

## Chapter 3

# Items, Variables, Values and Types

%% First Evaluation Complete...

### 3.1 Items and Values

Rosetta's basic semantic unit is called an *item*. Item structures result when Rosetta descriptions are parsed prior to manipulation. Although most users will never deal directly with items, they present an effective way to describe the relationships between variables, values and types.

Informally, an item consists of a *label* naming the item, a *value* the item represents, and a *type* from which specific item values must be chosen. When any structure is defined in a Rosetta specification, an item is created with the specified label. Variables, constants, terms, even facets themselves are items in a Rosetta specification. When a label is referred to in a specification, it refers to the value of the item it is associated with. An item's set of potential values is delineated by its associated type. In a legal Rosetta specification, every item's value is an element of its associated type. A more complete description of items can be found in Chapter 7.

Value items, or simply values, represent items that can be used as values for other items. There are three general classes of values: (i) elements; (ii) composite items; and (iii) functions. Elemental values represent primitive, atomic values that are directly manipulated by Rosetta. Elemental values include such things as integers, naturals, characters, bits and boolean values. Traditional programming languages refer to elemental values as scalar. Composite values are constructed from other values. Composite values include such things as sequences, sets, and facets.

Function values represent operations that by definition exhibit properties of mathematical functions.

The name `universal` is used to refer to all values. Universal is itself a type, but refers to the Rosetta term language.

All Rosetta types are *sets* where a set is simply a packaged collection of values. Functions and properties for sets are defined completely in Section 3.6.1. Throughout this document, the terms `set` and `subtype` are used interchangeably to refer to a subset of a set. The term `type` refers to any possible set.

The notation `a::T` is used to declare a new Rosetta item and constrain its type. Appearing in a declarative region, "`a::T`" declares a new item labeled `a` of type `T`. Specifically, `a` is a new item constrained by the type constraint `a in T` where `T` is a set. If the notation `a::T` appears in a non-declarative section, it serves as a mechanism for explicitly specifying the type of an expression when type inference produces ambiguous results.

By convention, we say that `v::T` in a declarative region of a facet declares a *variable* item of type `T` whose value is an element of set `T`. No expression is included to constrain the value of `v`, thus its value is not known.

Similarly, the notation `v::T is c` defines a **constant** item of type `T` whose value is given by the expression `c`. The constraint `c in T` must hold for the constant declaration to be consistent. Function definition is an exception to this rule where the expression `c` becomes the expression associated with the function, not an expression evaluated to obtain a value. This notation will be explained and used extensively in the following sections.

## 3.2 Elements

By definition, elements are values that are atomic and cannot be decomposed. Element types are sets of such values. Numbers such as `1`, `5.32`, and `-32`, characters such as `'a'`, `'B'`, and `'1'`, and boolean values such as `true` and `false` represents such atomic values. In contrast, composite values such as sequences and sets are not elemental in that each is defined by describing its contents. Element values are frequently called *scalars* in traditional programming languages.

The type `element` is comprised of the types `number`, `character`, and any new values created by enumeration declarations. The `element` type is largely a semantic construction with no common operations over all members of the type other than simple equality (`=`) and inequality (`/=`) operations.

## 3.3 Numbers

Numeric types include standard sets of values associated with traditional number systems. Predefined numeric types include `real`, `integer`, `natural`, `bit`, `imaginary`, `complex` and `boolean` and are listed in the following table:

<i>Type</i>	<i>Format</i>	<i>Subtype Of</i>
<code>complex</code>	<code>1+2*j</code> , <code>3*e(4*j)</code>	<code>number</code>
<code>real</code>	<code>-123.456</code> , <code>123.456</code> , <code>1.234e56</code>	<code>complex</code>
<code>posreal</code>	<code>123.456</code> , <code>1.234e56</code>	<code>real</code>
<code>negreal</code>	<code>-123.456</code> , <code>-1.234e56</code>	<code>real</code>
<code>rational</code>	<code>123/456</code>	<code>real</code>
<code>integer</code>	<code>123,0,-123</code>	<code>rational</code>
<code>natural</code>	<code>0,123,</code>	<code>integer</code>
<code>posint</code>	<code>123</code>	<code>natural</code>
<code>negint</code>	<code>-123</code>	<code>integer</code>
<code>bit</code>	<code>1,0</code>	<code>natural</code>
<code>imaginary</code>	<code>j, 5*j</code>	<code>complex</code>
<code>boolean</code>	<code>true, false</code>	<code>number</code>
<code>number</code>	<i>Any element of the above types</i>	<code>element</code>

Predefined operators defined over `number` and its subtypes include: (Assuming `A` and `B` are numbers)

```
% Ordering relations moved to boolean, real, and imaginary.
```

<i>Operation</i>	<i>Format</i>	<i>Valid For</i>
Negation	<code>- A</code>	<code>number</code>

### 3.3.1 Complex Numbers

Complex numbers form the most basic Rosetta number. All traditional number values are subtypes of `complex`. Traditional operations such as addition and subtraction are defined over complex numbers as anticipated. Projection functions extract real and imaginary values from complex values. `re` returns the

magnitude of a number's real part while `im` returns the magnitude of the imaginary part. For any complex value `n` expressed in the cartesian form:

```
n = re(n)+im(n)*j
```

The polar form may also be used. For any complex value `n` expressed in the polar form:

```
n = mag(n)*e^(arg(n)*j)
```

The `conj` operator evaluates to the complex conjugate of its complex argument. The operator `mag` evaluates to the magnitude of the vector associated with a complex number in the complex plane. Additional predefined operators defined over `complex` and its subtypes include: (Assuming `A` and `B` are numbers)

<i>Operation</i>	<i>Format</i>	<i>Valid For</i>
Addition and Subtraction	A+B, A-B	complex
Multiplication and Division	A*B, A/B	complex
Power	, A^ B	complex
Square Root	sqrt(A)	complex
Imaginary and Real	im(A), re(A)	complex
Magnitude, Conjugate and Argument	abs(A), conj(A), arg(A)	complex
Trig functions	sin(A), cos(A), tan(A) arcsin(A), arccos(A), arctan(A) sinh(A), cosh(A), tanh(A) arcsinh(A), arccosh(A), arctanh(A)	complex
Log and exponential	exp(A), log(A), log10(A), log2(A)	complex

## Real Numbers

The type `real` can be defined as the subtype of `complex` such that `im(x)=0`. Formally:

```
real::subtype(complex) is sel(x::complex | im(x)=0);
```

The additional operations `min`, `max`, `floor`, `ceiling`, `round`, and `sgn` are defined over `real` as well as traditional ordering relationships:

<i>Operation</i>	<i>Format</i>	<i>Valid For</i>
Minimum and Maximum	A min B, A max B	real
Floor, Ceiling and Truncate	floor(x), ceiling(x), trunc(x)	real
Round	round(x)	real
Signum	sgn(x)	real
Ordering Relations	A<B, A=<B, A>B, A>=B	real

`min` and `max` evaluate to the minimum and maximum value of their arguments respectively.

`floor` evaluates to the greatest integer number less than or equal to its argument. Conversely, `ceiling` evaluates to the least integer number greater than or equal to its argument. `trunc` always truncates its value towards zero.

`round` evaluates to `ceiling` if the fractional part of its argument is greater than or equal to 0.5. Otherwise, it evaluates to `floor`.

`sgn` is defined formally as (for  $x \neq 0$ ):

```
sgn(x::real)::integer is if -(x=0) then x / abs(x) else 1 end if;
```

Classical ordering relationships are provided and defined the traditional manner.

The constant values `e` and `pi` are provided as `real` numbers:

<i>Constant</i>	<i>Format</i>	<i>Valid For</i>
Exponential	<code>e</code>	<code>number</code>
Pi	<code>pi</code>	<code>real</code>

## Positive and Negative Real Numbers

The subtype `posreal` is defined as the subtype of `real` such that all values are greater than 0.0. The subtype `negreal` is defined as the subtype of `real` such that all values are less than 0.0. Formally:

```
posreal::subtype(real) is sel(x::real | x > 0);
negreal::subtype(real) is sel(x::real | x < 0);
```

No additional operators are defined over `posreal` or `negreal` beyond those defined for `real`.

Note that some operations such as negation are defined over `posreal` and `negreal`, but are not closed over those types.

## Imaginary Numbers

The type `imaginary` is the subtype of `complex` such that its real part is 0. Formally:

```
imaginary::subtype(complex) is sel(x::complex | re(x)=0);
```

Imaginary numbers are formed by multiplying a real number by the predefined imaginary constant `j`. One can think of this as a conversion operation from real number types to imaginary numbers. Multiplying an imaginary number by the imaginary constant results in a non-imaginary value. Thus, `5*j*j` is equivalent to `-5` as expected.

The following operators are defined over `imaginary` beyond those defined for `complex`:

<i>Operation</i>	<i>Format</i>	<i>Valid For</i>
Minimum and Maximum	<code>A min B, A max B</code>	<code>imaginary</code>
Ordering Relations	<code>A&lt;B,A=&lt;B,A&gt;B,A&gt;=B</code>	<code>imaginary</code>

`min` and `max` evaluate to the minimum and maximum value of their arguments respectively.

Classical ordering relationships are provided and defined the traditional manner.

## Rational Numbers

The type `rational` is the subtype of `real` such that each value is calculated as one integer value divided by another. Formally:

```
rational::subtype(real) is sel(x::real | exists(y,z::integer | x=y/z and z/=0));
```

No additional operators are defined over `rational` beyond those defined for `real`.

## Integer Numbers

The subtype `integer` is a subtype of `rational` such that all values are discrete. Formally:

```
integer::subtype(rational) is sel(x::rational | floor(x)=ceiling(x));
```

The additional `mod`, `div` and `rem` functions are defined over `integer` in addition to operators defined over `rational`. All operators are defined in the traditional fashion.

<i>Operation</i>	<i>Format</i>	<i>Valid For</i>
Modulo Arithmetic	<code>x mod y</code>	<code>integer</code>
Integer Division	<code>x div y</code>	<code>integer</code>
Integer Remainder	<code>x rem y</code>	<code>integer</code>

`integer` is closed under `+`, `-`, `*`, but not under `/`, or trigonometric operations.

## Natural Numbers

The subtype `natural` can be defined as the subtype of `integer` such that all values are greater than or equal to 0. Formally:

```
natural::subtype(integer) is sel(x::integer | x >= 0);
```

No additional operators are defined over `natural` beyond those defined for `integer`.

Note that some operations such as negation are defined over `natural` but are not closed over `natural`.

## Positive and Negative Integer Numbers

The subtype `posint` is defined as the subtype of `integer` such that all values are greater than 0. The subtype `negint` is defined as the subtype of `integer` such that all values are less than 0. Note that `posint` is a subtype of `natural` while `negint` is not. Formally:

```
posint::subtype(natural) is sel(x::natural | x > 0);  
negint::subtype(integer) is sel(x::integer | x < 0);
```

No additional operators are defined over `posint` or `negint` beyond those defined for `integer`.

Note that some operations such as negation are defined over `posint` and `negint` integers, but are not closed over those types.

## Bits

The subtype `bit` can be defined as a subtype of `natural` consisting of the values 0 and 1. Formally:

```
bit::subtype(natural) is {1,0};
```

Additional operators defined over `bit` include: (Assume the declarations `x,y::bit` and the definitions `a=%x`, and `b=% y`):

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
%	% 1, % true	Converts bits and boolean.
Inverse	not x	% -a
Conjunction and Disjunction	x and y, x or y	%(a and b), %(a or b)
Negated Conjunction and Disjunction	x nand y, x nor y	not(x and y), not(x or y)
Exclusive or and nor	x xor y, x xnor y	%(a xor b), %(a xnor b)

The % operation translates between `bit` and `boolean` in such a way that 1 is isomorphic with `true` and 0 is isomorphic with `false`.

It should be noted that as a subtype of `natural`, `bit` is not closed under arithmetic operations such as plus and minus.

