TYPE SYSTEMS FOR MODULAR PROGRAMMING: THEORY AND PRACTICE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Dinesh Katiyar
December 1994

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. John Mitchell
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. David Luckham

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Sriram Sankar

Approved for the University Committee on Graduate Studies:

Acknowledgements

This thesis owes much to the constant encouragement, indulgence and patience of Prof. John Mitchell, who taught me so much more about being a computer scientist than his duties as an advisor called for. His numerous hints and tips have helped me tremendously over the years. I also thank Prof. David Luckham and the rest of the PAVG research group at Stanford for providing the framework for the practical aspects of this work, and allowing me to be a part of it. A significant portion of this work was done in the ERL cafeteria over coffee with Dr. Sriram Sankar, who brought out some of the best in me and has been a mentor and friend in more ways than one.

I cannot describe the immensely inspiring roles of Dr. Somenath Biswas and J. Radhakrishnan in my initiation to the world of Computer Science. Since my undergraduate days at the I.I.T. Kanpur, these two have helped shaped my professional and personal attitudes in immeasureable ways.

I have been fortunate in meeting some of the most wonderful people during my stay at Stanford. Ashish, Geeta, Jake & Sangeetha, Rajeev, Ranjit, Sandhya, Sujata and numerous others have touched my life in a multitude of ways. Monica, Shefali and the Mittals have been my family away from home. The enthusiastic volunteers of ASHA have taught me the relevance and joy of occasionally being selfless.

I must also thank Stanford University, for letting me not work on my thesis when I should have been, and SUN Microsystems, for letting me work on my thesis when I shouldn't have been.

And, I fondly acknowledge my family for their undying support, for believing in me and for letting me choose my path; and Barkha, for bringing unknown dimensions of happiness into my life.

DK
12/94

Abstract

This thesis presents the design and implementation of a practical type system for languages intended for programming in the large.

Types have always played a significant role in programming languages. The notion of a type originally arose as a way to classify data in a program. This allowed for better utilization of memory, and avoided runtime checks, thus improving program efficiency. Another use of types has been as a programming discipline that allows for much better compile-time error detection. A more recent line of research has been the investigation of types as a way of supporting abstraction and modularity. This thesis is an effort in that direction – it proposes a type system for a language intended to support large-scale multi-authored software systems.

The thesis describes the Rapide type system. Rapide has been designed to support the prototyping of large systems. Types are separate from their implementations in Rapide. Interfaces are the central modularizing construct of the type system. The notion of subtyping provides a rich form of flexibility. Several flavors of parameterization and abstraction allow for very expressive interfaces. The thesis describes all of these in detail.

Theoretical foundations of the Rapide type system are provided. The Rapide type system has been designed to have a conceptually elegant basis. The thesis presents a calculus based on the dual notions of bounded universal and existential quantification. The former captures the different forms of polymorphism in Rapide, while the latter expresses the abstraction mechanisms. The underlying calculus is amenable to analytic studies to prove soundness and to explore expressiveness and efficiency issues. In particular, the investigation of the subtyping rules for bounded universal quantification yields interesting results.

The type system is shown to be expressive enough for practical use. All Rapide predefined types are defined using Rapide interfaces. A contemporary programming language with a rich set of features is shown to be loosely expressible in Rapide.

The type system has been implemented as part of the Rapide 1.0 compiler. The implementation and the Rapide programming environment are briefly described.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents the design and implementation of a practical type system for languages intended for programming in the large.

Types have always played a significant role in programming languages. The notion of a type originally arose as a way to classify data in a program. This allowed for better utilization of memory, and avoided runtime checks, thus improving program efficiency. Another use of types has been as a programming discipline that allows for much better compile-time error detection. A more recent line of research has been the investigation of types as a way of supporting abstraction and modularity. This thesis is an effort in that direction – we propose a type system for a language intended to support large-scale multi-authored software systems.

1.1 Motivation

The notion of a typical computer program has changed dramatically over the last few years. Most software was a few hundred lines of code typically written by a single author. However, as computers became faster and the needs that software could address became larger, the size of software grew in leaps and bounds. Today, a typical software project in the computer industry involves more than a handful of software developers, writing tens of thousands of lines of code, spanning hundreds of files. In this context, the ease of use of a programming language is now significantly determined by its support for large-scale cooperative software development.

Object-oriented programming emerged as a powerful tool in this respect, largely because of its support for abstraction and reusability. However, a quick investigation of popular object-oriented languages shows a whole plethora of features that aim to support permissive styles of programming but that make the languages themselves fairly complex. This leads to involved compiler technology and also complicates the building of programming environments for these languages.

Universal and existential quantification and their flavors (notably those using subtyping) have been shown to support a variety of programming styles. This thesis attempts to use exactly these two features to build the foundation of a programming language that supports large software development. Universal quantification provides the basis for very expressive forms of statically checkable polymorphism, which is a powerful tool for code reuse. Existential quantification, in turn, provides for useful flavors of abstraction and encapsulation, permitting different modules of a large system to be developed fairly independently.

We believe that programming languages need to address issues related to programming in the large, and we hope that this work provides the beginning of one possible direction for the evolution of languages.

1.2 Thesis overview

The thesis broadly consists of four sections.

Chapter 2 provides a description of the Rapide type system. Rapide has been designed to support the prototyping of large systems. In Rapide, types are separate from their implementations in Rapide. Interfaces are the central modularizing construct of the type system. The notion of subtyping provides a rich form of flexibility. Several flavors of parameterization and abstraction allow for very expressive interfaces. This chapter illustrates all of these. The work presented in this chapter represents the joint efforts of the PAVG group at Stanford.

Chapters 3 and 4 describe the theoretical foundations of the Rapide type system. The Rapide type system has been designed to have a conceptually elegant basis. Chapter 3 presents a calculus based on the dual notions of bounded universal and existential quantification. The former captures the different forms of polymorphism in Rapide, while the latter expresses the abstraction mechanisms. The underlying calculus is amenable to analytic studies to prove soundness and to explore expressiveness and efficiency issues. In particular, the investigation of the subtyping rules for bounded universal quantification yields interesting results. This and other details are provided in Chapter 4. The research presented in these chapters was done in part with Prof. John Mitchell and Dr. Sriram Sankar.

Chapter 5 illustrates the expressiveness of the Rapide type system. Rapide predefined types are used as one measure of expressibility and these are all show to be definable using Rapide interfaces. Also, a contemporary programming language with a rich set of features is shown to be loosely expressible in Rapide. Prof. John Mitchell assisted in this exposition.

Chapter 6 discusses the implementation of the RAPIDE type system. The type system has been implemented as part of the RAPIDE 1.0 compiler. The implementation of the type system including the subtype checker, and its role in the RAPIDE programming environment are briefly described. The author's implementation was part of the front end of the compiler that was largely developed by Walter Mann.

Chapter 7 summarizes related work and points towards future directions of research.

Chapter 2

Overview of the RAPIDE type system

This chapter gives an overview of the Rapide language framework and the Rapide type system and describes some of the motivation and rationale for the design.

Rapide is a programming language framework with an associated toolset. The framework consists of a set of languages, while the toolset provides related program development and diagnostic tools. The framework includes a type language, a default executable language, a default specification language, and an architecture language. Although these languages together provide a complete programming language, Rapide is intended to accommodate additional implementation and specification languages, besides the default ones provided by the framework. The constant of the framework is the type language, which is intended to be general enough to allow the types of relevant implementation languages to be expressed within it. The architecture language collects together those features of the type and executable languages that are typically used in defining the interfaces of system components and the dataflow connections between them.

2.1 The Rapide language

2.1.1 Language goals

Rapide was designed with certain goals in mind. These goals will reflect in many of the design decisions discussed in the rest of this chapter. The principle goals of Rapide were the support of the following:

Prototyping: Prototyping is the incremental development of systems by an iterative design and development process. In a typical project, partial system components, called *prototype components,* are built. These are analyzed separately and combined into a *system prototype,* which is analyzed to determine the adequacy of the design, to refine the specifications of the components, and to estimate system performance. The analysis of a prototype provides useful feedback for subsequent revision and refinement of the system design or of individual components. Prototyping is often an experimental activity. Alternate system designs can be evaluated by building prototypes for each and analyzing and comparing them. This allows crucial design decisions to be made as early in the design process as possible.

Multi-language systems: One goal of RAPIDE, and a reason for adopting a single, expressive type system, is to allow incremental migration of prototypes from one language to another. There are several reasons for working with multiple languages. For large systems where there is a premium on producing initial prototypes quickly and many revisions or modifications of the initial design may be expected, the default specification and executable languages have advantages. They are easy to program in due to general but sometimes inefficient programming constructs and due to the associated development tools. However, it may not be feasible to deliver a final program in the default executable language. An example may be in embedded systems, when limited computational resources are available on the final platform. Since interoperability of programs written in significantly different languages presents efficiency problems, compatibility with existing code might also dictate a particular programming language.

Different styles of programming: Another crucial language design goal is to support a range of programming styles within a single framework. One style of interest is the traditional program development by "step-wise refinement," using abstract data types and specifications. Another is the recent object-oriented style of programming which tends to provide better support for program development that involves successive refinement of working code, with substantial code reuse from one prototype to the next. The RAPIDE framework attempts to support both these modes of program development and evolution.

2.1.2 The RAPIDE framework

The languages of the framework

The RAPIDE language framework consists of a *type language* to define the interfaces of components, a *specification language* to specify and constrain the behavior of components, and an *executable language* to construct modules that implement system components. The *pattern language* is used by the executable and specification languages to characterize the execution of RAPIDE programs.

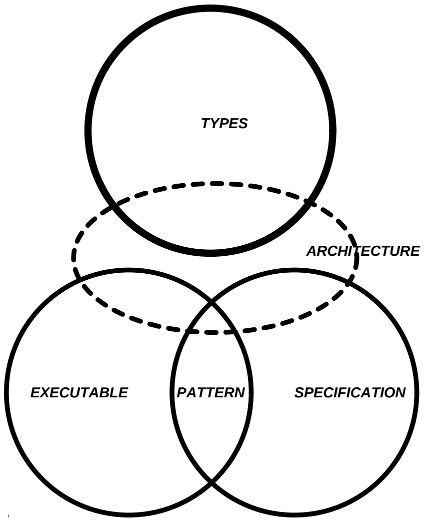

Figure 2.1: The RAPIDE language framework

The *architecture language* provides features for easily "wiring up" prototype components – *i.e.*, for defining explicitly the (possibly dynamic) communication links between modules.

Some languages of the RAPIDE framework are better understood in the context of the computational model associated with the default executable language. Computations are viewed as partially ordered sets of events (*posets*). The synchronization, concurrency, and timing aspects of a prototype are explicitly represented in such computations. Requirements on computations can be specified formally as constraints on these posets, and violations of these can be detected automatically. This model is described in greater detail in [LVB+93].

The standard languages of the framework are summarized below.

The type language: This language provides constructs for defining the interfaces of components and for deriving new interfaces from prior ones. "Derivation" resembles inheritance mechanisms in object-oriented languages. Since interfaces define the *types* of modules, this language also provides what is referred to as the RAPIDE *type rules*. Conformance with these type rules can be checked automatically, and provides a safeguard against many of the errors that can be expected to arise during prototyping – for example, in reconfiguring a system by changing the implementation of some of its components, or by introducing a new type of component. However the type rules are also flexible in allowing replacement of modules of one type by modules of another type according to type compatibility rules determined by their interfaces.

The executable language: The executable language is a concurrent reactive programming language in which one defines the executable behavior of components. Its principle constructs are independent (or concurrent) reactive processes that activate when patterns of events occur during execution. These pattern-triggered processes are used for two purposes:

1. to define architecture connections between components, and
2. to construct behaviors of components by rule-based, reactive programming techniques.

The executable language also provides standard kinds of Algol control structures, subprograms, exception raising and handling constructs, and timing features. The result of executing a RAPIDE program is a set of events with various partial orderings (such as time and causality) relating the events. This is referred to as the *poset execution model*.

The specification language: This language provides constructs for abstract specification of the behavior of a distributed system, including timing requirements. It is based on the poset execution model. The constraint-based specification language supplies a rich set of constructs for defining patterns of events that should or should not occur in any computation of a distributed system. It is also designed so that straightforward algorithms can be used to check any actual execution of a prototype for violations of pattern constraints.

The specification language supports various ways of using formal specifications in systems development. It provides a capability for automated diagnostic analysis of a prototype's behavior by checking consistency of executions and specifications. The use of formal specifications in *(i)* precise definition of component interfaces and system architecture, and *(ii)* automated analysis of simulation data from very large prototypes, plays a key role in large scale prototyping.

The pattern language: This sublanguage of the default specification and executable languages provides a convenient notation for characterizing posets. This capability is used in the specification language to impose constraints on event-based behaviors, and in the executable language to generate, detect and react to certain patterns of events. It is used by the architecture language to define communication between components.

The architecture language: This language provides basic mechanisms to wire together system components by defining the connections between them. Architectures are sets of component interfaces and connections. Connections allow components to synchronize and to communicate data. The need for this language is motivated by the prototyping of large distributed real-time systems, where the early stages of prototyping concentrate on system architecture. The separate architecture language allows the prototyping process to be started without the full knowledge of the rest of the framework.

The architecture language is defined as a natural extension of the type language. Component interfaces are defined as interfaces of the types language, extended with event-generating behaviors and constraints.

The RAPIDE toolset provides several design and diagnostic tools to aid program development and refinement. The architecture language has a graphical interface to allow component interfaces and connections to be defined easily. Graphical browsers facilitate navigating through module hierarchies. A separate tool allows inspection of the output of simulations (*i.e.* partially-ordered set of events) in several useful ways. Constraint checkers detect violations of constraints during execution of RAPIDE programs.

Why a framework?

Although most programming languages are presented as a single design, the notion of "language framework" actually dates to [Lan66], if not earlier. There are several reasons for formulating RAPIDE as a framework of interdependent sublanguages. A language framework allows users to deal with certain aspects of the design without having to learn the details of all the sublanguages. This is also useful because the different languages of the framework are typically associated with separate stages in the development process – one might first use the specification language to specify the requirements of the system, then use the types language to define the interfaces of the components, and finally use the executable language to implement the system. A framework also allows the use of an alternate implementation or specification language in conjunction with the same type language. Further, modules specified and implemented in different languages can be described uniformly in a single interface language (*i.e.* the types language). This is an essential first step towards tackling issues related to interoperability of languages, and facilitates the evaluation of the impact of migrating a prototype from one language to another. Finally, a framework makes it easier to incorporate evolutionary language changes that could arise from future research and experience. Independent changes may be made to each sublanguage, as long as the explicit conventions about the relationships between the sublanguages of the framework are maintained.

Interaction between the languages

The languages of the framework must satisfy certain compatibility requirements and conventions. These constraints are of different kinds:

- Certain restrictions apply globally. The languages of the framework must share certain visibility, scoping and naming rules. (These are just the rules of block-structured, statically-scoped languages.)
- The type system imposes syntactic and semantic restrictions on any executable or specification language used within the framework.

  The syntactic constraints essentially guarantee that the grammars of the framework languages can be combined to form an unambiguous context-free grammar. The types grammar provides explicit non-terminals that may be used in the other languages of the framework. Similarly, the types language requires the other languages to provide rules for some non-terminals that its own grammar relies on. The general approach is illustrated by following productions from the grammar for the RAPIDE types language.

```
type_expression ::=
      interface_type_expression
    | function_type_expression
    | type_constructor_application
    | FRAMEWORK_type_expression

interface_type_expression ::=
      interface
      {interface_declarative_region}
      end interface

interface_declarative_region ::=
      public_declarative_region
    | private_declarative_region
    | external_declarative_region
    | FRAMEWORK_declarative_region
```

Any language may use the non-terminal `type_expression` in its grammar. In exchange, other languages of the framework may provide the syntax for non-terminals such as *FRAMEWORK_type_expression* and *FRAMEWORK_declarative_region*. This is typically done in the sublanguage X by defining new non-terminals of the form *X_type_expression* and *X_declarative_region*. In an architecture language that extends interfaces with behaviors and constraints, for example, we would have a production of the form

```
ARCHITECTURE_declarative_region ::=
   interface_behavior_region
  | interface_constraint_region
```

The semantic conditions ensure that the operations provided by the executable language behave in a manner that is, loosely speaking, consistent with respect to the type system. The types language assumes that the executable language provides not only expressions for function bodies and modules, but also operators to apply functions to arguments and to select constituents from interfaces. Furthermore, these operators have to behave "correctly" with respect to the types declared. This form of correctness is formalized in the typing rules and exhibited in a form of modular type soundness proof developed for the RAPIDE language framework (see Section 3.2.4).

- The specification and executable languages may impose additional constraints on each other, independent of the types language. For instance, in the default set of languages, the executable language uses a computation model based on partially ordered sets of events. In this context, a specification language is most useful if it can meaningfully refer to partially-ordered computations. This is achieved in the default executable and specification languages by sharing a pattern language for defining and recognizing partially-ordered patterns of events.

A formal study of these interactions is a subject of future research.

2.1.3 Current status of RAPIDE

Over the past four years, the RAPIDE design has been tested, modified and evaluated in a variety of ways. An early implementation, based on a translation to Ada, was completed in 1991. Since then, several programming tools have been implemented, including a graphical interface to allow component interfaces and connections to be defined easily, a graphical browser to facilitate navigating through module hierarchies and to check subtype relationships and a separate tool to inspect the partially-ordered set of events in several useful ways. This allowed experimentation with the concurrent aspects of the language and some evaluation of the program-structuring mechanisms. Several large systems, including the ADAGE avionics system [TC93], the X-open distributed transaction processing architecture [Ken93], a "trusted" X-windows definition, as well as many small ones, have been expressed in the RAPIDE type system. The diagnostic information provided by the tools significantly contributed to the refinement of the framework design, including the type system. The second generation language and tool implementations, now well underway, are being done in C++.

2.2 The type system

This section starts by summarizing the novel aspects of the RAPIDE type system. It then motivates the notion of subtyping used in RAPIDE and goes on to describe the various concepts of the type system.

2.2.1 Salient features

This section describes some of the more fundamental concepts of the RAPIDE type system.

Types separate from implementations: The RAPIDE framework has separate types and executable languages. As a result, the types of components are entities distinct from their module implementations. This has several advantages. The interface language provides a uniform way, independent of the executable language, of describing the functionality that is provided by an implementation or that is desired of an implementation. Thus, the separate interface of a module serves as a useful form of documentation for that module. As explained later in this section, this separation also plays an important role in allowing for flexible and independent derivation mechanisms for the interfaces and implementations.

Interfaces as a unifying concept: A central, unifying idea in the RAPIDE type system is that all types are expressible using interface types, along with function types. Interfaces are the central construct of the type system. Interfaces define the types of both compile-time groupings of related types and operations, resembling, for instance, Ada packages [US 80] and as run-time objects of the form found in C++ [ES90] and other object-oriented languages. Section 2.2.3 illustrates this.

Subtyping independent from inheritance: The subtyping hierarchy is completely independent of the inheritance mechanisms. Ample motivation for this choice is provided in [Sny86, CHC90, Coo92]. In RAPIDE, inheritance is viewed purely as a technique for code reuse. This allows RAPIDE to provide independent derivation mechanisms for interfaces and implementations.

Inferred structural subtyping: The subtyping hierarchy is purely structural – it is inferred by examining the types, and is based on a notion of substitutivity (Section 2.2.2). The rules for subtyping are presented in Section 3.1.5, and some interesting algorithmic issues are discussed in Section 4.1.

