Integration of Griffin and Rapide

John Mitchell Dinesh Katiyar
Rapide Design Group, Stanford University

May 1, 1993

Abstract

This short document sketches a method for integrating Griffin into the Rapide language framework. This is done by expressing Griffin types as Rapide types. We give an overall view of the key concepts of the two languages, and a mapping of the concepts of Griffin into those of Rapide. We work through the translations of a few examples.

1 Premise

An important aspect of Rapide is the separation of the type and module language from the executable, or implementation, language. Designed with interoperability and MIF considerations in mind, this language separation allows modules in several different implementation languages to be described uniformly in a single interface language.

There is a general method for using an existing implementation language in conjunction with the Rapide types language. The first requirement is to define each "predefined" type of the implementation language in the Rapide type language. We then use this definition, called the "type preamble" of the implementation language, as a preamble to any module or set of modules written in the implementation language. We also require a method for translating user-defined types of the implementation language into Rapide interfaces and modules. By supplying type preambles and translations for several implementation languages, we may map all of the types occurring in any implementation module into the standard Rapide types language. This simplifies the problems of type compatibility and data conversion that arise in any multi-language context. Ideally, programs in each implementation language should only use the properties of a type that are guaranteed by its Rapide definition. However, this may be unrealistic in certain cases, and strategies to cope with deviation from this ideal are under consideration.

2 Proposal

Our proposal for integrating Griffin and Rapide is to treat Griffin as one possible implementation language to be used in conjunction with the Rapide type and module system. Although this approach may seem to take Rapide as the primary language, it is simply intended to establish a precise correspondence between the two languages.

The rest of this short document summarizes the main concepts in each type system and outlines a correspondence that may be used to write Griffin types in the Rapide type language. This correspondence is illustrated by translating some examples from the Griffin Language manual into Rapide types.

3 Main concepts in the Griffin type system

1. ADT Signature (a description of the set of operations that a type might have)
2. ADT Implementation (definitions for the operations)
3. Package Spec (like Ada)
4. Package Body (like Ada)
5. Class (collection of types sharing a common signature)
6. Class Membership (two forms of membership of a type in a class - explicit or implicit)
7. Polymorphism (type parameters, possibly constrained to be members of some class)
8. Union types
9. Subtyping (for Union types)
10. Inheritance

4 Main concepts in the Rapide type system

1. Interface
2. Module
3. Polymorphism
4. Subtyping
5. Inheritance

5 Outline of the translation from Griffin to Rapide

We list here a very top-level view of a scheme for expressing Griffin types as Rapide types.

Griffin		Rapide
ADT signature, package spec	$\longrightarrow$	interface
ADT implementation, package body	$\longrightarrow$	module
class, union	$\longrightarrow$	parameterized interface
class membership, subtyping on union types	$\longrightarrow$	subtyping
polymorphism	$\longrightarrow$	polymorphism
inheritance	$\longrightarrow$	inheritance

One reason for proposing a translation of Griffin into Rapide, rather than vice versa, is that this mapping is many-to-one. For instance, Griffin signatures and package specs are both special cases of Rapide interfaces. To give the converse map, we would perhaps need to separate Rapide interfaces into two cases - interfaces with just one type would map to ADT signatures while those with more than one type would have to be translated to package specs. A problem that arises is that although Rapide subtyping, for example, could be accounted for using Griffin union types, this only applies to Rapide interfaces that map to Griffin ADT's. For interfaces that map to Griffin package specifications, there does not appear to be any subtyping mechanism since Griffin package specifications are not considered types. (This is based on our best understanding of the current Griffin documentation.)

While the approximate mapping above suggests that the Rapide constructs are more general, this could be considered misleading. There are several distinctions in Griffin that are absent from Rapide, and thus may not be fully captured by Rapide types. The Rapide design has leaned towards a conceptually simpler high-level view, at a potential cost of some implementation brevity and efficiency.

6 Illustrative translations

In this section, we pick up a few examples from the Griffin LRM and demonstrate how one would go about expressing the same definitions in Rapide.

