Rosetta Semantics Strawman

Perry Alexander and *Cindy Kong*
Information & Telecommunication Technology Center
The University of Kansas
2291 Irving Hill Rd.
Lawrence, KS 66044-7541
{alex,ckong}@ittc.ukans.edu

David Barton
Averstar, Inc.
dlb@wash.inmet.com

September 1, 2000

Editor's note: All comments between versions will be formatted in *slanted* font with the initials of the author included. All mathematical definitions representing base Rosetta semantics are formatted in mathematical fonts. Mathematical definitions within Rosetta are defined using Rosetta expression syntax.

1 Introduction

2 Preliminary Definitions

Definition 1 (Rosetta Term Language) *$\mathcal{R}$ is the language consisting of all legal Rosetta strings.*

3 Items

The basic unit of Rosetta semantics is an *item* used to represent all Rosetta constructs. An item behaves as a 4-tuple consisting of a: (i) label; (ii) value; (iii) type; and (iv) string representation.

Definition 2 (Item) *An* item *is an abstract data structure defined as follows:*

- *l – is a* label *naming i and is referenced by the meta-function* `M__label(i::item)::label`
- *v – is a* value *associated with i and is referenced by the meta-function* `M__value(i::item)::universal>`
- *t – is a* bunch *representing the* type *of i and is referenced by the meta-function* `M__type(i::item)::bunch(universal)`
- *s – is a* string *representing the string representation of i and is referenced by the meta-function* `M__string(i::item)::string`

The following properties hold with respect to any item, i:

Axiom 1 (Value Consistency) `forall(i::item | M__value(i) :: M__type(i))`

Any item's value is an element of it's associate type bunch.

All Rosetta definitions are internally represented as items. The function `M__parse` is a predefined operation that takes any element of $\mathcal{R}$ and returns the item associated with it.

Definition 3 (Parsing) *M__parse(s::string)::item obeying the following axioms:*

Axiom 2 (Parse Consistency) *forall(i::item | (M__parse (M__string i))=i)*

Parsing the string representation of an item results in the original item.

Axiom 3 (String Consistency) *forall(s::string | s::$\mathcal{R}$ => (M__string (M__parse s)) = s)*

If s is a syntactically correct Rosetta language structure, then the string representation of the parsed string is the parsed string. This axiom is currently too strong as it does not consider variable renaming or whitespace issues.

Labels are used to reference items in Rosetta specifications. To aid in the referencing process, any label used in a specification for any action other than labeling new constructs refers to the value of the item associated with the label.

Axiom 4 (Referencing) *Let i be a Rosetta item with label l. In a Rosetta specification, reference to the label within an expression l refers to M__value(i).*

Several functions are defined that allow access of a labeled object in a bunch of objects. These functions are provided for shorthand purposes and do not add functionality to the definition.

Definition 4 (Dereference Function) *The deref function finds the bunch of items in a context associated with a specific label where the context is defined as a bunch of items. Specifically:*

```
M__deref(l::label, I::bunch(item))::item is sel(i::I | M__label(i)=l)
```

The `M__deref` function is typically used when performing semantic checking where item type and value are typically required to resolve type checking issues.

Definition 5 (Dereferenced Accessor Functions) *A collection of accessor functions is defined to retrieve an item's constitutent components from a bunch of items representing context. Specifically:*

- *M__deref_value(l::label, I::bunch(item))::universal is M__value(M__deref(l,I))*
- *M__deref_type(l::label, I::bunch(item))::bunch(universal) is M__type(M__deref(l,I))*
- *M__deref_string(l::label, I::bunch(item))::string is (M__string(M__deref(l,I))*

3.1 Variable and Constant Items

Variables and constant items are labels whose values are selected from a specific type bunch. The distinction is that a constant's value is fixed at definition time. Variable items are used to define logical variables, physical variables and parameters. In a sense, a variable is any Rosetta item whose label and type are known at specification time and whose value is determined by other definitions around it.