Expressive parameterization mechanisms: Parameterization may be used to define a family of related types, depending on either type parameters or value parameters. A type parameter in an interface or function declaration may be constrained by requiring it to be a subtype of some given type. As illustrated in Section 2.2.3, this allows the type system to express fairly flexible forms of polymorphism.

Expressive flavors of abstract data types: The RAPIDE type system allows one to express several flavors of abstract data types. Subtyping constraints can be imposed on type constituents in interfaces, allowing "partially abstract" types. Type constituents in interfaces can also be specified completely, giving RAPIDE the expressiveness of languages such as Standard ML [MTH90] while still allowing modules to be first-class. Section 2.2.3 discusses this in detail.

Flexible interface derivation mechanisms: The derivation mechanisms of the type language allow interfaces to be combined and reused; they are essentially a form of inheritance mechanism appropriate for defining related interfaces. The separation of inheritance from subtyping allows these to be fairly flexible. Section 2.2.3 provides details. The derivation mechanisms of the language can be applied to parameterized interfaces as well. This allows the type system to express popular notions such as *method specialization* and *mixins*. This is described in Section 2.2.3.

2.2.2 Subtyping and substitutivity

The fundamental relation between RAPIDE types is the subtype relation. The main principle associated with subtyping is *substitutivity*: if A is a subtype of B, then any expression of type A may be used without type error in any context that requires an expression of type B.

RAPIDE uses the notation "A <: B" to indicate that A is a subtype of B.

There are two significant ways in which subtyping is useful.

Specialized uniform behavior: Subtyping allows for uniform behavior over various types of data. In particular, subtyping makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some type. Consider, for example, a move function for a windowing environment that contains different kinds of windows such as text and graphic windows, amongst others. If all these kinds of windows are subtypes of some given window type, one can express the functionality of the move function. The examples in Sections 2.2.3 and 2.2.4 will further illustrate this.

Adding functionality: 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 B have some behavior which crudely approximates part of the system being prototyped, then it may be desirable to replace objects of type B with objects of another type A that have more realistic or accurate behavior. In many cases, the type A will be a subtype of B. By designing the language so that substitutivity is allowed, one may add functionality in this way without any modification to the original program.

RAPIDE provides rules for determining, by compile-time analysis of the structure of the types, if one type is a subtype of another. RAPIDE chooses this implicit structural approach over two other prevalent trends – *(i)* determining the subtype relationships directly from the derivation hierarchy, and *(ii)* requiring all subtype relationships to be explicitly stated. This is done for the following reasons:

- The subtyping-from-inheritance approach either compromises the substitutivity principle or places strong restrictions on the derivation mechanisms [CHC90]. It also causes the subtyping relationship to be greatly affected by the order in which the program components are written.

- The explicit subtyping approach is rather cumbersome, since many "obvious" relationships would have to be made explicit. Also, it is hard to fit a new type into an existing hierarchy. This approach does avoid the problem of "accidental" subtyping (caused when one type is determined to be a subtype of a completely unrelated type simply because they happen to have similar structures). However, accidents do not seem to be troublesome enough to warrant the added responsibility on the programmer. Some circumstantial evidence for this is provided in the work of [Rit91, RT91], where it is shown that types can be used to search for functions in libraries, generating very few accidental "hits".
- The RAPIDE approach makes it easier to reorganize code and to combine independently constructed subprograms. There is no need to change or introduce any subtyping relationships - they are all automatically determined from the program text.

The subtyping problem is fairly crucial to the process of type-checking and is the topic of recent research [Pie92, KS92, Kat92, GP94].

2.2.3 Interfaces

This section describes the central construct of the RAPIDE type system, namely, interfaces.

Simple interfaces

An *interface* is defined by a list of declarations. These may be declarations of object names (where an object is either a module or function) or types. A declaration of an object name associates a type with the name, but does not give an implementation. A type declaration in an interface may give only a type name, a type name with a subtype designation, or a complete type declaration (these are discussed in detail in Section 2.2.3). Overall, an interface is similar in spirit to an Ada package specification (except that packages are not types in Ada) [US 80], the signature part of a C++ class [ES90], or a Standard ML signature [MTH90] (except that Standard ML modules are only compile-time entities, and cannot be used as run-time values).

The objects of an interface type are modules containing type and object constituents matching the declarations listed in the interface. (The term *constituent* is used for components of a module to avoid conflict with the term *component* for component of a system or architecture.) All object constituents must have the type indicated in the interface. A type constituent may be *(i)* any type when the interface contains only a type name declaration, *(ii)* any subtype of the type given in the interface, if a subtype designation is used, or *(iii)* must be exactly the type given explicitly in the interface.

The interface of a module governs access to its constituents. The constituents that correspond to declarations in its interface are *visible,* which means that they may be referred to outside the module. If a module declares identifiers that do not occur in the module interface, then these are *hidden*, *i.e.*, not accessible from outside. In the case of visible type constituents, the subtype or type equality information given in the interface determines whether the type constituent will be considered a subtype of or equal to any other type in the program. This is discussed in Section 2.2.3.

As mentioned earlier, interfaces define the types of both object-like and package-like entities. The following two examples illustrate these respectively.

Example: *A Point interface*

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

The interface type Point contains three name declarations. Modules of type Point must have two integer constituents, called X_Val and Y_Val, and a function constituent called Distance. A function computing the distance between two points requires only one argument, since the function belongs to a particular point and therefore may compute the distance between the point passed as an actual parameter and the particular point to which the function belongs. In other words, the intended use of the Distance function constituent of a point object x is to compute the distance between x and another point object y, by a call of the form x.Distance(y). Of course, since the interface gives only signature information, the Distance constituent is not forced to compute the distance between two points. If the user wishes to specify that Distance must compute distance, this may be added to the interface using the specification sublanguage.
□

Example: *A Stack package*

```
type Int_Stack_Package is interface
    type Stack;
    New : function() return Stack;
    Push : function(X : Integer, S : Stack) return Stack;
    Pop : function(S : Stack) return Stack;
    Top : function(S : Stack) return Integer;
end interface;
```

This interface illustrates the traditional abstract data type style. An object of Int_Stack_Package must have a type constituent Stack and the function constituents listed above. It is intended that these constituents provide the standard operations on stacks. Since only the type name, Stack, is given in the interface, the type constituent of an object of type Int_Stack_Package will be hidden. More specifically, outside a module Stack_Impl of type Int_Stack_Package, the type Stack_Impl.Stack will not be recognized as equal to or a subtype of any other type in the program.

□

In addition to name declarations, an interface may also contain *specifications*. These are constraints on the behavior of the executable objects, and are written in the specification language. The purpose of specifications in interfaces is to document the intended semantics of modules formally and more precisely than can be achieved by types alone *e.g.* input/output constraints on parameter values of operations or restrictions on the order in which the operations can occur. Executions can be checked for violations of specifications by an automated checker at runtime. Details of the specification language can be found in [RSp].

In the examples above, all interface declarations are *public*. In general, a Rapide interface may also declare names as *private* or *external*. These are described in Section 2.2.5.

Interface types have been used to write the interfaces for all the predefined types supported by the executable language of the framework. The exercise of specifying these predefined types as interfaces has illustrated the expressiveness and flexibility of the type system. Section 2.2.6 describes this aspect further, and complete details can be found in [RPr].

Interface derivation mechanisms

Rapide derivation mechanisms provide a flexible form of reuse. The purpose of a derivation (or inheritance) mechanism is to allow a declaration made in one place for one purpose to be used in another place for a similar, related, or perhaps even different purpose. In Rapide, interface derivation declarations define textual dependencies between interfaces. Any change to an interface imposes similar changes to all interfaces derived from it. This is an important feature for rapid prototyping since it simplifies program modification and maintenance by reducing the amount of redundant or repeated code, but it must be used carefully to avoid unexpected changes.

In contrast with Smalltalk [GR83], C++ and other object-oriented languages, Rapide has distinct inheritance mechanisms for interfaces and implementations. This has proven very useful, for the general reason that interface and implementation inheritance are often used independently and/or in opposite ways. For instance, one would want to define the interface for double-ended queues by extending that for simple queues, whereas the latter can be easily implemented from the former [Sny86].

Similarly, the Smalltalk library shows several cases of conflict between the implementation hierarchy and a reasonable interface hierarchy [Coo92].

The simplest form of derivation declaration is

include type_expression

Essentially, a declaration "**include** A" inside an interface type B has the effect of inserting all the declarations of A into B. Since this requires knowing the interface A completely, the type expression that is **include**d must denote an interface type that can be determined completely at compile time. This is an important restriction on the use of derivation declarations.

A simple example is the derivation of the interface for colored points from the Point interface given in Section 2.2.3.

Example: *Deriving Colored Point from Point*

```
type Colored_Point is interface
    include Point;
    Col : function() return Color;
end interface;
```

Colored points are derived from points using an **include** declaration. The result is that the Colored_Point interface has four function constituents, X_Val, Y_Val, Distance and Col.
□

It is often useful to be able to include only a part of a previously declared interface. Rapide provides the following forms of derivations for this purpose

include type_expression **only** identifier_list

include type_expression **except** identifier_list

which include only the listed constituents, or include all constituents except those listed (respectively).

Unlike other languages that allow a single entity to be derived from more than one other (*i.e.*, multiple inheritance), Rapide does not have any standard or default rules for resolving conflicts.

One reason is that such rules are generally complicated, and unpredictable in certain cases. Moreover, it is anticipated that the RAPIDE type system will be used in conjunction with multiple executable languages, and these may each have their own conflict resolution rules for modules. However, RAPIDE makes it possible to avoid conflicts by renaming included constituents. For instance, one can write

```
include A  only  x,y
           rename  x  to  new_x,
                   y  to  new_y
```

Parameterized interfaces

One often needs to define interfaces uniformly for various types – consider writing set packages for integers, for reals, for strings, and so on. This may be done in Ada using *generics*, or in C++ with *templates*. RAPIDE has a similar capability that is more flexible because type parameters may be constrained to be subtypes of given types. The flexibility and expressiveness that is thus made possible is illustrated through the following example.

Example: *A Set package*

A generic set interface can be written as following, using a type parameter as the type of elements in the set.

```
type SetType(type Elem) is interface
    Insert : function(E: Elem);
    Delete : function(E: Elem);
    Union : function(S: SetType(Elem));
    Is_Member : function(E: Elem) return Boolean;
end interface;
```

The above declaration defines a type constructor. When applied to a particular type T, this gives an interface type SetType(T). Each object of type SetType(T) will have the four function constituents listed above. The type of each function constituent may be computed by substituting T for Elem in the declaration above. For example, if S has type SetType(T), then the type of S.Insert is **function**(E: T), meaning that S.Insert is a function that requires an actual parameter of type T and returns no result. The intended use of this type constructor is to give the general form of set type, where each module implements a single set in the usual object-oriented way.

Now consider the following. Since the set interface provides a function constituent Is_Member, it might be reasonable to expect the SetType constructor to be applied only to element types that have a notion of equality defined on them. Using the subtype constraints that can be imposed on type parameters, this can be expressed as follows:

```
type Equality(type Elem) is interface
    Eq: function(E: Elem) return Boolean;
end interface;

type SetType(type Elem <: Equality(Elem)) is interface
    Insert: function(E: Elem);
    Delete: function(E: Elem);
    Union: function(S: SetType(Elem));
    Is_Member: function(E: Elem) return Boolean;
end interface;
```

□

There is a slight subtlety in the form of the subtype constraint imposed in the above example. The two variants of such forms of restricted polymorphism are illustrated in the context of polymorphic functions in Section 2.2.4.

Parameterization can be combined with the interface derivation mechanisms to express useful notions such as *mixins* and *method specialization*.

- **Mixins:** It is often useful to include constituents of an interface that is defined using a type constructor. By applying a type constructor to the interface that is being defined, one may specialize the types of included constituents. This, in fact, gives RAPIDE the power to program so-called *mixins* (in the terminology of CLOS [Kee89]) without requiring a separate concept. This is illustrate in the following example, which adds equality to points.

  Example: *Points with Equality*

  ```
  type Equality(type T) is interface
      Eq : function(X : T) return Boolean;
  end interface;

  type Eq_Point is interface
      include Point;
      include Equality(Eq_Point);
  end interface;
  ```

```
type Eq_Point_Elab is interface
    X_Val : function() return Integer;
    Y_Val : function() return Integer;
    Eq : function(X : Eq_Point_Elab) return Boolean;
end interface;
```

The type constructor Equality(type T) returns an interface with one function constituent. The type of this constituent contains the formal type parameter T, and therefore will depend on the actual parameter of the type constructor. Interface Eq_Point includes the constituents of Point and Equality(Eq_Point). Since Equality is applied to the actual type parameter Eq_Point, the resulting Eq function constituent has type

function (X : Eq_Point) **return** Boolean

obtained by replacing the formal parameter T with the actual parameter Eq_Point. Therefore, the Eq function constituent of an Eq_Point object accepts another Eq_Point as an argument. The interface Eq_Point is equivalent to Eq_Point_Elab, which is written directly without any derivation declarations.
□

- **Method specialization:** Another use for type constructors in derivation declarations arises when specializing recursively-defined interfaces. In languages without derivations, interface definitions can simply use the name of the interface being defined to make recursive interface definitions. However, when one permits these definitions to be reused in defining new interfaces, one would like the recursive occurrences of the interface name to now refer to the new interface being defined. Hence, such languages typically provide a separate keyword for these references – for example, Eiffel [Mey92] uses the term "*like current*". In RAPIDE, however, we can use type constructors to overcome the need for a new feature. More specifically, instead of writing

```
type T is interface ... like current...;
```

one would write

```
type TGen(type t) is interface ...t...;
type T is TGen(T);
```

Derivations are then done using the constructor TGen instead of T. Since RAPIDE has no special type symbol *like current*, there is no need for special typing rules of the form found in Eiffel, for example, or the concomitant typing difficulties (see [Coo89b]).

Interfaces as abstract types

In Section 2.2.3, the Int_Stack_Package example showed how one could define *simple abstract types* in RAPIDE. In that example, the type declaration in the interface contained only a type name Stack, which declared a hidden "private" type. However, one can also declare a type name with a subtype designation, or a complete type declaration. Subtype designations give "partially hidden" types, which shall be referred to as *bounded abstract types*. Complete type declarations result in "public" types that shall be called *exact abstract types*.

syntax	*visibility*	*term*
type t;	private	simple abstract type
type t <: T;	partially hidden	bounded abstract type
type t = T;	public	exact abstract type

Figure 2.2: RAPIDE abstract types

- **Bounded abstract types**

  When a type in an interface is declared as a type name only, the corresponding type constituent of any module is highly abstract, in the sense that outside the module, this type is not equal to nor related by subtyping to any other type in the program. Such non-subtypeable types have an important pragmatic consequence. An implementation of such a type can assume that all objects of that type will be represented in exactly the same way, and therefore can implement the operations of the object more efficiently.

  However, there is a slight inconvenience with such abstract types by virtue of their not having any supertypes either. Since nothing is known about their representation, objects of such types cannot be used in any context but of the operations on the abstract type. The use of subtype constraints on type constituents provides a useful compromise. In this case, the types are only partially hidden, since some information about the operations on the type is specified in the subtype designation. This makes the type non-subtypeable, while still allowing supertypes, and hence permitting its use in contexts that are satisfied with the partial information about the type. This is illustrated in the following examples:

  Example: *Unique implementations*

```
type Uniq_Spec is interface
    type Uniq <: interface
                        ... public operations on instances of Uniq...
                    end interface;
  New_Uniq_Element : function() return Uniq;
  end interface;
```

This example illustrates the general technique of using bounded abstract types to declare non-subtypeable types. For any implementation Uniq_Spec_Impl of the interface Uniq_Spec, only partial information about the type Uniq_Spec_Impl.Uniq is known outside the module, thus making it impossible for any instances of this type to be created other than through the New_Uniq_Element function of Uniq_Spec_Impl.

□

Example: *Partially hidden clock type*

```
type Clock_Spec is interface
  type Clock <: interface
                  tick:function();
                  read:function() return integer;
                end interface;
  new_clock : function() return Clock;
end interface;
```

This example shows how to specify some constituents, without giving an explicit type definition. The interface declares a type Clock together with a function new_clock that returns a "new" clock. If Clock_Impl is a module with interface Clock_Spec, then Clock_Impl.Clock is a type that is known to be a subtype of the interface given explicitly above. Therefore, if C is an object of type Clock_Impl.Clock, we can call function constituents C.tick() and C.read(), the first returning no value and the second returning an integer. Since the type Clock is not given explicitly in the interface, one cannot define a type that is a subtype of Clock_Impl.Clock. Therefore, the only elements of type Clock_Impl.Clock will be those produced by calls to Clock_Impl.new_clock.

□

Example: *Time-stamped events*

```
type Event_Spec is interface
    type Time_Stamped_Event <:
         interface
             event_name:string;
             precedes:function(E:Time_Stamped_Event)
                        return boolean;
         end interface;
    new_event:function(Name:string)
                   return Time_Stamped_Event;
end interface;

module Event_Impl() return Event_Spec is
    type Time_Stamped_Event =
         interface
             event_id:integer;
             event_name:string;
             precedes:function(E:Time_Stamped_Event)
                        return boolean;
         end interface;
    new_event:function(Name:string)
                   return Time_Stamped_Event is
          module
              event_id:integer = System.Clock();
              event_name:string = Name;
              precedes:function(E:Time_Stamped_Event)
                         return boolean =
                       return(event_id < E.event_id);
          end module;
end module;
```

This example illustrates how non-subtypable types can lead to more efficient implementations. Event_Impl is an implementation of Event_Spec. As explained earlier, one cannot define subtypes of Event_Impl.Time_Stamped_E
This allows Event_Impl to assume that the argument E of the precedes function has the same unique internal representation for events. Therefore, the precedes function can access the internal representation of its argument – in this case, this is done through the event_rep constituent.
□

Non-subtypeable types are analogous to the "sealed classes" of Dylan [App92].

- **Exact abstract types**

  Full type declarations permit the type-checking of incremental definitions of related types, as in the style developed for Standard ML [Mac86, Tof89]. (The relationship between Standard ML type constituents, which are considered "public" by default, and full type declarations in interfaces is discussed in [HL94, Ler94].)

Example: *Extending the functionality of abstract types*

```
type T_Package is interface
    type T;
    f : function(x:T) return T;
end interface;

type Ext_T_Package(t:T_Package) is interface
    type T' is t.T;
    g : function(x:T') return T';
end interface;
```

The type T_Package defines an abstract type and a function on that type. Given an instance t of this package, one can define values of type t.T and apply the function t.f to them.

Now suppose we wish to extend the functionality of the T_package type with another function g such that the new function can be applied to instances of the old package type. This can be done as shown above using a full type declaration. For each implementation of T_Package, Ext_T_Package defines an interface with a type constituent that is equal to that of the particular T_Package. (Note that one couldn't define Ext_T_Package using the interface derivation mechanisms of Section 2.2.3, since that would result in a new independent abstract type.)

□

Example: *Adding functionality to Stacks*

```
type Int_Stack_Package is interface
    type Stack;
    Push : function ...
    Pop : function ...
    Top : function ...
end interface;

A : Int_Stack_Package is ...;
```

```
type Stack_A_Ext is interface
    type Stack = A.Stack;
    PushList : function(L : List(Integer), S : Stack) return Stack;
end interface;
```

This example provides a concrete illustration of the motivation of the previous example. If one wishes to add the function Pushlist that should still be applicable to instances of stacks created using the A implementation, one could use the above style to do so.
□

2.2.4 Functions

Function types in RAPIDE are enriched with fairly expressive notions of parameterization. These are identical to those provided for interfaces as well. The various forms are summarized in Figure 2.3, and are illustrated in the examples below.

syntax	*characteristic*	*term*
f(type t,...)	uniform behavior over all types	simple polymorphism
f(type t <: T,...)	parametric over simple constraints	bounded polymorphism
f(type t <: G(t),...);	parametric over "circular" constraints	F-bounded polymorphism

Figure 2.3: RAPIDE polymorphic types

Simple polymorphic functions allow one to specify behavior that is uniform over all types. In other words, the function does not have to make any assumptions about the type of elements involved.