### 3.3.2 Lexical Structure of Number Constants

Numeric constants are represented by a strings of digits and optional sign, decimal point, exponential and radix indicators. Specifically:

- A number may be preceded by an optional “-” operator that inverts the sign of its argument. The number `-123` is equivalent to negative 123.
- A single decimal point may be included in a number. The number `1.23` is interpreted in the traditional manner.
- A single exponent indicator may be included in a number. The number `1.234e7` is equivalent to 1.234 times 10 to the 7<sup>th</sup> power.
- An optional radix may be included using the notation `R\N\eE` where R is the radix value (up to 16), N is a number, and E is an exponent. The radix value and the exponent value are always expressed in base 10 while the number value is specified in the indicated radix. The number `2\10001.1001\` is the base 2 representation of the binary real value 10001.1001. Note that “\” is not a function, but a part of the number token itself.
- Imaginary numbers are formed by multiplying a `real` value by the complex root, `j`. The number `5.3*j` is interpreted in the traditional fashion.
- Complex numbers are formed by adding a real number to an imaginary number or using the polar form. The number `5+2*j` is interpreted as the complex number whose real part has magnitude 5 and imaginary part has magnitude 2. The number `6*e(pi*j)` is interpreted as the complex number whose magnitude is 6 and whose argument is `pi`.

### 3.3.3 Boolean

The Rosetta `boolean` type is defined by the two element set `{true,false}` and is a subtype of `number`. Although `boolean` is a number type, it is not a subtype of `complex` or `natural`.

The following operators are defined over `boolean` beyond those defined for `complex`:

<i>Operation</i>	<i>Format</i>	<i>Valid For</i>
Minimum and Maximum	A min B, A max B	<code>boolean</code>
Ordering Relations	A<B,A=<B,A>B,A>=B	<code>boolean</code>
Bit/Boolean Conversion	%A	<code>boolean</code>
Logical Operations	A and B, A or B, A xor B A nand B, A nor B, A xnor B not A	<code>boolean</code>

`min` and `max` evaluate to the minimum and maximum value of their arguments respectively. The maximum `boolean` value is `true` and the minimum `false`. Classical ordering relationships are provided and defined the traditional manner given the definitions of `true` and `false` as maximum and minimum values respectively.

The % operator converts between `boolean` and `bit` in the classical manner.

Classical logical operations including `and`, `or`, `xor`, `nand`, `nor`, `xnor`, and `not` are provided. `and` and `or` are synonyms for `min` and `max` while `not` is a synonym for the unary operation “-”.

When treated as numeric values, `true` and `false` follow the following equivalence and ordering rules:

<i>Property</i>	<i>Meaning</i>
<code>false = -true</code>	<code>false</code> is equivalent to not <code>true</code>
<code>-false = true</code>	<code>true</code> is equivalent to not <code>false</code>
<code>forall(x::number   x /= true =&gt; x &lt; true)</code>	<code>true</code> is the greatest number
<code>forall(x::number   x /= false =&gt; x &gt; false)</code>	<code>false</code> is the least number

Boolean values obey ordering laws, but addition, subtraction, multiplication, division and other traditional operations are not defined. As numbers, `true` acts like positive infinity and `false` acts like negative infinity. Consequences of this convention are numerous and useful. They include: (i) `max` is semantically the same as `or`; (ii) `min` is semantically the same as `and`; and (iii) `=<` is semantically the same as implication (`=>`) and `>=` is semantically the same as implied by (`<=`). The same laws apply to these operations in all cases, and the different signs are taken to be synonyms of each other, maintained here for the sake of historical recognition. Thus, we have the following table defining `min` and `max` over `true` and `false`:

<i>A</i>	<i>B</i>	<i>A min B</i>	<i>A max B</i>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>

This is identical to the truth table defining `and` and `or`. The negation operator, `-`, also follows directly from the numeric interpretation of `boolean`. The greatest positive number negated is the least negative number. Thus, `-true = false`. As negation is its own inverse, we know that `-(-x) = x` for any boolean value `x`. Thus, `-false = true`. The resulting truth table has the form:

<i>A</i>	<i>-A</i>
<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>

Definitions for other logical operations follow directly. Of particular interest is the definition of implication as:

$$A \Rightarrow B == \neg A \text{ or } B$$

By definition, this is equivalent to `-A max B`. Again, consider the truth table generated by the definition of `true` and `false` as numeric values:

<i>A</i>	<i>B</i>	<i>-A</i>	<i>-A max B</i>
<code>false</code>	<code>false</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>

This is semantically the same as the definition of implication. Reverse implication works similarly and the definition of equivalence (`A=B`) is consistent with the above definition. Further, when values are restricted to `boolean`, the following equivalences hold:

$$A \Rightarrow B == A \Leftarrow B$$

$$A \Leftarrow B == A \Rightarrow B$$

$$A \Leftarrow B \text{ and } B \Leftarrow A == A = B$$

It should be noted that Rosetta does not define logical equivalence, `iff`, separately from numerical equivalence. Given the mathematical definition of booleans, the normal equivalence operations are sufficient.

**Example 11 (Number Constants)** *Examples of defining number constants, including `complex`, its subtypes, and `boolean` include:*

Number	Interpretation
<code>12</code>	The standard decimal constant 12
<code>-12</code>	The standard decimal constant -12
<code>1.2</code>	The standard decimal constant 1.2
<code>1.23e4</code>	The decimal constant $1.23 * 10^4$
<code>1.23e-4</code>	The decimal constant $1.23 * 10^{-4}$
<code>16\E.1F\e5</code>	The hexadecimal constant $E.1F_{16} * 16^5$
<code>2\1101\e-7</code>	The binary constant $1101_2 * 2^{-7}$
<code>false</code>	The boolean constant false
<code>true</code>	The boolean constant true

### 3.4 Characters

The type `character` is a subtype of `element` and is defined as the collection of unicode values.

Given `a::character`, `b::character`, and `n::natural` in the range of Unicode code values, operators on `character` include:

Operation	Format	Definition
Ord and character	<code>ord(a)</code> , <code>char(n)</code>	Unicode value
Ordering Relations	<code>a&lt;b</code> , <code>a=&lt;b ...</code>	<code>ord(a)&lt;ord(b)</code> , <code>ord(a)=&lt;ord(b) ...</code>
Raise and lower case	<code>uc(a)</code> , <code>dc(a)</code>	Raise/lower case

#### 3.4.1 ASCII Type

The type `ascii` is a subtype of `character` and is defined as the subset of characters that represent ASCII values. Formally:

```
ascii::subtype(character) is map(char,{0,..255});
```

No new functions are defined on ASCII other than those defined over `character` values.

```
%% Should add other character sets besides ascii.
```

#### Lexical Structure of Character Constants

Unicode literals are expressed using the standard notation `'X'` where `X` is a Unicode character, `'U+XXXX'` where `XXXX` is a 4-digit, hexadecimal number. The enclosing ticks are significant and must be included. `character` values that have no printable form must be specified using their Unicode hex value.

**Example 12 (Character Constants)** *Examples of defining character constants:*

Character	Interpretation
<code>'U+DD01'</code>	Unicode character associated with hex <code>DD01</code>
<code>'1'</code>	ASCII character 1
<code>'a'</code>	ASCII character <code>a</code>

```
%% Need some decent examples of unicode constants
```

## 3.5 Enumerations

Enumerations provide a mechanism for declaring new elemental values and types by extension. When an enumeration is declared, two semantic operations are performed on the list of value items associated with the enumeration. First, the list of values associated with the enumeration are added to `element` as elemental types if they are not already present. Second, the list of value items is assembled into a set associated with the construction. For example, the following notation:

```
enumeration(apple,orange,pear)
```

is semantically equivalent to adding the new value items `apple`, `orange`, and `pear` to `element` and assembling them into a new set. Intellectually (and semantically) the `enumeration` construct can be evaluated as:

```
enumeration(apple,orange,pear) == {apple, orange, pear}
```

The distinction between the `enumeration` former and the `set` former is that the value items comprising the enumeration need not exist before the enumeration is formed. In the example above, if `apple` is not a value item prior to generating the enumeration, then it is declared as one by the enumeration.

The `enumeration` former is only allowed in a declarative region and cannot be used to generate sets in terms.

Enumerations can be used to define new types using the canonical Rosetta notation:

```
fruit :: subtype(element) is enumeration(apple,orange,pear);
```

This declaration creates a new item whose supertype is `element` and whose value is the set containing the value items `apple`, `orange`, and `pear`. A variable defined as:

```
x :: fruit;
```

must take its value from the set `{apple,orange,pear}`.

Only elemental values may be included in enumeration declarations. Thus, the pair of declarations:

```
x :: integer;
c :: enumeration( x,1,z );
```

is illegal because `x` is already defined as an integer item, and so can't be used as a label item. Assuming that if `z` has been defined, it is defined as a value item, the declaration:

```
c :: enumeration( 1,z );
```

is semantically legal. A new value item `z` is created in the current scope and the enumeration evaluates to the set `{1,z}`. Note that having been defined as a value item by the enumeration, `z` cannot be used as anything else in the current scope. Upon leaving the enumeration's scope, `z` is again available for use as any item name.

```
%% Move the definition of label to the meta package.
```

**Summary:** The following predefined elemental types are predefined for all Rosetta specifications:

- **element** — All atomic values including **number**, **character**, and value items generated by **enumeration** formers.
- **number** — Subtype of **element** consisting of **complex** and **boolean** values.
- **complex** — Subtype of **number** and root of the numeric type tree. Formed as the the sum of any **imaginary** and any **real**. Thus,  $7.0e2 + 2.1e4*j$  is **complex**.
- **imaginary** — Subtype of **complex** where the real part is 0. Formed by any multiple of **j** and a **real**. Thus, **j** is **imaginary** as is  $5e3*j$ .
- **real** — Subtype of **complex** where the imaginary part is 0. Thus, 4,  $4.3e2$ , and  $-4.3e2$  are all **real**.
- **posreal** — Subtype of **real** where all values are greater than 0. Thus, 4 and  $4.3e2$  are both **posreal**.
- **negreal** — Subtype of **real** where all values are less than 0. Thus,  $-4$  and  $-4.3e2$  are both **negreal**.
  
- **rational** — Subtype of **real** where values are fractions of integers. Formed by dividing one integer value by another. Thus,  $5/4$  is a **rational** constant.
- **integer** — Subtype of **rational** where values are integral numeric values. Formed when no decimal point or negative exponent are included in the number definition. Thus, 1, 12,  $12e3$  and  $-12e3$  are all **integer** constants.
- **natural** — Subtype of **integer** where all values are positive or zero. Thus, 0, 5 and  $12e3$  are all **natural** constants.
- **posint** — Subtype of **natural** where all values are greater than zero. Thus, 5 and  $12e3$  are all **posint** constants.
- **negint** — Subtype of **integer** where all values are less than zero. Thus,  $-5$  and  $-12e3$  are **negint** constants.
  
- **bit** — Subtype of **natural** consisting of the values 0 and 1. Operations on **bit** elements correspond to operations on the booleans in the canonical fashion.
- **boolean** — Subtype of **number** consisting of the named values **true** and **false**. **True** is the greatest number value while **false** is the smallest. The boolean operators **and**, **or** and **not** correspond to **min**, **max** and “-” respectively.
  
- **character** — Subtype of **element** consisting of all unicode values. Formed using notation ‘x’ where x is either a printable character, or a four digit, hexadecimal value. Thus, ‘a’, ‘1’, ‘U+0024’ and ‘U+EF37’ are **character** constants.
- **ascii** — Subtype of **character** consisting of all ascii values. Formed using the same notation as **character** values. Thus, ‘a’ and ‘1’ are **ascii** values.
  
- **enumeration values** — Value items formed by use of the **enumeration** former. An **enumeration** declaration may add new value items to the **element** type. The notation **enumeration(x,y,z)** is used to define three new **element** values and assembles them into a set that can be used as a type.

## 3.6 Composite Types

Composite types make complex values by combining simpler values. There are two mechanisms for structuring: (i) containment and (ii) indexing. Containment groups items together into collections of items. Sets and sequences are both used as containers for multiple items of the same type. Sets provide a container for a specified type that is not indexed and does not contain duplicate items. Indexing establishes a function from the natural numbers (from zero to the size of the structure minus one) to the elements of the structure. Sequences effectively index sets allowing individual elements within the sequence to be accessed.