A variable definition is achieved in Rosetta by the following declaration:

```
V::T
```

where `V` is the variable label (traditionally called its name) and `T` is its associated type. By definition of the member operator (`::`) and the Referencing axiom, this definition states the following:

```
M__value(V)::M_value(T)
```

Thus, all values associated with `V` must be in the bunch associated with `M__value(T)`.

A constant definition is achieved similarly by defining a specific value for the value field:

```
C::T is v
```

where `C` is the constant label, `T` is the constant type, and `v` is the constant value. The same definition can be achieved using the two definitions:

```
C::T; C=v
```

Using the definition of variables, this definition states the following:

```
M__value(V)::M_value(T) + M_value(C)=v
```

Thus, all values associated with `C` must be in the bunch associated with the value of `T` and those values must be equal to `v`.

3.2 Literal Item

A *literal item* is a special constant item representing a specific, atomic Rosetta value. Specifically, a value item is an item whose value is constant and known. Each value item's label is the same as the string associated with its value. Thus, the label for the item associated with the value 5 is the string "5". When the label "5" appears in a Rosetta specification, it resolves to the value 5. `literal` values are necessarily of type `element`.

Example 1 (Literal Item Example) *The value 5 is represented by the item i such that:*

- *M__label(i)= ''5''*
- *M__type(i)=element*
- *M__value(i) = 5*
- *M__string(i) = ''5''*

Axiom 5 (Literal Parse and String) *forall(i::item | M__value(i)=M__label(i) => M__string(i)=M__value(i))*

If an item's label is equivalent to the string associated with it's value, then printing the item prints only the label.

Definition 6 (Literal Items) *The M__literal function is true if and only if its argument is or references a value item. In general, a literal item is an item that satisfies the Literal Parse and String axiom and M__literal can be defined as:*

```
M__literal(i::item)::boolean is
   M__value(i)=M__label(i) => M__string(i)=M__value(i);


%% I'm not sure this literal function is necessary.  Even if it is,
%% this definition is pretty lame.
```

3.3 Type Item

A *type item* is a variable or constant item representing a Rosetta type. The type item label is the name of the type. The type item value is the bunch representing the possible values associated with the type. The type item's type is the supertype of the type. In Rosetta, an unintepretted type is defined as a variable while interpretted types are typically defined as constants. It should be noted that the definition of these types parallels that of variable and constant definition. Rosetta types are simply variables and constants whose values are bunches.

3.3.1 Uninterpretted Types

An uninterpretted type definition is achieved in Rosetta by the following declaration:

```
T::type(universal);
```

where `T` is the name of the type and `type(universal)` represents an arbitrary bunch.

An uninterpretted subtype definition is achieved in Rosetta by the declaration:

```
T::type(R);
```

where `R` is a known type. In this definition, `T` is contained in `R`, but it's actual value is left unspecified. Although `T` is known to be a subtype of `R`, its actual value is not known. The distinction between the definitional styles is that when the supertype is known, some type compatibility decisions can be made. When the supertype is not known, the type is not guaranteed to be compatible with any other type. Note that when defining types, `type == bunch`. Thus `T::type(R)` is equivalent to `T::bunch(R)`.

Example 2 (Uninterpretted Type Item) *The type `R::type` is represented by an item `t` such that:*

- *`M__label(t) = ``R''`*
- *`M__type(t) = universal`*
- *`M__value(t) = undefined`*
- *`M__string(t) = ``R::type''`*

```
%% Check out the string above.  I think this is correct, but I'm not
%% certain.
```

3.3.2 Interpretted Types

An interpretted type definition is achieved in Rosetta by the following declaration:

```
T::type(R) is B;
```

where `T` is the type name, `R` is the supertype, and `B` is the bunch defining the type value. As with other constant definitions, this type definition is equivalent to:

```
T::type(R); T = B;
```

It should be noted that the bunch `B` may be any expression of type bunch such that `B::type(R)`. Specifically, types may be expressed by comprehension over the subtype or any other type so long as the result is contained in `R`.

