to C$ such that $(f(a))(b) = g(a, b)$.
    d. For arbitrary function $g : A \to B_\perp$, there exists a unique function $f : A_\perp \to B_\perp$ such that $f(\perp) = \perp$ and $f(a) = g(a)$, for $a \in A$.

13. The function notation defined in this chapter is a descendant of a symbol manipulation system known as the *lambda calculus.* The abstract syntax of lambda expressions is defined as:

    $$E ::= (E_1\ E_2) \mid (\lambda I.E) \mid I$$

    Lambda expressions are simplified using the β-rule:

    $$((\lambda I.E_1)E_2) \Rightarrow [\,E_2/I\,]E_1$$

    which says that an occurrence of $((\lambda I.E_1)E_2)$ in a lambda expression can be rewritten to $[\,E_2/I\,]E_1$ in the expression. All bound identifiers in E_1 are renamed so as not to clash with the free identifiers in E_2. We write $M \Rightarrow^* N$ if M rewrites to N due to zero or more applications of the β-rule.

    a. Using the β-rule, simplify the following expressions to a final (*normal*) form, if one exists. If one does not exist, explain why.

    i. $((\lambda x.\, (x\, y))(\lambda z.\, z)$
    ii. $((\lambda x.\, ((\lambda y.\, (x\, y))x))(\lambda z.\, w))$
    iii. $((((\lambda f.\, (\lambda g.\, (\lambda x.\, ((f\, x)(g\, x)))))(\lambda m.(\lambda n.\, (n\, m))))(\lambda n.\, z))p)$
    iv. $((\lambda x.(x\, x))\, (\lambda x.\, (x\, x)))$

v. $((\lambda f.((\lambda g.((f f) g))(\lambda h.(k h))))(\lambda x.(\lambda y.y)))$
vi. $(\lambda g.((\lambda f.((\lambda x.(f(x x)))(\lambda x.(f(x x)))))g))$

b. In addition to the β-rule, the lambda calculus includes the following two rules:

α-*rule* : $(\lambda x.E) \Rightarrow (\lambda y.[y/x]E)$
η-*rule* : $(\lambda x.(E\,x)) \Rightarrow E$ where x does not occur free in E

Redo the simplifications of i-vi in a, making use of the η-rule whenever possible. What value do you see in the α-rule?

c. A famous result regarding the lambda calculus is that it can be used to simulate computation on truth values and numbers.

i. Let **true** be the name of the lambda expression $(\lambda x.\lambda y.x)$ and let **false** be the name of the lambda expression $(\lambda x.\lambda y.y)$. Show that $((\mathbf{true}\,E_1)\,E_2) \Rightarrow^* E_1$ and $((\mathbf{false}\,E_1)\,E_2) \Rightarrow^* E_2$. Define lambda expressions **not**, **and**, and **or** that behave like their Boolean operation counterparts, e.g., $(\mathbf{not\ true}) \Rightarrow^* \mathbf{false}$, $((\mathbf{or\ false})\mathbf{true}) \Rightarrow^* \mathbf{true}$, and so on.

ii. Let **0** be the name of the lambda expression $(\lambda x.\lambda y.y)$, **1** be the name of the expression $(\lambda x.\lambda y.(x\,y))$, **2** be the name of the expression $(\lambda x.\lambda y.(x(x\,y)))$, **3** be the name of the expression $(\lambda x.\lambda y.(x(x(x\,y))))$, and so on. Prove that the lambda expression **succ** defined as $(\lambda z.\lambda x.\lambda y.(x\,((z\,x)\,y)))$ rewrites a number to its successor, that is, $(\mathbf{succ\ n}) \Rightarrow^* \mathbf{n+1}$. There also exists a lambda expression **pred** such that $(\mathbf{pred\ 0}) \Rightarrow^* \mathbf{0}$ and $(\mathbf{pred\ n+1}) \Rightarrow^* \mathbf{n}$ (but we won't give it here, as it is somewhat ungainly).

d. Recursively defined functions can also be simulated in the lambda calculus. First, let **Y** be the name of the expression $(\lambda f.((\lambda x.(f(x\,x)))(\lambda x.(f(x\,x)))))$.

i. Show that for any expression E, there exists an expression W such that $(\mathbf{Y}\,E) \Rightarrow^* (W\,W)$, and that $(W\,W) \Rightarrow^* (E\,(W\,W))$. Hence, $(\mathbf{Y}\,E) \Rightarrow^* E(E(E(\cdots E(W\,W)\cdots)))$.

ii. Using the lambda expressions that you defined in the previous parts of this exercise, define a recursive lambda expression **add** that performs addition on the numbers defined in part ii of b, that is, $((\mathbf{add\ m})\,\mathbf{n}) \Rightarrow^* \mathbf{m+n}$. (Hint: first define an expression **IF** such that $(((\mathbf{IF\ 0})\,E_1)\,E_2) \Rightarrow^* E_1$ and $(((\mathbf{IF\ n+1})\,E_1)\,E_2) \Rightarrow^* E_2$.) Say that your definition of **add** has the form $\mathbf{add} = \lambda x.\lambda y.\ \cdots \mathbf{add} \cdots$. Let **ADD** be the lambda expression $(\mathbf{Y}\,(\lambda h.\lambda x.\lambda y.\ \cdots h \cdots))$. Show that $((\mathbf{ADD\ m})\,\mathbf{n}) \Rightarrow^* \mathbf{m+n}$.

14. a. Give examples of recursive definitions of the form $n = \cdots n \cdots$ that have no solution; have multiple solutions; have exactly one solution.

b. State requirements under which a recursively defined function $f: Nat \rightarrow A$ has a unique solution. Use mathematical induction to prove your claim. Next, generalize your answer for recursively defined functions $g: A^* \rightarrow B$. How do your requirements resemble the requirements used to prove Theorem 3.13?

15. Show that each of the following recursively defined sets has a solution.

a. $Nlist = Unit + (\mathbb{N} \times Nlist)$
b. $N = Unit + N$
c. $A = A$
d. $Blist = \mathbb{B} \times Blist$.

Do any of them have a unique solution?