### 3.6.1 Sets

Rosetta sets are collections of items that exhibit properties traditionally defined in classical set theory. In the following, assume that  $S$  and  $T$  are sets. The first table lists functions that form sets from items or other sets:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Formation	$\{1\}, \{1,2,3\}$	<i>Forms a set from a collection of items</i>
Comprehension	$\text{sel}(x::T \mid p(x))$	$\{x \mid x \in T \text{ and } p(x)\}$
Union	$S+T$	$\{x \mid x \in T \vee x \in S\}$
Intersection	$S*T$	$\{x \mid x \in T \wedge x \in S\}$
Difference	$S-T$	$\{x \mid x \in S \wedge x \notin T\}$
Power Set	$\text{set}(T)$	$(S \text{ in } \text{set}(T)) == S=<T$
Integer Sequence	$\{i, \dots j\}$	$\text{sel}(x::\text{integer} \mid x \geq i \text{ and } x \leq j)$
Image	$\text{image}(f, S)$	<i>f applied to each element of S</i>

The basic set former takes an arbitrary collection of items and forms a set by extension. Each argument to the set former is treated and evaluated as an expression. The `sel` operation provides a set comprehension capability where one set is filtered to form another. In the table, elements of  $T$  are filtered by the boolean predicate  $p$  to form a new set. Operations for intersection, union, and difference are defined in the classical manner. The `set` function is equivalent to the set of all subsets of its argument and is typically used to define new set items. The sequence operation generates sets from sequences of integers. The notation  $\{1, \dots 4\}$  generates the set  $\{1, 2, 3, 4\}$ .

Finally, the `image` operation takes a function and applies it to all elements of a set. (`image` is synonymous with `ran` defined later.)

Classical relations between sets are defined and are listed in the following table:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Equality	$S = T$	$S =< T \text{ and } T =< S$
Inequality	$S \neq T$	$\text{not}(S =< T) \text{ or } \text{not}(T =< S)$
Subset	$S =< T, T \geq S$	$\forall x : S \cdot x \in T$
Proper Subset	$S < T, T > S$	$S =< T \text{ and } S \neq T$
Element	$a \text{ in } S$	$a \in S$
Size	$\# S$	$ S $
Empty Set	$\{\}$	$\forall x : \text{universal} \cdot \text{not}(x \in \{\})$

Equivalence is equivalence of contents. Subset and proper subset are defined in the classical manner from element. The `in` operation defines the set theoretic concept of “element of.” Size returns the cardinality of the set while `{}` names the empty set.

Defining items of a particular set type is achieved using the set type former or the power set former. The following notation defines  $x$  to be an element of the set of all possible subsets (the power set) of another set  $S$ :

```
x::set(S);
```

The declaration may intuitively be read as “ $x$  is an element of the power set  $S$ ” or alternatively as “ $x$  is a subset of  $S$ .” This is in contrast to the notation:

```
x::S;
```

that defines  $x$  to be a single element of the set  $S$ .

Like any Rosetta definition, it is possible to make a set valued item constant using an `is` clause to associate the item with a value. The following notation defines a set of integers that is equal to the set containing -1, 0 and 1:

```
trivalue::set(integer) is {-1,0,1};
```

Similarly, set comprehension can be used to define a set value:

```
natural::set(integer) is sel(x::integer | x >= 0);
```

In both cases, the type correctness restriction requires that the specified expression be an element of the type. In each of the above cases, the expressed value is a set of integers and is thus a legal value. The following expression:

```
trivalue::set(integer) is {-0.1,0.0,0.1}
```

Is not type correct because the specified set value is not a set of integers.

### 3.6.2 Sequences

Sequences are indexed collections of elements that combine the features of arrays and lists into a single, indexed container data structure. Sequences differ from sets in two important ways. First, they are indexed from 0 and allow random access of elements via their index. If  $s=[1,2,1]$  is a sequence, then  $s(0)=1$ ,  $s(1)=2$  and so forth. Second, they allow multiple instances of the same value in the container. In the example  $s=[1,2,1]$  the value 1 appears in both the first and last position. The simplest sequence is  $[],$  the empty sequence. If  $S$  and  $T$ , are sequences,  $n$  a natural number,  $e$  an element, and  $I$  a sequence of natural numbers, the following operations are defined to form sequences from items or other sequences:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Formation	$[1,2,1,4]$	<i>Forms a sequence containing 1,2,1,4 in the specified order</i>
Subscription	$S \text{ sub } I$	<i>Subsequence from S corresponding to integer sequence I</i>
Catenation	$S\&T$	<i>Concatenation</i>
Integer Sequence	$[i, \dots j]$	<i>Sequence of integers from i to j</i>
Replacement	$n \rightarrow e   S$	<i>Copy S with element n replaced by value e</i>
Empty Sequence	$[],$	<i>The empty sequence</i>
Head, tail and cons	$\text{head}(S), \text{tail}(S), \text{cons}(h, t)$	$\text{cons}(\text{head}(S), \text{tail}(S)) = S$
Mapping	$\text{map}(f, S)$	$\text{map}(f, [s_0, s_1, \dots]) = [f(s_0), f(s_1), \dots]$
Reduction	$\text{reduce}(f, S, i)$	$f(f(f(i, s(0)), s(1)), s(2)) \dots$
Filtering	$\text{filter}(p, S)$	<i>Include only elements satisfying p.</i>

The sequence former,  $[],$  forms sequences by extension with ordering of elements in the sequence the same as the lexical ordering in the former.

Subscription is an extraction mechanism where elements from a sequence are extracted to form a new sequence. Give  $S$  and an integer sequence  $I$ ,  $S \text{ sub } I$  extracts the elements from  $S$  referenced by elements of  $I$  and forms a new sequence. For example:

```
[A,B,C,D] sub [0,2,1] == [A,C,B]
```

The catenation operator, `&`, concatenates two sequences.

The notation `[i,..j]` forms an integer sequence from `i` running to `j`. As an example, the functions `head`, `tail`, and `cons` can be defined using subscription and integer sequence as follows:

```
head(S) = S(0)
tail(S) = S sub [1,..(#S-1)]
cons(x,S) = [x]&S
```

`S(0)` returns the first element in the sequence. The integer sequence former `[1,..(#S-1)]` forms the integer sequence from 1 to the length of `S` minus 1. Extracting elements of `S` associated with 1 through `#S-1` includes all elements except the first and thus defines `tail` in the canonical fashion.

The replace operation allows replacement of an element within a list. The notation `n->e | S` generates a new sequence with the element in position `n` replaced by `e`. For example:

```
2 -> 5 | [1,2,3,4,5] == [1,2,5,4,5]
```

The `map` and `filter` operators provide mechanisms for applying a function to each element of a sequence and filtering a sequence respectively. They correspond to `ran` and `sel` for functions and sets. The operation `map(f,S)` applies `f` to each element of `S` in order, generating a new sequence. Given the definition of an increment function:

```
inc(x::natural)::natural is x+1;
```

then:

```
map(inc,[0,1,2,3]) == [1,2,3,4]
```

`filter(p,S)` applies `p` to each element of `S` generating a new sequence of only those elements satisfying `p`. Given the definition of a greater than zero operation:

```
gtz(x::integer)::boolean is x>0;
```

then:

```
filter(gtz,[0,1,2,3]) == [1,2,3]
```

`reduce(f,S,i)` applies the binary function `f` recursively through the sequence `S`. The initialization value `i` is paired with `S(0)` to start the process. For example, the addition operator can be used to implement summation:

```
reduce(_+_,[1,2,3],0) == 6
```

The following operations define relationships between sequences and properties of sequences:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Equality and inequality	$S = T, S \neq T$	<i>Lexical equivalence</i>
Access	$S(n)$	<i>nth element of S from 0</i>
Ordering Relations	$S < T, S = < T, S = T, S > T$	<i>Lexicographical ordering</i>
Size	$\# S$	<i>Size</i>
Min and max	$S \max T, S \min T$	<i>Order defined on elements</i>
Contents	$\sim S$	<i>Set of elements from the sequence</i>

Equal and not equal take their canonical meanings.

The ordering operations, `min` and `max` are lexicographic ordering relations. If `cons(x,S) < cons(y,T)`, then either `x < y` or `x = y` and `S < T`. Note that for any sequence, `S != []` implies that `S > []`. `S = < T` is defined as `S < T` or `S = T`. The `S min T` and `S max T` operators return the minimum sequence of `S` and `T` and the maximum sequence respectively.

The contents of a sequence can be extracted as a set using the notation `~ A`. Duplicates are removed as well as indexing and the ordering imposed by the indexing. For example:

```
~ [2,1,1,1] == {1,2}
```

To define an item of type `sequence` containing only elements from type `B`, the following notation is used:

```
x :: sequence(B);
```

To define an item of type `sequence`, the following notation is used:

```
x :: sequence(universal);
```

where `x` is the new item and `sequence(universal)` refers to the set of all possible Rosetta sequences. Thus, `sequence(universal)` is any sequence while `sequence(B)` restricts possible sequences to the elements of `B`. Semantically, `sequence(B)` generates the set of all finite sequences created from `B` and thus the type containing all finite sequences of `B`.

## Bitvectors

A special case of a sequence is the `bitvector` type. Formally, `bitvector` is defined as:

```
bitvector :: type is sequence(bit);
```

Operations over `bit` are generalized to `bitvector`'s of the same length by performing each operation on similarly indexed bits from the two bit vectors. Assuming that `op (o)` is any `bit` operation, the `bitvector`, `C`, result of applying the operation over arbitrary `bitvector` items `A` and `B` is defined by:

```
forall(n :: {0, .. (#A-1)} | C(n) = A(n) o B(n))
```

If either `A` or `B` is longer, then the shorter `bitvector` is padded to the left with `0s`. to achieve the end result.

In addition, the following operations are defined over items of type `bitvector`: (Assume `A :: bitvector` and `n :: natural`)

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Bitwise Logic	A or B, A and B, A xor B A nand B, A nor B, not A	<i>Logical operators</i>
Conversion	bv2n(A), n2bv(n)	<i>Convert between bitvectors and naturals</i>
2's complement	twos(A)	<i>Generate two's complement</i>
Shift Operations	ashr(A), ashl(A), lshr(A), lshl(A)	<i>Logical and arithmetic shift</i>
Rotate	rotr(A), rotl(A)	<i>Rotate right and left</i>
Pad Operations	padr(A,1,n), padl(A,0,n)	<i>Pad with value to n bits.</i>

```
%% Add examples for the bitwise operations
```

```
%% Add examples for direct specification of bitvectors using strings
%% (or whatever we decide to use).
```

The operations `bv2n` and `n2bv` provide standard mechanisms for converting between binary and natural numbers. It is always true that `bv2n(n2bv(x))=x`.

The operation `twos` takes the two's complement of a binary value. The `lshr` and `lshl` operations provide logical shifts right and left while `ashr` and `ashl` provide arithmetic shifts right and left. The distinction being that logical shift operations shift in 0s while arithmetic shift operations shift in 1. The `rotr` and `rotl` operations provide rotation or circular shift. Note that none of the complement, shift, or rotate operations change the length of the bit vector.

The `padr` and `padl` operations pad or concatenate a bit vector. Each takes three arguments: (i) the bitvector being manipulated; (ii) the pad value (1 or 0); and (iii) the resulting length. If the length value is less than the length of the argument vector, `padr` removes bits to the right and `padl` removes bits to the left resulting in a vector of length `n`. In this case, the pad value is ignored.

A special subtype of `bitvector` is defined to allow definition of bitvectors with specific lengths. The `wordtype` type former takes a single natural number argument and generates the type containing bitvectors of that length:

```
wordtype(n::integer)::set(bitvector) is sel(b::bitvector | #b = n);
```

The type definitions:

```
word::subtype(bitvector) is wordtype(16);
byte::subtype(bitvector) is wordtype(8);
nybble::subtype(bitvector) is wordtype(4);
```

defines new types called `word`, `nybble` and `byte` that consist of all bitvectors of lengths 16, 4 and 8 respectively. It should be noted that `wordtype` is simply a function that returns a set of bitvectors. Usage of functions in this way is defined in a subsequent chapter.

## Strings

A special case of a sequence is the `string` type. Formally, `string` is defined as:

```
string::subtype(sequence(character)) is sequence(character);
```

A shorthand for forming strings is the classical notation embedding a sequence of characters in quotations. Specifically:

```
"ABcdEF" == ['A', 'B', 'c', 'd', 'E', 'F']
```

Functions defined over strings include those defined for general sequences. In particular, the notation `"abc"` & `"def"` is appropriate for concatenation of strings. It is important to note that the ordering operations for sequences provide lexicographical ordering for strings. No additional function definitions are required.