Example 3 (Interpretted Type Item) *The type `T::type(R) is B` is represented by an item `t` such that:*

- *`M__label(t) = ''T''`*
- *`M__type(t) = R`*
- *`M__value(t) = B`*
- *`M__string(t) = ''T::type = B''`*

It is important to note that types behave as variables and constants in all ways. Keeping this in mind makes this section somewhat redundant as rules for variable and constant definition are simply repeated for types.

3.3.3 Type Compatibility

We say that items of type `T` are compabible with type `R` if any value from `T` can be used in an expression involving `R`. Formally, `T` is compatible with `R` if it can be shown that `T::R`.

Definition 7 (Type Compatibility) *Given two types `T` and `R`, `T` is* compatible *with `R` if and only if `T::R` can be proven.*

Example 4 (Type Compatibility) *Assume the following declarations:*

```
T1::type(universal);
T2::T1;
T3::T2 = sel(t::T2 | P(i))
```

The uninterpretted type `T1` is not compatible with either `T2` or `T3` because `T1::T2` and `T1::T3` cannot be proven.

The uninterpretted type `T2` is compatible with type `T1` because `T2::T1` is true by definition.

The interpretted type `T3` is compatible with type `T2` because `T3::T2` by expansion of the definition of `T3`. The inverse is not true unless it can be shown that `sel(t::T2 | P(i))=T2`. In this case, `T3=T2`.

3.4 Term Items

Terms are special Rosetta objects that represent declarations within the facet body. Specifically, any declaration using the syntax `l:t` where `l` is a label and `t` is a string is considered a term declaration. Officially, declarations of variables and types can also be viewed as terms, but typically are not.

For any term item t, the term item label names the term, providing a reference for it. If a term's label is undefined, the term cannot be referenced by name. The term item value is an expression in the term algebra defined by the facet domain. The term item type is the language of all terms defined by the facet domain. Alternatively, the term type is the set of all syntactically legal strings as defined by the including facet. A term value is simply an element of that set.

A term definition is achieved in Rosetta by the following declaration:

```
L:T
```

where `T` is the term expression and `L` is the label assigned to the term.

Example 5 (Term Item) *The term `l:F(x)=5;` defined in the logic domain is represented by an item `t` such that:*

- *M__label(i) = ``l''*
- *M__type(i) = $\mathcal{L}$*
- *M__value(i) = ``F(x)=5''*
- *M__string(i) = ``l:F(x)=5''*

where $\mathcal{L}$ is the language describing logical expressions in the domain associated with the term.

3.5 Facet Items

3.5.1 Facet Abstract Syntax

Facet types are defined as tuples of sets containing semantic elements of facets.

Definition 8 (Facet Type) *Any* facet, *F, is a value of the following form:*

F=(D,T,B,P,V,I)

where the following hold:

```
%% Need an image function for the labels operation.  Range might
%% work.  It's still rather fouled up.
```

- *D is a facet defining the domain of the facet and M__domain(F)=D*
- *T is the set of terms defined in the facet and M__terms(F)=T*
- *B is the set of types defined in the facet and M__types(F)=B*
- *P is the set of physical variables defined in the facet and M__pvars(F)=P*
- *V is the set of visible labels defined in the facet and M__visible(F)=V*
- *I is the set of parameters defined in the facet and M__params(F)=I*
- *M__items(f::facet)::bunch(item) is D++T++B++P++I> is the set of all items associated with a facet*
- *M__labels(f::facet)::bunch(label) is dom(i::M__items(f) | M__label(i)) is the set of all labels used in the facet*

A facet item, `f`, is an item whose value is of type facet. Specifically:

```
f::facet;
```

declares a variable item, `f`, of type `facet`. A facet constant is defined in the canonical Rosetta fashion:

```
f::facet is <exp>;
```

where <*exp*> is an expression of type facet, typically defined using the facet algebra.