Example: *Simple polymorphism : reversing lists*

```
Reverse : function(type T, L : List(T) )
               return List(T)
```

Lists can be reversed independently of the type of elements stored in the list. Hence the Reverse function on lists can be assigned a simple polymorphic type, as shown above.

□

However, it is often the case that one wishes to define some behavior that is uniform over all types that have certain functionality. One useful way of permitting such behavior is to allow functions that are generic over all subtypes of a given type.

Example: *Bounded Polymorphism : moving windows*

```
Wd_Move : function(type W <: Window, x:W)
              return W;
```

Consider a function Wd_Move that can move windows of different kinds - graphic windows, text windows, and so on. If all these are subtypes of a type Window which defines all constituents that the Wd_Move function requires, one could write the type of this function as

Wd_Move : **function**(x : Window) **return** Window;

However, since the return type of this function is Window, the type system is forced to view all graphics and text windows as simple windows, once they are moved. But the bounded polymorphic type of the example above allows the type system to be more "knowledgeable" and return the more specific type.

□

The flavor of bounded polymorphism above allows generic behavior over all subtypes of a given type. Another very useful form of polymorphism allows the constraint on the type parameter to be more general - specifically, the type bound is allowed to refer to the type itself. This style of *F-bounded polymorphism,* as it is called, is further motivated and illustrated in [CCH+89].

Example: *F-Bounded Polymorphism : sorting lists*

```
Sort : function(type T <: Ordered(T);
                          L : List(T))
           return List(T);

type Ordered(type T) is interface
    Less : function(X : T) return Boolean;
end interface;

type Integer is interface
    ...
    Less : function(N:Integer) return Boolean;
    ...
end interface;
```

To sort lists, one needs to assume that the elements of the list are “comparable”. This can be specified, as above, by requiring the element type of the lists to have a constituent function Less which permits two elements of that type to be compared. The interface for Integer above shows how the constraint Integer ¡: Ordered(Integer) would be satisfied, thus allowing Integer lists to be sorted.
□

2.2.5 Private and external interfaces

In Section 2.2.3, it was said that interface types consist of a list of declarations. This was a slightly simplified view of interfaces. The constituents mentioned there are part of what is called the *public interface*. Constituents of the public interface are visible to the entire system. Interfaces can contain two other subparts - a *private interface* and an *external interface*.

Private interfaces

The private interface contains a list of declarations similar to that in a public interface. However, these constituents are visible only within modules of this interface type. The need for this arises from the difficulty in implementing binary operations in the object-oriented style of programming. Usually implementing a binary operation efficiently requires some information about the representation of the two arguments. However, in the object-oriented style, binary operations only have access to the internals of their first implicit argument; they can only use the publicly available functions to look at the second. Private interfaces provide a means of “leaking” as many details of the internal representation as may be required, but only to objects of the same type. This is made clear in the following example:

Consider an interface for sets of integers.

```
type IntegerSetType is interface
    Insert : function(N : Integer);
    Delete : function(N : Integer);
    Union : function(S : IntegerSetType);
    Is_Member : function(N : Integer) return Boolean;
end interface;
```

Now consider how one might implement the union function. The natural and efficient way of doing it requires the ability to iterate over the elements of the set passed as an argument. But the publicly available functions on sets don't provide an apparent way of doing this. The most realistic solution would be to add a new function in the interface which returns a list of all the members of the set. However, this means making more operations public than required. Also, adding a new constituent during program development may require changes in certain derived interfaces. Using the private interface, one can write:

type IntegerSetType **is interface**
public
  Insert : **function**(N : Integer);
  Delete : **function**(N : Integer);
  Union : **function**(S : IntegerSetType);
  Is_Member : **function**(N : Integer) **return** Boolean;
private
  List_of_Members : **function**() **return** Integer_List;
end interface;

Now, only other integer sets can invoke the List_of_Members functions.

Private interfaces are similar in spirit to the C++ friend concept.

External interfaces

An interface may also have an external part which consists of a set of external declarations. This is a list of object names with specified types that modules belonging to this interface can assume to be visible. The executable language is expected to provide means of associating actual objects with these declarations.

The use of external declarations is strongly motivated by the use of the Rapide type system to specify architectures and their connections.

Consider the interface Timer that describes timing devices. A timer can provide various timing utilities as long as it has access to some clock. This may be specified by using the external declarations to require that a clock be made available to every instance of the type Timer.

```
type Timer is interface
public
    Start : function();
    Stop : function();
    Reset : function();
    Time_Elapsed : function() return Time;
extern
    Clk : Clock;
end interface;
```

2.2.6 Predefined types

The interface types of the Rapide type system have been used to write the interfaces for all the predefined types supported by the executable language of the framework. These include several types with interesting semantics such as action and reference types (for details, see [KLM94]). The exercise of specifying these predefined types as interfaces has illustrated the expressiveness and flexibility of the type system. Section 5.1 describes in brief some of these interfaces. The detailed interfaces for the predefined types and type constructors are outlined in [RPr].

Chapter 3

Type-theoretic foundations for RAPIDE

The design of the RAPIDE framework was tightly coupled with the description of its type system, which in turn was consistently embedded in a concise theoretical framework. The existence of a simple underlying theory is important in that it strongly suggests the conceptual simplicity of the programming language based on this foundation. The framework also provides a useful medium for the design and analysis of type-checking algorithms for the programming language, including the investigation of issues such as expressiveness and efficiency of the algorithms. Another useful application of the underlying conceptual framework is that it provides a representation model for the implementation of the language.

In this chapter, we first present a typed calculus with its typing and subtyping rules, and we demonstrate the existence of sound type-checking algorithms. Finally, we show that all the "features" of the RAPIDE types language map onto the type system for this calculus.

3.1 The calculus

3.1.1 Overview

The basic primitive notions of the calculus are function types, record types, bounded universal and existential quantification. The calculus is an extension of the second-order polymorphic λ-calculus with record types, abstract types and subtyping, which allows one to capture the more general forms of polymorphism and abstraction described in Chapter 2.

We first describe the notation we shall follow in presenting the calculus.

In writing types and variables, we use the following symbols:

$$\begin{array}{rl}
\text{basic types}: & a \\
\text{type variables}: & s, t, \ldots \\
\text{types}: & S, S_1, S_2, \ldots, T, T_1, T_2 \ldots \\
\text{type constructors}: & F, G, \ldots \\
\text{term variables}: & x, y, \ldots \\
\text{terms}: & e, f, \ldots \\
\text{record labels}: & \ell
\end{array}$$

Typing environments are written as:

$$\begin{array}{rl}
\text{typing environment}: & \Gamma \\
\text{environment with assertion on term variables}: & \Gamma, x{:}T \\
\text{environment with assertion on type variables}: & \Gamma, s{<:}t \\
& \Gamma, t{<:}F(t) \\
& \Gamma, t = F(t)
\end{array}$$

To refer to the various types, and to rules pertaining to these types, we shall use the following:

$$\begin{array}{rl}
\text{function types}: & FUN \\
\text{record types}: & RCD \\
\text{polymorphic types}: & UNV \\
\text{abstract types}: & EXS \\
& \text{(with subscripts 1 and 2 for bounded and exact abstract types)}
\end{array}$$

In the rules of the calculus, we shall find it useful to refer to the sets of free type and term variables. For that purpose, we use

$$\begin{array}{rl}
\text{free variables in an expression}: & FV(...) \\
\text{free type variables in an expression}: & FTV(...)
\end{array}$$

Also, in referring to the rules of the calculus, we shall adopt the following naming conventions for these rules.

$$\begin{aligned}
\text{subtyping rules}: &\quad ST - ...\\
\text{typing rules}: &\quad TY - ...\\
\text{type equality rules}: &\quad TE - ...\\
\text{value equality rules}: &\quad VE - ...\\
\text{reduction rules}: &\quad RR - ...\\
\text{type introduction rules}: &\quad ... - I\\
\text{type elimination rules}: &\quad ... - E\\
\text{renaming rules}: &\quad ... - \alpha\\
\text{reduction (rewrite) rules}: &\quad ... - \beta\\
\text{extensionality rules}: &\quad ... - \eta
\end{aligned}$$

3.1.2 Types

The types of the calculus are summarized in Figure 3.1.

$$\begin{array}{lll}
T ::= & a & (BAS)\\
| & T_1 \rightarrow T_2 & (FUN)\\
| & \{\ell_1 : T_1, \ldots, \ell_n : T_n\} & (RCD)\\
| & \forall t <: F(t).T & (UNV)\\
| & \exists t <: F(t).T & (EXS_1)\\
| & \exists t = F(t).T & (EXS_2)\\
| & \texttt{Top} & (TOP)\\
| & \texttt{fst}(e) & (EXS{-}E_1)
\end{array}$$

Figure 3.1: Grammar for the types of the calculus

The types are as follows:

1. basic or primitive types.
2. function types of the form $T_1 \rightarrow T_2$.
3. record types where the type $\{\ell_1 : T_1, \ldots, \ell_n : T_n\}$ is the type of all records with fields $\ell_1, \ldots, \ell_n$ having types $T_1, \ldots, T_n$ respectively.

4. bounded universally quantified types of the form $\forall t <: F(t).T$. These types denote expressions that are polymorphic over all the subtypes of a given type. The bounds imposed on the polymorphic type variable can contain the variable itself.

5. bounded existentially quantified types of the form $\exists t <: F(t).T$. These types are abstract types, where bounds can be imposed on the type variable that is being abstracted. A special case is the form $\exists t = F(t).T$ which are the exact abstract types.

6. `Top`, which is the supertype of all types. As explained later, this type allows one to write unbounded polymorphic types and completely opaque abstract types as special cases of bounded polymorphic types and bounded abstract types respectively (by using `Top` as the bound in each case).

7. extracted types of the form $\mathtt{fst}(e)$. We need a way to refer to the type component of terms of abstract types. Since we shall view such terms as being pairs (the first component being the hidden type and the second the term over which the type is abstracted), we use $\mathtt{fst}(e)$ to denote the type component.

For simplicity, we omit the description of recursive types in this thesis.

3.1.3 Terms

Figure 3.2 gives the grammar for the expressions of the calculus.

Besides variables, the calculus has terms corresponding to the introduction and elimination of the complex types. Hence the expressions are as follows:

1. variables.

2. expressions of the form $\lambda x{:}T.e$ that create functions, and $e_1 e_2$ that apply them.

3. $\{\ell_1 = e_1, \ldots, \ell_n = e_n\}$, which denotes a record with fields $\ell_1, \ldots \ell_n$ that have values $e_1, \ldots e_n$ respectively, and $e.\ell$ which selects the value of the field ℓ.

4. bounded polymorphic terms of the form $\Lambda t <: F(t).e$, and the instantiation of such terms as expressions of the form eT.

5. terms of abstract types of the forms $\langle \exists t <: F(t) = T, e \rangle$, and $\langle \exists t = F(t) = T, e \rangle$, corresponding to bounded and exact abstract types, and terms of the form $\mathtt{snd}(e)$ that correspond to the extraction of the value component of terms of abstract types (remember that we view such terms as a type-value pair).

Note that we do not need any special expressions of type `Top`.

$$
\begin{array}{rll}
e ::= & x & (VAR) \\
| & \lambda x{:}T.e & (FUN\!-\!I) \\
| & e_1 e_2 & (FUN\!-\!E) \\
| & \{\ell_1 = e_1, \ldots, \ell_n = e_n\} & (RCD\!-\!I) \\
| & e.\ell & (RCD\!-\!E) \\
| & \Lambda t <: F(t).e & (UNV\!-\!I) \\
| & eT & (UNV\!-\!E) \\
| & \langle \exists t <: F(t) = T, e \rangle & (EXS_1\!-\!I) \\
| & \langle \exists t = F(t) = T, e \rangle & (EXS_2\!-\!I) \\
| & \mathtt{snd}(e) & (EXS\!-\!E_2)
\end{array}
$$

Figure 3.2: Terms of the calculus

3.1.4 Typing rules

We have a typing rule for each of the different forms of expressions and one for subsumption. Typing assertions are of the form

$$\Gamma \vdash e : T$$

which means that under the assumptions of typing environment Γ, expression e has type T.

A typing environment Γ is a set of assertions on term and type variables. Assertions of the form $x{:}T$ assert that the term variable x has type T, and those of the form $t{<:}X$ assert that the type variable t is a subtype of the type or type variable X. A typing environment can contain atmost one assertion about any term or type variable. We write "Γ, α" to conveniently indicate a typing environment that contains the assertion α and all the assertions of environment Γ.

The typing rules are:

1. Variables simply have the type that the environment assumes them to have.

$(TY\!-\!VAR)$ $$\Gamma, x{:}T \vdash x : T$$

2. Function abstractions behave as usual.

$(TY\!-\!FUN\!-\!I)$ $$\frac{\Gamma, x{:}S \vdash e : T}{\Gamma \vdash \lambda x{:}S.e : S \rightarrow T}$$

3. The rule for function application is also just the standard one.

$$(TY\!-\!FUN\!-\!E) \qquad \frac{\Gamma \vdash f : S \rightarrow T \qquad \Gamma \vdash e : S}{\Gamma \vdash fe : T}$$

4. The type of a record is simply the list of field names in that record along with the types of the values attached to those labels.

$$(TY\!-\!RCD\!-\!I) \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 {:} T_1, \ldots, \ell_n {:} T_n\}}$$

5. The type of a selected field is simply the type corresponding to that field name in the record type.

$$(TY\!-\!RCD\!-\!E) \qquad \frac{\Gamma \vdash e : \{\ell_1 {:} T_1, \ldots, \ell_n {:} T_n\}}{\Gamma \vdash e.\ell_i : T_i} \quad i \in \{1 \ldots n\}$$

6. Bounded polymorphic functions allow abstractions over types. Furthermore,one can impose a type bound on that variable.

$$(TY\!-\!UNV\!-\!I) \qquad \frac{\Gamma, t {<:} F(t) \vdash e : T}{\Gamma \vdash \Lambda t {<:} F(t).e : \forall t {<:} F(t).T}$$

7. Polymorphic functions can be instantiated with a particular type as long as the type satisfies the constraint imposed by the bound in the polymorphic type.

$$(TY\!-\!UNV\!-\!E) \qquad \frac{\Gamma \vdash e : \forall t {<:} F(t).T \qquad \Gamma \vdash S <: F(S)}{\Gamma \vdash eS : [S/t]T}$$

8. The type component of a term of a bounded abstract type must satisfy the constraint imposed by the bound on the hidden type. The value component must have the corresponding type, assuming the type instantiation of the hidden type.

$$(TY\!-\!EXS_1\!-\!I) \qquad \frac{\Gamma \vdash S <: F(S) \qquad \Gamma \vdash [S/t]e : [S/t]T}{\Gamma \vdash \langle \exists t {<:} F(t) = S, e \rangle : \exists t {<:} F(t).T}$$

9. The type of the value component of a term of an abstract type is simply its type with the hidden type variable instantiated to the type component of the term.

$$(TY\!-\!EXS_1\!-\!E_2) \qquad \frac{\Gamma \vdash e : \exists t {<:} F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

10. The rule for exact types is similar to that for bounded types except that here the type component must satisfy a stricter equality constraint.

$$(TY\!-\!EXS_2\!-\!I) \qquad \frac{\Gamma \vdash S = F(S) \qquad \Gamma \vdash [S/t]e : [S/t]T}{\Gamma \vdash \langle \exists t = F(t) = S, e \rangle : \exists t = F(t).T}$$

11. For the value component, the rule is the same as that for bounded abstract types.

$$(TY\!-\!EXS_2\!-\!E_2) \qquad \frac{\Gamma \vdash e : \exists t = F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

12. We also have a rule that expresses the property of subsumption, namely that any expression of type T also belongs to all the supertypes of T.

$$(TY\!-\!SUB) \qquad \frac{\Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e : T}$$

3.1.5 Subtyping rules

The subtyping relation has the properties of a pre-order, along with a subtyping rule for each of the types.

1. Subtyping is reflexive.

$$(ST\!-\!REF) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2}{\Gamma \vdash T_1 <: T_2}$$

2. The subtype relation is also transitive.

$$(ST\!-\!TRA) \qquad \frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

3. The only subtyping information we have about variables is that in the environment.

$$(ST\!-\!VAR_1) \qquad \Gamma, t{<:}T \vdash t <: T$$

$$(ST\!-\!VAR_2) \qquad \Gamma, t = T \vdash t <: T$$

4. Functions have the usual subtyping rule - contravariant in their domain types and covariant in the range types.

$$(ST\!-\!FUN) \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 <: S_1 \\ \Gamma \vdash S_2 <: T_2\end{array}}{\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

5. For a subtype of a record type, one can either add extra fields to a record type or specialize the types of existing fields.

$$(ST\!-\!RCD) \qquad \frac{\Gamma \vdash S_1 <: T_1 \qquad \ldots \qquad \Gamma \vdash S_n <: T_n}{\Gamma \vdash \{\ell_1 : S_1, \ldots, \ell_n : S_n, \ldots\} <: \{\ell_1 : T_1, \ldots, \ell_n : T_n\}}$$

6. The subtyping rules for bounded polymorphic types is rather subtle. We explain it using substitutivity of subtypes. Suppose a context expects an element of the type $\forall t{<:}G(t).T_2$. This means it needs a function that takes an argument type satisfying the constraint $t{<:}G(t)$, and returns a value of type T_2. If we instead substitute an element of the type $\forall t{<:}F(t).T_1$, we are supplying a function that takes a type satisfying $t{<:}F(t)$ and returns a value of type T_1. Now such a substitution will not cause any type errors if (1) any type argument that was originally plausible (*i.e.* any type t such that $t{<:}G(t)$) satisfies the constraint of the substituted element (*i.e.* $t{<:}F(t)$), and (2) for any plausible type argument (*i.e.* any type t such that $t{<:}G(t)$), the value returned by the substituted type (*i.e.* elements of T_1) are values that were originally acceptable (*i.e.* elements of T_2). Hence we have the following rule:

$$(ST\!-\!UNV) \qquad \frac{\begin{array}{c}\Gamma, t{<:}G(t) \vdash t <: F(t) \\ \Gamma, t{<:}G(t) \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

7. The subtyping rule for bounded abstract types is similar in appearance. In this case, some context expects an element of the type $\exists t{<:}G(t).T_2$, which means it needs a term with a value component of type T_2 and a hidden type component t that is a subtype of $G(t)$. If we instead substitute an element of the type $\exists t{<:}F(t).T_1$, we are supplying a term with a value component of type T_1 and a hidden type component t that is a subtype of $F(t)$. This substitution would be type-correct if under the assumptions of the substituted hidden type (namely, $t <: F(t)$), none of the original needs are violated. These are (1) the hidden type is a subtype of $G(t)$, and (2) the value component has type T_2. This gives rise to the following rule:

$$(ST\!-\!EXS_1) \qquad \frac{\begin{array}{c}\Gamma, t{<:}F(t) \vdash t <: G(t) \\ \Gamma, t{<:}F(t) \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \exists t{<:}F(t).T_1 <: \exists t{<:}G(t).T_2}$$

8. The subtyping rules for exact abstract types is similar in nature.

$$(ST\!-\!EXS_2) \qquad \frac{\begin{array}{c}\Gamma, t = F(t) \vdash t = G(t) \\ \Gamma, t = F(t) \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \exists t = F(t).T_1 <: \exists t = G(t).T_2}$$

Note that the first precedent is essentially requiring that every fixpoint of F is a fixpoint of G.