6.1 Example: Simple sets of integers

Let us start by defining and implementing sets of integers - something one might expect in any standard predefined types library. In an attempt to start with simple ideas, we will consider sets of integers only. We shall look at polymorphic examples later.

Griffin

The class for integer sets in Griffin would contain all the operations that one might want to perform.

```
%% Griffin code
class Int_Set = {
    fun create : () → mytype;
    fun insert : (integer; mytype) → mytype;
    fun member : (integer; mytype) → bool;
    fun delete : (integer; mytype) → mytype;
}
```

Each type in this class would have its own internal representation of sets on which it chooses to implement the operations on the set. For example, one that uses linked lists would look as follows:

```
%% Griffin code
type myset::Int_Set = {
    var set_impl : integer_list;
    fun create() ⇒ nil;
    fun insert(n:integer, s:Int_Set) ⇒ cons(n, s.set_impl);
    ...
}
```

Rapide (ADT style)

In Rapide, we would use an interface with a type constituent to define the class of integer sets. This would be done as follows:

```
%% Rapide code
type Int_Set is interface
    type t;
    create : function() return t;
    insert : function(n:integer, s:t) return t;
    member : function(n:integer, s:t) return bool;
    delete : function(n:integer, s:t) return t;
end interface;
```

A module that implements this interface would instantiate the type in the interface to the representation type it chooses to use for implementing the sets. For instance, one that uses linked list would look like this:

```
%% Rapide code
module myset:Int_Set is
    begin
        ilp : integer_list_package;
        type t = ilp.t;
        create = function() return(ilp.nil);
        insert = function(n:integer, s:t) return(ilp.cons(s,n));
        ...
    end
```

Rapide (object-oriented style)

Rapide supports both the ADT style used in Griffin, and an object-oriented style in which the operations are viewed as components of an object. (We do not intend to start a religious war on the subject of "object-oriented." In this document, we will use "ADT style" and "object-oriented style" as convenient phrases for distinguishing two possible Rapide styles. The reader may substitute "Style A" and "Style B" for these phrases if desired.)

In the object-oriented style, each object can be viewed as being capable of understanding a set of messages, and the interface lists these.

```
%% Rapide code
type Int_Set is interface
    insert : function(n:integer);
    member : function(n:integer) return bool;
    delete : function(n:integer);
end interface;
```

Note that each operation has one fewer parameter than in the ADT-style interface. This is because each set is an "implicit argument" to each of its operations. In addition, the type constituent has been removed since modules of this interface will be sets, as opposed to implementations of a set ADT in the ADT style.

Instances have a local updateable state, along with means to initialize this state by default. The operations on a set are implemented in terms of operations on its state.

```
%% Rapide code
module myset:Int_Set is
    begin
        s : integer_list;
        insert = function(n:integer) s.cons(n);
        ...
        initial s = nil;
    end
```

6.2 Example: Types with ordering relations

We now look at the definition of a simple type class taken from the Griffin library of predefined types. This is the class of types that have the usual ordering relations on their elements.

Griffin

The class definition lists all the operations that all type members of this class should have in their interface.

```
%% Griffin code
class Ordered = {
    fun < : (mytype; mytype) → bool;
    fun <= : (mytype; mytype) → bool;
    fun > : (mytype; mytype) → bool;
    fun <= : (mytype; mytype) → bool;
};
```

The special symbol **mytype** is used to refer to the type being defined.

A particular type that belongs to this class would have the form:

```
%% Griffin code
type ord::Ordered = {
    fun <(x:ord; y:ord) ⇒ ⟨ implementation ⟩
    ...
};
```

A function that accepts two arguments of any type belonging to this class and returns the minimum of the two, may be defined as follows:

```
%% Griffin code
fun min(O::Ordered; x,y:O) ⇒ if x<y then x else y;
```

Rapide (ADT style)

In Rapide, we would start by writing an interface type with an abstract type in its interface. The interface corresponding to the `Ordered` class is:

```
%% Rapide code
type Ordered is interface
    type t;
    < : function(x,y:t) return bool;
    <= : function(x,y:t) return bool;
    > : function(x,y:t) return bool;
    >= : function(x,y:t) return bool;
end interface;
```

