Subtyping, assignment and cloning in a concurrent object-oriented language

John C. Mitchell *†‡ Dinesh Katiyar† David C. Luckham‡ Neel Madhav‡
Sigurd Meldal‡ Sriram Sankar‡

Abstract

This paper describes subtyping, assignment, equality and cloning in a concurrent object-oriented language called RAPIDE. Based on experiments and language design considerations, we separate the subtype relation from the derivation (or inheritance) hierarchy. This avoids various anomalies and has the added benefit of reducing the syntactic and design complexity of programs that use intricate subtype relationships. A relatively sophisticated algorithm to decide subtyping is also needed for implementation. Objects may contain multiple threads, and different objects with the same type may have different implementations. This makes it difficult to provide a meaningful, predefined form of "copy" assignment or predefined equality tests. Therefore, the primitive form of assignment in RAPIDE is a form of pointer assignment. Special provision is made for user-defined "cloning," so that an object may produce a meaningful copy of itself. Equality may be user-defined for any type, in some cases overriding an "inherited" predefined equality.

1 Introduction

RAPIDE is a language framework for rapid construction of prototypes of large concurrent/distributed realtime systems. The RAPIDE framework has three parts:

- A *type/module language* for designing components of a software system and their interfaces. A component is characterized by an interface outlining the way it interacts with other components and a module providing an executable prototype of the component.
- A *specification language* that allows an abstract specification of behavior to be included in an interface.
- An *executable language* for giving executable behavior to modules and architectures (interconnections between components).

Our preliminary implementation of RAPIDE includes syntactic analysis for the type/module and executable languages and code generation and analysis tools for a subset of the executable language. A second generation implementation of the type/module and executable languages in currently underway. The primary basis for the specification language is Anna, for which a tool suit already exists [LvHKBO87, LHM+86].

Prototyping is the incremental development and analysis of systems. Our primary goals in the development of RAPIDE are to explore prototyping processes and to produce a language and support tools for prototyping. In our view, prototyping is an experimental activity in which the information gained by executing one prototype is critical in the design of the next prototype. We believe that recent progress in type systems, object-oriented program development and formal methods can be used to support powerful prototyping methods. RAPIDE is intended to support reuse and rapid evolution of components, automated

*****Address:** Department of Computer Science, Stanford University, Stanford, CA 94305. mitchell@cs.stanford.edu (415) 723-8634

†Supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation; NSF grant CCR-8814921 and the Wallace F. and Lucille M. Davis Faculty Scholarship.

‡Supported by DARPA under ONR N00014-90-J-1232.

analysis of behaviors of complex concurrent prototypes, and standardizing specifications of components in modular systems.

The reason for designing a *language framework,* rather than a fixed language, is to allow for evolutionary changes based on future experience in the area of prototyping distributed, time sensitive systems. We are also interested in prototypes written in multiple languages and issues involved in migrating a prototype from one language to another. This paper describes the relationship between subtyping and inheritance in RAPIDE and aspects of the type/module language that effect the interaction with the other sublanguages.

An outline of an early version of RAPIDE (then called *Reality*) is given in [BL90] and a preliminary report on the type and module system appears in [MMM91]. After eighteen months of design work and programming experiments, we believe that the types/module language will remain relatively stable, providing standard structuring primitives. Within this framework, the specification and executable sublanguages may change freely, as long as certain general criteria for compatibility between the three languages are maintained. These involve visibility, scoping and naming rules and adherence to a basic storage and assignment model. If additional types and modules are incorporated, these will be considered part of the standard preamble or a library, not the basic language. Although we expect the concurrent computation model to remain stable, the type/module language is largely independent of this model.

The main language features discussed in this paper are:

- The decision to separate subtyping from interface derivation (inheritance) and its pragmatic consequences.
- The adoption of a storage and assignment model based on an abstraction of pointers.
- The treatment of equality based on our "object-oriented" framework and reference equality.
- A provision for user-defined copying (cloning) of executable modules.

Due to space restrictions, we are limited in our ability to provide comparisons with related programming languages. We would expect to provide more comparative information in a conference version of the paper. Our main influences are Smalltalk [GR83], C^{++} [Str86], Eiffel [Mey92], Self [US87] and Standard ML [Mil85, Mac85, HMT87].

For the reader familiar with [MMM91], we briefly summarize changes in the language since Fall, 1990. The main change is in the definition of subtyping, which is now determined in a way that is unrelated to the inheritance mechanism. This allows us to replace three interface inheritance primitives, extend, restrict and copy, by a single primitive we now call include. A terminological change is that we have substituted the word "interface" for what was formerly called a "specification." We now use "specification" to refer to axioms included in an interface. We have found little use for expressions that define module values directly. Instead, we use the keyword module for a function that returns a module, also called a "module generator." In the terminology of Standard ML, our module declarations define "functors" instead of "structures." We have also changed the way that module derivations result in dynamic binding of constituent names to executable bodies. In [MMM91], we explained a mechanism based on textual copy of module constituents. We have replaced this with the more common and predictable method of identifying dynamically resolved calls with the special symbol self.