9. All types are subtypes of the type `Top`.

$(ST-TOP)$ $$\Gamma \vdash t <: \texttt{Top}$$

10. The type component of an element of a bounded abstract type always satisfy the subtyping constraint imposed by the abstract type on the hidden type component.

$(ST-EXS_1-E_1)$ $$\frac{\Gamma \vdash e : \exists t <: F(t).T}{\Gamma \vdash \texttt{fst}(e) <: F(\texttt{fst}(e))}$$

11. For an exact abstract type, the type component satisfies a two-way subtyping constraint.

$(ST-EXS_2-E_1)$ $$\frac{\Gamma \vdash e : \exists t = F(t).T}{\Gamma \vdash \texttt{fst}(e) <: F(\texttt{fst}(e)) \Gamma \vdash F(\texttt{fst}(e)) <: \texttt{fst}(e)}$$

3.2 Soundness

We now prove the soundness of the type system described in this chapter. The proof is based on the notion of syntactic soundness proposed in [WF94]. Briefly, the approach is based on *subject reduction* and on the use of *rewriting* to specify the operational semantics of the programming language. A modification of the Wright-Felleisen approach is that our proof structure is modular in the kinds of types in the type system. This will be explained in Section 3.2.3.

3.2.1 Defining type soundness

Before we present our soundness proof, let us take a look at the common definitions of type soundness. We present these in very informal language.

Weak Soundness : Weak soundness of a type system implies that if a program is well-typed, then its execution does not result in a type error. Weak soundness essentially states that the type system prevents type errors.

Strong Soundness : Strong soundness of a type system implies that if a program is well-typed and its execution terminates, then the result is a value of the same type. Thus, strong soundness gives guarantees not only about the execution but also the result of the program execution.

As the names suggest, strong soundness implies weak soundness.

There are two distinct approaches that can be taken to prove the soundness of a type system. In either, the goal is to prove something stronger, based on the framework used to reason about programs.

Semantic Soundness : This notion of soundness is based on the semantics of the language. Semantic soundness of a type system implies that if any program fragment has a certain type in a certain type context, then the semantic meaning of that fragment in any program context honoring the type context is a value with the same type.

For example, if one uses denotational semantics to provide a soundness proof, one would use the denotational meaning of the program fragment, whereas with operational semantics, one would use the reducibility of the program fragment.

Simply put, semantic soundness assures type correctness not only for the entire program but for all meaningful fragments as well.

Syntactic Soundness : This notion of soundness is based on the operational formulation of language semantics by *rewriting* (and not purely on the syntax, as the name might suggest!). Syntactic soundness of a type system implies firstly that if a program is well-typed, then all intermediate states in the evaluation of the program have the same type, and secondly that all irreducible expressions that are not acceptable values ("faulty expressions") are untypable.

Both semantic and syntactic soundness imply strong soundness.

We use the syntactic approach to soundness.

3.2.2 The soundness theorem

The soundness theorem that we prove is stated as:

Theorem 3.1 (Syntactic Soundness) *If $\vdash e : T$ then either e diverges or e evaluates to an irreducible value v such that $\vdash v : T$.*

We prove this as an obvious consequence of the following lemmas:

Lemma 3.2 (Subject Reduction) *If $\vdash e_1 : T$ and $e_1 \mapsto e_2$, then $\vdash e_2 : T$*

Lemma 3.3 (Faulty Expressions) *If e is faulty, then there is no T such that $\vdash e : T$*

3.2.3 The general approach

Our approach to proving type soundness uses certain observations on the ways values of a certain type are created and used. The proof style is based on the facts that for every kind of type in the type system, there are corresponding *introduction* and *elimination* rules, and that for each corresponding intro-elim pair, there is normally an associated rewrite rule. Also, faulty expressions are easily definable as constructed terms that do not reduce.

Let us look at a generic case now - some aspects may become clearer when we get to the first concrete case in the proof.

Consider some type K that is typical of any of the kinds of constructed types. For elements of this type, there would be a rule that determines under what circumstances this would be true. This rule shows how values of a certain type can be "constructed".

$$(TY\!-\!K\!-\!I) \qquad \frac{\alpha_I(\Gamma, e)}{\Gamma \vdash \mathcal{I}_K(e) : K}$$

Here, $\mathcal{I}_K$ denotes the structure of a constructed element of type K, and hence is a "term schema", while α_I represents the typability conditions on the subterms of such elements, and hence is a "type judgement schema".

Similarly, given elements of this type, there would be a rule determining how such an element can be used "destructed".

$$(TY\!-\!K\!-\!E) \qquad \frac{\begin{array}{c}\Gamma \vdash f : K \\ \alpha_E(\Gamma', f)\end{array}}{\Gamma' \vdash \mathcal{E}_K(f) : K'}$$

α_E is again a "term schema" for terms obtained by using terms of type K.

Expression reductions occur when a destructor is applied to a constructor of a term of compatible types.

$$(RR\!-\!K) \qquad \mathcal{E}_K(\mathcal{I}_K(e)) \longrightarrow e'$$

But one can, by combining the $(TY\!-\!K\!-\!I)$ and $(TY\!-\!K\!-\!E)$ rules, deduce the type for $\mathcal{E}_K(\mathcal{I}_K(e))$.

$$(TY\!-\!K\!-\!IE) \qquad \frac{\begin{array}{c}\Gamma \vdash \mathcal{I}_K(e) : K \\ \alpha_I(\Gamma, e) \\ \alpha_E(\Gamma', \mathcal{I}_K(e))\end{array}}{\Gamma' \vdash \mathcal{E}_K(\mathcal{I}_K(e)) : K'}$$

However, subject reduction requires that reductions preserve type. In other words, the reduct e' in $(RR\!-\!K)$ above should have the same type as $\mathcal{E}_K(\mathcal{I}_K(e))$. So this leads us to the following requirement, to prove subject reduction for the rewrite rule corresponding to types of this kind.

$$(K\!-\!Rule) \qquad \alpha_I(\Gamma, e) \ \& \ \alpha_E(\Gamma', \mathcal{I}_K(e)) \Longrightarrow \Gamma' \vdash e' : K'$$

This rule is fairly straightforward. $\alpha_I(\Gamma, e)$ are well-formedness conditions on the structure of subterms of values of the concerned type, while $\alpha_E(\Gamma', \mathcal{I}_K(e))$ are well-formedness conditions on the structure of terms of the concerned type. So, in essence, proving subject reduction for the type system boils down to one rule for each kind, and each rule is of the general form : "If one constructs a well-formed term of this type from well-formed subterms, then the reducts obtained by applying the reduction rule corresponding to this kind should be of a certain type".

Finally, faulty expressions are defined as all terms that contain a constructed (non-basic) subterm that does not reduce. We first identify the class of "answers" – the irreducible terms that are acceptable as final values. These are defined as:

$$\begin{aligned} \mathcal{A} = \{ \quad & x,\ \lambda x{:}T.e,\ \{\ell_1 = e_1, \ldots, \ell_n = e_n\}, \\ & \Lambda t <: F(t).e,\ \langle \exists t <: F(t) = T, e\rangle,\ \langle \exists t = F(t) = T, e\rangle \} \end{aligned}$$

Faulty expressions are now defined as:

$$(FE) \qquad \mathcal{F} = \{ f \mid \exists e \subseteq f \text{ such that } e \notin \mathcal{A} \text{ and } e \neq \mathcal{E}_K(\mathcal{I}_K(x)) \}$$

3.2.4 The proof

We now prove the two lemmas.

Subject Reduction

Let us look at the cases for the different types individually.

Function types: We do the first case in a bit more detail to better illustrate the general mechanism of this proof style.

We have seen the function type intro-elim rules:

$$(TY\!-\!FUN\!-\!I) \qquad \frac{\Gamma, x{:}S \vdash e : T}{\Gamma \vdash \lambda x{:}S.e : S \rightarrow T}$$

$$(TY\!-\!FUN\!-\!E) \qquad \frac{\Gamma \vdash f : S \rightarrow T \qquad \Gamma \vdash e : S}{\Gamma \vdash fe : T}$$

The reduction rule for function types is the application of a function abstraction to an argument.

$$(RR{-}FUN) \qquad (\lambda x{:}\, S.e)e' \longrightarrow [e'/x]e$$

Combining the rules gives us:

$$(TY{-}FUN{-}IE) \qquad \frac{\Gamma, x{:}\, S \vdash e : T \quad \Gamma \vdash e' : S}{\Gamma \vdash (\lambda x.e)e' : T}$$

Thus, proving subject reduction is reduced to showing the following:

$$(FUN{-}Rule) \qquad \Gamma, x{:}\, S \vdash e : T \ \&\ \Gamma \vdash e' : S \Longrightarrow \Gamma \vdash [e'/x]e : T$$

which is proven by induction on the derivation of the type if e. The proof is provided in Section 3.2.4.

Record types: For records, the reduction rule is

$$(RR{-}RCD) \qquad \{\ell_1 = e_1, \ldots, \ell_n = e_n\}.\ell_i \longrightarrow e_i$$

and subject reduction is reduced to proving:

$$(RCD{-}Rule) \qquad \Gamma \vdash e_1 : T_1 \ldots \Gamma \vdash e_n : T_n \Longrightarrow \Gamma \vdash e_i : T_i, \quad i \in \{1 \ldots n\}$$

which is trivially true.

Polymorphic types: The reduction rule is:

$$(RR{-}UNV) \qquad (\Lambda t{<:}F(t).e)S \longrightarrow [S/t]e, (if\, S <: F(S))$$

Subject reduction follows from:

$$(UNV{-}Rule) \qquad \Gamma, t{<:}F(t) \vdash e : T \ \&\ \Gamma \vdash S <: F(S) \Longrightarrow \Gamma \vdash [S/t]e : [S/t]T$$

This, again, is proved by induction on the derivation of the type of e. The proof is supplied in Section 3.2.4.

Abstract types: The reduction rule for bounded abstract types is:

$$(RR{-}EXS) \qquad \mathtt{snd}(\langle \exists t{<:}F(t) = S, e\rangle) \longrightarrow [S/t]e$$

Note that there are two "destructors" for abstract types. The other results in a type equality.

$$(TE{-}EXS) \qquad \texttt{fst}(\langle \exists t{<:}F(t) = S, e\rangle) = S$$

Subject reduction corresponds to:

$(EXS{-}Rule)$

$$\Gamma \vdash [S/t]e : [S/t]T \ \& \ \Gamma \vdash S <: F(S) \Longrightarrow \Gamma \vdash [S/t]e : [\texttt{fst}(\langle \exists t{<:}F(t) = S, e\rangle)/t]T$$

This follows trivially because of $(TE{-}EXS)$). The case for exact abstract types is identical.

It must be noted that we have made a simplifying assumption about the scopes of `fst`() and `snd`(). The details of this are discussed in Section 4.2.

Faulty Expressions

The property that faulty expressions are untypable is proved by showing the following two facts:

Fact 3.4 (Subterm typability of typable terms) $\vdash e : T \Rightarrow \forall f \subseteq e, \exists \Gamma, S$ such that $\Gamma \vdash f : S$

ie. All subterms of a typable term are typable under some context.

Proof: The proof follows from going through the list of all the typing rules (Section 3.1.4) and noticing that the precedents for each rule include the typability of all subterms of the term being typed in the consequence.

1. Variables have to be assigned types by the environment.

$$(TY{-}VAR) \qquad \Gamma, x{:}T \vdash x : T$$

2. The only subterm of an abstraction $\lambda x{:}S.e$ is e which, by the assumptions, is typable in the context of the abstraction along with a type assignment to the abstraction variable.

$$(TY{-}FUN{-}I) \qquad \frac{\Gamma, x{:}S \vdash e : T}{\Gamma \vdash \lambda x{:}S.e : S \rightarrow T}$$

3. Both subterms of a function application have to be typable.

$$(TY{-}FUN{-}E) \qquad \frac{\Gamma \vdash f : S \rightarrow T \qquad \Gamma \vdash e : S}{\Gamma \vdash fe : T}$$

4. The subterms of a record are the values of each of the fields, and the rule requires these to be typed by the type of the corresponding field in the record type.

$$(TY\!-\!RCD\!-\!I) \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : T_1, \ldots, \ell_n : T_n\}}$$

5. The subterm of a record selection is the record itself, which the selection rule requires to be typed as a record type.

$$(TY\!-\!RCD\!-\!E) \qquad \frac{\Gamma \vdash e : \{\ell_1 : T_1, \ldots, \ell_n : T_n\}}{\Gamma \vdash e.\ell_i : T_i} \quad i \in \{1 \ldots n\}$$

6. The subterm of a polymorphic function has to be typable in the environment of the function along with a subtyping constraint on the type variable.

$$(TY\!-\!UNV\!-\!I) \qquad \frac{\Gamma, t <: F(t) \vdash e : T}{\Gamma \vdash \Lambda t <: F(t).e : \forall t <: F(t).T}$$

7. The subterm of a polymorphic function application is the polymorphic function, which has to be typable as a polymorphic type.

$$(TY\!-\!UNV\!-\!E) \qquad \frac{\Gamma \vdash e : \forall t <: F(t).T \qquad \Gamma \vdash S <: F(S)}{\Gamma \vdash eS : [S/t]T}$$

8. The subterms of abstract terms are the value components of these terms, and these have to be typable (although with a type substitution).

$$(TY\!-\!EXS_1\!-\!I) \qquad \frac{\Gamma \vdash S <: F(S) \qquad \Gamma \vdash [S/t]e : [S/t]T}{\Gamma \vdash \langle \exists t <: F(t) = S, e \rangle : \exists t <: F(t).T}$$

$$(TY\!-\!EXS_2\!-\!I) \qquad \frac{\Gamma \vdash S = F(S) \qquad \Gamma \vdash [S/t]e : [S/t]T}{\Gamma \vdash \langle \exists t = F(t) = S, e \rangle : \exists t = F(t).T}$$

9. One can select the value component of an abstract term if that abstract term, which is the only subterm of the selection, has an abstract type.

$$(TY\!-\!EXS_1\!-\!E_2) \qquad \frac{\Gamma \vdash e : \exists t <: F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

$$(TY\!-\!EXS_2\!-\!E_2) \qquad \frac{\Gamma \vdash e : \exists t = F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

Fact 3.5 (Subterm untypability of faulty terms) $e \in \mathcal{F} \Rightarrow \exists f \subseteq e$ such that for no $\Gamma, S, \Gamma \vdash f : S$

ie. Every faulty term has a subterm that is untypable under any context.

Proof: We know that every faulty term has, by definition, a subterm that is neither an answer nor is it of the form of a type destructor applied to a constructor of that same type. Now if this subterm is typable, then since it is not an answer, it must be typable by one of the following rules:

$$(TY\!-\!FUN\!-\!E) \qquad \frac{\Gamma \vdash f : S \rightarrow T \qquad \Gamma \vdash e : S}{\Gamma \vdash fe : T}$$

$$(TY\!-\!RCD\!-\!E) \qquad \frac{\Gamma \vdash e : \{\ell_1 {:} T_1, \ldots, \ell_n {:} T_n\}}{\Gamma \vdash e.\ell_i : T_i} \quad i \in \{1 \ldots n\}$$

$$(TY\!-\!UNV\!-\!E) \qquad \frac{\Gamma \vdash e : \forall t {<:} F(t).T \qquad \Gamma \vdash S <: F(S)}{\Gamma \vdash eS : [S/t]T}$$

$$(TY\!-\!EXS_1\!-\!E_2) \qquad \frac{\Gamma \vdash e : \exists t {<:} F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

$$(TY\!-\!EXS_2\!-\!E_2) \qquad \frac{\Gamma \vdash e : \exists t = F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

These are simply the typing rules for all non-answer terms. But as is obvious, each of these rules requires that the term being typed be of the form $\mathcal{E}_K(\mathcal{I}_K(x))$, which we know our subterm is not. Hence, the subterm is untypable.

Proof of subject reduction for function types

Lemma 3.6 (Subject Reduction for Function Types)

$$\Gamma, x{:}\, S \vdash e : T \ \&\ \Gamma \vdash e' : S \Longrightarrow \Gamma \vdash [e'/x]e : T$$

Proof: The proof is by induction on the derivation of the type of e.

The base case, for single-step derivations is when e is a variable, say y. There are two cases here.

1. $\underline{y \neq x.}$ In this case, we need to show

$$\Gamma, x{:}\, S \vdash y : T \Longrightarrow \Gamma \vdash y : T$$

   which is trivially true.

2. $\underline{y = x.}$ In this case, we need to show

$$\Gamma, x{:}\, S \vdash x : T \;\&\; \Gamma \vdash e' : S \Longrightarrow \Gamma \vdash e' : T$$

   Now, $\Gamma, x{:}\, S \vdash x : T$ is true only if $S <: T$. But if $S <: T$, then $\Gamma \vdash e' : S \Longrightarrow \Gamma \vdash e' : T$.

For the induction, we show that if we assume the hypothesis to be true for all derivations of length less than that of the derivation of e, we can prove the hypothesis for e. In particular, remember that the derivations of all subterms of e are smaller than the derivation for e.

There are several cases for the last step of the derivation of e, depending on the structure of e.

1. $\underline{e = \lambda y{:}\, T_1.f}$

   Note that we choose the variable y such that it does not occur in Γ. Also, T must be of the form $T_1 \rightarrow T_2$.

   $\Gamma, x{:}\, S \vdash e : T \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}\, S \vdash \lambda y{:}\, T_1.f : T_1 \rightarrow T_2 \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}\, S, y{:}\, T_1 \vdash f : T_2 \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}\, S, y{:}\, T_1 \vdash f : T_2 \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow \Gamma', x{:}\, S \vdash f : T_2 \;\;\&\;\; \Gamma \vdash e' : S$
   (where $\Gamma' = \Gamma, y{:}\, T_1$)
   $\Rightarrow \Gamma', x{:}\, S \vdash f : T_2 \;\;\&\;\; \Gamma' \vdash e' : S$
   (since y doesn't occur in e')
   $\Rightarrow \Gamma' \vdash [e'/x]f : T_2$
   (by the induction hypothesis)
   $\Rightarrow \Gamma, y{:}\, T_1 \vdash [e'/x]f : T_2$
   $\Rightarrow \Gamma \vdash \lambda y{:}\, T_1.[e'/x]f : T_1 \rightarrow T_2$
   $\Rightarrow \Gamma \vdash [e'/x](\lambda y{:}\, T_1.f) : T$
   $\Rightarrow \Gamma \vdash [e'/x]e : T$

2. $\underline{e = fg}$

   $\Gamma, x{:}\, S \vdash e : T \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}\, S \vdash fg : T \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}\, S \vdash f : T' \rightarrow T \;\;\&\;\; \Gamma, x{:}\, S \vdash g : T' \;\;\&\;\; \Gamma \vdash e' : S$
   $\Rightarrow (\Gamma, x{:}\, S \vdash f : T' \rightarrow T \;\;\&\;\; \Gamma \vdash e' : S) \;\;\&\;\; (\Gamma, x{:}\, S \vdash g : T' \;\;\&\;\; \Gamma \vdash e' : S)$
   $\Rightarrow \Gamma \vdash [e'/x]f : T' \rightarrow T \;\;\&\;\; \Gamma \vdash [e'/x]g : T'$
   (by two uses of the induction hypothesis)

$\Rightarrow \Gamma \vdash [e'/x]f.[e'/x]g : T$

$\Rightarrow \Gamma \vdash [e'/x](f.g) : T$

$\Rightarrow \Gamma \vdash [e'/x]e : T$

3. $\underline{e = \{\ell_1 = e_1, \ldots, \ell_n = e_n\}}$

   T must be of the form $\{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\}$

   $\Gamma, x{:}S \vdash e : T \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\} \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S \vdash e_1 : T_1 \ldots \Gamma, x{:}S \vdash e_n : T_n \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma \vdash [e'/x]e_1 : T_1 \ldots \Gamma \vdash [e'/x]e_n : T_n$

   (using the induction hypothesis for $e_1 \ldots e_n$)

   $\Rightarrow \Gamma \vdash \{\ell_1 = [e'/x]e_1, \ldots, \ell_n = [e'/x]e_n\} : \{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\}$

   $\Rightarrow \Gamma \vdash [e'/x]\{\ell_1 = e_1, \ldots, \ell_n = e_n\} : T$

   $\Rightarrow \Gamma \vdash [e'/x]e : T$

4. $\underline{e = f.\ell}$

   $\Gamma, x{:}S \vdash e : T \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S \vdash f.\ell : T \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S \vdash f : \{\ldots \ell{:}T \ldots\} \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma \vdash [e'/x]f : \{\ldots \ell{:}T \ldots\}$

   (by the induction hypothesis)

   $\Rightarrow \Gamma \vdash [e'/x]f.\ell : T$

   $\Rightarrow \Gamma \vdash [e'/x]e : T$

5. $\underline{e = \Lambda t{<:}F(t).f.}$

   T must be of the form $\forall t{<:}F(t).T'$.