**Summary:** The following predefined composite types are available in a Rosetta specification:

- **set** — A set is a packaged collection of items that obeys basic principles of set theory. Sets are formed by extension using the set former or by comprehension using `sel`. The notation `set(S)` refers to the power set of `S`. `subtype` is a synonym for `set` used in declarations. The notation `type` is a synonym for `set(universal)`, the power set of all possible Rosetta items.
- **sequence** — A **sequence** is an indexed collection of items. The notation `sequence(S)` refers to any sequence of Rosetta items formed from the elements of set `S`. Sequences are formed by extension using the sequence former or by filtering, mapping or folding sequences using `reduce` and `map`. The declaration `sequence(universal)` refers to any indexed collection of Rosetta items.
- **string** — The type `string` is a special sequence defined as `string=sequence(character)`.
- **bitvector** — The type `bitvector` is a special sequence defined as `bitvector=sequence(bit)`. Operations from `bit` are defined as bitwise operations over `bitvector`.
- `wordtype(n::integer)` — The function `wordtype(n::natural)` is a special function that generates the set of bitvectors of length `n`.

## 3.7 Functions

Defining functions in Rosetta is a simple matter of defining mappings between types. Functions are extensive mappings between two different types, called the *domain* and *range* of the function. Functions are defined by defining their domain and stating an expression that transforms elements of the domain into elements of the range. This is achieved by introducing variables of the domain type whose scope is confined to the function definition and defining a result expression using that variable. The domain is given by an expression that describes the domain type. The range is given by an expression using the variable introduced by the function, called the result or result function.

### 3.7.1 Direct Definition

The direct definition mechanism for defining functions is to define the function's signature and an expression that relates domain values to a range value. Functions are typically defined by providing a signature and an optional expression. The syntax:

```
add(x,y::natural)::natural is x+y;
```

In this definition, `add` is the function name, `x,y::natural` defines the domain parameters, and `natural` is the return type. Together, these elements define the signature of `inc` as a function `add` that accepts two arguments of type `natural` and evaluates to a value of type `natural`.

Following the signature, the keyword `is` denotes the value of the return expression, in this case `x+y`. The return expression is a standard Rosetta expression defined over visible symbols whose type is the return type. Literally, what the function definition states is that anywhere in the defining scope `add(x,y)` can be replace

by  $x+y$  for any arbitrary `natural` values. Whenever a function appears in an expression in a fully instantiated form, it can be replaced by the result of substituting formal parameters by actual parameters and evaluating the resulting expression. Specifically, if the function instantiation `add(3,4)` appears in an expression, it can be replaced by the expression `3+4` and simplified to 7. Any legal expression can be encapsulated into a function in this manner.

Like any Rosetta definition, `add` is an item with an associated type and value. In this case, the type of `add` is a function type defining a mapping from two naturals into the natural. The value of `add` is known and is a function encapsulating the expression  $x+y$ . The specific value resulting from function evaluation is determined by its associated expression.

A function signature can be defined separately by specifying its arguments and return type without an expression. The notation:

```
add(x,y::natural)::natural;
```

defines the signature of a function `add` that maps pairs of `natural` into `natural`. Because this `add` definition has no associated expression, it is referred to as a function signature. Allowing the definition of a signature without an associated expression supports flexibility in the definition style. In this case the expression associated with `add` can be defined directly using equality or indirectly by defining properties. Function signatures help dramatically in reducing over-specification in definitions.

### 3.7.2 Anonymous Functions and Function Types

Anonymous functions, frequently called lambda functions, are defined by excluding the name and encapsulating the function definition in the function former `<* *>`. The definition:

```
<* (x,y::natural)::natural is x+y*>
```

defines an anonymous function identical to the function `add` above, except the anonymous function has no label. Such function definitions can be used as values and are evaluated in exactly the same manner as named functions. Specifically:

1. `<* (x,y::natural)::natural is x+y *>(1,4)`
2. `== <* ()::natural is 1+4 *> == 5`

Anonymous function signatures can also be defined in a similar manner. The definition:

```
<* (x,y::natural)::natural *>
```

is equivalent to the function signature defined above without associating the signature with a name. We call this a function type because it defines a collection of functions mapping pairs of natural numbers to natural numbers. Technically, it defines a set of functions that map two natural numbers to a third natural number. This is true because the function's expression is left unspecified. Because the definition represents a set of functions, it can be used as a type in formal definitions. The definition:

```
add::<* (x,y::natural)::natural *>
```

is semantically equivalent to the earlier `add` signature definition. The definition says that `add` is of type `<*(x,y::natural)::natural*>` or that `add` is a function that maps two natural numbers to a third natural number. The earlier signature definition is a shorthand for this notation provided to make definitions easier to read and write.

The definition:

```
add::<* (x,y::natural)::natural *> is <* (x,y::natural)::natural is x+y *>
```

is equivalent to the first definition of `add` above and semantically defines the function definition shorthand. The `add` function is defined as a variable of type `function`. The declaration asserts that the `add` function is equivalent to the anonymous function value mapping two integers to their sum. Any function label can be replaced by its value, if defined.

The notation `<* *>` is referred to as a function former because it encapsulates an expression with a collection of local symbols to form a function. The brackets form a scope for locally defined parameters. When no parameters are present, the function former brackets can be dropped as there is no need to define parameter scope.

Rosetta provides the special type `function` that contains all functions definable in a specification. Stating that `f::function` says that `f` is a function, but does not specify its `domain` or `range` values.

A function is an element of a function type if it is an element of the set of functions associated with the type. Given a function, `f`

```
f(x::T)::R is expr;
```

and a function type `F`:

```
F(x::M)::N;
```

then `f in F` if the following relationship holds:

```
f in F == T =< M and forall(x::T | f(x) in N)
```

The domain of `f` must be a subset of the domain of `F` and applying `f` to each element of its domain must result in an element of `F`'s specified domain.

For example, given the standard definition of `inc`:

```
inc(x::integer)::integer is x+1;
```

`inc` is an element of the function type:

```
<* (x::integer)::integer *>
```

because it is one specific example of a function mapping `integer` to `integer`. Specifically, the domains of the function and function type are the same and applying `inc` to any integer results in an integer.

A function type is a subtype of another function type if a subset relationship holds between them. Specifically, given two function signatures defining function types:

```
F1(x::D1)::R1;  
F2(x::D2)::R2;
```

Then `F1 =< F2` and `F1::subtype(F2)` if the following holds:

```
%% Change to make function subtyping contravariant. This follows  
%% from Hehner and is also supported by comments from Roberto.
```

```
F1 =< F2 == D2 =< D1 and R1 =< R2;
```

The distinction between subtype and type inclusion is the return type of the function type is used rather than its range. This is because the largest possible range associated with an element of a type must be the return type associated with its function type.

For example, given the function types:

```
F1 == <*(x::real)::integer*>
F2 == <*(x::integer)::integer*>
```

we know that  $F1 =< F2$  because  $integer =< real$  and  $integer =< integer$  hold for the domains and return types respectively.

Functions defined over function types and anonymous functions include: (Assume that  $f$  and  $g$  are functions and  $F$  and  $G$  are a function types)

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Type Equivalence	$F=G, F/=G$	$F \geq G$ and $G \geq F$ $-(F \geq G)$ or $-(G \geq F)$
Type Containment	$F=<G, F<G, F>G, F>=G$	Subset and proper subset relationships.
Containment	$f \text{ in } F$	Type membership

Type containment is define as above. Proper containment,  $F>G$  occurs when  $F>=G$  and  $-(F=G)$ . Type equivalence,  $F=G$  occurs when the types have the same elements. A function is a member of a type,  $f \text{ in } F$  if its domain is a subset of the type's domain and its range is a subset of the type's return type.

### 3.7.3 Function Evaluation

Evaluation of Rosetta functions follows the semantics of  $\lambda$ -calculus and allows for both currying and partial evaluation. All Rosetta functions are evaluated in a lazy fashion. Arguments can be instantiated and functions evaluated in any order. Consider the following use of `inc` and `add`:

```
inc(add(4,5))
```

Rather than evaluating in the traditional style, expand the function definitions Using the canonical definitions of `inc` and `add` results in the following anonymous function:

```
<*(z::natural)::natural is z+1 *><*(x,y::natural)::natural is x+y *>(4,5))
```

Instantiating the argument to what was the increment function results in the following definition:

```
<*()::natural is <*(x,y::natural)::natural is x+y *>(4,5) + 1 *>
```

As the resulting function has no arguments, the outer function former can be dropped resulting in the new anonymous function:

```
<*(x,y::natural)::natural is x+y *>(4,5) + 1
```

Instantiating the  $x$  parameter of the new function by replacing the formal parameter with its associated actual parameter results in:

```
<*(y::natural)::natural is 4+y *>(5) + 1
```

Finally, instantiating the `y` parameter in the same manner results in:

```
<* ()::natural 4+5 *> + 1
```

When no arguments are defined in the scope of an anonymous function, the function former can be dropped resulting in the expected result:

```
4+5+1 == 10
```

In general, the notation `<* ()::T is e *>` is equivalent to simply stating `e`.

The same result occurs regardless of the order of instantiation. The following shows a different order resulting in the same result:

1. `<* (z::integer)::integer is z+1 *>(add(4,5))`
2. `== <* ()::integer is add(4,5)+1 *>`
3. `== <* ()::integer is <*(x,y::integer)::integer is x+y *>(4,5)+1 *>`
4. `== <* ()::integer is <*(x::integer)::integer is x+5 *>(4) + 1 *>`
5. `== <* (x::integer)::integer is x+5 *>(4) + 1`
6. `== <* ()::integer is 4+5 *> + 1`
7. `== 4+5+1`
8. `== 10`

### 3.7.4 Currying, Partial Evaluation, Function Composition and Selective Union

Thus far, all Rosetta functions have been defined by using an expression in the function definition. Rosetta provides three additional mechanisms for function construction: (i) curried functions and partial evaluation; (ii) function composition; and (iii) selective union. Partial evaluation generates new functions by substituting values for some parameters and simplifying. Function composition is simply an application of function definition techniques that allow a new function to be constructed from existing functions. Selective union allows functions to be specified by extension.

#### Currying Multi-Parameter Functions

Technically, evaluation of multi-parameter function is achieved by a process based on the concept of a curried function. This process provides the basis of the evaluation process used previously. All Rosetta functions can be expressed as functions of a single argument, or curried functions. Specifically, the function:

```
<* (x::R;y::S)::T is exp *>
```

can be expressed equivalently as:

```
<*(x::R)::<*(y::S)::T*> is <*(y::S)::T exp*>*>
```

The new function is expressed is unary function over items of type `R` that returns another unary function that maps items of type `S` to type `T`. Given a function `f(x::R; y::S)::T` and `r::R` and `s::S`, the following equivalence holds:

```
f(r,s) == f(r)(s)
```

Given the definition above, this equivalence generalizes to functions of arbitrarily many variables.

Consider again the definition of `add` defined over two integer numbers:

```
add(x,y::integer)::real is x+y;
```

Using the previous notation, `add` can be expressed in a curried fashion as:

```
add(x::integer)::<*(y::integer)::integer*> is
  <*(y::integer)::integer is x+y*>;
```

The use of `x` in the expression is perfectly legal as the second function definition is done in the scope of `x`.

The `add` function is now defined over a single parameter of type `integer`. Its return type is no longer an `integer` value, but a function that maps one `integer` onto another. Using this notation, adding two values `a` and `b` is achieved using `add(a)(b)` - exactly the notation discussed previously.

Now consider application of the `add` function using the curried function approach:

```
1. add(1,2)
2. == add(1)(2)
3. == <*(x::real)::<*(y::real)::real is x+y*>>(1)(2)
4. == <*(y::real)::real is 1+y*>(2)
5. == <*(::real is 1+2*>
6. == 3
```

The function is evaluated by substituting an actual parameter for a formal parameter in first the original `add` function and then in the unary function returned by `add`, exactly as it was done in the previous section.

It is particularly interesting to note the following equivalence:

```
1. add(1)
2. == <*(x::real)::<*(y::real)::real is x+y*>>(1)
3. == <*(y::real)::real is 1+y*>
```

This process, called currying, defines the semantics of multi-parameter functions and is the basis for all function evaluation.