Some developments that are not covered in this paper are the treatment of internal state in specifications, the relation between the reference model and efficient implementation of polymorphism (this is not much different from Standard ML), and issues of separate compilation. We also do not have space to discuss "mixins" (modules whose constituents are not executable as they stand, but are useful when inherited) and the degree to which all predefined types may be reduced to interface declarations.

2 Overview of type and module language

2.1 General features of RAPIDE

RAPIDE is a statically typed language, with parametric polymorphism and subtyping. The predefined types include Integer, Boolean, String and other familiar types. The predefined type constructors include Record, Union and Function types with the usual operations. The main form for user-defined type is the Interface, discussed in the next subsection. The primary relation between predefined and/or user-defined types is subtyping, rather than type equality.

Modules are the basic program-structuring facility of the language. A module may contain a number of related parts, called its *constituents.* A module may be as large as an independent subsystem of a large prototype, or as small as a single data object. We have gone to some length to give large and small modules the same status in the language. Two reasons are language uniformity and simplicity. A more specific objective is to support prototype development that may involve expanding small modules into large, hierarchically-structured subsystems. If the interface to a module is fixed, we do not want internal changes in the implementation to effect the set of legal operations or interactions with the module.

Each module has a type, called its *interface.* An interface describes the visible constituents of a module and the way that the module interacts with its environment. Since a module may contain multiple threads, a module may interact both through calls to accessible subprograms (functions and procedures) and execution of concurrent communication events called actions. In keeping with the way that RAPIDE is factored into separate sublanguages, very little is assumed about the nature of subprograms or actions in the type/module language. For example, actions could be realized as broadcast communication, point-to-point communication over channels, or assignment to shared variables without effecting their treatment in the type/module language. The interface of a module includes a list of accessible subprograms (functions and procedures) and their types, names of types declared in the module, and the names of in and out actions and the types of associated events. It is important to note that there may be many modules with the same interface, just as there may be many records (or other kind of data) with the same type; unlike classes in other object-oriented languages, we do not link the type to the internal structure of a module (our form of object).

2.2 Interfaces

An interface has two parts: an *external interface* and an *internal interface*. The external interface specifies the interaction between a module and its environment. The internal interface describes module constituents that are accessible only to other modules of the same type. The primary motivation for the internal interface is to solve a problem associated with binary operations in typed object-oriented languages. This is discussed in some detail in [MMM91].

Both external and internal interfaces have a set of *constituent* declarations, *derivation declarations* and *specifications.* A constituent may be type, a parameterized type (type constructor), or a constant of any type (including function or procedure). A derivation declaration is used to derive parts of an interface from previously declared interfaces; this mechanism provides "interface inheritance." Specifications, in the current language, are algebraic or temporal axioms that specify the behavior of module constituents.

The syntactic form of an interface declaration is

```
type <name> is
interface
    include <name> ( except | only ) <name> rename (<name> to <name>, ... );
    type <name> [ <parameters> ]; -- (Parameterized) type name.
    <name> : <type-exp>;   -- Constant of indicated type.
    action <name> (<parms>); -- Action declaration.
constraint
    <specifications>
internal
    <interface_body>
end interface;
```

where the interface body after the keyword **internal** has the same form as the external interface body sketched here. We have shown one declaration of each kind, but an interface may contain any number of derivation, type and constant declarations.

Interfaces may be *parameterized* by types and values.

A trivial example, used here because of its familiarity, is an interface for points with integer components.

```
type Point is
interface
      function X_Val() return Integer;
      function Y_Val() return Integer;
      function Distance(P : Point) return Integer;
end interface;
```

The elements of the type Point defined by this interface are modules with at least three constituents: X_Val, Y_Val and Distance. Each module of this type must have a subprogram body of the correct type for each of these constituents.

2.3 Modules

A module encapsulates a set of declarations and processes, generally related in some way. Each module has a specified type (an interface) and belongs to every supertype of this type. A module must declare the types, parameterized types, and constants given in its interface and may include other "private" declarations. Each process in a module has a separate thread of control.

Although there is an in-line form (omitted from this paper), modules are typically created by *module generators.* A module generator is a function with zero or more type and value parameters that returns a module. (A module generator is analogous to a Standard ML "functor.") The general syntactic form of a module generator follows:

```
module <name> (<parms>) return <type-exp> is
      include <derivation-decls>
      <type-decls>
      <value-decls>;
      initial(<parms>) <code>;
      processes <process-list>;
      final <code>;
      handle <exception-handler-list>;
end module;
```

When called, storage is allocated for the declared constituents (or pointers to appropriate code). The initialization code is then executed and the process in the process list enabled. Control is then returned to the calling program. The finalization code is executed when the module is destroyed. The given handlers are invoked in response to exceptions raised in the dynamic call chains of the given processes and subprograms. The purpose of the optional parameter list following the keyword **initial** is described in Section 6.

Module generators may have type and value parameters. A simple example follows:

```
module Create_Point(X_Init, Y_Init: Integer) return Point is
      X : ref(Integer) := refto X_init;
      Y : ref(Integer) := refto Y_init;
      function X_Val() return Integer is begin return ! X end;
      function Y_Val() return Integer is begin return ! Y end;
      function Distance(P : Point) return Real is
            begin return Sqrt(Sqr(P.X_Val - ! X) + Sqr(P.Y_Val- ! Y)); end;
end module;
```

The keywords **ref()** and **refto** and the dereference operator ! are explained in Section 4. The constituents of a module may be *selected* using the "dot" notation.