   $\Gamma, x{:}S \vdash e : T \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S \vdash \Lambda t <: F(t).f : \forall t{<:}F(t).T' \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S, t{<:}F(t) \vdash f : T' \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma', x{:}S \vdash f : T' \;\;\&\;\; \Gamma \vdash e' : S$

   (where $\Gamma' = \Gamma, t{<:}F(t)$)

   $\Rightarrow \Gamma' \vdash [e'/x]f : T'$

   (by the induction hypothesis)

   $\Rightarrow \Gamma, t{<:}F(t) \vdash [e'/x]f : T'$

   $\Rightarrow \Gamma \vdash \Lambda t <: F(t).[e'/x]f : \forall t{<:}F(t).T'$

   $\Rightarrow \Gamma \vdash [e'/x](\Lambda t <: F(t).f) : T$

   $\Rightarrow \Gamma \vdash [e'/x]e : T$

6. $\underline{e = fT'}$

   $\Gamma, x{:}S \vdash e : T \;\;\&\;\; \Gamma \vdash e' : S$

   $\Rightarrow \Gamma, x{:}S \vdash fT' : T \;\;\&\;\; \Gamma \vdash e' : S$

$\Rightarrow \Gamma, x{:}S \vdash f : \forall t{<:}F(t).S'$ & $\Gamma \vdash T' <: F(T')$ & $\Gamma \vdash e' : S$ (where $T = [T'/t]S'$)
$\Rightarrow \Gamma \vdash [e'/x]f : \forall t{<:}F(t).S'$ & $\Gamma \vdash T' <: F(T')$
(by the induction hypothesis)
$\Rightarrow \Gamma \vdash ([e'/x]f)T' : [T'/t]S'$
$\Rightarrow \Gamma \vdash [e'/x](fT') : T$
$\Rightarrow \Gamma \vdash [e'/x]e : T$

7. $\underline{e = \langle \exists t <: F(t) = T', f \rangle.}$

   T must be of the form $\exists t{<:}F(t).T'$.

   $\Gamma, x{:}S \vdash e : T$ & $\Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}S \vdash \langle \exists t <: F(t) = S', f \rangle : \exists t{<:}F(t).T'$ & $\Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}S \vdash [S'/t]f : [S'/t]T'$ & $\Gamma \vdash S' <: F(S')$ & $\Gamma \vdash e' : S$
   $\Rightarrow \Gamma \vdash [e'/x]([S'/t]f) : [S'/t]T'$ & $\Gamma \vdash S' <: F(S')$
   (by the induction hypothesis)
   $\Rightarrow \Gamma \vdash [S'/t]([e'/x]f) : [S'/t]T'$ & $\Gamma \vdash S' <: F(S')$
   $\Rightarrow \Gamma \vdash [S'/t]([e'/x]f) : [S'/t]T'$ & $\Gamma \vdash S' <: F(S')$
   $\Rightarrow \Gamma \vdash \langle \exists t <: F(t) = S', [e'/x]f \rangle : \exists t{<:}F(t).T'$
   $\Rightarrow \Gamma \vdash [e'/x](\langle \exists t <: F(t) = S', f \rangle) : T$
   $\Rightarrow \Gamma \vdash [e'/x]e : T$

8. $\underline{e = \langle \exists t = F(t) = T', f \rangle}$

   This case is identical to the previous one.

9. $\underline{e = \mathtt{snd}(f)}$

   $\Gamma, x{:}S \vdash e : T$ & $\Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}S \vdash \mathtt{snd}(f) : T$ & $\Gamma \vdash e' : S$
   $\Rightarrow \Gamma, x{:}S \vdash f : \exists t{<:}F(t).T'$ & $\Gamma \vdash e' : S$
   (where $[\mathtt{fst}(f)/t]T' = T$)
   $\Rightarrow \Gamma \vdash [e'/x]f : \exists t{<:}F(t).T'$
   (by the induction hypothesis)
   $\Rightarrow \Gamma \vdash \mathtt{snd}([e'/x]f) : [\mathtt{fst}(f)/t]T'$
   $\Rightarrow \Gamma \vdash [e'/x](\mathtt{snd}(f)) : T$
   $\Rightarrow \Gamma \vdash [e'/x]e : T$

Proof of subject reduction for polymorphic types

Lemma 3.7 (Subject Reduction for Polymorphic Types)

$$\Gamma, t{<:}F(t) \vdash e : T \;\;\&\;\; \Gamma \vdash S <: F(S) \Longrightarrow \Gamma \vdash [S/t]e : [S/t]T$$

Proof: This proof also proceeds by induction on the derivation of e, and is very similar to the above. The base case for variables follows trivially from the subsumption rule. We look at the various induction cases depending on the structure of e.

1. $\underline{e = \lambda y{:}T_1.f}$

   T must be of the form $T_1 \rightarrow T_2$.

   $\Gamma, t{<:}F(t) \vdash e : T \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t) \vdash \lambda y{:}T_1.f : T_1 \rightarrow T_2 \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t), y{:}T_1 \vdash f : T_2 \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t), y{:}T_1 \vdash f : T_2 \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma', t{<:}F(t) \vdash f : T_2 \ \ \& \ \ \Gamma \vdash S <: F(S)$

   (where $\Gamma' = \Gamma, y{:}T_1$)

   $\Rightarrow \Gamma', t{<:}F(t) \vdash f : T_2 \ \ \& \ \ \Gamma' \vdash S <: F(S)$

   (since y doesn't occur in $S, F(S)$)

   $\Rightarrow \Gamma' \vdash [S/t]f : [S/t]T_2$

   (by the induction hypothesis)

   $\Rightarrow \Gamma, y{:}T_1 \vdash [S/t]f : [S/t]T_2$

   $\Rightarrow \Gamma \vdash \lambda y{:}T_1.[S/t]f : T_1 \rightarrow [S/t]T_2$

   $\Rightarrow \Gamma \vdash \lambda y{:}T_1.[S/t]f : [S/t]T$

   (since y does not occur in T_1)

   $\Rightarrow \Gamma \vdash [S/t](\lambda y{:}T_1.f) : [S/t]T$

   $\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

2. $\underline{e = fg}$

   $\Gamma, t{<:}F(t) \vdash e : T \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t) \vdash fg : T \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t) \vdash f : T' \rightarrow T \ \ \& \ \ \Gamma, t{<:}F(t) \vdash g : T' \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow (\Gamma, t{<:}F(t) \vdash f : T' \rightarrow T \ \ \& \ \ \Gamma \vdash S <: F(S)) \ \ \& \ \ (\Gamma, t{<:}F(t) \vdash g : T' \ \ \& \ \ \Gamma \vdash S <: F(S))$

   $\Rightarrow \Gamma \vdash [S/t]f : [S/t](T' \rightarrow T) \ \ \& \ \ \Gamma \vdash [S/t]g : [S/t]T'$

   (by two uses of the induction hypothesis)

   $\Rightarrow \Gamma \vdash [S/t]f : [S/t]T' \rightarrow [S/t]T \ \ \& \ \ \Gamma \vdash [S/t]g : [S/t]T'$

   $\Rightarrow \Gamma \vdash [S/t]f.[S/t]g : [S/t]T$

   $\Rightarrow \Gamma \vdash [S/t](f.g) : [S/t]T$

   $\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

3. $\underline{e = \{\ell_1 = e_1, \ldots, \ell_n = e_n\}}$

   T must be of the form $\{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\}$

   $\Gamma, t{<:}F(t) \vdash e : T \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t) \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\} \ \ \& \ \ \Gamma \vdash S <: F(S)$

   $\Rightarrow \Gamma, t{<:}F(t) \vdash e_1 : T_1 \ldots \Gamma, t{<:}F(t) \vdash e_n : T_n \ \ \& \ \ \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma \vdash [S/t]e_1 : [S/t]T_1 \ldots \Gamma \vdash [S/t]e_n : [S/t]T_n$

(using the induction hypothesis for $e_1 \ldots e_n$)

$\Rightarrow \Gamma \vdash \{\ell_1 = [S/t]e_1, \ldots, \ell_n = [S/t]e_n\} : \{\ell_1 \colon [S/t]T_1, \ldots, \ell_n \colon [S/t]T_n\}$

$\Rightarrow \Gamma \vdash [S/t]\{\ell_1 = e_1, \ldots, \ell_n = e_n\} : [S/t]\{\ell_1 \colon T_1, \ldots, \ell_n \colon T_n\}$

$\Rightarrow \Gamma \vdash [S/t]e : T$

4. $\underline{e = f.\ell}$

$\Gamma, t{<:}F(t) \vdash e : T \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma, t{<:}F(t) \vdash f.\ell : T \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma, t{<:}F(t) \vdash f : \{\ldots \ell \colon T \ldots\} \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma \vdash [S/t]f : [S/t]\{\ldots \ell \colon T \ldots\}$

(by the induction hypothesis)

$\Rightarrow \Gamma \vdash [S/t]f : \{\ldots \ell \colon [S/t]T \ldots\}$

$\Rightarrow \Gamma \vdash [S/t]f.\ell : [S/t]T$

$\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

5. $\underline{e = \Lambda t'{<:}G(t').f.}$

T must be of the form $\forall t'{<:}G(t').T'$.

$\Gamma, t{<:}F(t) \vdash e : T \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma, t{<:}F(t) \vdash \Lambda t' <: G(t').f : \forall t'{<:}G(t').T' \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma, t{<:}F(t), t' <: G(t') \vdash f : T' \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma', t{<:}F(t) \vdash f : T' \;\;\&\;\; \Gamma \vdash S <: F(S)$

(where $\Gamma' = \Gamma, t'{<:}[S/t]G(t')$)

$\Rightarrow \Gamma' \vdash [S/t]f : [S/t]T'$

(by the induction hypothesis)

$\Rightarrow \Gamma, t'{<:}[S/t]G(t') \vdash [S/t]f : [S/t]T'$

$\Rightarrow \Gamma \vdash \Lambda t' <: [S/t]G(t').[S/t]f : \forall t'{<:}[S/t]G(t').[S/t]T'$

$\Rightarrow \Gamma \vdash [S/t](\Lambda t' <: G(t').f) : [S/t](\forall t'{<:}G(t').T')$

$\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

6. $\underline{e = fT'}$

$\Gamma, t{<:}F(t) \vdash e : T \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma, t{<:}F(t) \vdash fT' : T \;\;\&\;\; \Gamma \vdash S <: F(S)$

$\Rightarrow \Gamma, t{<:}F(t) \vdash f : \forall t'{<:}G(t').S' \;\;\&\;\; \Gamma, t{<:}F(t) \vdash T' <: G(T') \;\;\&\;\; \Gamma \vdash S <: F(S)$

(where $T = [T'/t]S'$)

$\Rightarrow \Gamma \vdash [S/t]f : [S/t](\forall t'{<:}G(t').S') \;\;\&\;\; \Gamma, t{<:}F(t) \vdash T' <: G(T')$

(by the induction hypothesis)

$\Rightarrow \Gamma \vdash [S/t]f : \forall t'{<:}[S/t]G(t').[S/t]S' \;\;\&\;\; \Gamma, t{<:}F(t) \vdash T' <: G(T')$

$\Rightarrow \Gamma \vdash [S/t]f : \forall t'{<:}[S/t]G(t').[S/t]S' \;\;\&\;\; \Gamma \vdash [S/t]T' <: [S/t]G(T')$

$\Rightarrow \Gamma \vdash [S/t]f : \forall t'{<:}[S/t]G(t').[S/t]S' \;\;\&\;\; \Gamma \vdash [S/t]T' <: [S/t]G([S/t]T')$

$\Rightarrow \Gamma \vdash ([S/t]f)([S/t]T') : [[S/t]T'/t]S'$
(by the induction hypothesis again)
$\Rightarrow \Gamma \vdash [S/t](fT') : [S/t]T$
$\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

7. $\underline{e = \langle \exists t' <: G(t') = T', f \rangle.}$

T must be of the form $\exists t' {<:} G(t').T'$.

$\Gamma, t{<:}F(t) \vdash e : T \;\; \& \;\; \Gamma \vdash S <: F(S)$
$\Rightarrow \Gamma, t{<:}F(t) \vdash \langle \exists t' <: G(t') = S', f \rangle : \exists t' {<:} G(t').T' \;\; \& \;\; \Gamma \vdash S <: F(S)$
$\Rightarrow \Gamma, t{<:}F(t) \vdash [S'/t']f : [S'/t']T' \;\; \& \;\; \Gamma, t{<:}F(t) \vdash S' <: G(S') \;\; \& \;\; \Gamma \vdash S <: F(S)$
$\Rightarrow \Gamma \vdash [S/t]([S'/t']f) : [S/t][S'/t']T' \;\; \& \;\; \Gamma, t{<:}F(t) \vdash S' <: G(S')$
(by the induction hypothesis)
$\Rightarrow \Gamma \vdash [[S/t]S'/t']([S/t]f) : [[S/t]S'/t'][S/t]T' \;\; \& \;\; \Gamma \vdash [S/t]S' <: [S/t]G(S')$
$\Rightarrow \Gamma \vdash [[S/t]S'/t']([S/t]f) : [[S/t]S'/t'][S/t]T' \;\; \& \;\; \Gamma \vdash [S/t]S' <: [S/t]G([S/t]S')$
$\Rightarrow \Gamma \vdash \langle \exists t' <: [S/t]G(t') = [S/t]S', [S/t]f \rangle : \exists t' {<:} [S/t]G(t').[S/t]T'$
(by the induction hypothesis again)
$\Rightarrow \Gamma \vdash [S/t](\langle \exists t' <: G(t') = S', f \rangle) : [S/t](\exists t' {<:} G(t').T')$
$\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

8. $\underline{e = \langle \exists t = F(t) = T', f \rangle}$

This case is identical to the previous one.

9. $\underline{e = \mathtt{snd}(f)}$

$\Gamma, t{<:}F(t) \vdash e : T \;\; \& \;\; \Gamma \vdash S <: F(S)$
$\Rightarrow \Gamma, t{<:}F(t) \vdash \mathtt{snd}(f) : T \;\; \& \;\; \Gamma \vdash S <: F(S)$
$\Rightarrow \Gamma, t{<:}F(t) \vdash f : \exists t' {<:} G(t').T' \;\; \& \;\; \Gamma \vdash S <: F(S)$
(where $[\mathtt{fst}(f)/t']T' = T$)
$\Rightarrow \Gamma \vdash [S/t]f : [S/t](\exists t' {<:} G(t').T')$
(by the induction hypothesis)
$\Rightarrow \Gamma \vdash [S/t]f : \exists t' {<:} [S/t]G(t').[S/t]T'$
$\Rightarrow \Gamma \vdash \mathtt{snd}([S/t]f) : [\mathtt{fst}([S/t]f)/t'][S/t]T'$
$\Rightarrow \Gamma \vdash [S/t](\mathtt{snd}(f)) : [S/t]([\mathtt{fst}(f)/t']T')$
$\Rightarrow \Gamma \vdash [S/t]e : [S/t]T$

3.3 Expressiveness

In this section, we show how the theoretical framework used in this chapter captures the notions of Rapide as described in Chapter 2.

Figure 3.3 provides a top-level summary of this mapping. By simple interfaces and function types, we mean those not involving the polymorphism and abstraction mechanisms.

RAPIDE	*type-theoretic construct*
simple interfaces	record types
simple function types	function types
polymorphism	bounded universal quantification
abstraction	bounded existential quantification

Figure 3.3: Mapping of RAPIDE into a typed calculus

We look at how the various examples of Chapter 2 can be expressed in the above framework.

- The point interface

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

  is expressed as

  $$\text{Point} = \{\ \text{X\_Val} : \text{Integer}, \ \text{Y\_Val} : \text{Integer}, \ \text{Distance} : \text{Point} \rightarrow \text{Integer}\ \}$$

- The Stack package

  type Int_Stack_Package **is** **interface**
  type Stack;
  New : **function**() **return** Stack;
  Push : **function**(X : Integer, S : Stack) **return** Stack;
  Pop : **function**(S : Stack) **return** Stack;
  Top : **function**(S : Stack) **return** Integer;
  end **interface**;

  is expressed as

  $$\text{Int\_Stack\_Package} = \exists\ \text{Stack}.\ \{\ \text{New} : \text{Stack}, \ \text{Push} : \text{Integer}\ X\ \text{Stack} \rightarrow \text{Stack}, \ \text{Pop} : \text{Stack} \rightarrow \text{Stack}, \ \text{Top} : \text{Stack} \rightarrow \text{Integer}\ \}$$

- Set packages with equality on elements, defined by

```
type Equality(type Elem) is interface
    Eq : function(E : Elem) return Boolean;
end interface;

type SetType(type Elem <: Equality(Elem)) is interface
    Insert : function(E : Elem);
    Delete : function(E : Elem);
    Union : function(S : SetType(Elem));
    Is_Member : function(E : Elem) return Boolean;
end interface;
```

are expressed as

$$
\begin{aligned}
\text{Equality} &= \forall\ \text{Elem}.\ \{\ \text{Eq} : \text{Elem} \rightarrow \text{Boolean}\ \} \\
\text{SetType} &= \forall\ \text{Elem} <: \text{Equality(Elem)}. \\
&\qquad \{\ \text{Insert} : \text{Elem} \rightarrow \text{Top}, \\
&\qquad\ \ \text{Delete} : \text{Elem} \rightarrow \text{Top}, \\
&\qquad\ \ \text{Union} : \text{SetType(Elem)} \rightarrow \text{Top}, \\
&\qquad\ \ \text{Is\_Member} : \text{Elem} \rightarrow \text{Boolean}\ \}
\end{aligned}
$$

- The Clock type

```
type Clock_Spec is interface
  type Clock <: interface
                  tick:function();
                  read:function() return integer;
                end interface;
  new_clock : function() return Clock;
end interface;
```

is expressed as

$$
\begin{aligned}
\text{Clock\_Spec} = \exists\ \text{Clock} <: \{\ \text{tick} : \text{Top},\ \text{read} : \text{Integer}\ \}. \\
\{\ \text{new\_clock} : \text{Clock}\ \}
\end{aligned}
$$

- The Stack package implementation extension

  type Stack_A_Ext **is interface**
  type Stack = A.Stack;
  PushList : **function**(L : List(Integer), S : Stack) **return** Stack;
  end interface;

  is expressed as

  $$\text{Stack\_A\_Ext} = \exists\ \text{Stack} = \texttt{fst}(\text{A}) \ \{\ \text{PushList} : \text{List(Integer)}\ X\ \text{Stack} \rightarrow \text{Stack}\ \}$$

  Note that multiple packaged types have to be ordered and the appropriate combination of `snd`() and `fst`() operations have to be applied to extract the right type. In the above example, since Stack was the only type in the Int_Stack_Package type, we can use `fst`(A) to denote A.Stack.