## Partial Evaluation

Partial evaluation is the process of taking a function and instantiating only a subset of it's parameters. The semantics of partial evaluation are defined using currying as described previously. Here, only the usage and application of partial evaluation are discussed.

Consider the following definition of `f` over real numbers:

```
f(x::real;y::real;z::real)::real is (x+y)/z;
```

Application of `f` follows the traditional rules of substituting actual parameters for formal parameters in the expression and substituting the expression for the function. Partial evaluation will perform the same function, but will not require instantiating all parameters. Consider a situation where `f` is applied knowing that in all cases the value of `z` will be fixed at 2 to perform an average. The following syntax partially evaluates `f` and assigns the resulting function to the new function name `avg`:

```
avg(x::real;y::real)::real;
```

```
avg=f(_,,2);
```

In this definition, the “\_” symbol is used as a placeholder for a parameter that will not be instantiated. To calculate the value of  $f(_,,2)$ , we simply follow the instantiate and substitute rule:

```
f(_,,2) == <(x::real;y::real)::real is (x+y)/2*>;
```

The result is a 2-ary function that returns a real value. As noted, this function calculates the average of its two arguments. An alternate, more compact notation for the definition is:

```
avg::<(x::real;y::real)::real*> is f(_,,2);
```

This general approach is applicable to functions of arbitrarily many values.

## Function Composition

Function composition is an application of function definition capabilities. Assume that two functions,  $f$  and  $g$  exist and that  $\text{ran}(g) \subseteq \text{dom}(f)$ . We can define a new function  $h$  as the composition of  $f$  and  $g$  using the following definition style:

```
h(x::R)::T is f(g(x));
```

The approach extends to other definition styles in addition to the direct definition style.

Consider the definition of `inc` and a function `sqr` defined as:

```
sqr(y::integer)::integer is y^2;
```

The definition of a function whose value is  $(x + 1)^2$  can be defined as:

```
<(z::integer)::integer is sqr(inc(z))*>
```

Expanding the definitions of `sqr` and `inc` gives the following function:

```
<(z::integer)::integer is
  <(y::integer)::integer is
    y^2*><(x::integer)::integer is x+1*>(z))*>
```

The only available simplification is to substitute  $y$ 's actual parameter in to the expression for `sqr` giving:

```
<(z::integer)::integer is < (< (x::integer)::integer is x+1*>(z))^2 *> *>
```

Continuing to substitute, replacing formal parameters with actuals and eliminating function formers when parameters are replaced gives:

```
1. <(z::integer)::integer is < (< z+1 *>)^2 *> *>
2. == <(z::integer)::integer is < (z+1)^2 *> *>
3. == <(z::integer)::integer is (z+1)^2 *>
```

The result is a new function defined over  $z$  that gives the result of composing `inc` and `sqr`.

## Selective Union

Selective union of two functions  $f$  and  $g$  is defined formally as:

```
(f|g)(x) = if x in dom(f)
           then f(x)
           else if x in dom(g)
                 then g(x)
                 end if;
           end if;
```

Using the `if` construct insures that the function associated with the first including domain will be called. In the above example, if  $\text{dom}(f)=\text{integer}$  and  $\text{dom}(g)=\text{real}$ , then an integer value will cause  $f$  to be selected while only a `real` value that is not an `integer` will cause  $g$  to be selected. If the domains are reverse, *i.e.*  $\text{dom}(f)=\text{real}$  and  $\text{dom}(g)=\text{integer}$ ,  $g$  will never be selected because any element of `integer` is also in `real`. If the type of  $x$  is in none of the domains specified, then the result of evaluating the function is undefined.

The domain and range of  $(f|g)$  is defined as:

```
dom(f|g) == dom(f)+dom(g);
ran(f|g) == ran(f)+ran(g);
```

Selective union is highly useful for implementing a form of polymorphism. An example of a function defined by selective union is the simple `non_zero` function:

```
non_zero(n::number)::boolean is
  (<*(n::0) is true*> |
   <*(n::sel(x::integer | x>0))::boolean is false*> |
   <*(n::sel(x::integer | x<0))::boolean is false*>)
```

This is a rather pedestrian use of selective union and there are better definitions of the `non_zero` function. However, it does demonstrate how domain values can be used to select from among different function definitions.

### 3.7.5 Function Extension

Selective union provides a semantics for combining different function instances to form a single function. When these instances represent functions implementing the same general operation on different data types, a primitive form of overloading and polymorphism can be implemented. A special notation is provide to define a function that extends a named function already defined. Specifically:

```
overloading-function(parameters)::return-type
  extending overloaded-function is
  expression;
```

defines a new function, *overloading-function*, by taking the selective union of *overloaded-function* and the function:

```
<*(parameters)::return-type is expression*>
```

This mechanism is handy when defining a function that extends a function defined in an included package or in the containing scope. For example:

```
package example::logic is
begin
  inc(x::real)::real is x+1.0;

  facet extension-example::logic is
    inc(x::character)::character extends example.inc is x;
  begin
    ...
  end facet extension-example;
```

In the facet `extension-example`, a new function `inc` is defined for characters that extends the increment function defined in the facet's scope. The new function, called `inc` in the facet, is defined as:

```
inc::<*(x::character+real)::character+real is
  <*(x::character)::character is x*> |
  <*(x::real)::real is x+1*>;
```

The new function overloads the existing function because it comes first in the selective union. Within the facet `extension-example`, the function `inc` can be called legally with a parameter of either type `character` or `real` with the appropriate result. Note that the extension mechanism does not add semantics, only syntax to the existing definition.

It is also possible to define a function abstractly by excluding the expression. The following notation defines a new function that extends an existing function, but does not define the specifics of that extension. Such functions are useful when defining requirements where complete information is not available.

```
overloading-function(parameters)::return-type
  extending overloaded-function;
```

### 3.7.6 The If Expression

The Rosetta `if` expression is a polymorphic function that supports choice between options. The syntax of the `if` expression is:

```
if exp1 then exp2 else exp3 end if;
```

where `exp1` must be of type `boolean` while `exp2` and `exp3` may be of arbitrary types.

The rules for evaluating an `if` expression differ from the `if` statement in an imperative language. When `exp1` is `true`, the expression evaluates to `exp2`. When `exp1` is `false`, the expression evaluates to `exp3`. Specifically:

```
if true then a else b end if == a
if false then a else b end if == b
```

The `else` clause may be omitted:

```
if exp1 then exp2 else end if;
```

In this case, the if expression evaluates to *exp2* if *exp1* is true and is undefined otherwise. Specifically:

```
if true then a end if == a
if false then a end if == undefined
```

The domain of an if expression is simply `boolean` while the ranges is the union of the types of *exp2* and *exp3*.

For convenience, an `elsif` construct is provided to nest if statements. The notation:

```
if exp1 then exp2
  elsif exp3 then exp4
  elsif exp5 then exp6
  ...
  else expn
end if
```

is semantically equivalent to:

```
if exp1 then exp2
  else if exp3 then exp4
    else if exp5 then exp6
      ...
      else expn end if
    end if
  end if
end if
```

### 3.7.7 The Case Expression

```
%% The case statement section is dependent on resolution of bugs 156
%% and 157
```

The `case` expression supports selection from multiple options in a manner similar to using the `if` construct with the `elsif` extension.

The general form of a case statement is:

```
case exp0 is
  s1 -> exp1 |
  s2 -> exp2 |
  s3 -> exp3 |
  ...
  sn -> exprn
end case;
```

where *s1*-*sn* are sets and *exp0*-*expn* are expressions. The `case` statement evaluates to *expk* when *exp0* in *sk* holds. If this relationship is satisfied by multiple sets, the `case` expression evaluates to the expression associated with the first such set. A default case can be achieved using `universal` as the set expression. The equivalence check performed in most traditional languages is performed by using singleton sets. For example:

```

case x is
  sel(x::integer | x > 0) -> false |
  {0} -> true |
  sel(x::integer | x < 0) -> false
end case;

```

implements a zero test on the `x`.

Please note that the case statement is semantically equivalent to application of a unary function defined using selective union. Specifically:

```

(<*(x::sel(i::integer | x > 0))::boolean is false *> |
 <*(x::0)::boolean is true *> |
 <*(x::sel(i::integer | x < 0))::boolean is false *> )

```

is identical to the previous `case` statement.

### 3.7.8 The Let Expression

Function application provides Rosetta with a mechanism for defining expressions over locally defined variables. An additional language construct, the `let` expression generalizes this providing a general purpose `let` construct. The Rosetta `let` is much like a Lisp `let` in that it allows definition of local variables with assigned expressions. The general form of a `let` expression is:

```

let (x::T be ex1) in ex2 end let;

```

This expression defines a local variable `x` of type `T` and associates expression `ex1` with it. The expression `ex2` is an arbitrary expression that references the variable `x`. Each reference to `x` is replaced by `ex1` in the expression when evaluated.

The `be` keyword is semantically similar to `is`. The distinction is that `is` clauses are evaluated when their associated variable declarations are evaluated. `be` clauses are evaluated when the `let` form is evaluated. Unlike traditional variable declarations, `let` parameter definitions may not omit the `be` clause. All `let` parameters must have values when the `let` form is interpreted.

The syntax of the `let` expression is defined by transforming the expression into a function application. Specifically, the semantic equivalent of the previous `let` expression is:

```

<*(x::T)::universal is ex2*>(ex1)

```

When the function is applied, all occurrences of `x` in `ex2` are replaced by `ex1`. This process is identical to the application of any arbitrary function to an expression. Assume the declaration `i::integer` and consider the following `let` expression:

```

let (x::integer be i+1) in i'=x end let;

```

The semantics of this `let` expression is:

```

<*(x::integer)::universal is i'=x*>(i+1)

```

Evaluation of the function application gives:

```
i'=i+1
```

The usefulness of `let` becomes apparent when an expression is used repeatedly in a specification. Consider a facet with many terms that reference the same expression. The `let` construct dramatically simplifies such a specification.

`Let` expressions may be nested in the traditional fashion. In the following specification, the variable `x` of type `T` has the associated expression `ex1` while `y` of type `R` has the expression `ex2`. Both may be referenced in the expression `ex3`.

```
let (x::T be ex1) in
  let (y::R be ex2) in ex3 end let
end let;
```

This expression may also be written as:

```
let (x::T be ex1, y::R be ex2) in ex3 end let;
```

The semantics of this definition are obtained by applying the previously defined semantics of `let`:

```
<*(x::T)::universal is <*(y::R)::universal is ex3 *>*>(ex1)(ex2)
```

Consider the following specification assuming that `i` is of type integer and `fnc` is a two argument function that returns an integer:

```
let (x::integer be i+1, y::integer be i+2) in i'=fnc(x,y) end let;
```

When evaluated, the following function results:

```
<*(x::integer)::universal is <*(y::integer)::universal is i'=fnc(x,y)*>*>(i+1)(i+2)
```

The result of evaluating this function is:

```
<*(x::integer)::universal is <*(y::integer is i'=fnc(x,y)*>*>(i+1)(i+2) ==
<*(y::integer)::universal is i'=fnc(i+1,y)*>(i+2) ==
i'=fnc(i+1,i+2)
```

In order for normal argument substitution to work, the expressions in the Rosetta `let` expression must not be mutually recursive. If recursion is necessary, the expressions must be represented as normal Rosetta rules or predicates.

**Summary:** A Rosetta function is defined by specifying a domain, range and an expression defining a relationship between domain and range elements. The notation:

```
f(d::domain)::range is exp;
```

Defines a function mapping `domain` to `range` where `exp` is an expression defined over domain parameters and gives a value for the associated range element. The notation:

```
f(d::domain)::range;
```

defines *f* as an element of the set of all functions relating *domain* to *range* without specifying the precise mapping function.

As an example, the increment function is defined over naturals using the notation:

```
inc(x::natural)::natural is x+1;
```

Applying a function is simply substitution of an actual parameter for a formal parameter. Evaluating `inc(2)` involves replacing *x* with 2 and applying the definition of a function. Specifically:

```
inc(2) = 2+1 =  
inc(2) = 3
```

Function types are specified as anonymous functions using the notation:

```
<(d::domain)::range*>
```

This function type specifies the set of all functions mapping *domain* to *range*. The definition:

```
f::<(d::domain)::range*>
```

says that *f* is a function that maps *domain* to *range*. The function former (`<* *>`) defines the scope of the named parameter *x* and forms a function constant in the same manner as `{}` forms a set and `[]` forms a sequence.