Module derivation declarations allow a module generator to *inherit* and *rename* constituents defined by previously declared module generators. This is largely parallel to the interface derivation mechanism, except for the issue of dynamic binding. The preliminary description in [MMM91] described derivation via syntactic copying and substitution, but the language has since been revised in favor of explicit use of the keyword **self** to indicate dynamic interpretation of references to module constituents.

The decision to separate interfaces from modules is discussed at some length in [MMM91], and also in papers of other authors, such as [Sny86]. Our experience since [MMM91], and communication with our colleagues, continues to support this decision. A particular situation illustrating the advantage of this separation, as opposed to the more common approach of Smalltalk, C^{++} and Eiffel, involves library routines and modules. In writing components for a library, it is important to specify the interfaces of arguments to library functions. However, it is inconvenient and cumbersome to have to give generic implementations for these interfaces, especially if (as in C^{++}) giving an arbitrary implementation will restrict the way that all other modules with related (subtype) interfaces will be implemented.

As mentioned earlier, different modules of the same type (interface) may be implemented in different ways. This may appear to cause various difficulties. However, due to subtyping, the same issues would arise as if we had required a one-to-one correspondence between interfaces and modules (as in C^{++}, for example). More specifically, suppose we declare type A and give a single implementation, then declare a subtype B with a different implementation. Since every value of type B also belongs to type A, the implementation of subtype B gives us additional implementation for values of type A. Although certain languages impose restrictions on the difference between the implementations of A and B, no reasonable restrictions are sufficient to statically determine the storage size or exact format of values.

3 Subtyping

3.1 Motivation for subtyping

The fundamental principle associated with subtyping is substitutivity: if A is a subtype of B, then any expression of type B may be used without type error in any context that requires an expression of type A. This is an ideal, and most languages deviate from it in some way. The main reason is that complete substitutivity may be at odds with a clean or understandable definition of the subtype relation. An extreme example is Eiffel [Mey92], where contravariance of function arguments is overlooked in some cases (see [Coo89]). In RAPIDE, we have made substitutivity a guiding principle, with our separation between subtyping and derivation allowing us to avoid the Eiffel problem. There are, however, some subtleties surrounding modules with type constituents.

The primary motivation for subtyping in a prototyping language is to allow functionality to be added to a prototype with minimal modification to the system. This is widely appreciated in the object-oriented programming community. In short, if objects of some type A have some behavior which crudely approximates part of the system being prototyped, then we may want to replace objects of type A with objects of another type B that have more realistic or accurate behavior. In many cases, the type B will be a subtype of A. By designing the language so that substitutivity is allowed, we may add functionality in this way without any modification to the remainder of the program. Another advantage of subtyping is uniform operation over various types of data, regardless of whether these types were added incrementally or anticipated from the earliest design stage.

3.2 Explicit versus implicit subtype hierarchy

In many object-oriented languages, the subtype relation is derived from the inheritance, or derivation, relation. For example, in C^{++}, class A is a subclass of B only if A is derived from B (and B is a public base class). We will call this the *explicit hierarchy* approach, since the subtype hierarchy is given explicitly in the program. An alternative is to examine types separately to see if it makes sense to consider one a subtype of the other. This requires inspection of the types actually defined, and does not depend on whether derivation mechanisms are used to define a type. We will call this the *implicit hierarchy* approach, since the subtype hierarchy is inferred from the meaning of type declarations in some way.

In RAPIDE, we follow the implicit hierarchy approach. Some advantages of this over the explicit hierarchy approach are:

- *supports both top-down and bottom-up paradigms:* In program development, we want to add both subtypes (to specialize previously existing types) and supertypes (to organize existing components by behavior, and to generalize certain subprograms). In the explicit approach, it is very cumbersome to fit a new type into an already existing hierarchy. The new type has to be carefully defined by derivations from a set of already existing types in such a way that the desired subtype relations hold. As a consequence, the explicit approach seems practical only when new types are always subtypes of existing types.

- *permits easy reorganization of program code:* In program development, and especially so in prototyping, we frequently reorganize programs by moving code from one place to another, and by adding and deleting "dead code" and "dead types". This process destroys the existing derivation hierarchy. Subtype relationships based on an explicit approach may therefore change in unexpected ways unless types are carefully rewritten.

- *reduces syntactic complexity of programs:* An implicit hierarchy is more permissive and reduces the syntactic complexity of programs. In particular, when a program is composed from independently constructed subprograms, it is not necessary to introduce relationships by inserting derivation declarations. An algorithm will do this automatically instead. Hence, some complexity is transferred from the program to the compiler or other language tool.

- *allows more flexible derivation mechanisms:* There are useful means of deriving new types from old ones that do not necessarily result in subtypes. With an implicit approach, these may be used freely, with subtyping inferred when appropriate.