A particular implementation of this (as a module) would be written as follows:

```
%% Rapide code
module ord:Ordered is
    begin
        type t = ⟨ some specific type ⟩;
        < = function(x,y:t) ⟨ implementation ⟩;
        ...
    end
```

The `min` function is written in the form:

```
%% Rapide code
min = function(O:Ordered; x,y:O.t)
        begin
            if O.<(x,y) then x else y;
        end
```

Rapide (object-oriented style)

For the sake of completeness, we show how we might express the same notions in Rapide object-oriented style. In this case, Griffin type classes map to Rapide parameterized interfaces, with the type parameter substituting for the Griffin special symbol **mytype**. Griffin class membership corresponds to the subtype relationship on the resulting Rapide interfaces.

```
%% Rapide code
type Ordered_Gen(type t) is interface
    < : function(Elem:t) return bool;
    <= : function(Elem:t) return bool;
    > : function(Elem:t) return bool;
    >= : function(Elem:t) return bool;
end interface;
```

Rapide achieves the effect of Griffin's **mytype** with parameterization in the type declarations (type constructors in Rapide parlance). The main purpose for defined `Ordered_Gen` is to write polymorphic functions (such as `min` defined below) that are applicable to any ordered type.

A specific ordered type may be defined by type constructor application as follows:

```
%% Rapide code
type Ordered is Ordered_Gen(Ordered);
```

A typical module would be:

```
%% Rapide code
module ord:Ordered is
    begin
        < = function(Elem:ord) ⟨ implementation ⟩;
        ...
    end
```

However, since `Ordered` has only the four comparison functions listed and nothing else, it is not a very useful type. (Note that the contravariant occurrence of `Ordered` in the functions of its interface precludes it from having any useful subtypes either). But the type constructor can be used to define more useful examples such as the following:

```
%% Rapide code
type ord_int is interface
    include Ordered_Gen(ord_int);
    ...
end interface;
```

Finally, to write the `min` function, we impose a subtype constraint on the type argument. This is where we need the constructor form of the above interface definitions.

```
%% Rapide code
min = function(O <: Ordered_Gen(O); x,y:O)
          begin
              if x.<(y) then x else y;
          end
```

Note that Griffin declarations of the form *type :: class* corresponds directly to Rapide declarations of the form *type* ¡: *constructor(type)*, where *constructor* is defined from *class* by making the special symbol **mytype** a formal type parameter.

6.3 Example: Types with comparable elements

We now look at an extension of the above example. We would like to define another class of types that have the equality operations defined on their elements. Then we would like to combine the definitions for classes of ordered types and classes of types with equality to define classes of types whose elements are comparable.

Griffin

We first define a class that contains the equality operations.

```
%% Griffin code
class Eq = {
    fun = : (mytype; mytype) → bool;
    fun /= : (mytype; mytype) → bool;
};
```

We can now define the set of types that have both the ordering relations as well as the equality operations as follows:

```
%% Griffin code
class Comparable = Eq + Ordered
```

Rapide (ADT style)

The first interface is similar to the one defined in the last example.

```
%% Rapide code
type Eq is interface
    type t;
    = : function(x,y:t) return bool;
    /= : function(x,y:t) return bool;
end interface;
```

The combination of the two interfaces is a bit more subtle. Since they each contain one abstract type, we need to make sure we don't get both.

```
%% Rapide code
type Comparable is interface
    include Ordered;
    include Eq except t;
end interface;
```

Rapide (object-oriented style)

We define another constructor for the equality operations.

```
%% Rapide code
type Eq_Gen(type t) is interface
    = : function(Elem:t) return bool;
    /= : function(Elem:t) return bool;
end interface;
```

Inheritance is achieved by combining these constructor forms as follows:

```
%% Rapide code
type Comparable_Gen(type t) is interface
    include Eq_Gen(t);
    include Ordered_Gen(t);
end interface;
```

We get the final type definition in the manner described earlier.

```
%% Rapide code
type Comparable is Comparable_Gen(Comparable);
```

6.4 Example: Polymorphic trees

Both Griffin and Rapide allow type parameterization. Hence we can define "generic" interfaces for trees which can be then instantiated for trees with elements of a particular type.