An anonymous function is a function having no assigned label. It is treated like a lambda function in Lisp programming languages in that it can be evaluated like any other function, but has no name by which to reference it.

```
<(d::domain)::range is exp*>
```

This anonymous function specifies the function mapping *domain* to *range* using the expression *exp*. It is semantically the same as `f(d::domain)::range`. The definition:

```
f::<(d::domain)::range is exp*>
```

says that *f* is a function that maps *domain* to *range* using the expression *exp*. It is semantically the same as `f(d::domain)::range is exp`.

The `let` expression provides a mechanism for defining local variables and assigning expressions to them. This provides shorthand notations that can dramatically simplify complex specifications by reusing specification fragments. The syntax of the general `let` expression is:

```
let (v1::T1 be e1, v2::T2 be e2, ..., vn::Tn be en) in exp end let;
```

where *v1* through *vn* define variables, *T1* through *Tn* define the types associated with each variable, and *e1* through *en* define expressions for each variable.

Evaluating the `let` expression results in the expression *exp* with each variable replaced by its associated expression. The semantics of the `let` expression are defined using function semantics. It is sufficient to realize that the `let` provides local definitions for expressions.

The `if` expression provides a simple mechanism for expressing choice. The general form:

```
if expression then cond1 else cond2 end if;
```

evaluates to *cond1* if *expression* evaluates to **true** and *cond2* if *expression* evaluates to **false**.

The **case** expression provides a general purpose selection method used to choose from more than two, potentially non-exclusive options. The form of the case statement is:

The general form of a case statement is:

```
case exp0 is
  s1 -> exp1 |
  s2 -> exp2 |
  s3 -> exp3 |
  ...
  sn -> expn |
  otherwise -> exp
end case;
```

where *exp*<sub>*k*</sub> are expressions and *s*<sub>*k*</sub> are sets or expressions that result in sets. The **case** expression evaluates to the first *exp*<sub>*k*</sub> such that *exp*<sub>0</sub> in *s*<sub>*k*</sub>. The term **otherwise** is a synonym for **universal** and provides a default case. If the match condition holds for none of the **case** terms and an **otherwise** term is not included, the **case** statement is undefined.

```
%% Stop First Evaluation Complete...
```

```
%% Start Evaluation Here...
```

### 3.8 Set Construction and Quantification

In Rosetta, all quantifiers are functions defined over other functions. A number of second order functions such as **min** and **max** are defined and will be presented here. Given that  $F(x::\text{universal})::\text{universal}$  and  $P(x::\text{universal})::\text{boolean}$ , the following quantifier and set constructor functions are defined:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Function Former	<*(rank)::return is exp *>	<i>Forms a function value or type.</i>
Domain	dom(F)	<i>Function's actual domain</i>
Return Type	ret(F)	<i>Signature's return type</i>
Range	ran(F)	<i>Function's Range or Image</i>
Maximum and Minimum	max(F),min(F)	<i>Maximum and minimum value in range</i>
Selection	sel(P)	<i>Set Comprehension</i>
Exists	exists(P)	<i>Existential quantifier</i>
Forall	forall(P)	<i>Universal quantifier</i>
Summation and Product	+B, *B	<i>Summation and Product</i>

```
%% The unary versions of + and * are being considered for removal in
%% bug 166.
```

The signature for the **min** function is:

```
min(f::<(x::universal)::universal*>)::universal
```

The `min` function accepts an arbitrary function and returns the minimum value associated with the range of the argument function. Recall that the range of the argument function is the result expression applied to each element of the domain. Consider the following function application:

```
min(<*(x::{1,2,3,4})::natural is x*2 *>)
```

The domain of the argument is the set  $\{1,2,3,4\}$ . Although it is unusual to define a set by extension in these circumstances, it is perfectly legal. The range of the argument function is the expression applied to each element of the domain. Specifically,  $\{2,4,6,8\}$ . The `min` function then returns the minimum value in  $\{2,4,6,8\}$  or 2.

If the unaltered minimum value associated with the input set is desired, the `min` function can be applied using an identity function as in:

```
min(<*(x::{1,2,3,5})::natural is x *>)
```

The `max` function is defined similarly and operates in the same manner.

### 3.8.1 Domain, Range and Return Type

The `ret` function takes a function and returns its defined return type. This is the type specified in the function definition following parameters and limits the values that can be returned by the function definition. The `ran` function is similar to a set mapping function and returns the image of a function with respect to its domain. It returns the set resulting from applying the parameter function's expression to each element of the domain. By definition, `ran(F) in ret(F)`, but it is not necessary for the range of a function to be equal to its return type. Consider the following example where `ran` is used to add one to each element of set B:

```
ran(<*(x::B)::natural is x+1*>)
```

Given that  $B=\{1,2,3\}$ , the expression above evaluates to  $\{2,3,4\}$ . This is precisely the application of `x+1` to each element of the range set. Applying `ret` to the same function would return `natural` as the return type.

The `dom` function is defined similarly to the range function but instead returns the domain associated with its function argument. For example:

```
dom(<*(x::T)::natural is x+1*>)
```

evaluates to the set T.

The domain and range functions present a greater challenge when dealing with functions of arity other than 1. The domain of a nullary function is defined as the `null` type while the range of a nullary function is the result of its evaluation:

```
dom(<*( )::natural is 3+2 *>) == null
ran(<*( )::natural is 3+2 *>) == {5}
```

Using this identity, one can define evaluation of a fully instantiated function as taking the range of that function. Specifically, if all arguments to a function are known, then the range of that instantiated function is the same as evaluating the function.

Currying is applied when looking at the domain and range of functions with arity greater than one. Recall that any n-ary function can be treated as application of a series of unary functions. Thus, the domain of functions with arity greater than one is defined as the type of the first parameter. Thus, for the `add` function defined by:

```
add(x,y::natural)::natural is x+y;
```

The value of `dom(add)` is defined as:

```
dom(add) == natural;
```

or the set of values the curried form of `add` can be applied to.

The range of such a function is the set of functions that result from currying over all possible domain values. Literally, it is the set of values, albeit function values, that result from applying the curried function form to the domain values. Thus for the `add` function:

```
ran(add) == image(add,natural);
```

the range is defined as the set of functions that result from applying `add` to every domain element. The elements of the resulting set are functions of the form:

```
f(x::natural)::natural is x+n
```

where `n` is any `natural`.

The return type is defined in the same manner as for unary functions. It is the type used to define the function signature in its declaration. Thus, for `add`:

```
ret(add) == natural
```

It is important to note that the range and return type of a function are two different concepts. `ran(f)` is a function of `f` and its application across its domain. `ret(f)` is associated with the signature of `f` and need not be equal to its domain.

### 3.8.2 Quantifiers

As previously defined, the functions `min` and `max` provide minimum and maximum functions over function ranges. Over boolean valued functions, `min` and `max` provide quantification functions `forall` and `exists`. As noted earlier, `and` and `or` correspond to the binary relations `min` and `max` respectively. As `forall` and `exists` are commonly viewed as general purpose `and` and `or` operations, `forall` and `exists` should correspond to `min` and `max`.

Consider the following application of `forall` to determine if a set, `S`, contains only integers greater than zero:

```
forall(<* (x::S)::boolean is x>0 *>)
```

Here, the domain of the argument function is the set `S` and the result expression `x>0`. To determine the range of the argument function, `x>0` is applied to each element of `S`. Assume that `S={1,2,3}`. Substituting into the above expression results in:

```
forall(<*(x::{1,2,3})::boolean is x>0 *>)
```

Applying the result expression to each element of the domain, the range of the function becomes:

```
{true,true,true} == {true}
```

As `true` is greater or equal to all boolean values, the minimum resulting value is `true` as expected. Assuming `S={-1,0,1}` demonstrates the opposite effect. Here, the range of the internal function becomes:

```
{false,false,true} == {false,true}
```

As `false` is less than `true`, the minimum resulting value is `false`. Again, this is as expected.

### 3.8.3 Selection

The function `sel` provides a comprehension operator over boolean functions. The signature for `sel` is defined as follows:

```
sel(<*(x::universal)::boolean*> is set(universal)*>)
```

Like `min` and `max`, `sel` observes the range of the input function. However, instead of returning a single value, `sel` returns a set of values from the domain that satisfy the result expression. Consider the following example where `sel` is used to filter out all elements of `S` that are not greater than 0:

```
sel(<*(x::S)::boolean is x>0*>)
```

Assuming  $S=\{1,2,3\}$ ,  $x>0$  is true for each element. Thus, the above application of comprehension returns  $\{1,2,3\}$ . If  $S=\{-1,0,1\}$  then  $x>0$  holds only for 1 and the instance of `sel` evaluates to  $\{1\}$ .

```
%% A select one operation, selone, is being discussed in bug 169
```

### 3.8.4 Shorthand Notation

A shorthand notation is provided to make specifying `forall`, `exists`, `sel`, `min` and `max` expressions simpler. Notationally, the following statement:

```
forall(x::S | x>0);
```

is equivalent to:

```
forall(<*(x::S)::boolean is x>0*>);
```

and returns `true` if every `x` selected from `S` is greater than 0. The notation allows specification of the domain on the left side of the bar and the expression on the right. The domain of the expression is assumed to be boolean for `forall`, `exists`, and `sel`. For `min` and `max`, the domain is taken from the expression. This notation is substantially clearer and easier to read than the pure functional notation. Note that the original notation is still valid for specifying quantified functions.

The notation extends to n-ary functions by allowing parameter lists to appear before the “|” to represent parameter lists. The format of these lists is identical to the format of function parameter lists. Specific examples include:

```
forall(x,y::integer | x+y>0)
exists(x,y::integer | x+y>0)
sel(x,y::integer | x+y>0)
```

It is important to remember that like `forall` and `exists`, `sel` observes the function range and selects appropriately. Interpreting the notation in the standard way results in the definitions:

```
forall(<*(x,y:integer)::boolean is x+y>0 *>)
exists(<*(x,y:integer)::boolean is x+y>0 *>)
sel(<*(x,y:integer)::boolean is x+y>0 *>)
```

```
%% Need to update summary section
```

**Summary:** Quantifier functions operate on other functions. Each generates the range of their function argument and returns a specific value associated with that range. `min` and `max` return the minimum and maximum range values respectively and are synonymous with `forall` and `exists`. `sel` and `ran` provide comprehension and image functions respectively. `sel` applies a specific boolean expression to a function's range and returns a set of domain elements satisfying the expression. `dom` returns the domain of a function defined as the application of the result expression to every domain element.

### 3.8.5 Function Containment

Function containment,  $f1 \leq f2$ , holds when a function is fully contained in another function. Assuming  $f1(x::d1)::r1$  and  $f2(x::d2)::r2$  where  $d1$ ,  $d2$ ,  $r1$  and  $r2$  are types representing domain and range respectively:

$$f1 \leq f2 == d1 \leq d2 \text{ and } \text{forall}(x::d1 \mid f1(x) = f2(x))$$

$f1$  is contained in  $f2$  if and only if the domain of  $f1$  is contained in the domain of  $f2$  and for every element of  $f1$ 's domain,  $f1(x)$  is equal to  $f2(x)$ .

Consider the case of determining if increment is contained in identity over natural numbers. In this case, the law should not hold:

1. `inc`  $\leq$  `id`
2. `==`  $\langle*(x::\text{natural})::\text{natural is } x+1*\rangle \leq \langle*(x::\text{natural})::\text{natural is } x*\rangle$
3. `==` `dom(inc)`  $\geq$  `dom(id)` and `forall(x::natural | inc(x) = id(x))`
4. `==` `natural in natural` and `forall(x::natural | x+1 = x)`
5. `==` `true` and `false`

`false` is obtained from the second expression by the counter example provided by  $x=0$  as  $0+1 \neq 0$ .

Assume that  $f(x::df)::rf$  and  $g(x::dg)::rg$  are functions. The following operations are defined over two functions:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Equivalence	$f=g, f \neq g$	$f \leq g$ and $g \leq f, \neg(f \leq g)$ or $\neg(g \leq f)$
Containment	$f \leq g, g \geq f$	$\text{dom}(f) \leq \text{dom}(g)$ and <code>forall(x::dom(f)   f(x)=g(x))</code>
Proper Containment	$f < g, g > f$	$f \geq g$ and $f \neq g$

Functional equivalence checks to determine if every application of  $f$  and  $g$  to elements from the union of their domains results in the same value. Specifically,  $f(x) = g(x)$  for every  $x$  in either domain. Function inequality is defined as the negation of function equality.