- The Reverse function

  Reverse : **function**(**type** T, L : List(T))
  return List(T)

  is expressed as

  $$\text{Reverse} : \forall\ \text{T}.\ (\ \text{List(T)} \rightarrow \text{List(T)}\ )$$

- The Move function for windows

  Wd_Move : **function**(**type** W <: Window, x:W)
  return W;

  is expressed as

  $$\text{Wd\_Move} : \forall\ \text{W} <: \text{Window}.\ (\ \text{W} \rightarrow \text{W}\ )$$

- The Sort function

  Sort : **function**(**type** T <: Ordered(T);
  L : List(T))
  return List(T);

  is expressed as

  $$\text{Sort} : \forall\ \text{T} <: \text{Ordered(T)}.\ (\ \text{List(T)} \rightarrow \text{List(T)}\ )$$

Note:

- An encoding for private interfaces is shown in Section 4.3.

- External interfaces, at the type level, can be encoded by parameterizing the external declarations.

- There is no encoding for inheritance, since inheritance in Rapide is simply pre-compilation shorthand.

Chapter 4

A closer look at the type system

This section looks at some of the theoretical details of the RAPIDE type system.

4.1 Polymorphism

The typing rules for bounded polymorphism are the focus of most of our theoretical investigation. The study of the calculus, in this case, allows us to explore expressiveness and efficiency issues related to the type checking of RAPIDE.

4.1.1 Decidability for bounded quantification

The subtyping rule for simple bounded polymorphic types has generated a fair amount of controversy, and has a long history of research.

Cardelli-Wegner : Cardelli and Wegner [CW85], in their original `FUN` language, had chosen the following rule:

$$(\mathit{bd\_poly\_fun}) \qquad \frac{\Gamma, t{<:}F \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F.T_1 <: \forall t{<:}F.T_2}$$

In other words, the bounds on the two types had to be identical. This syntactic restriction trivially yielded a decision procedure for type checking, but had no natural semantic motivation, and was also fairly restrictive.

Curien-Ghelli : [CG91] went on to propose the following rule in their formulation of bounded quantification.

$$(bd\_poly\_orig) \qquad \frac{\Gamma \vdash G <: F \qquad \Gamma, t{<:}G \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F.T_1 <: \forall t{<:}G.T_2}$$

Based on this rule, a subtyping algorithm was developed and proved partially correct, which meant that if the algorithm terminated, it did so correctly. Subsequently, Ghelli [Ghe90] presented a termination proof for this algorithm. However, a mistake was discovered in this termination proof, following which Pierce [Pie92] presented a proof showing that the subtyping problem for bounded polymorphic types is undecidable.

Katiyar-Sankar : In [KS92], we investigated the causes for the undecidability of subtyping and proposed a restriction that makes the subtype checking problem decidable. Our restriction of polymorphism, called *completely bounded quantification*, uses the original subtyping rule (*bd_poly_orig*), but requires all bounds of polymorphic types to be structurally completely specified - this is achieved by not permitting the type *Top* in any such bound. We also show how this restriction can be relaxed so as to regain the expressiveness of unbounded quantification. The details of this restriction and the proof of its decidability is presented in Appendix A, and the interested reader is referred to that section.

Ghelli : One of the factors that contributes to the divergence of subtype checking with the (*bd_poly_orig*) rule is the use of the stricter type bound in the typing environment of the second premise. A plausible variant of the rule uses the less stricter bound instead.

$$(bd\_poly\_local) \qquad \frac{\Gamma \vdash G <: F \qquad \Gamma, t{<:}F \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F.T_1 <: \forall t{<:}G.T_2}$$

While the decidability of the subtyping problem associated with this rule remains an open problem, Ghelli[Ghe93] demonstrated that this rule caused the transitivity of the subtyping relation to no longer hold true, making the behavior of subtyping unintuitive.

Castagna-Pierce : More recently, Castagna and Pierce [GP94] study yet another variant of the subtyping rule for bounded polymorphic types.

$$(bd\_poly\_orig) \qquad \frac{\Gamma \vdash G <: F \qquad \Gamma, t{<:}Top \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F.T_1 <: \forall t{<:}G.T_2}$$

This rule is easily shown to make subtype checking decidable, while enjoying several nice syntactic properties. But more interestingly, this weaker rule seems to suffice for all practical situations that bounded quantification has been used for.

4.1.2 Subtyping for F-bounded quantification

In this section, plausible subtyping rules for F-bounded quantification are developed and compared for their expressiveness.

As mentioned above, in the standard formulation of simple bounded quantification, the following rule was proposed.

$$(\mathit{bd\_poly\_orig}) \qquad \frac{\Gamma \vdash G <: F \qquad \Gamma, t{<:}G \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F.T_1 <: \forall t{<:}G.T_2}$$

Given this rule for the simple case, a first guess for the rule for F-bounded polymorphic types might be:

$$(\mathit{fbd\_poly\_naive}) \qquad \frac{\Gamma \vdash G(t) <: F(t) \qquad \Gamma, t{<:}G(t) \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

However, the first premise for this rule is stronger than necessary, since it requires that $G(t) <: F(t)$ for all types t, while we know that we need this to hold only when $t <: G(t)$. This can be demonstrated by the following subtyping problem:

$$\Phi \vdash \forall t{<:}\alpha{\rightarrow}(\alpha{\rightarrow}t).T <: \forall t{<:}\alpha{\rightarrow}t.T$$

The subproblem arising from the first premise (the second is trivially true), that is,

$$\Phi \vdash \alpha{\rightarrow}t <: \alpha{\rightarrow}(\alpha{\rightarrow}t)$$

is provable only if we assume that $t <: \alpha{\rightarrow}t$.

A better subtyping rule seems to be:

$$(\mathit{fbd\_poly\_ok}) \qquad \frac{\Gamma, t{<:}G(t) \vdash G(t) <: F(t) \qquad \Gamma, t{<:}G(t) \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

On the other hand, if one attempts to deduce the subtyping rule from first principles, the results are a bit different. Suppose some context expects an instance of the type $\forall t{<:}G(t).T_2$, which means it needs a function that takes an argument type satisfying the constraint $t{<:}G(t)$, and returns a value of type T_2. If we instead substitute an instance of the type $\forall t{<:}F(t).T_1$, we are supplying a function that takes a type satisfying $t{<:}F(t)$ and returns a value of type T_1. Now such a substitution will not cause any type errors if

- any type argument that was originally plausible (*i.e.* any type t such that $t{<:}G(t)$) satisfies the constraint of the substituted instance (*i.e.* $t{<:}F(t)$), and
- for any plausible type argument (*i.e.* any type t such that $t{<:}G(t)$), the value returned by the substituted type (*i.e.* instances of T_1) are values that were originally acceptable (*i.e.* instances of T_2).

These requirements lead us to the rule

$$(\mathit{fbd\_poly\_correct}) \qquad \frac{\Gamma, t{<:}G(t) \vdash t <: F(t) \qquad \Gamma, t{<:}G(t) \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

Let us look at the difference between these last two rules. The second, *fbd_poly_correct* seems more flexible since it would allow us to deduce (correctly) assertions such as:

$$\Phi \vdash \forall t{<:}t.T <: \forall t{<:}\alpha.T$$

where the first premise reduces to

$$t <: \alpha \vdash t <: t$$

which would hold by reflexivity of the subtype relation. If we use *fbd_poly_ok*, then this premise reduces to

$$t <: \alpha \vdash \alpha <: t$$

which cannot be shown.

However, we argue below that in relatively simple type systems, this extra flexibility arises only when one has types of the form $\forall t{<:}t.T$, which seems like a rather awkward formulation of the type $\forall t.T$ since the constraint is trivially satisfied by all types.

The argument is based on the formulation of bounded quantification presented in [CG91]. We use the partially correct algorithm presented there to reason about the proof rules for subtyping.

Consider the first premise of *fbd_poly_correct*, namely

$$\Gamma, t{<:}G(t) \vdash t <: F(t)$$

In the formulation of the subtype checking algorithm, a subtyping judgement about a type variable can be proven in the following ways:

- using the reflexivity rule, which can be done only if $F(t) = t$. This means the subtype is one of the "awkward" kinds just mentioned above.
- using the rule which says that every type is a subtype of the maximal type. This means that $F(t)$ has to be the maximal type, which in turn implies that the corresponding premise of *fbd_poly_ok* will also succeed by the same rule.
- using the deterministic form of the transitivity rule, which basically proceeds by trying to prove that the bound of the variable t in the environment (*i.e.* $G(t)$) is a subtype of $F(t)$. But this is exactly the premise of *fbd_poly_ok*.

Hence, the only cases where the use of *fbd_poly_ok* results in an outcome inconsistent with the use of *fbd_poly_correct* are those that involve types of the form $\forall t{<:}t.T$. These types, because of their unnatural formulation, seldom occur in practice. This is true of all examples written in RAPIDE to date. Hence, we propose that the following rule for subtyping F-bounded polymorphic types suffices in practice.

$$\frac{\begin{array}{c}\Gamma, t{<:}G(t) \vdash G(t) <: F(t) \\ \Gamma, t{<:}G(t) \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

4.2 Abstract types

The calculus presented in Chapter 3 presents existential types in a slightly simplified manner compared to the abstraction mechanisms in RAPIDE. The simplification is in the mode that the type and value components of an abstract type are declared. We look at the case for bounded abstract types - exact abstract types are identically handled.

The rules in Chapter 3 present existential types as what are known as "strong sums". In this case, **fst**(e) and **snd**(e) can occur independently and there are separate reduction rules for each.

$$\mathtt{snd}(\langle \exists t{<:}F(t) = S, e\rangle) \longrightarrow [S/t]e$$

$$\mathtt{fst}(\langle \exists t{<:}F(t) = S, e\rangle) = S$$

As one might expect, there is another style corresponding to "weak sums". In this case, the occurrence of existential types is restricted to let-expressions of the form

$$\text{let}\, s = \mathtt{fst}(e)\, \text{in}\, \{\, \text{let}\, x = \mathtt{snd}(e)\, \text{in}\, M\}$$

and the reduction rule is

$$\text{let}\, s = \mathtt{fst}(e)\, \text{in}\, \{\, \text{let}\, x = \mathtt{snd}(e)\, \text{in}\, M\} \longrightarrow [S/s][f/x]M$$

when $e = \langle \exists t{<:}F(t) = S, f\rangle$.

However, in RAPIDE existential types are subtly different and correspond to the following expressions where the scopes of the type and value components of the abstract type are more tightly coupled:

$$\text{let}\, x = \{\, \text{let}\, s = \mathtt{fst}(e)\, \text{in}\, \mathtt{snd}(e)\}\, \text{in}\, M\}$$

The reduction rule to be used is:

$$\text{let}\, x = \{\, \text{let}\, s = \mathtt{fst}(e)\, \text{in}\, \mathtt{snd}(e)\}\, \text{in}\, M\} \longrightarrow [[S/s]f/x]M$$

for $e = \langle \exists t{<:}F(t) = S, f\rangle$.

The more accurate forms of exact abstract types are similarly defined.

4.3 Private interfaces

In this section, we explore how bounded abstract types are sufficient to express the functionality provided by private interfaces as described in Section 2.2.5. This approach is akin to that taken in [PT93b].

Briefly, the encoding is as follows. The use of private interfaces is extra visibility when there are multiple implementations of the same type. So suppose we have a type I with implementations I1 and I2 as follows:

```
type I is interface
public
    ... public constituents...
private
    ... private constituents...
end interface;

I1 : I = ... implementation one ... ;
I2 : I = ... implementation two ... ;
```

We can express this as

```
type I_Package is interface
    type I_x <: interface
                        ... public constituents ... ;
                end interface;
    I1_x : I_x;
    I2_x : I_x;
  end interface;

type I_Package_Impl is module
    type I_x = interface
                      ... public constituents ... ;
                      ... private constituents ... ;
               end interface;
    I1_x = ... implementation one ...;
    I2_x = ... implementation two ...;
  end module;

  type I is I_Package_Impl.I_x;
  I1 = I_Package_Impl.I1_x;
  I2 = I_Package_Impl.I2_x;
```

The implementations of I1_x and I2_x can make use of the private constituents of each other because of the extended definition of the type I_x in that scope.

In fact, the very mechanism shown above can be used to model a more general feature akin to the *friends* notion of C++, where any two given types can have added visibility into the internals of each other. This would be done simply by packaging the two types instead of the single one above.

A shortcoming of this translation, however, is that the multiple implementations of the type with private constituents have to be collected together and wrapped into one package.

Chapter 5

Expressiveness of RAPIDE types

This chapter illustrates the expressiveness of the Rapide type system, indicating that the set of concepts chosen as primitive in our formulation does not result in a loss in programming ease.

The chapter presents two studies of the expressiveness of Rapide types that the author is significantly responsible for. In Section5.1, we define all the predefined types of Rapide using Rapide interfaces and function types, and show that these definitions exhibit the same behavioral and subtyping properties as the predefined types they define. In Section5.2, we investigate a contemporary language Griffin, and demonstrate how its features can be expressed in Rapide.

5.1 Rapide predefined types

In this section, we illustrate the use of Rapide interface and function types to define the predefined types of Rapide. These definitions, besides serving as proof of the expressiveness of the Rapide type system, provide documentation of the functionality of the various predefined types of Rapide and are the basis of the exposition in the Rapide predefined types manual [RPr].

For each predefined type, we show how one declares instances of it, what operations may be performed, and how objects of that type can be created. We also mention the subtyping rules in all the non-trivial cases. A significant property of these Rapide definitions of the Rapide predefined types is that the subtyping relationships that one expects for each of these types are, infact, those that would be inferred from the Rapide definitions.

For details about any of these, the reader is referred to [RPr].

5.1.1 Boolean

The declaration and use of booleans and operations that can be performed on them are described by the following:

boolean_type_expression ::= Boolean

type Boolean **is** **interface**
  function Not() **return** Boolean;
  function And(rhs : Boolean) **return** Boolean;
  function Or(rhs : Boolean) **return** Boolean;
  function Xor(rhs : Boolean) **return** Boolean;
  function =(rhs : Boolean) **return** Boolean;
  function /=(rhs : Boolean) **return** Boolean;
end **interface**;

boolean_literal ::= True | False

5.1.2 Equal

Most predefined types have equality, inequality and other ordering functions defined on their objects. Instead of repeating the definition of these functions for each predefined type, Rapide defines a type constructor Equality, which has functions = and /=.

equality_type_expression ::= Equality'('*type_expression*')'

type Equality(**type** T) **is** **interface**
  function =(rhs : T) **return** Boolean;
  function /=(rhs : T) **return** Boolean;
end Equality;

5.1.3 Order

This is similar to the Equality type constructor.

order_type_expression ::= Order'('*type_expression*')'

```
type Order(type T) is interface
 include Equality(T);
      function <=(rhs : T) return Boolean;
      function <(rhs : T) return Boolean;
      function >(rhs : T) return Boolean;
      function >=(rhs : T) return Boolean;
 end interface;
```

5.1.4 Numeric

The numeric predefined types are Integer, Float, Natural and Positive. The type Numeric defines the functionality common to floating-point numbers and integers.

numeric_type_expression ::= Integer | Float | Natural | Positive

```
type Numeric (type t) is interface
      include Order(t);
      function +() return t;
      function −() return t;
      function +(rhs : t) return t;
      function −(rhs : t) return t;
      function ∗(rhs : t) return t;
      function ∗∗(rhs : t) return t;
      function /(rhs : t) return t;
      function %(rhs : t) return t;
      function Abs() return t;
end Numeric;

type Integer is interface
      include Numeric(Integer);
      function Pred() return Integer;
      function Succ() return Integer;
end Integer;
```

```
Max_Integer : Integer;
Min_Integer : Integer;

type Float is interface
    include Numeric(Float);
    function Floor() return Integer;
end interface;

type Natural is Integer;
        (with the constraint that all instances are non−negative)

type Positive is Natural
        (with the constraint that all instances are positive)
```

numeric_literal ::= *integer_literal* | *float_literal*
integer_literal ::= { *digit* }
float_literal ::= *integer_literal* '.' *integer_literal* [*exponent*]
    | *integer_literal* '#' { *hex_digit* } '.' *integer_literal* '#' [*exponent*]

digit ::= '0' | '1' | ... | '9'
hex_digit ::= *digit* | 'a' | ... | 'f' | 'A' | ... | 'F'
exponent ::= ('e' | 'E') *integer_literal*

Note that Positive, Natural and Integer are only distinct because of the constraints imposed, and are hence indistinguishable by the type system. Also, there is no subtyping relationship between Integer and Float.

5.1.5 Character

character_type_expression ::= Character

```
type Character is interface
    include Order(Character);
    function Code() return Integer;
end Character;
```

function Code_To_Char(i:Integer) **return** Character;

character_literal	::=	’’’ *unquoted_character_literal* ’’’
unquoted_character_literal	::=	’a’ | ... | ’z’ | ’A’ | ... | ’Z’ |
		’\n’ | ’\\’ | ’\’’ |
		’*integer_literal*’

The escape characters on the second line represent, respectively, the characters for new line, backslash and single quote. Literals of the form `'\N'`, where N is an integer, simply denote the character returned by Code_To_Char(N).

5.1.6 String

Strings are essentially sequences of Character objects.

string_type_expression ::= String

type String **is** **interface**
  include Order(String);
  function Is_Null() **return** Boolean;
  function Length() **return** Integer;
  function [ ](Position : Integer) **return** Character;
  function &(S : String) **return** String;
  function Append(C : Character) **return** String;
  function Prepend(C : Character) **return** String;
  function Slice(Lbound, Ubound : Integer) **return** String;
  function String_To_Integer(S : String) **return** Integer;
end String;

function &(S : String, C : Character) **return** String;
function &(C : Character, S : String) **return** String;

string_literal ::= ’"’ { *unquoted_character_literal* } ’"’

5.1.7 Record

A record is a finite, heterogeneous collection of labeled objects. Another way to describe records, which are familiar from Pascal and many subsequent languages, is as tuples whose components are selected by component name rather than position in the collection.

record_type_expression ::=
  record
    [*field_declarations*]
  end [**record**] [*identifier*]

field_declaration_list ::= *field_declaration* { *field_declaration* }
field_declaration ::= *identifier_list* ':' *type_expression* [**is** *expression*]';'
  ::= **include** *type_expression*
    [[**only** | **except**] *derived_name_list*]
    [**replace** *renamings*]

Each Record type:

type R **is** **record**
  F1 : T1;
  F2 : T2;
  ...
  Fn : Tn;
end **record**;

where Fi are identifiers and Ti are types, is equivalent to the interface type:

type R **is** **interface**
  F1 : T1;
  F2 : T2;
  ...
  Fn : Tn;
end **interface**;

A Record literal is a list of field name, field object pairs written in the following manner

record_literal ::= '(' [*field_object_list*] ')'
field_object_list::= *identifier* '=>' *expression* { ',' *identifier* '=>' *expression* }

A Record type R1 is a subtype of a Record type R2 if the set of fields of R2 is a subset of the set of fields of R1, and for every field F of R2, the type of F in R1 is a subtype of the type of F in R2. This may be viewed as two separate rules for subtyping — adding fields and replacing types of fields by subtypes. Subtyping for recursive record types is determined similar to that for interface types.