Most facets are defined using the concrete facet syntax:

```
facet f(p1::T) is
  p2::R;
begin logic
  t1::P(p1,p2);
end f;
```

where `f` names the facet, `p1` is a facet parameter, `p2` is a facet variable, `logic` is the facet domain and `t1` labels the single facet term `P(p1,p2)`.

Example 6 (Facet Item) *Let the following be a hypothetical facet item:*

```
facet f(p1::T) is
  p2::R;
begin logic
  t1::P(p1,p2);
end f;
```

Given that `i=M_parse(f)`, the following definitions hold:

- *`M_label(i)=``f''`*
- *`M_type(i) = facet`*
- *`M_value(i) = v`*
- *`M_string(i) = ``facet f(p1::T) is ...''`*

and the following definitions hold for the value, `v`, of `i`:

- *`M_items(v)={logic,t1,T,R,p1,p2}`*
- *`M_domain(v)=logic`*
- *`M_terms(v)={t1}`*
- *`M_types(v)={T,R}`*
- *`M_pvars(v)={p2}`*
- *`M_visible(v)={p1,p2,t1}`*
- *`M_params(v)={p1}`*

Definition 9 (Visibility) *We say that elements of `M_visible(v)` are the facet's visible labels. As such, each label may be referenced. For each visible label, `l`, in facet `f` we define the nullary function `f.l` such that:*

```
f.l(f::facet,l::label)::item is M_deref(l,M_items(f))
```

Note that `f.l` is equivalent to the visible item, not its label. Thus, when `f.l` appears in a Rosetta specification, it is dereferenced exactly like a traditional label. The difference being reference to an item in another item space.

```
%% Some problems remain here.  When f.l is used in a domain, it
%% refers to the object and is not its label.  Another idea would be to
%% have f.l label the same item in the including facet.  This may
%% present dereferencing problems that we might not want to deal
%% with.
```

3.5.2 Facet Semantics

A facet's semantics is represented as a pair representing its `domain` and `terms` that extend that domain. This pair corresponds to the concepts of a formal system and theory presentation in traditional formal sytems. In traditional definitions, the presentation is defined with the formal system implicitly present. As Rosetta supports interaction between domains, the formal system must be explicitly present in the facet specification.

Definition 10 (Facet Semantics) *The semantics of a facet is defined as:*

$$F_n = (D_n, T_n)$$

where D_n is the semantic domain of F_n and T_n is the term set of F_n. D_n formally defines the formal system associated with the specification in terms of: (i) a formal langauge; (ii) an inference mechanism; and (iii) a semantic basis. The formal language, $\mathcal{L}_n$ is an extension of the basic Rosetta syntax. For most domains, $\mathcal{L}_n = \mathcal{L}$ implying that the domain syntax is the same as the base Rosetta syntax. The inference mechanism, $\mathcal{I}_n$, is a collection of inference rules and axioms that together define when an element of the term language follows from a presentation in the language. A term t follows from a term set T_n and D_n if it can be derived using rules from $\mathcal{I}_n$. This relationship is stated as:

$$T_n \vdash_{D_I} t$$

The definition of semantic correctness is simply consistency of terms with respect to the specified semantic domain. If no term or declaration introduces an inconsistency, then the facet definition is semantically correct.

Definition 11 (Semantic Correctness) *A facet is semantically correct, indicated by `M_consistent`(F_1), if its items do not introduce an inconsistency with respect to its domain. Specifically:*

$$\neg(T_1 \vdash_{D_1} false)$$

The semantic correctness of any given facet is dependent on both its term set and its domain. Thus, it is impossible to determine semantic correctness without knowing the specification domain. This is expected as in Rosetta, the domain is specified explicity with each facet.