3.3 Determining subtyping

Subtype relationships between RAPIDE types are determined from their *canonical forms* [1]. The most interesting case is an interface, with derivation and renaming. The canonical form of an interface is obtained by expanding out the derivation constructs and accounting for type recursion and associativity and commutativity in some canonical way. After this transformation, each interface contains in its external and internal parts a set of types, type constructors, and constants. Each element of these sets is identified by a unique name, possibly different from the literal name in the program if derivation involves renaming. To give a simple example, the canonical form of the interface Point in Section 2 contains three constants X_Val, Y_Val, and Distance in its external part and nothing in its internal part. If we derive colored points from points and rename Distance to Location, the canonical form of colored points will include Distance, not Location, so that we may correctly find that colored points are a subtype of points. (Except for explicit renaming, names must match exactly.)

Given two interface types T_1 and T_2, the general guideline used to determine whether T_1 is a subtype of T_2 is that T_1 contains all the constituents of T_2, and possibly more. For example, creating a new type by adding more constituents to the Point interface results in a subtype of Point.

There are a few more details:

- The external part of T_1 must contain all the constituents of the external part of T_2. Also, the set of all constituents of T_1 must contain all of the constituents of T_2. This rule implies that moving constituents from the internal part of a type to its external part results in a subtype of the original type.

- The corresponding constant constituents of T_1 and T_2 need not be identical. Although the canonical names of the constituents must be the same, the type of the constituent in T_1 may be a subtype of the type of the constituent in T_2.

[1] See Appendix A for a more detailed description of the canonical forms of various RAPIDE types.

In the formal notation explained in Appendix B, the subtype rule for interfaces may be written as follows:

$$\frac{\Gamma \vdash A_i <: A'_i \qquad \Gamma \vdash B_j <: B'_j \qquad \Gamma \vdash C_k <: C'_k}{\Gamma \vdash (\,\langle a_i\colon A_i, b_j\colon B_j, e_n\colon E_n\rangle, \langle c_k\colon C_k, d_m\colon D_m\rangle\,) <: (\,\langle a_i\colon A'_i\rangle, \langle b_j\colon B'_j, c_k\colon C'_k\rangle\,)}$$

In this rule, an interface is written as a pair of tuples $(\langle\ldots\rangle, \langle\ldots\rangle)$, the first tuple corresponding to the external part, and the second corresponding to the internal part. This rule does not consider type and parameterized type constituents for brevity.

3.4 Issues in subtype checking

The subtyping algorithm is similar to those in [CM91, Ghe90] with the following additional complications:

recursive types: Interfaces may be defined recursively. For example, the Point interface contains a reference to itself. Subtype checking becomes a bit more involved in such situations, since the recursion may have to be "unfolded" a few times before the check can be performed. Amadio and Cardelli [AC91] present an algorithm for subtyping recursive function types, based on a coding of recursive types as infinite trees. We extend the algorithm to handle recursion in interfaces. The algorithm essentially compares types by finger-pointing through the graph representations of the types. Some book-keeping has to be done to keep track of the recursive bindings, and to avoid comparing the same nodes again. The algorithm is a direct implementation of the rules listed in Appendix B.

bounded types: RAPIDE allows polymorphic types which are generic functions that accept type arguments. Each type argument may be restricted to be a subtype of some type expression, which we refer to as the bound. Bounds may also be specified for type constituents of interfaces. However, we disallow the bounding expression to involve any types with bounds. This restriction is placed because the subtyping problem for the general case is undecidable. The undecidability arises in a rather subtle fashion, as is evident from the plausible algorithmic claims of [Ghe90], later shown incorrect by the undecidability result of [Pie92].

While specifying a bound for a type, it is often useful to be able to mention the type itself. For instance, while writing a generic *sort* routine, one would like to specify that the interface for the type argument has a function which can compare it with another element of the same type. We have formulated a subtyping rule for this case that handles the most important cases of such *recursive* bounds in practice and has a decidable subtyping algorithm. We are currently investigating the possibility of more flexible rules.

parameterized types: RAPIDE allows user-defined type constructors, which are parameterized type forms. All occurrences of type constructors, except those occurring as type declarations in interface types, can be instantiated to actual types before we check for subtyping. The remaining case is treated by a method similar to function types. Specifically, one type constructor is a "subtype" of another if its parameter bounds are supertypes of the bounds on corresponding parameters in the other. Recursion in type constructors is similar to recursive type bounds, mentioned above.

4 Storage model and assignment primitives

4.1 Three assignment alternatives: alias, copy or clobber

Suppose X and Y are identifiers bound to some kind of structured objects. There are three ways we could try to interpret an assignment X := Y. The first is that after assignment, the identifier X now refers to the same object as Y. We refer to this as *aliasing assignment.* The second alternative would be to let X refer to a " copy " of the object referred to by Y. We call this *copy assignment.* The third option is to modify the object referred to by X so that it thereafter behaves as if it were a copy of Y. Since this overwrites the internal state of X, we call this *clobber assignment.* Clobber assignment is similar in effect to copy assignment, except that there is no need to allocate space. This gives clobber assignment an efficiency advantage. In addition, depending on how we interpret aliasing assignment, these two may have different effects on an identifier Z that initially refers to the same object as X. However, if X and Y do not have the same internal representation, then clobber assignment may not be feasible.