5.1.8 Union

Union types are *tagged unions*, (sometimes called *discriminated unions*) of one or more types. If U is the tagged union of S and T and o is an object of type U, then it is possible to extract from o an object of type S or type T, depending on the *tag* of o. The types S and T are called the *member types* of U.

union_type_expression ::= **union**
{ *object_name_declaration* }
end [**union**]

A union type declaration:

type U **is union**
TAG1 : T1;
TAG2 : T2;
...
TAGn : Tn;
end union;

where TAGi are identifiers and Ti are type expressions, is shorthand for the declaration of a polymorphic function type with two parameters:

```
type U is function ( -- two parameters: A type and a record of functions
    type T;
    record
        TAG1 : function (o : T1) return T;
        TAG2 : function (o : T2) return T;
        ...
        TAGn : function (o : Tn) return T;
    end record ) return T;
```

A union type literal consists of a tag and an object of the type associated with the tag.

union_literal ::= '(' *identifier* ',' *expression* ')'

Functions for extracting a value based on a tag or to provide tag-based discrimination can be written based on this functional representation.

A union type U1 is a subtype of a union type U2 if the set of tags of U1 is a subset of the set of tags of U2, and for every tag T of U1, the type associated with T in U1 is a subtype of the type associated with T in U2.

5.1.9 Enum

An enumeration type is a type that is defined by explicitly listing its member objects.

enumeration_type_expression ::= **enum** *identifier* {',' *identifier* } **end** [**enum**]

An enumeration type expression is shorthand for a corresponding union type expression. The declaration

type E **is** **enum** I1, I2, ..., In **end** **enum**

represents the type declaration

type E **is** **union** I1, I2,..., In: **triv**; **end** **union**

An enumeration literal is simply an identifier which must belong to the enumeration type.

Enumeration_literal ::= *identifier*

5.1.10 Iteration

The following set of interfaces defines a set of type constructors that are meant for inclusion in the definition of other interface types to add standard operations for iteration.

```
iteration_type_expression ::=
      Iterator'('type_expression')' |
      Iterable'('type_expression')' |
      Discrete'('type_expression')'

type Iterator (type T) is interface
   Item  : function () return T;
   More  : function () return Boolean;
end Iterator;

type Iterable (type T) is interface
   Iterator : module () return Iterator(T);
end Iterable;

type Discrete (type T) is interface
   <    : function (rhs : T) return Boolean;
   =    : function (rhs : T) return Boolean;
   Succ : function () return T;
end Discrete;
```

Type T is said to be *iterable* if T <: Iterable(T) and *discrete* if T <: Discrete(T).

The subtyping relationships for types created by applying these constructors are as follows:

- Iterator(T1) is a subtype of Iterator(T2) iff T1 is a subtype of T2.
- Iterable(T1) is a subtype of Iterable(T2) iff T1 is a subtype of T2.
- Discrete(T1) is a subtype of Discrete(T2) iff T1 is equal to T2.

5.1.11 Array

An array type declares a mapping from objects of its *index* type to objects of the array's *element* type.

array_type_expression ::=
    array '['*type_expression* {',' *type_expression* }']' **of** *type_expression*

The types between the brackets are the index types of the array type.

Also,

array [T1, T2, ...] **of** T

is equivalent to

array [T1] **of** **array** [T2] **of** ... **of** T

In the most general case, the type expression

array [Index_Type] **of** Elem_Type

is equivalent to the expression

Array(Index_Type, Elem_Type)

where the Array type constructor is defined as:

```
type Array (type Index; type Element) is interface
    include Iterable(Index);
    [ ] : function (I : Index) return Element;
end Array;
```

However, if the Index_Type is discrete (in other words, if Index_Type <: Discrete(Index_Type)), the type expression

array [Index_Type] **of** Elem_Type

is equivalent to

Discrete_Array(Index_Type, Elem_Type)

where the Discrete_Array type constructor is defined as:

```
type Discrete_Array (type Index <: Discrete(Index);
                     type Element) is interface
    include Array(Index, Element);
    Slice        : function (First, Last : Index) return Discrete_Array(Index, Element);
    =, /=        : function (rhs : Discrete_Array(Index, Element)) return Boolean;
    First, Last  : function () return Index;
    Length       : function () return Integer;
end Discrete_Array;
```

Array literals are written using an aggregate notation with the following syntax:

```
array_literal ::= '(' [bounds] association { ',' association } ')'
bounds        ::= expression '..' expression ','
association   ::= positional_association | named_association | others_association

positional_association ::= expression
named_association      ::= expression '=>' expression
others_association     ::= others '=>' expression
```

An array type A1 is a subtype of array type A2 if the index type of A2 is a subtype of the index type of A1, and the element type of A1 is a subtype of the element type of A2.

5.1.12 Set

```
set_type_expression ::=  Set'('type_expression')'

type Set(type Element <: Equality(Element)) is interface
     include Equality(Set(Element));
     function Is_Member(E : Element) return Boolean;
     function +(E : Element) return Set(Element);
     function &(S : Set(Element)) return Set(Element);
     function −(E : Element) return Set(Element);
     function −(S : Set(Element)) return Set(Element);
     function <(S : Set(Element)) return Boolean;
     function Cardinality() return Natural;
```

end Set;

function Empty_Set(**type** Element <: Equality(Element)) **return** Set(Element);

set_literal ::= '{' '}' | '{' *expression* {',' *expression*} '}'

Set(E1) is a subtype of Set(E2) iff E1 is type equivalent to E2.

5.1.13 Ref

References *refer to* objects. References are objects of reference types. A reference type expression R must mention the type of the referenced objects.

ref_type_expression ::= Ref'('*type_expression*')'

type Ref(**type** T) **is** **interface**
    function :=(rhs : T) **return** Ref(T);
    function !() **return** T;
    function Is_Nil() **return** Boolean;
end Ref;

function Ref_To(**type** T; rhs : T) **return** Ref(T);

function Nil(**type** T) **return** Ref(T);

The operation R.:=(X) makes the reference R refer to X. The operation !R returns the object that R refers to. The operation R.Is_Nil() checks whether R refers to an object.

The call Ref_To(T,X) creates a reference of type Ref(T) which refers to X.

The call Nil(T) creates a reference of type Ref(T).

Ref(T1) <: Ref(T2) if and only if T1 and T2 are type-equivalent.

5.1.14 Clocks

Clocks are monotonically increasing counters which provide a notion of simulation time. Events are timed with respect to (one or more) clocks and have start and finish times with respect to each of these clocks.

clock_type_expression ::= Clock | Synchronous_Clock | Regular_Clock | Slaved_Clock

type Clock **is interface**
    type Ticks <: Natural;
    function Now() **return** Ticks;
    function Start(E : **event**()) **return** Ticks;
    function Finish(E : **event**()) **return** Ticks;
    function Length(E : **event**()) **return** Ticks;
end Clock;

function Make_Clock() **return** Clock;

The interfaces for synchronous, regular and slaved clocks are more complex and the interested reader can find these in [RPr].

5.1.15 Event

Events are record-like structures that are generated by modules as part of the action-based communication primitives of Rapide. For more details, refer to [REx].

event_type_expression ::= **event** '(' *param_list* ')'

param_list ::= *identifier_list* ':' *type_expression*
{ ';' *identifier_list* ':' *type_expression* }

Every event type is defined in terms of its *parameter record type* and a set of operations available on all events.

For every event type E of the form **event**(F1:T1, ... Fn:Tn), the associated parameter record type E_Rec is defined as:

```
type E_Rec is record
        F1 : T1;
        F2 : T2;
        ...
        Fn : Tn;
end record;
```

Also, all events have a default set of operations defined on them. These allow an event to be compared with another and allow the originating module and action to be determined. These operations are described in the following interface.

```
type Event_Operations is interface
      =, /=, <, >, <=, >=, || : function (rhs : event()) return Boolean;
      From_Module : interface end;
      Action_Name : String;
end interface;
```

The event type E of the form **event**(F1:T1, ... Fn:Tn) is now defined as:

```
type E is interface
      include E_Rec;
      include Event_Operations;
end interface;
```

Event types can be defined from other event types in the same way that record types are.

The subtyping rules for event types are as follows:

- Every event type is a subtype of its parameter record type.
- For two event types E1 and E2, E1 is a subtype of E2 iff E1's parameter record type is a subtype of E2's parameter record type.

5.1.16 IO

Users perform input and output (I/O) operations on files.

IO_type_expression ::= File

type File **is** **interface**
  Go_To_Start : **function**();
  Go_To_End : **function**();
  New_Line : **function**();
  Next_Line : **function**();
  At_End : **function**() **return** Boolean;
  Read_Bool : **function**() **return** Boolean;
  Put_Bool : **function**(x : Boolean);
  Read_Int : **function**() **return** Integer;
  Put_Int : **function**(x : Integer);
  Read_Float : **function**() **return** Float;
  Put_Float : **function**(x : Float);
  Read_Char : **function**() **return** Char;
  Put_Char : **function**(x : Char);
  Read_String : **function**() **return** String;
  Put_String : **function**(x : String);
end File;

function Set_Error_File(f : File);
function Set_Input_File(f : File);
function Set_Output_File(f : File);

function File_Open(filename : String) **return** File;

IO_literal ::= std_in | std_out | std_err

The File interface defines the default operations that all files must provide. These include operations to move in a file and to read and write objects of the basic types. The function File_Open is implementation-dependent and translates strings to some concrete I/O device that can be treated as a file.

The executable language provides default file objects for the std_in, std_out and std_err literals.

5.2 Integration of Griffin and Rapide

This section 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.

5.2.1 Premise

An important aspect of Rapide is the separation of the type and module language from the executable, or implementation, language. Designed with interoperability 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.

5.2.2 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.

5.2.3 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

5.2.4 Main concepts in the RAPIDE type system

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

5.2.5 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	$\longrightarrow$	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.

5.2.6 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.

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
```

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.

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);
```

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)
```

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);
```

Chapter 6

Implementation of the RAPIDE type system

This chapter describes some of the aspects of the implementation of the front end of the Rapide 1.0 compiler, and the type system in particular. The author's contribution to this implementation was the the design and development of the type and subtype checking algorithms for the language, along with support for tools that would use these routines. The algorithms were designed by extending the proof rules for the underlying theoretical framework (see Chapter 3).

We use the term "semanticize" to refer to the process of checking syntactic representations against semantic information about those representations, and annotating the same with such information.

6.1 An overview of the compiler

Figure 6.1 describes the overall layout of the Rapide 1.0 compiler.

The front end of the compiler consists broadly of two stages:

Parsing: The parser essentially generates an *Abstract Syntax Tree (AST)* from the program code. Some syntactic abbreviations are also handled at this stage through simple source code transformations.

Semanticizing: The semanticizer checks the AST for some forms of correctness, and annotates the abstract syntax tree with semantic information, generating the *Static Semantics Tree (SST)*.

The front end and the components of the semanticizer are described in greater detail later in this chapter.

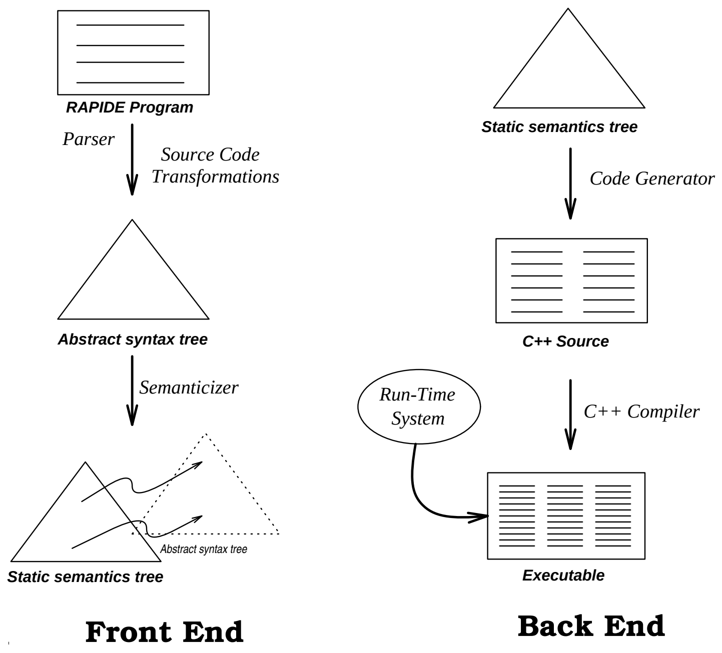

Figure 6.1: The Rapide 1.0 compiler

The back end consists of the following stages:

Code Generation: The code generator accepts the annotated syntax tree and generates C++ code.

Compilation: A C++ compiler then generates executable code from the C++ code.

The back end includes a component called the Rapide *Run-Time System (RTS)*. This plays an important role in generating the executable. The run-time system is responsible for tasks such as event management, modelling concurrent processes, scheduling processes and logging computations.

Integrated with the Rapide 1.0 programming environment are the following tools:

- a poset browser
- interfaces and modules hierarchy browsers
- an interface conformance checker (subtype checker, in other words)
- a constraint checker

One of the more interesting tools planned for a later release is a *Graphical Architectures Editor* currently under design at TRW.

6.2 The front end

The front end of the RAPIDE compiler, as mentioned earlier, consists of two stages. We now look at these more closely.

6.2.1 Source code transformations & the AST

Before generating the abstract syntax tree, the RAPIDE front end performs several simple source code transformations. These are normalizations that convert RAPIDE code into some normal source form. Examples of such transformations are:

- coalescing interface constituents
- rewriting object (and name) declarations with more than one identifier as separate object (and name, respectively) declarations
- rewriting parameter declarations with more than one identifier as separate parameter declarations
- rewriting architectural constructs in terms of the simpler interface and module concepts
- syntactic abbreviations for variables are rewritten in terms of reference types
- shorthand function declarations are rewritten as object declarations of function types

Each node of the AST is an instance of class `Node`. Each node has an ordered list of sons. The class `Node` defines operations to access and manipulate these sons efficiently. To make annotation of the AST easy and extensible, each node has associated with it a set of attributes, and provides functions for adding and retrieving attributes. Semantic information is thus attachable in a fairly flexible manner, and can be much more elaborate in nature than would be possible easily in a symbol table.

6.2.2 Semanticization & the SST

The semanticizer checks the AST for some forms of semantic correctness and decorates the tree with information that is useful to the code generator and other tools. The nodes of the AST are mapped into semantic symbols, generating the SST. The semantic symbols contain as little redundant information as possible - most "as is" information is still accessed through the corresponding node, while most "inferred" information is stored in the semantic symbols. Hence the SST is tightly linked with the AST.

Examples of actions performed by the semanticizer related to typing information are:

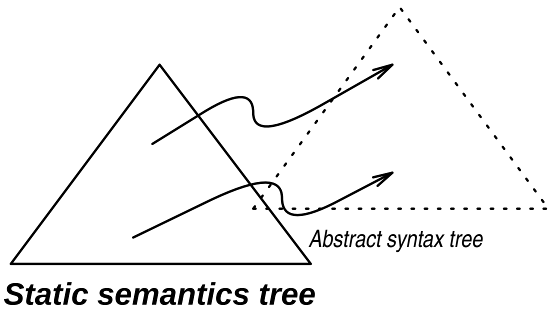

Figure 6.2: SST - AST information sharing

- Refine type (name) declaration nodes into different semantic symbols corresponding to simple type (name) declarations and type constructor (name) declarations.
- For interface types, elaborate all derivations and provide attributes that return the normalized form of the different kinds of constituents (public, private, extern, etc). For function types, provide attributes for the argument list and return type.
- For all nodes with a parameter list (for example, function types and type constructors), normalize and semanticize each of the parameters, and provide an attribute in the corresponding semantic symbol that returns this parameter list.
- For each type expression corresponding to a predefined type, decorate the corresponding semantic symbol with the information relevant for the particular predefined type.

The *subtype checker* appears as a subsystem of the semanticizer. It is the unit responsible for type conformance checking. This is described in greater detail in Section 6.3.

Another feature of interest is the *semantics external interface*, which is the interface to building and retrieving semantic information, and is the only set of functions that other tools need to have access to, in order to be able to completely use the SST for their implementation. This interface includes simple decorating and retrieving functions, and the interfaces to the various semantic symbols the SST recognizes.

6.3 The subtype checker

Section 2.2.2 describes the relevance and motivation behind the notion of subtyping. Section 3.1.5 enumerates the subtyping rules for the underlying calculus of Rapide. The subtype checker extends and implements these rules for the full language.

More specifically, the subtype checker takes as arguments, two semantic symbols corresponding to type expressions, and returns a boolean value depending on whether or not one type expression is determined to be a subtype of the other.

6.3.1 Side-effects of the subtype checker

The following side-effects of the subtype checker also describe some of the interesting aspects of the checker:

- A subtype check is invoked whenever an instance x of a certain type T_1 is used in a context that demands some other type T_2. In such cases, it is usually quite likely that x will be used as a T_2 on more than one occasion, in reasonably quick succession. Instead of performing the entire subtype check each time, the algorithm uses an attribute on type expressions to remember the last few types it was successfully compared with. Thus, a successful subtype check has the side-effect of annotating the subtype with this information.

- There is a slight subtlety involved with the subtype checking of recursive types. While checking whether a recursive type T_1 is a subtype of another recursive type T_2, one is permitted to assume that this is true and then proceed with the rest of the check. This is once again implemented using an attribute on type expressions. As an aside, the same attribute can be used for both this and the previous annotation.

- The subtype checker generates a trace of its actions. This trace allows a user to inspect in greater detail why a subtype check succeeded or failed (the latter case being naturally the more useful one). The tool described later in this section makes use of this trace. The checker produces a trace for the following six forms of actions (with relevant arguments) :

  subtp_success: Success of the check

  subtp_failure: Failure of the check

  subtp_trace_in: Start of a recursive subtype check

  subtp_trace_out: End of a recursive subtype check

  subtp_trace: Atomic check performed

  subtp_error: An error encountered

  The interface to the trace generator is orthogonal to the subtype checker and very easily modifiable.

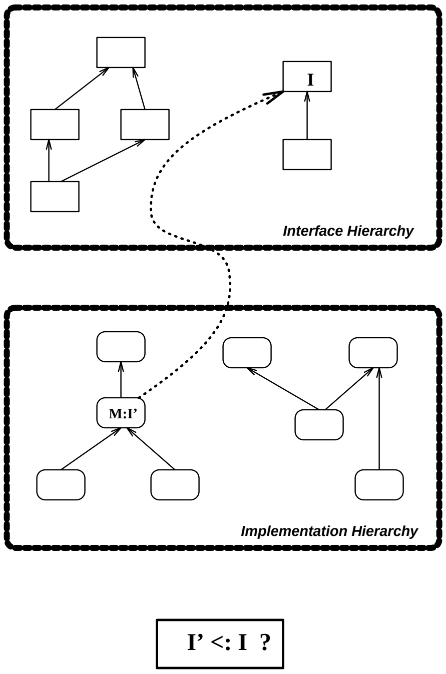

Figure 6.3: Static checking while browsing

6.3.2 Interaction with tools

The subtype checker adds functionality to the interface and module browsers, making them useful in an evolutionary programming environment.

In RAPIDE, interfaces are separate from their implementations. In a typical development scenario, some set of initial interfaces are agreed upon, and several developers work independently on their implementations. At any point in this process, the browsers allow one to view the interface and module hierarchies separately. Normally one might be required to write and attempt to compile a test program to check whether a module satisfies its interface. In other words, all static checking is done on a complete program. However, by hooking the subtype checker into the browsers, a fair amount of checking can be done at a fairly early stage without having to wait for other developers. The exact functionality provided is described in Figure 6.3.