Semantic correctness as consistency is not decidable in the general case. Thus, pragmatics of semantic checking insist on a human assisted process. Where appropriate, Rosetta will be restricted to assure automatic semantic correctness determination. Such situations necessarily include operational facets where executability needs to be insured.

The values associated with an item consist of the bunch of items the item can legally take on. This is necessarily a sub-bunch of the item's type. Using the domain and context of the actual parameter, possible values are found by comprehension over the item's assigned type. All values resulting in a consistent assignment are included in the set of legal values. This quantity is a generalized form of the function operation `ran` extended to all items. The `M_ran` function is defined as the range of all values legally taken by any item. The `ran` operation for functions is simply the `M_ran` function assuming the current facet.

Definition 12 (Meta Range) *The bunch of values legally taken by an item is the sub-bunch of the item's type defined by values that do not result in an inconsistent facet. This is refered to as the* range *of an item. Specifically:*

$$\texttt{M\_ran}(\texttt{i}, \texttt{f}) = \texttt{sel}(\texttt{v} :: \texttt{M\_type}(\texttt{i}) | \texttt{M\_consistent}((\texttt{D}_\texttt{f}, \texttt{T}_\texttt{f} + \{\texttt{i} = \texttt{v}\})))$$

A formal parameter can be replaced by an actual parmater if the actual parameter is *type compatible* with the formal parameter. A formal parameter is type compatible with an actual parameter if all legal instances of the actual parameter are type safe with respect to the formal parameter. Although substitution is a purely syntactic operation, the objects associated with labels must be referenced to determine the safeness of the substitution.

Definition 13 (Type Compatbility) *An actual parameter, `a`, is type compatible with respect to a formal parameter, `p`, if and only if:*

```
M__compatible(a,p::label) :: boolean is M__ran(a) :: M__type(p);
```

3.5.3 Parameterization and Instantation

Facet parameters as all Rosetta parameters are treated as universally quantified variables. The definition:

```
facet A(x::T,y::R) is
  ...
end A;
```

can be viewed conceptually as:

```
forall(x::T |
  forall(y::R |
    facet A is
      ...
    end A;
  )
);
```

Although not a legal Rosetta definition, the facet reflects the behavior of a parameter. Instantiating parameters is a process of applying the standard universal elimination operation in classical logic. Specifically, replacing a formal parameter with an item of compatible type and eliminating the universal quantifier associated with the variable. This process is refered to as *instantiation* of a facet.[1]

Definition 14 (Facet Instance) *A facet instance is defined as a collection of terms that are consistent with the facet definition and potentially extend the facet definition. Specifically, given a facet F_n, F_m defines an instance of F_n if and only if:*

$$\texttt{M\_\_consistent}(\texttt{F}_\texttt{m}) \wedge \forall \texttt{t} :: \texttt{M\_\_terms}(\texttt{F}) \cdot \texttt{M\_\_terms}(\texttt{G}) \vdash_{\texttt{D}_\texttt{n}} \texttt{t} \tag{1}$$

F_m is an instance of F_n if it is consistent and every term in F_n can be derived from F_m.

Instantiating a parameterized item is replacement of a formal parameter with the label of an actual parameter that is type compatible. Instantiating parameters is the only syntactic mechanism for generating facet instances.

Definition 15 (Instantiation) *Given a facet with formal parameter `i` and an actual parameter `j`, such that `M__compatible(j,i)` holds, the following defines the result of instantiating `i` with `j`:*

[1] Note that facet instantiation is defined identically to function application.