Function containment,  $f \leq g$ , occurs when  $\text{dom}(f) \leq \text{dom}(g)$  and `forall(x::dom(f) | f(x) = g(x))`. Proper containment occur when simple containment holds and the functions are not equal.

`%% Working here...`

`%% Consider moving this section. Technically, these are second order funtions, but they are specific to real and complex valued fuctions. Don't know where to move them though...`

### 3.8.6 Limits, Derivatives and Integrals

A special class of functions for defining limits, derivatives and integrals are provided for use with real valued functions. These functions exist primarily to allow specification of differential equations (both ordinary and partial) over real valued functions. Given a real valued function  $f(x::\text{real})::\text{real}$ , the following definitions are provided:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Limit	<code>lim(f,x,n)</code>	$\lim_{x \rightarrow n} f(x)$
Derivative	<code>deriv(f,x)</code>	$\frac{df}{dx}$
Indefinite Integral	<code>antideriv(f,x,c)</code>	$\int f(x)dx + c$
Definite Integral	<code>integ(f,x,u,l)</code>	$\int_l^u f(x)dx$

The derivative of a function is defined with using limit in the canonical fashion. The following axiom is defined for all real valued functions and real valued nonzero  $\delta$ :

$$\text{deriv}(f,x) = \text{lim}((f(x+\delta)-f(x))/(x+\delta)-x,\delta,0)$$

In the derivative function,  $f$  is the object function and  $x$  is the label of the parameter subject to the derivative. In the above function, the following holds:

$$\text{deriv}(f,x) = \frac{df}{dx}$$

The derivative function is generalizable to expressing partial derivatives. Assuming that  $g$  is defined over multiple parameters, such as  $g(x::\text{real};y::\text{real};z::\text{real})::\text{real}$ , then:

$$\text{deriv}(g,x) = \frac{\partial g}{\partial x}$$

Antiderivative, or indefinite integral, is the inverse of derivative. The antiderivative of  $f$  with respect to  $x$  is expressed as:

$$\text{antideriv}(f,x,c) = \int f(x)dx + c$$

$f$  being the function in question,  $x$  being the variable integrated over, and  $c$  being the constant of integration.

As antiderivative is the dual of derivative, the following axiom is defined for all real valued functions:

$$\text{antideriv}(\text{deriv}(f,x),x,0) == \text{deriv}(\text{antideriv}(f,x,0),x) == f$$

The definite integral of  $f$  with respect to  $x$  over the range  $u$  to  $l$  is expressed as:

$$\text{integ}(f,x,l,u) == \int_l^u f(x)dx$$

The definite integral is defined as the difference of the indefinite integral applied at the upper and lower bounds:

$$\text{integ}(f,x,l,u) == \text{antideriv}(f,x,0)(u) - \text{antideriv}(f,x,0)(l)$$

It is possible to express a definite integral over an infinite range using the notation:

$$\text{integ}(f,x,\text{false},\text{true}) = \int_{-\infty}^{\infty} f(x)dx$$

It should be noted that limit, derivative, antiderivative and integral functions are defined over real valued functions only. Further, the functions provide a mechanism for expressing these operations and some semantic basis for them. Solution mechanisms are not provided.

## 3.9 Universal Type

The type `universal` is now introduced as the supertype of all Rosetta types. This includes `element`, `set`, `sequence`, constructed types, `function` and facet types. Declaring:

```
x :: universal;
```

results in an item `x` that can literally contain any Rosetta value. Declaring:

```
f(x::universal)::universal;
```

results in a function `f` that can accept any Rosetta value and may result in any Rosetta value. Declaring:

```
t :: subtype(universal);
```

results in a type variable, `t`, whose value may be any set of Rosetta values.

## 3.10 User Defined Types

### 3.10.1 Sets and Types

As noted earlier, all Rosetta types are sets and any Rosetta set can be used as a type. To support clarity in specifications, several notational shorthands are provided to support defining types and subtypes. The item declaration notation:

```
i :: integer;
```

defines an item named `i` whose value is restricted to single elements from the set `integer`. When `integer` is viewed as a set, this restriction can be represented as:

```
i in integer;
```

Similarly, the notation:

```
natural :: set(integer);
```

implies that `natural` is a set of elements from `integer`. This constraint can be expressed using `in` by equating `set(integer)` power set of integers:

```
natural in set(integer);
```

the set `natural` is contained in the power set of integers and is a subset of `integer`. The value of `natural` can be restricted to a single element of the power set that appropriately defines naturals using the notation:

```
natural :: set(integer) is sel(x::integer | x >= 0);
```

Now the set `natural` is constrained to be equal to the set of elements from `integer` that are greater than or equal to 0. Because the expression defines a set value by comprehension over `integer`, we know that the expression is contained in `set(integer)`.

In Rosetta, sets are first class items and any set can be used as a type. Thus, `natural` from the previous declaration can be used as a type in subsequent declarations. Thus, the declaration:

```
n :: natural;
```

declares a new item named `n` that is an element of the set `natural` used as a type.

## Subtype and Type

In Rosetta, one type is a subtype of another if all its elements are contained in the second type. Specifically, `S` is a subtype of `T` if `S=<T` holds. When defining a new set using the notation:

```
natural::set(integer);
```

it is known that `natural` is a subset of `integer` and thus that `natural` is a subtype of `integer`. Thus, Rosetta provides an operation, `subtype` that is semantically equivalent to `set`:

```
natural::subtype(integer);
```

The `subtype` notation is equivalent to the `set` notation. Both define a new set that includes possible subsets of `integer`. The `subtype` notation is simply syntactic sugar that provides a mechanism that a set will be used as a type.

It is also possible to define a new type that is not explicitly defined as a subtype of any existing type. Using the `set` notation, such a type is defined as:

```
T::set(universal);
```

or alternatively using `subtype`:

```
T::subtype(universal);
```

Both notations define a new set, `T`, whose elements are simply Rosetta items. No other type restriction is made. Such sets are frequently used when defining abstract types whose construction is not specified or known. Thus, Rosetta provides a keyword `type` that is equivalent to the definition `subtype(universal)`. Specifically:

```
T::type;
```

is equivalent to the previous notations and defines a new type that has no subtype relationships with other types.

Both `subtype` and `type` definitions can be used to define constants in a manner identical to that for any other Rosetta item. The `is` clause is included to provide a constant value for the symbol. The type `natural` is defined using this technique:

```
natural :: subtype(integer) is sel(n::integer | n >= 0);
```

The following example defines a type that includes sets of exactly four integers:

```
set4 :: subtype(set(integer)) is sel(x::set(integer) | #x=4);
```

where `set4` is the set of subsets of `integer` that contain exactly four elements. In this case, `set4` is a set of integer sets, not simply a set of integers. Thus, the `set` operation is used to generate the power set and the new type `set4` is chosen from the power set of the power set. In other words, it is itself a set of integer sets. The `sel` operation uses the cardinality operator to choose integer subsets that contain 4 element sets only. The notation `z::set4` declares `z` to be an element of `set4`, or simply a subset of `integer` containing four elements.

The `type` notation can be used similarly to define the natural numbers:

```
natural::type is sel(x::integer | x >= 0)
```

Using this definition, `natural` is still a subset of `integer` and is thus a subtype of `integer`. However, this information must be inferred rather than directly found in the definition. The `type` declaration should be used carefully and only when defining or types not defined by filtering existing sets. If a subtype relationship exists, then it should be specified explicitly in the definition. Even if the type's value is not known, expressing a subtype relationship in the type definition aids automated analysis and readability.

In addition to constructing new types comprised of elements, the `subtype` construct can be used to define types comprised of composite values. The following definition:

```
bv::subtype(sequence(bit));
```

defines a new type named `bv` that is comprised of bitvectors. Similarly, it is possible to define types containing sets and constructed types.

### 3.10.2 Parameterized Type Formers

Any function returning a set can be used to define a Rosetta parameterized type. Consider the following function definition:

```
word(n::natural)::subtype(bitvector) is  
  sel(b::bitvector | $b = n);
```

Remembering that `subtype` is a synonym for `set`, the function signature defines a mapping from natural numbers to a set of bitvectors. That set of bitvectors is defined by the `sel` operations to be those whose lengths are equal to the parameter `n`. Thus, `word` will return the set of bitvectors of length equal to its parameter. We can now use `wordtype` as a type definition construct.

The notation:

```
reg::wordtype(8);
```

defines `reg` to be a bitvector of length 8.

The notation:

```
bv8::subtype(bitvector) is word(8);
```

defines `bv8` to be the set of all bitvectors of length 8.

```
%% Summary section needs to be updated.
```

**Summary:** User defined types are declared exactly as are other Rosetta variables and constants. While the notation `x::T` forces `x` to be a singleton element of `T`, the notation `x::subtype(T)` allows `x` to be a subset for `T`. Types can be formed from any element or composite type.

Uninterpreted types are defined as subtypes of the `universal` type.

Parameterized types are defined by using functions to return set as types.

## 3.11 Constructed Types

```
%% Working to determine if the type parameters from the function
%% should also be expressed as parameters to the data declaration.
```

### 3.11.1 Defining Constructed Types

Rosetta provides a general shorthand for defining types in a constructive fashion. Constructor, observer and recognizer functions are defined for the type and encapsulated in a single notation. These types are called *constructed types* and are created with the special `data` keyword and notation. As an example, consider a definition for a binary tree of integers:

```
Tree(a::type) :: type is data(a::type)
  null::nullp |
  node(L::Tree(a),v::a,R::Tree(a))::nodep;
```

This definition provides two constructors for `Tree`: (i) the nullary function `null`; and (ii) the ternary function `node`. The `null` function creates an empty tree while the `node` function creates a node from a value and a left and right subtree. A tree of integers can be defined as:

```
IntTree :: Tree(integer);
```

A tree with one node whose value is 0 can be generated with the following function instantiation:

```
node(null(),0,null());
```

A balanced tree with 0 as the root and 1 and 2 as the left and right nodes respectively can be generated:

```
node(node(null(),1,null()),0,node(null(),2,null()));
```

The recognizers `nullp` and `nodep` indicate the constructor used to generate a tree. Specifically, `nullp` is true if its argument is `null()` and `nodep` is true if its argument is an instantiation of the `node` function. Semantically, these functions are defined as follows:

```
nullp(x::tree(integer))::boolean is x=null();
nodep(x::tree(integer))::boolean is
  exists(lt::tree(integer), v::integer, rt::tree(integer) | node(lt,v,rt)=x);
```

Finally, parameter names are used to generate observer functions that return actual parameters from constructor functions. Specifically, the following functions are generated from the integer tree definition:

```
lt(t::sel(x::tree(integer) | nodep(x)))::tree(integer)
rt(t::sel(x::tree(integer) | nodep(x)))::tree(integer)
v(t::sel(x::tree(integer) | nodep(x)))::integer
```

These functions return the actual parameter instantiation of their associated formal parameter. For example:

```

lt(node(null(),1,node(null(),2,null()))) == null()
v(node(null(),1,node(null(),2,null()))) == 1
v(rt(node(null(),1,node(null(),2,null())))) = 2

```

The syntax for creating unparameterized constructed type definitions is:

```

T :: type is data
  f1(b11::T11, b12::T12 ... b1i::T1i)::r1 |
  f2(b21::T21, b22::T22 ... b2j::T2j)::r2 |
  ...
  fn(bn1::Tn1, bn2::Tn2 ... bnm::Tnm)::rn;

```

This data type definition defines  $n$  functions that create and recognize all elements of type T. Instantiating any of the  $f_k$  functions creates an element of type T. This set of functions are referred to as *constructors* of the type T.

Associated with each constructor,  $f_k$ , is a boolean *recognizer* function  $r_k$  that is true when its argument was created with the associated constructor function. Specifically,  $r_k$  will return true when its argument was created using  $f_k$ :

```

r(k)(t1, t2, t3, ..., ti)==true;

```

Associated with each constructor function parameter is an *observer* function of the same name that observes the parameter. Given an instantiated constructor function, the observer associated with a parameter will return the actual parameter instantiating it:

```

b(k)(t1, t2, t3, ..., ti)==tk;

```

Like any other function, constructor functions can be partially evaluated. If this is the case, then the results of applying observer functions associated with uninstantiated parameters are not defined.