In a language with subtyping, clobber assignment is a problematic language primitive. The reason is that if the type of Y is a subtype of the type of X, the values referred to by these identifiers may have different sizes or storage formats. Since it is not possible to determine statically which subtype Y belongs to, there is likely to be some loss of information (or loss of encapsulation) in clobber assignment. This is the case for structure or object assignment in C^{++}.

The basic form of assignment provided in RAPIDE is an aliasing assignment. The reasons for this are that aliasing assignment has an efficient implementation independent of the size or type of objects involved and, in combination with other language features, is sufficient to define copy assignment. Meaningful cases of a clobber assignment X := Y may be achieved without any special language provision. All that is required is an X procedure constituent (method) which, given Y as a parameter, updates the internal state of X appropriately.

Copy assignment is expressible using aliasing assignment by the form X := Y.Copy, where Copy is a function constituent (method) that returns a copy of Y. Some examples that illustrate the benefits of this approach to copy assignment are:

- Suppose X is a sparse matrix and Y is a dense matrix. In RAPIDE, these could have the same type, but different internal representations. It does not seem desirable to interpret X := Y as clobber assignment, since this would involve stuffing a dense matrix into a data structure intended for a sparse matrix. However, both aliasing assignment and copy assignment would make sense. In the case of copy assignment, we want X to refer to a copy of Y with the same internal representation as Y. This is achieved through by X := Y.Copy when := is aliasing assignment.

- Suppose X and Y are both points, but Y is a "directed point," meaning that Y is oriented in some direction and this orientation determines the result of Y.Move, where Move is a component of all subtypes of points. In a clobber assignment, there would be no provision in the internal structure of X to accommodate the orientation of Y. Therefore, clobber assignment would result in a loss of functionality. (This happens in C^{++}, for example.) As for matrices, both aliasing and copy copy assignment are meaningful and have potential uses.

We describe RAPIDE assignment in more detail in the rest of this section. A language mechanisms for copying is described in Section 6.

4.2 Reference types

In RAPIDE, as in Standard ML [Mil85], assignment is an operation associated with reference types. A consequence is that a variable of type A has a different type from a constant of type A.

There is a predefined type constructor called *reference,* giving us a type **ref**(A) of references to values of type A, for each type A. A subtle but important difference between a reference and a pointer is that a reference refers to a value, while a pointer points to a location containing a value. There are four basic operations on references:

creation: if E is an expression of type A, then ref_to(E) is a new reference, initially referring to the value of E.

dereference: if E is an expression of type ref(A), then !E is an expression for the value referred to by E.

assignment: if E is an expression of type ref(A) and E' an expression of type A, then E := E' is a statement that changes the reference given by E to refer to the value of E'.

equality: if E and E' are expressions of type ref(A), then E = E' is a boolean expression that is true exactly if E and E' have the same reference value. In other words, E = E' tests whether E and E' are aliases. (Nb. This is NOT the same as testing whether E and E' point to the same location; there is no location pointed to, unless A itself is a reference type.)

Ordinary pointers may be represented as references to references.

4.3 Advantages of references over pointers

In manipulating C^{++} objects, it is common to use pointers and pointer assignment, since assignment with ordinary variables is interpreted as clobber assignment, with associated pathologies. Therefore, we initially considered pointer assignment as the natural basic form of assignment for RAPIDE. However, there is a disadvantage to the form of pointer assignment found in C^{++}, Pascal, and elsewhere when we try to interpret *all* assignment as pointer assignment. This eventually led us to reject pointers in favor of references. The main design criteria in our choice of assignment primitive were that (i) assignment should be efficient unless the user writes something inefficient, and (ii) assignment should behave uniformly across all types of data, from integers to large modules with many concurrent threads. The first rules out copy assignment as the default primitive, and the second tips the balance away from standard pointers.

The problem with pointers is that a pointer is a reference to a *location* that contains a value. It is the location containing the value that causes problems. If we wish to view integer assignment as pointer assignment, we need to simulate (or perhaps actually build) a location for each integer value. This is cumbersome and inefficient both because of the space allocation involved and the otherwise unnecessary indirection required for each integer access. Moreover, this gives us two forms of equality on integers: whether they occupy the same location and whether they are the same integer. This is awkward and redundant. Therefore, we adopt a variant of pointers in which the locations pointed to are not visible. This allows the implementation to either place values in locations and use ordinary pointer assignment, or in the case of immutable values such as integers, dispense with the locations pointed to and implement assignment in the ordinary way. A benefit of adopting ML reference types is that it gives us a simple solution to the related problem of designing efficient and uniform tests for equality across all types of data.

5 Equality

It is difficult to provide a meaningful predefined equality on all types for many of the reasons mentioned with regard to copy assignment. Therefore, our approach to equality has been to make this a predefined operation on some types only. These are the predefined types and the predefined type constructors (assuming equality on the argument types), except for function types. In addition, equality is predefined for reference types. This is superficially similar to the treatment of equality in Standard ML [GGM91], except for two important features of RAPIDE. The first is that due to subtyping, we do not have to maintain a special category of "equality types." Instead, types with equality are identified as subtypes of the type with equality operation (see, e.g., the example treatment of ordered types in [MMM91] or the examples in [CCH+89]). Second, since operations are constituents of values, there is a natural place for "ad hoc" operations such as equality that have different semantics for different types. Thus basic features from object-oriented programming, together with bounded polymorphism (see [CW85, CCH+89, MMM91]) allow us to treat an otherwise problematic area relatively simply.