Griffin

```
%% Griffin code
class Tree[t::Any] = {
    fun insert(mytype; t) : mytype;
    fun member(mytype; t) : bool;
};
```

Rapide (ADT style)

```
%% Rapide code
type Tree_ADT(type t) is interface
    type tree;
    insert : function(x:tree, e:t) return tree;
    member : function(x:tree, e:t) return bool;
end interface;
```

Rapide (object-oriented style)

```
%% Rapide code
type Tree_Gen(type MT, type t) is interface
    insert : function(x:t) return MT;
    member : function(x:t) return bool;
end interface;

type Tree(type t) is Tree_Gen(Tree, t)
```

6.5 Example: Sets of elements with equality

As seen earlier, Griffin uses type class membership to impose constraints on type arguments. This is achieved in Rapide with a subtyping constraint on the type argument (F-bounded polymorphism).

Griffin

```
%% Griffin code
class BasedSet[elem::Eq] = {
    fun in(elem; mytype) : bool;
    fun with(mytype; elem) : mytype;
    fun less(mytype; elem) : mytype;
    fun setunion(mytype; mytype) : mytype;
    ...
};
```

Rapide (ADT style)

```
%% Rapide code
type BasedSet_ADT(elem:Eq) is interface
    type BasedSet;
    in : function(x:elem.t, s:BasedSet) return bool;
    with : function(s:BasedSet, x:elem.t) return BasedSet;
    less : function(s:BasedSet, x:elem.t) return BasedSet;
    setunion : function(s1,s2:BasedSet) return BasedSet;
    ...
end interface
```

Rapide (object-oriented style)

```
%% Rapide code
type BasedSet_Gen(type MT, type elem <: Eq_Gen(elem)) is interface
    in : function(x:elem) return bool;
    with : function(x:elem) return MT;
    less : function(x:elem) return MT;
    setunion : function(s:MT) return MT;
    ...
end interface

type BasedSet(type elem) is BasedSet_Gen(BasedSet, elem);
```

A Salient features of the Rapide type system

In this section, we briefly explain some of the design decisions of the Rapide types language along with the rationale that motivated each.

A.1 A unified view of types

explanation:

At the core of the Rapide type system is a single essential type concept - the interface. Interfaces along with functions and means of parameterization constitute the type system.

motivation:

- This leads to a conceptually simple type system.
- All standard types can be viewed as interfaces.
- The user doesn't necessarily have to remember all the idiosyncratic syntax for predefined types.

A.2 Types are separate from implementations

explanation:

Type definitions (interfaces) are independent and separate from their implementations (modules).

motivation:

- This allows the types and implementation languages to be viewed as separate entities in the same framework.
- The same interface can have several implementations.
- The same implementation can implement more than one interface.

- Derivation (inheritance) mechanisms can be used for the interfaces and modules independently. Code reuse in one does not force it in the other, allowing for more flexible coding.
- Types separate from their implementations gives a cleaner way of supporting abstract classes. One does not need to resort to notions of "virtual methods" or "incomplete types" or something similar.

A.3 Inheritance is unrelated to subtyping

explanation:

The subtyping relationship is determined independently of the derivation mechanisms.

motivation:

- By trying to make all forms of derived interfaces subtypes of the original interface, one is forced to either drastically restrict the derivation mechanisms or make the substitutivity property that one expects from subtyping untrue. This approach allows one to have very flexible derivation mechanisms.
- Given a set of types, there is ideally a unique subtyping hierarchy that can be imposed on them. However, the inheritance hierarchy can be one of many, depending on factors such as implementation order, efficiency, etc. This is similar to the notion expressed in William Cook's paper (OOPSLA 92) where he looks at the Smalltalk class library and demonstrates that the client's view of the library (which corresponds to the interface hierarchy) is different from the implementer's view (which corresponds to the inheritance hierarchy).