The general expression above is equivalent to the following definitions and laws (where the definitions are in the definition section and the laws in the predicate section):

```

T :: type;
f1(b11::T11, b12::T12 ... b1i::T1i)::T;
f2(b21::T21, b22::T22 ... b2j::T2j)::T;
...
fn(bn1::Tn1, bn2::Tn2 ... bni::Tni)::T;

r1(t::T)::boolean is exists(b11::T11, b12::T12 ... b1i::T1i |
                             f1(b11, b12 ... b1i)=t)
r2(t::T)::boolean is exists(b21::T21, b22::T22 ... b2j::T2j |
                             f2(b21, b22 ... b2j)=t)
...
rn(t::T)::boolean is exists(bn1::Tn1, bn2::Tn2 ... bni::Tnk |
                             fn(bn1, bn2 ... bni)=t)

rn(t::T)::boolean is exists(bn1::Tn1, bn2::Tn2 ... bnm::Tnm |
                             fn(bn1, bn2 ... bnm)=t)

b11(t::f1(x,-,- ... -))::T11 is x;
b12(t::f2(-,x,- ... -))::T12 is x;

```

```

...
begin
  t1: forall(x::T | exists(x1::b11,...,xi::bi | f1(x1,x2, ... xi) = x) or
    exists(x1::b21,...,x2::b2j | f2(x1,x2, ... xi) = x) or
    ...
    exists(xn::bn1,...,xn::bnm | fn(x1,x2, ... xi) = x) or

```

The syntax for creating parameterized constructed type definitions adds a collection of type parameters to the definition:

```

F(p1::type,...,pn::type):: type is data (p1::type,...,pn::type)
  f1(b11::T11, b12::T12 ... b1i::T1i)::r1 |
  f2(b21::T21, b22::T22 ... b2j::T2j)::r2 |
  ...
  fn(bn1::Tn1, bn2::Tn2 ... bnk::Tnk)::rn ;

```

In this case, the result is a type definition function that can be used to create new subtypes of the new type `F`. Specifically, when instantiated `F(p1,...,pn)` creates a new constructed type with constructed type variables instantiated.

The tree example is one such parameterized constructed type definition. The new type, `Tree(a)`, is parameterized over a single value that is used as a type in subsequent definitions:

```

Tree(a::type) :: type is data(a::type)
  null::nullp |
  node(L::Tree(a),v::a,R::Tree(a))::nodep;

```

Thus, the definition:

```

IntTree :: type is Tree(integer);

```

This definition creates a new type called `IntTree` that is formed by instantiating the `Tree` constructed type with `integer`. Alternatively:

```

AnIntTree :: Tree(integer);

```

creates a single new integer tree named `AnIntTree` that is of the type created by the parameterized constructed type instantiation.

### 3.11.2 Records

In Rosetta, no special syntax for defining records is defined as record structures follow directly from constructed types. A record type is a constructed type with a single constructor function that associates values with parameters used as field names. A typical record type will be defined with the following constructive technique:

```

record::type is data
  recordFormer(f0::T0 | f1::T1 | ... fn::Tn)::recordp;

```

where `recordFormer` is the single constructor, `f1` through `fn` are the names of the various fields and `T1` through `Tn` are the types associated with those fields. The recognizer `recordp` is also defined, but is largely unused. To define a specific record type that represents Cartesian coordinates, the following notation is used:

```
cartesian::type is data
  cartFormer(x::real, y::real, z::real)::cartp;
```

To define an item of this type, the standard Rosetta declaration syntax is used:

```
c :: cartesian;
```

Values can be associated with record items using the canonical `is` form:

```
origin :: cartesian is cartFormer(0,0,0);
```

Accessing individual fields of the record is achieved by applying one of the observer functions associated with a field name. To access field `y` in the record `c`, the following notation is used:

```
y(c)
```

Forming a record is achieved by calling the constructor function:

```
recordFormer(v1,v2,...,vn)
```

where `v1` through `vn` name the specific values for fields `f1` through `fn`. Defining a coordinate in Cartesian space using the definition above is achieved by:

```
cartFormer(1,0,0);
```

Accessing the result is achieved using the observer functions:

```
x(cartFormer(1,0,0))==1;
y(cartFormer(1,0,0))==0;
z(cartFormer(1,0,0))==0;
```

Using the “`_`” notation, it is possible to create records whose specific field values are not known. The following creates a cartesian coordinate whose `x` and `y` values are known, but whose `z` value is not specified:

```
cartFormer(1,0,_);
```

Should the function `z` be instantiated with this record, the return value is undefined.

```
%% Working here...
```

### 3.11.3 Pattern Matching

Pattern matching in parameter lists dramatically simplifies defining observer functions over type constructors. Parameter matching takes advantage of the mechanism used to create its input parameters. Consider the integer tree definition presented above. Two constructor functions, `null` and `node` are defined to construct two different types of trees. Viewed differently, they also partition trees into the subclasses constructed by those individual functions. Specifically, the empty and nonempty trees. Viewed in this manner, it follows the the constructor functions can be used to generate types like any other types. For example:

```
nonemptyIntTree :: type is ran(node)
emptyTree :: type is ran(null)
```

Due to the nature of constructed types, the constructor for a particular instance of the type is always known. This fact can be utilized to perform pattern matching when instantiating function parameters. Consider the following definition of `is_empty` using selective union:

```
is_empty(t::tree(integer))::boolean is
  (<*(t::null)::boolean is true *> |
   <*(t::node(lt,v,rt))::boolean is false*>);
```

The first function accepts a single parameter of type `null`. This shorthand is equivalent to saying that `t` is contained in the set of all trees generated by `null`. Of course, this contains the single `null` tree. In the second definition, the type `node(lt,v,rt)` refers to all trees that can be constructed with `node`. Furthermore, `lt`, `v` and `rt` become parameters in the function that are bound to the actual parameters of any invocation of `node`. Specifically, in the following function call:

```
<*(t::node(lt,v,rt))::boolean is false*>(node(null(),5,node(null(),6,null())))
```

`lt = null()`, `v=5`, and `rt=node(null(),6,null())` within the scope of the function. These values are determined by matching the constructor function `node` with the parameter specification for `t`. The parameters are implicitly defined and their associated types determined from the constructor specification. Specifically, `lt` and `rt` are of type `tree(integer)` while `v` is of type `integer`.

A more interesting case is defining accessor functions for the left and right subtrees of a nonempty tree. This is accomplished using the following definitions:

```
lTree(t::node(lt,v,rt))::tree(integer) is lt;

rTree(t::node(lt,v,rt))::tree(integer) is rt;
```

The utility of pattern matching is more obvious here. The two functions return actual parameters associated with the constructor function `node`. Furthermore, both functions are defined only over trees constructed with `node` and are not defined over trees constructed with `null`. This is the desired result for high level specification.

In the definitions of `lTree`, `rTree` and `is_empty`, some or all of the constructor parameters are not used in the internal function. Thus, they need not be named in the definition. We use “\_” to designate such a parameter as in the following:

```
is_empty(t::tree(integer))::boolean is
  (<*(t::null)::boolean is true *> |
   <*(t::node(_,_,_))::boolean is false*>);

lTree(t::node(lt,_,_))::tree(integer) is lt;

rTree(t::node(_,_,rt))::tree(integer) is rt;
```

In both cases, parameters that are not used are not named or available in the function definition.

```
%% Must edit to reflect changes in facet operations and the root of
%% the facet type hierarchy. Defer details to the domains chapter.
```

## 3.12 Facet Items

Because Rosetta is a reflective language, specification structures such as facets are items defined in the language. Like any other item, a facet item consists of a label, type and value. A facet's type is a set of facets that define the possible values of the facet item. A facet's value is simply an element of that set. The semantics of facet operations and types are defined in Chapter 7, but are included here as they are treated in the same manner as any Rosetta item.

In Chapter 2 a format for defining facets directly is provided. Specifically, the following defines a simple facet that increments an input value and outputs it:

```
facet inc(i::in integer; o::out integer)::state_based is
begin
  l1: o'=i+1;
end inc;
```

Treated as a Rosetta item, the label of this facet is `inc`, the type `state_based`, and the value the algebra associated with the definition.

### 3.12.1 Facet Operations

The *facet algebra* defines a collection of operations over facet types. These operations allow composition of individual facets into new facets and the definition of relationships between facets.

Facet composition operators can be used to define new facets as compositions of other facets. To achieve this, a facet is declared and assigned to the composition of other facets. An example from Chapter 2 describes the composition of requirements and constraints for a sorting component. Specifically:

```
sort :: logic is sort_req + sort_const;
```

This declaration follows the definitional style used for all Rosetta declarations. The label `sort` names the facet while the built-in type `logic` defines the facet type. In this case `+`, pronounced sum, forms a new facet from `sort_req` and `sort_const`. Specifically, sum forms the co-product of `sort_req` and `sort_const`.

Like types, parameterized facets may be defined using the function notation. The `facet` type is a type like any other and can be returned by functions. Thus, the signature of a parameterized `sort` facet definition is:

```
sort(qs::boolean)::facet is
  sort_const + (if qs then quick_sort_req else sort_req);
```

In this definition, the parameter `qs` selects whether requirements for a quicksort or more general sorting requirements are included in the sum.

The following operators are defined over facets:

<i>Operation</i>	<i>Format</i>	<i>Definition</i>
Sum	$F + G$	<i>co-product of F and G</i>
Product	$F * G$	<i>product of F and G</i>
Implies	$F \Rightarrow G$	<i>homomorphism from G to F</i>
Equivalence	$F = G$	$F \Rightarrow G$ and $G \Rightarrow F$
Functor	$F(f::\text{domain})::\text{domain}$	Mapping from one facet type to another

The properties of sum and product are defined by the category theoretic notations of co-product and product. When the co-product of two items is formed, the new item must have the properties of both the original items. Specifically, the facet  $F+G$  must have all properties of both  $F$  and  $G$ . When the product of two items is formed, the new item must have the properties of one or the other of the original items. Specifically, the facet  $F*G$  must have either properties of  $F$  or  $G$ .

Facet implication is a relation between facets that occurs when a homomorphism exists between them. When  $F \Rightarrow G$ , pronounced “ $F$  implies  $G$ ” holds, all properties of  $F$  are also properties of  $G$  and a homomorphism exists from  $G$  to  $F$ . Note that the homomorphism works in the opposite direction as implication.

Facet equivalence is defined as the existence of an isomorphism between two facets. If  $F \Rightarrow G$  and  $G \Rightarrow F$ , then an isomorphism exists between  $F$  and  $G$  and  $F=G$ .

A functor is a special function that maps elements of one facet type onto elements of another facet type. Functors play an important role in Rosetta as mechanisms for moving information between domains. Application of functors and their semantics are defined fully in Chapter 6 and Chapter 7 respectively.

### 3.12.2 Facet Types and Subtypes

A facet’s type is defined by the use of a domain in its definition. For example, a facet  $f$  defined as follows:

```
facet f(x::in integer, z::out integer)::finite_state is
begin
  t1: ... ;
  t2: ... ;
end f;
```

is considered to be of type `finite_state`. Thus, the declaration:

```
f::finite_state;
```

could be used to declare the facet signature. Note that the details defined in terms and declarations from the previous facet are not included in this declaration.

The facet type defined by a facet domain is the collection of all consistent facets that are defined based on the domain. In effect, every facet that references a specific domain is an element of that domain.

Facet subtypes provide a *domain polymorphism* capability. In the same way that `integer` is a subtype of `real` because integers are defined by restricting reals, the `finite_state` domain is a sub-domain of the `state_based` domain. This is true because the `finite_state` domain is formed from the `state_based` domain by *extension*, adding new definitions to constrain the `state_based` domain. Thus, a homomorphism exists between the `state_based` and `finite_state` domains.

The signature of the `finite_state` domain is:

```
domain finite_state::state_based;
```

indicating that the domain `finite_state` is an extension of the domain `state_based`. It then follows that the facet type associated with `finite_state` is a subtype of the facet type associated with `state_based`. The semantics and uses of facet types and subtypes are defined in Chapters 6 and 7. Facet types and subtypes are among the most important language contributions of the Rosetta system.

```
%% End Evaluation Here...
```