6 User-defined cloning

6.1 Motivation

A certain kind of copying seems essential in practice. Given a module with mutable state, we need to be able to create (at run-time) another module in a behaviorally equivalent state, but occupying distinct memory locations so that subsequent updates to one module do not effect the other. As in Eiffel [Mey92] and Self [US87], we call this form of copying *cloning.* To give an example where cloning is useful, consider a module generator returning a new module. When useful data is passed as a parameter to the module generator, it is common to initialize the state variables to copies of this data. While copying is achieved in other languages by storage allocation and assignment, RAPIDE reference operations alone do not accomplish this.

In Eiffel and Self, there is a built-in language primitive for cloning. The predefined cloning method for any class may be redefined as needed. However, it is not feasible to provide a predefined cloning operation for all RAPIDE modules. The main reason is that with concurrent threads, the language processor does not have sufficient information to determine the amount of locking required to do a copy, nor is it necessarily clear how to accomplish this in the presence of various concurrency primitives. Instead, we have taken the position that cloning should be under user control.

Even where a default, predefined cloning operation could be given, there are cases where a user will wish to override this. A common case is a linked data structure, such as a list. If we clone the module that is the first element in the list, a compiler cannot determine whether it is appropriate to recursively clone all reachable modules.

6.2 Problems in user-defined copying

Without special provision, it is difficult to implement cloning without compromising data abstraction. We will illustrate this using the following interface and module, which show the simplest pattern where problems seem to arise. The module contains a function whose behavior depends on the hidden internal state of the module (in this case, a single integer variable), and there is no procedure for setting the internal state directly. A concrete example might be a random number generator where the internal state is the last number generated or an encryption device where the internal state contains a private key. In general, it is not desirable to add a procedure that resets the state to the public interface of this module, but we do wish to add a public Clone function.

```
type Coder is interface
    function Encode(...) return ... is ... ;
end interface;

module Mk_Enigma() return Coder is
    Key : Integer := 0;
    function Encode(...) return ... is
        ... Key ...
    end Encode;
end module;
```

The basic problem in adding a Clone function to Mk_Enigma is that a module value produced by Mk_Enigma has hidden internal state, and there is no way to set this state using the given module constituents.

6.3 Attempt to use the internal interface

An approach that is unsatisfactory in general is to add a procedure that sets the internal state of the module. Since this procedure should not be completely visible, we put in the best available place, the internal interface. Using the new procedure, we can implement Clone by calling the module generator Mk_Enigma to get a new module value, and then calling the new procedure to set the internal state of the new module appropriately.

– Example code omitted –

This has the correct functionality, but requires the extra procedure to set the internal state of the module. The main disadvantage is that the type of the parameter to Set_State is given in the interface to all modules of this type. However, not all modules of a type will have the same internal structure. Therefore, it is a significant limitation to fix the parameter type for all modules. Also, it would not be desirable to make public the encryption key of a module to all modules of type Coder.

6.4 Solution: internal calling sequence

When we implement cloning using the internal interface approach above, we create a module and then set its internal state by calling a procedure designed for this purpose. A more direct approach is to modify the call to the module generator so that we may directly create a new object with correct initial state. In order to preserve the signature of the module generator, we only allow the modified calling sequence within the module body itself. This gives the module implementor the ability to establish and maintain invariants in a simple way.

In this approach, we return to the original interface given in Section 6.2.

Recall that the general form of module generator includes an initialization sequence beginning with the keyword **initial**. As mentioned briefly in Section 2.3, the keyword **initial** may be followed by an optional parameter list. When the module generator is called from an arbitrary point in its scope, the parameter list from the top of the declaration is used. For a call to the generator from within the generator body, both parameter lists may be supplied. This allows the body of a clone function to call the generator with enough additional parameters so that the initial state of the new module can be set appropriately. However, since the **initial** parameter list is not visible outside the module body, this feature does not substantially complicate formal or informal proofs of module invariants.

To show how this works, we continue the running example.

```
module Mk_Enigma() return Coder is
      Key : Integer;
      function Encode(...) return ... is begin
            ... Key ...
      end Encode;
      function Clone() return Coder is begin
            return Mk_Enigma()(Key); end Clone;
initial (K : Integer);
      if Passed?(K) then Key := K else Key := 0;
end module;
```

This mechanism gives us the ability to define efficient cloning functions without compromising data abstraction.

References

[AC91] R.M Amadio and L. Cardelli. Subtyping recursive types. In *Proc. ACM Symp. Principles of Programming Languages*, pages 104–118, 1991.

[BL90] F. Belz and D.C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proc. ACM Tri-Ada'90 Conference*, December 1990. To appear.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991. Summary in *Math. Foundations of Prog. Lang. Semantics,* Springer LNCS 442, 1990, pp 22–52.