A.4 Implicit structural subtyping

explanation:

Whether one type is a subtype of another is automatically determined by the type system, and this is done based on the structure of the two types.

motivation:

- It is more cumbersome to define new types in an explicitly subtyped system, since one has to place the new type appropriately in the hierarchy. Similarly, it is harder to delete old types while maintaining the rest of the hierarchy.
- It is easier to combine independent subprograms; all the subtyping relationships are determined by the compiler.
- Accidental subtyping (determining two completely unrelated types to be compatible just because their structures happen to be so) is not a very frequent phenomena.

A.5 Emphasis on reuse

explanation:

The language provides very flexible derivation and parameterization mechanisms.

motivation:

- The language framework is aimed at rapid prototyping, and hence tries to provide as many techniques for the reuse of code as possible.
- Flexible inheritance is possible because of its decoupling from subtyping.
- Implementations can be reused independent of how the corresponding interfaces were defined, and vice versa. This is because of the separation of interfaces and implementations.
- It is useful to be able to define uniform behaviour over all types or over a set of types. (The latter is done by imposing a supertype constraint on the type parameter. The supertype is allowed to refer to the type parameter itself - ie, the language allow F-bounded polymorphism).

A.6 No need for "self"

explanation:

Other type systems require the capability to have some way of referring to a type in its definition (eg Self, Current, MyType). This language does not.

motivation:

- The parameterization techniques of the language allow an easy way to achieve the same effect. One simply defines a type constructor that corresponds to having the special symbol as a formal type parameter. A specific type is defined by applying the type constructor to the type being defined.

  So, instead of

  > %% *Code in some language with "Self"*
  > **type** T **is** **interface** ...**self**...;

  one would write

  > %% *Rapide code*
  > **type** TGen(**type** t) **is** **interface** ...t...;
  > **type** T **is** TGen(T);

  Derivations are then done using the constructors.
- The use of special symbols often forces a resolution rule in the semantics of the language if it allows nested type definitions. This is because the use of the special symbol in a nested definition is potentially ambiguous. In this framework, one can explicitly make unambiguous references.

A.7 No need for renaming of interfaces

explanation:

The derivation mechanisms of the language do not provide a way of renaming the name of the interface that is being inherited, even though this is often useful for derivations from recursive interface definitions.

motivation:

Once again, the parameterization techniques of the language allow an easy way to achieve the same effect.

For instance, to achieve the following definitions,

```
type S is interface
            f1:function(a:A) return S;
        end interface;
type T is interface
            f1:function(a:A) return T;
            f2:B;
        end interface;
```

one might wish to proceed as follows:

```
type S is interface
            f1:function(a:A) return S;
        end interface;
type T is interface
            include S rename S to T;
            f2:B;
        end interface;
```

However, the rename operation that is used in the above code is not permitted. But instead, one can write:

```
type S_Gen(type t) is interface
                        f1:function(a:A) return t;
                    end interface;
type S is S_Gen(S);
type T is interface
            include SGen(T);
            f2:B;
        end interface;
```

A.8 No need for separate mix-ins

explanation:

Some other languages provide special means of being able to define behaviour that doesn't need the knowledge of all of the rest of the program component. eg. Virtuals in C++ provide means of doing simple mixins. This framework allows for the possibility of even more powerful mixins without the need for a separate concept.

motivation:

Since one can define and inherit parameterized interfaces, one can program mixins without needing any extra tool. For instance, to express something similar to:

```
%% Code in some language with mix-ins
mixin ordered is interface
                        less_than: function(x,y:self) return bool;
                    end interface;
type ordered_set is interface
                        include set;
                        include ordered;
                        ...
                    end interface;
```

one would write the following:

```
%% Rapide code
type ordered(type t) is interface
                        less_than: function(x,y:t) return bool;
                    end interface;
type ordered_set_Gen(type t) is interface
                        include set;
                        include ordered(t)
                        ...
                    end interface;
type ordered_set is ordered_set_Gen(ordered_set)
```