- *M__items(M__instantiate(f,i,j))=M__items(f)-{M__item(i)}+{M__itemj}*
- *M__domain(M__instantiate(f,i,j))=M__domain(f)*
- *M__terms(M__instantiate(f,i,j))=M__terms(f)[i/j]*
- *M__types(M__instantiate(f,i,j))=M__types(f)[i/j]*
- *M__pvars(M__instantiate(f,i,j))=M__pvars(f)[i/j]*
- *M__visible(M__instantiate(f,i,j))=M__visible(f)[i/j]*
- *M__params(M__instantiate(f,i,j))=M__params(f)-{i}*

4 Facet Contexts

```
%% Still needs some work to deal with the static nature of types and
%% terms over time.  Specifically, l1: P(x) defines a constant of
%% type term whose value is P(x).  It cannot change over time.  The
%% value of x may change over time.  This needs to be cleaned up and
%% and some text added.
```

To this point, facets are static items with no temporal properties. When describing systems, it is necessary to specify how sequences of input changes effect the system's state and output. The Rosetta *context* provides the ability to reference facet instantiations at various points in time. This mechanism is provided syntactically using the "`@`" operation to explicity reference a context.

Various domains use the notation `x@t` to reference the value of item `x` in some context `t`. This usually refers to a state or time value. Within domain specifications, the behavior of context objects is described without reference to their structure or interface behavior. Here we provide mechanisms that allow specifying the semantics of expressions as well as referencing items in various contexts.

`x@t` refers to the value of `x` in context `t`. Using the concept of context introduced in Section **??**, the semantics of `x@t` is defined as `M__deref_value(x,t)` or the value of `x` in context `t`.

Definition 16 (x@s) *The semantics of `x@t` is defined simply as `x` in context `t`:*

$$x@t == M\_deref\_value(x,t)$$

Thus, when the notation `x@t` appears in an expression, it is in interpretted as the value of `x` at `t`. When a variable appears without explicit reference to context, the default context is defined as the current context. Thus, `x` refers to the value of `x` at the current time or in the current state.

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

5 Composition Operations

5.1 Label Distribution

Label distribution states and labeling operations distrubute over declarations. Thus, `L:T1 ∘ L:T2 = L:T1 ∘ T2` for any Rosetta logical operator.

Definition 17 (Label Distribution) *For any logical operator ∘, label `L` and item definitions `T1` and `T2`, the following distrubution law holds:*

$$\mathtt{L:T1} \circ \mathtt{L:T2} = \mathtt{L:(T1} \circ \mathtt{T2)}$$

Example 7 (Label Distrubution Over Terms) *Assume the following term definitions in a facet:*

L:P(x); L:Q(x);

By label distribution, this is equivalent to:

L:P(x) and Q(x)

5.2 Type Composition

A direct application of label distribution is type composition. In Rosetta, an item is frequently viewed from multiple, interacting specifications. One such example occurs when facets are composed and parameter lists are unioned. In such cases, a label refering to an item may be interpretted differently in the domains of both facets. In such situations, the type of the parameter in the newly formed facet is the conjunction of the original two types. This follows directly from the definition of type composition:

Definition 18 (Type Composition) *Assume the following two variable definitions:*

v::T1; v::T2;

By label distribution, this is equivalent to:

v::(T1 and T2)

Note that in this context `and` is not a logical connective, but a composition operator. `v::(T1 and T2)` means that the item `v` is both of type `T1` and `T2` simultaneously. This does not imply that the resulting type is the intersection of `T1` and `T2`, but is similar to a set of ordered pairs of elements from `T1` and `T2`. Elements of the composed type can be viewed as either type. The semantics of this will be better understood when considering facet composition operators later in this section.

5.3 Facet Composition

```
%% White paper composition definitions here...
```

5.4 Domain Interaction

```
%% White paper interaction semantics here...
```

6 Types and Values

7 Open Issues

- Assuming that `A::B` is true when `A` is an atomic element and in `B`. Specifically, that `A` is not a bunch. If this axiom is removed, then bunch values for atomic items becomes possible. For example, if `i::integer` is defined, then `i=1,2` is fine if `1,2::integer` is true.
- String functions are defined over items, not just over values. The string/parse thing needs some thought. It's not clear to me that it's as simple as we're assuming it to be at this point.