[Coo89] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[GGM91] C.A. Gunter, E. Gunter, and D.B. MacQueen. An abstract interpretation of ML equality kinds. In *Int'l Conf. Theor. Aspects of Computer Software*, page ??, 1991.

[Ghe90] Giorgio Ghelli. *Proof Theoretic Studies about a minimal type system integrating inclusion and parametric polymorphism.* PhD thesis, University of Pisa, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation.* Addison Wesley, 1983.

[HMT87] R. Harper, R. Milner, and M. Tofte. The semantics of Standard ML. Technical Report ECS–LFCS–87–36, Lab. for Foundations of Computer Science, University of Edinburgh, August 1987.

[LHM+86] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler. Task sequencing language for specifying distributed ada systems. In *Lecture Notes in Computer Science, Number 275*, pages 249–305. Springer-Verlag, Berlin, May 1986.

[LvHKBO87] David Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. Anna - A Language for Annotating Ada Programs. In *Lecture Notes in Computer Science, Number 260.* Springer-Verlag, Berlin, July 1987.

[Mac85] D.B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), 1985. 35 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mil85] R. Milner. The Standard ML core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[MMM91] J.C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 270–278, January 1991.

[Pie92] B. Pierce. Bounded quantification is undecidable. In *Proc. 19th ACM Symp. Principles of Programming Languages*, page ??, 1992. To appear. Available as Carnegie Mellon Technical Report CMU-CS-91-161.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Str86] B. Stroustrop. *The C^{++} Programming Language.* Addison-Wesley, 1986.

[US87] D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 227–241, 1987.

A Canonical forms

Each type in RAPIDE has a *canonical form* representation. This representation is determined from the original description of the type in RAPIDE by expanding the type names used in its definition, and in-lining the parts of the type defined by reuse (*i.e.* expanding all the *includes*). Below, we informally explain how to determine subtyping relationships between canonical forms (a more detailed and rigorous description in the form of inference rules is given in Appendix B). This essentially defines the subtype relationship for RAPIDE types.

There are four forms of canonical representations, which correspond to function types, interface types, union types and polymorphic types.

$$Ct ::= Cf \mid Ci \mid Cu \mid Cp$$

- **Canonical Function Types**

  A canonical function type has an argument type and a result type. We use the following concise notation -

  $$Cf ::= Ct_a \rightarrow Ct_r$$

  where Ct_a is the type of the argument, and Ct_r is the return type of the function.

  Multi-argument function types in RAPIDE can be "curried" into a canonical representation using only single-argument function types.

The subtyping rule for these canonical function types is straightforward. The canonical function type F_1 is a subtype of F_2 if the argument type of F_1 is a supertype of the argument type of F_2, and the result type of F_1 is a subtype of that of F_2.

- **Canonical Interface Types**

  A canonical interface type is divided into two parts corresponding to the external and internal interfaces respectively. Each of these parts contains a list of type declarations and a list of named fields. Each type declaration may optionally specify an upper bound on the type.

$$\begin{aligned} Ci &::= (\ Ci'_{ext}\ ,\ Ci'_{int}\ ) \\ Ci' &::= \langle \mathit{type\_decls},\ \ell_1\colon Ct_1, \ldots, \ell_n\colon Ct_n \rangle \\ \mathit{type\_decls} &::= \mathit{type}\ t_1[<:Ct_1], \ldots \mathit{type}\ t_k[<:Ct_k] \end{aligned}$$

  The subtyping rule for the canonical interface types is a bit more involved. The interface type (E_1, I_1) is a subtype of (E_2, I_2) if for every type declaration and field in the latter, there is one with the same name in the corresponding part in the former, such that

  – the type attached to every field in the supertype is a supertype of the type attached to the corresponding field in the subtype, and
  – bounded type declarations in the supertype have bounds that are supertypes of the bounds on the corresponding type declarations in the subtype

  However, we relax the above by allowing the subtype to make more types and fields external than the supertype does. So essentially, a subtype can "move" types and fields from the internal interface to the external.

- **Canonical Union Types**

  A canonical union type represents the tagged union of a set of types.

$$Cu ::= l_1\colon Ct_1 + \ldots + l_n\colon Ct_n$$

  A union type U_1 is a subtype of U_2 if for every labeled type in U_1, there is a supertype of that type in U_2 with the same label.

- **Canonical Polymorphic Types**

  A canonical polymorphic type is a generic function type that accepts a type argument, which may be bounded by a type expression.

$$Cp ::= \forall t{<:}Ct.Ct'$$

  However, we disallow the bounding expression to involve any polymorphic types itself. This restriction is placed because the subtyping problem for the general case is undecidable. The undecidability arises in a rather subtle fashion, and it was only after Ghelli [Ghe90] gave an "algorithm" for the problem that Pierce [Pie92] proved its undecidability.

  The type $\forall t{<:}A_1.B_1$ is a subtype of $\forall t{<:}A_2.B_2$ if

  – A_2 is a subtype of A_1, and
  – B_1 is a subtype of B_2 assuming the stricter of the two restrictions on t, namely that t is a subtype of A_2.

B Inference Rules

The subtype checking algorithm is specified as a set of inference rules, where execution proceeds by following the rules backwards from the conclusions to the assumptions (certain priorities have to be established between rules to completely determine the execution, but these are not mentioned for brevity). An essential property of the following deterministic set of rules is that subtyping defined by these rules is transitive. In other words, if we extend the system with a transitivity rule, we do not change the set of provable subtyping judgements. This is not a completely trivial property to prove.

In the following rules,

- Γ is the subtyping environment. It is of the form $\{t_1{<:}T_1, t_2{<:}T_2, \ldots t_n{<:}T_n\}$. It is the list of subtyping relationships that are known and can be used in further inference.
- $\mathcal{M}$ is the environment of recursive bindings. It is of the form $\{t_1 \equiv T_1, t_2 \equiv T_2, \ldots t_m \equiv T_m\}$ where each T_i is the recursive definition of the type t_i, and hence t_i occurs in T_i. $\mathcal{M}$ is essentially used to unfold the recursion in recursive types.
- $\mathcal{A}$ is the list of assumptions made while checking the subtyping relationship between recursive types. It is of the form $\{T_1{<:}T'_1, T_2{<:}T'_2, \ldots T_k{<:}T'_k\}$. This list is maintained to avoid comparing the same pair of nodes after a recursive type has been unfolded.

For simplicity in this conference abstract, we omit details concerning types in interfaces and the rules solely for manipulating Γ, $\mathcal{M}$ and $\mathcal{A}$.

(*reflexivity*)
$$\Gamma, \mathcal{M}, \mathcal{A} \vdash A{<:}A$$

(*arrow*)
$$\frac{\begin{array}{c}\Gamma, \mathcal{M}, \mathcal{A} \vdash B_1{<:}A_1 \\ \Gamma, \mathcal{M}, \mathcal{A} \vdash A_2{<:}B_2\end{array}}{\Gamma, \mathcal{M}, \mathcal{A} \vdash A_1 \rightarrow A_2{<:}B_1 \rightarrow B_2}$$

(*recursive*)
$$\frac{\Gamma \cup \{s{<:}t\}, \mathcal{M} \cup \{s \equiv A, t \equiv B\}, \mathcal{A} \vdash A{<:}B}{\Gamma, \mathcal{M}, \mathcal{A} \vdash \mu s.A{<:}\mu t.B}$$

(*interfaces*)
$$\frac{\begin{array}{c}\Gamma, \mathcal{M}, \mathcal{A} \vdash A_i{<:}A'_i \\ \Gamma, \mathcal{M}, \mathcal{A} \vdash B_j{<:}B'_j \\ \Gamma, \mathcal{M}, \mathcal{A} \vdash C_k{<:}C'_k\end{array}}{\Gamma, \mathcal{M}, \mathcal{A} \vdash (\,\langle a_i\!: A_i, b_j\!: B_j, e_n\!: E_n\rangle, \langle c_k\!: C_k, d_m\!: D_m\rangle\,){<:}(\,\langle a_i\!: A'_i\rangle, \langle b_j\!: B'_j, c_k\!: C'_k\rangle\,)}$$

(*unions*)
$$\frac{\Gamma, \mathcal{M}, \mathcal{A} \vdash A_i{<:}B_i \qquad i \in \{1 \ldots n\}}{\Gamma, \mathcal{M}, \mathcal{A} \vdash \{\ell_1\!: A_1 + \ldots + \ell_n\!: A_n\}{<:}\{\ell_1\!: B_1 + \ldots + \ell_n\!: B_n + \ldots\}}$$

(*polymorphic*)
$$\frac{\begin{array}{c}\Gamma, \mathcal{M}, \mathcal{A} \vdash A_2{<:}A_1 \\ \Gamma \cup \{t{<:}A_2\}, \mathcal{M}, \mathcal{A} \vdash B_1{<:}B_2\end{array}}{\Gamma, \mathcal{M}, \mathcal{A} \vdash \forall t{<:}A_1.B_1{<:}\forall t{<:}A_2.B_2}$$

(*trans_var_bound*)
$$\frac{\Gamma \cup \{t{<:}A\}, \mathcal{M}, \mathcal{A} \vdash A{<:}B}{\Gamma \cup \{t{<:}A\}, \mathcal{M}, \mathcal{A} \vdash t{<:}B}$$

(*unfold₁*)
$$\frac{\Gamma, \mathcal{M} \cup \{t \equiv A\}, \mathcal{A} \cup \{t{<:}B\} \vdash A{<:}B}{\Gamma, \mathcal{M} \cup \{t \equiv A\}, \mathcal{A} \vdash t{<:}B}$$

$(unfold_2)$ $$\frac{\Gamma, \mathcal{M} \cup \{t \equiv A\}, \mathcal{A} \cup \{B{<:}t\} \vdash B{<:}A}{\Gamma, \mathcal{M} \cup \{t \equiv A\}, \mathcal{A} \vdash B{<:}t}$$

(var_env) $$\Gamma \cup \{t{<:}A\}, \mathcal{M}, \mathcal{A} \vdash t{<:}A$$

$(loop\_detect)$ $$\Gamma, \mathcal{M}, \mathcal{A} \cup \{A_1{<:}A_2\} \vdash A_1{<:}A_2$$