While viewing the browsers, the user is allowed to click on a module M of type I' in the module browser, drag it and drop it into any interface I in the interface browser. This is viewed as a request to check whether the module M implements the interface I. The subtype checker is fired off, and checks whether I' is a subtype of I. The user can then demand an explanation of the subtype checker's outcome. This is reconstructed from the trace generator that was mentioned in the previous section.

Chapter 7

Conclusion

This concluding chapter summarizes other research work addressing approximately the same issues as this thesis, and points towards future directions of research along these lines.

7.1 Related Work

Foundational work in the design and semantics of object-oriented languages can be broadly categorized into three approaches.

Class-based languages: Such languages have the notion of a *class* as a definition of common behavior and *instances* as the actual manifestations of that behavior. Several seminal papers have opened new avenues of investigations in this area. Cardelli [Car88] proposed record models for features of object-oriented languages, and this led to more involved investigations in [Coo89a], [CM89] and [Car91]. Cardelli and Wegner [CW85] prompted the use of quantification to model language features. Mitchell and Plotkin [MP88] explored the use of existential types to model abstract types, and Pierce and Turner [PT93a] proposed models for object-oriented languages that modelled objects as existential types. Cook and others [CCH+89] proposed F-bounded quantification as a useful tool.

Our work is a continuation of these various avenues - we view objects as records and combine universal and existential bounded quantification in a uniform framework to model features of object-oriented languages. We have also drawn heavily from the design and features of languages such as SMALLTALK, ML, C++ and ADA. Fig 7.1 gives a rough graphic depiction of this. The entries above RAPIDE in the picture depict other contemporary language design work that is similar in flavor, while the entries below ([HL94], [Ler94] and [Pie91]) reflect some of the recent theoretical investigations along the same lines.

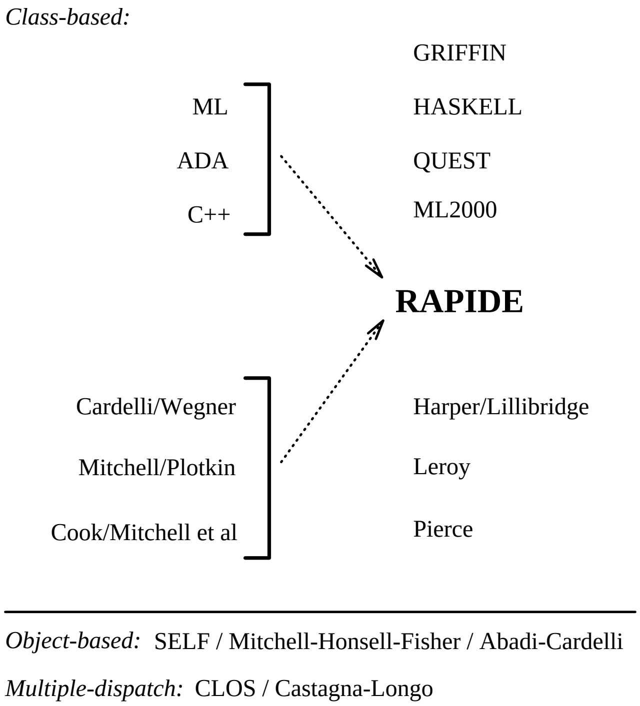

Figure 7.1: Related Work

Object-based languages: In these languages, the behavior of objects is determined by the object itself and not from some class that they belong to. Delegation rather than inheritance is the mode for code-reuse. SELF [US87] is the leading proponent of this style, although the essential ideas come from earlier work by [Lie86] and [Ste87]. Object-based theoretical models have been studied more recently in [AC94] and [MHF93].

Multiple-dispatch based languages: These are languages where the encapsulation boundaries are weakened in favor of improved ways of specifying the behavior of multi-methods. CLOS [Kee89] made this programming style popular, and Castagna et. al. [CGL92] have been investigating this further.

7.2 Future Work

This work represents initial progress towards type systems and static support for large software environments. Some directions for future work that the author envisions are:

- *sound design of multi-language systems:* Programming environments that include components built on a range of operating systems in a variety of programming languages are becoming popular. This is evidenced, for instance, by the growing acceptance of the Object Management Group's standard on distributed software, CORBA. An imminent direction for the evolution of type systems in general is the support of such multi-language "programs" and a notion of type safety in their runtime behavior.

- *the evolution of type systems in the context of software architectures:* The development environments of large software projects, especially those that are used in embedded systems, tend to use the notion of software architectures, where software is developed in components that are then hooked up with simple control flow connections. An interesting area of research is the use of type systems in supporting the static development of such architectures. For example, types enhances with a notion of protocols might be one style of evolution in this context.

- *applications:* Type systems have much to offer to and learn from domains such as client-server languages, schema languages for heterogeneous databases, and scripting languages, amongst others. The static guarantees required and the runtime assumptions made in each of these cases are vastly different, and the design of typed versions of these languages should be useful both for the use of such languages and as a playground for new ideas for conventional languages.

Appendix A

Decidability proof for completely bounded quantification

A.1 Introduction

$F_{<:}$ [CW85, CG91] is a typed λ-calculus with subtyping and bounded second-order polymorphism. The importance of $F_{<:}$ in programming language design is that it provides a simple context for studying the typing problems that arise when subtyping and bounded quantification are added to polymorphic languages such as ML.

Curien and Ghelli [CG91] developed a subtyping algorithm for $F_{<:}$ and proved its partial correctness. Subsequently, Ghelli [Ghe90] presented a termination proof for this algorithm. A mistake was discovered in this termination proof, following which Pierce [Pie92] presented a proof showing that the subtyping problem for general $F_{<:}$ types is undecidable.

This section shows how one can make the subtyping problem decidable by imposing some restrictions on $F_{<:}$ types. We first prove the termination of Curien and Ghelli's algorithm when the bounds on all polymorphic types involved do not contain *Top*. *i.e.*, In all types of the form $\forall \alpha <: \tau_1 . \tau_2$, τ_1 does not contain *Top*. Such a bound completely determines the structure of α, hence we refer to this as "completely bounded". We later show that this restriction can be relaxed to allow unbounded quantification. Adding records and unions to our system causes it to become undecidable.

In Section A.2, we present Curien and Ghelli's algorithm. We prove the termination of this algorithm for completely bounded quantification in Section A.3. Section A.4 shows how we can relax our restrictions to allow unbounded quantification. Section A.5 presents examples of records and unions that cannot be handled by simple extensions of Curien and Ghelli's algorithm.

A.2 Curien and Ghelli's algorithm

A $F_{<:}$ type τ is either a simple type (such as *Int*), a variable, a function type $(\tau_1 \rightarrow \tau_2)$, or a polymorphic type $(\forall\alpha{<:}\tau_1.\tau_2)$. Given a list of assumptions Γ and two types σ and τ, the subtyping problem is to determine whether or not $\Gamma \vdash \sigma <: \tau$ — *i.e.*, whether or not σ is a subtype of τ given the assumptions in Γ. The assumptions in Γ will all be of the form $\alpha <: \tau$, where α is a type variable and τ is a type. (This convention — that α's refer to type variables and σ's and τ's refer to types — is used for the rest of the paper. We shall also use α to refer to simple types — the context will make it clear whether a particular α is a variable or a simple type.) τ in this case is called the *bound* of α and is referred to as $\Gamma(\alpha)$. Free variables in τ may only be *bounded* in other assumptions in Γ to the left (earlier in the list) of the assumption containing τ.

Curien and Ghelli's algorithm is presented as a list of axiom schemas and inference rules. These schemas and rules contain templates of subtyping problems. The algorithm proceeds by applying the inference rules backwards to the subtyping problem. If the subtyping problem matches the template below the line of an inference rule, it reduces to subtyping problems that can be derived from the templates above the line of the inference rule. If the subtyping problem matches the template of an axiom schema, the algorithm reports success. In all other cases, the algorithm reports failure.

The axiom schemas and inference rules are listed below:

(*NTop*)
$$\Gamma \vdash \sigma <: Top$$

(*NRefl*)
$$\Gamma \vdash \alpha <: \alpha$$

(*NVar*)
$$\frac{\Gamma \vdash \Gamma(\alpha) <: \tau}{\Gamma \vdash \alpha <: \tau}$$

(*NArrow*)
$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2}$$

(*NAll*)
$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma, \alpha{<:}\tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall\alpha{<:}\sigma_1.\sigma_2 <: \forall\alpha{<:}\tau_1.\tau_2}$$

A.3 Proof of termination

We prove termination by defining a complexity metric which is finite and positive for each subtyping problem, and show that the application of inference rules causes the complexity of the new subtyping problems generated to decrease. Since the complexity cannot decrease indefinitely, the algorithm will have to terminate.

In A.3.1, we define the complexity metric. In A.3.2, we show how the complexity metric is affected by applying the various inference rules. We conclude with a condition (Theorem A.3.3) that, if satisfied, will guarantee the termination of the subtyping algorithm. Finally, in A.3.3, we show how our restrictions on bounds of polymorphic types satisfies the condition of Theorem A.3.3.

A.3.1 The complexity metric

We define the *size* of a type τ with respect to a list of assumptions Γ, and refer to this as $size(\tau)_\Gamma$. The *complexity* of a subtyping problem $\Gamma \vdash \sigma <: \tau$ is defined as:

$$complexity(\Gamma \vdash \sigma <: \tau) \stackrel{\text{def}}{=} size(\sigma)_\Gamma + size(\tau)_\Gamma$$

$size(\tau)_\Gamma$ is determined by recursively replacing variables in τ with their respective bounds and then computing the textual size of the resulting expression. $size(\tau)_\Gamma$ is formally defined as:

$$\begin{array}{rcl} size(Top)_\Gamma & = & 1 \\ size(\alpha)_\Gamma & = & \begin{cases} size(\Gamma(\alpha))_\Gamma & \text{if } \Gamma(\alpha) \text{ is defined} \\ 1 & \text{otherwise} \end{cases} \\ size(\tau_1 \rightarrow \tau_2)_\Gamma & = & size(\tau_1)_\Gamma + size(\tau_2)_\Gamma \\ size(\forall\alpha{<:}\tau_1.\tau_2)_\Gamma & = & size(\tau_1)_\Gamma + size(\tau_2)_{\Gamma,\alpha<:\tau_1} \end{array}$$

Examples:

1. $size(Top \rightarrow \alpha)_{\alpha <: Int \rightarrow Int}$
   $= size(Top)_{\alpha <: Int\rightarrow Int} + size(\alpha)_{\alpha <: Int \rightarrow Int}$
   $= 1 + size(Int \rightarrow Int)_{\alpha <: Int\rightarrow Int}$
   $= 1 + size(Int)_{\alpha <: Int\rightarrow Int} + size(Int)_{\alpha <: Int\rightarrow Int}$
   $= 1 + 1 + 1 = 3.$

2. $size(\forall\alpha_1 <:(\alpha_2 \rightarrow Int).\alpha_1)_{\alpha_2 <: Int\rightarrow Int}$
   $= size(\alpha_2 \rightarrow Int)_{\alpha_2 <: Int\rightarrow Int} + size(\alpha_1)_{\alpha_2 <: Int\rightarrow Int, \alpha_1 <: \alpha_2 \rightarrow Int}$

$$= size(\alpha_2)_{\alpha_2 <: Int \to Int} + size(Int)_{\ldots} + size(\alpha_2 \to Int)_{\alpha_2 <: Int \to Int, \alpha_1 <: \alpha_2 \to Int}$$
$$= size(Int \to Int)_{\ldots} + 1 + size(\alpha_2)_{\alpha_2 <: Int \to Int, \alpha_1 <: \alpha_2 \to Int} + size(Int)_{\ldots}$$
$$= 2 + 1 + size(Int \to Int)_{\ldots} + 1$$
$$= 2 + 1 + 2 + 1 = 6.$$

Lemma A.3.1 *$size(\tau)_\Gamma$ is always finite and positive.*

Proof. It is obvious from the definition of *size* that it has to be positive. We prove that it is finite by showing that evaluation of *size* terminates for all τ and Γ. A complexity metric similar to that used in Ghelli's flawed termination proof actually works in this case.

The complexity metric to prove the termination of the evaluation of $size(\tau)_\Gamma$ is obtained by first ordering all the variables that occur in τ and Γ such that the following property is satisfied: If α_i is defined in the bound of α_j, then α_i occurs to the left of α_j in the ordering. It is possible to obtain such an ordering given the structure of $F_{<:}$ (there may be multiple orderings that satisfy this condition in which case, one of them is chosen arbitrarily). The *depth* of each variable is then defined as the number of variables that occur to the left of it in this ordering.

The complexity of any subproblem $size(\tau')_{\Gamma'}$ that arises during the evaluation of $size(\tau)_\Gamma$ is the tuple $\langle D, S\rangle$, where D is the maximum depth over all the variables that occur in τ' and S is the textual length of τ'. There may be variables in τ' and Γ' that do not occur in either of τ or Γ. These variables are created when an existing variable is duplicated in the reduction process, thus requiring one of the uses to be renamed. The depth of the renamed variable is defined to be the same as its unrenamed counterpart. The key to this proof is that defining the depth of renamed variables in this manner maintains the condition based on which the initial ordering was created.

It is easy to see that the complexity decreases (the ordering between $\langle D, S\rangle$ tuples is lexicographic) during the evaluation of $size(\tau)_\Gamma$. For all reductions other than $size(\alpha)_\Gamma = size(\Gamma(\alpha))_\Gamma$, the D component of the complexity metric either remains the same or decreases while the S component decreases; whereas in the abovementioned reduction, the D component decreases.

Since this complexity metric cannot decrease indefinitely, the evaluation of *size* for any τ and Γ has to terminate. □

A.3.2 The effect of inference rule application on complexity

NVar

$$complexity(\Gamma \vdash \alpha <: \tau) = size(\alpha)_\Gamma + size(\tau)_\Gamma = size(\Gamma(\alpha))_\Gamma + size(\tau)_\Gamma = complexity(\Gamma \vdash \Gamma(\alpha) <: \tau)$$

i.e., The complexity metric remains the same after application of the inference rule *NVar*. However, *NVar* may be applied continuously at most as many times as there are variables in Γ before one of the other rules has to be applied. The complexity metric reduces when any of the other rules are applied, so there is no problem.

NArrow

It is quite obvious that $complexity(\Gamma \vdash \tau_1 <: \sigma_1)$ and $complexity(\Gamma \vdash \sigma_2 <: \tau_2)$ are both less than $complexity(\Gamma \vdash \sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2)$.

NAll

It is quite obvious that $complexity(\Gamma \vdash \tau_1 <: \sigma_1)$ is less than $complexity(\Gamma \vdash \forall\alpha{<:}\sigma_1.\sigma_2 <: \forall\alpha{<:}\tau_1.\tau_2)$. We shall now simplify $complexity(\Gamma \vdash \forall\alpha{<:}\sigma_1.\sigma_2 <: \forall\alpha{<:}\tau_1.\tau_2) - complexity(\Gamma, \alpha{<:}\tau_1 \vdash \sigma_2 <: \tau_2)$ to determine the conditions under which the complexity metric decreases when reducing to the second rule above the line in *NAll*. This expression, which must be positive for the complexity metric to decrease, simplifies as follows:

$complexity(\Gamma \vdash \forall\alpha{<:}\sigma_1.\sigma_2 <: \forall\alpha{<:}\tau_1.\tau_2) - complexity(\Gamma, \alpha{<:}\tau_1 \vdash \sigma_2 <: \tau_2)$
$= (size(\forall\alpha <: \sigma_1.\sigma_2)_\Gamma + size(\forall\alpha <: \tau_1.\tau_2)_\Gamma) - (size(\sigma_2)_{\Gamma,\alpha <: \tau_1} + size(\tau_2)_{\Gamma,\alpha <: \tau_1})$
$= (size(\sigma_1)_\Gamma + size(\sigma_2)_{\Gamma,\alpha <: \sigma_1} + size(\tau_1)_\Gamma + size(\tau_2)_{\Gamma,\alpha <: \tau_1}) - (size(\sigma_2)_{\Gamma,\alpha <: \tau_1} + size(\tau_2)_{\Gamma,\alpha <: \tau_1})$
$= size(\sigma_1)_\Gamma + size(\tau_1)_\Gamma + (size(\sigma_2)_{\Gamma,\alpha <: \sigma_1} - size(\sigma_2)_{\Gamma,\alpha <: \tau_1})$

We concentrate on the part $size(\sigma_2)_{\Gamma,\alpha <: \sigma_1} - size(\sigma_2)_{\Gamma,\alpha <: \tau_1}$ from the last line above. If this expression is non-negative, then the overall expression will be positive, and therefore the complexity metric will decrease on the application of *NAll*. We need to present the following lemma before we can proceed further.

Lemma A.3.2 *For any types σ, τ, and for any list of assumptions Γ, such that the variable α is not bounded in any of them, and also does not occur anywhere in Γ, $size(\sigma)_{\Gamma,\alpha <: \tau} = size(\sigma)_\Gamma + n(size(\tau)_\Gamma - 1)$, where $n \geq 0$ and depends only on σ.*

The evaluation of $size(\sigma)_{\Gamma'}$ (for any Γ') will reduce to zero or more evaluations of the form $size(\alpha)_{\Gamma''}$ in addition to other reductions. Reductions to the form $size(\alpha)_{\Gamma''}$ may be either due to occurrences of α in σ, or occurrences of α in bounds in Γ' which are used to replace the variables they bound. If Γ' does not contain any bound that contains α, then the number of times $size(\sigma)_{\Gamma'}$ reduces to the form $size(\alpha)_{\Gamma''}$ depends only on σ. Suppose this number is n.

Therefore, $size(\sigma)_\Gamma$ reduces to n evaluations of the form $size(\alpha)_{\Gamma''}$, each of which evaluates to 1 since α is not bounded anywhere. So we can write $size(\sigma)_\Gamma$ as $m + n$ where m is the result of the evaluation of the remainder of the reductions.

Similarly, $size(\sigma)_{\Gamma,\alpha<:\tau}$ reduces to n evaluations of the form $size(\alpha)_{\Gamma''}$, while the remainder of the reductions evaluate to m. Each reduction to $size(\alpha)_{\Gamma''}$ further reduces to $size(\tau)_{\Gamma''}$. Since Γ'' is of the form $\Gamma, \alpha <: \tau, \ldots,$ τ does not depend on the portion of Γ'' to the right of Γ. Therefore, $size(\tau)_{\Gamma''} = size(\tau)_{\Gamma}$. So we can write $size(\sigma)_{\Gamma,\alpha<:\tau}$ as $m + n(size(\tau)_{\Gamma})$, which is the same as $size(\sigma)_{\Gamma} + n(size(\tau)_{\Gamma} - 1)$. □

We now simplify $size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} - size(\sigma_2)_{\Gamma,\alpha<:\tau_1}$.

$size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} - size(\sigma_2)_{\Gamma,\alpha<:\tau_1}$
$= (size(\sigma_2)_{\Gamma} + n(size(\sigma_1)_{\Gamma} - 1)) - (size(\sigma_2)_{\Gamma} + n(size(\tau_1)_{\Gamma} - 1)) \qquad$ (for some $n \geq 0$)
$= n(size(\sigma_1)_{\Gamma} - size(\tau_1)_{\Gamma})$

We are now ready to present the first of our main results.

Theorem A.3.3 *During the execution of Curien and Ghelli's algorithm, if $size(\sigma_1)_{\Gamma} \geq size(\tau_1)_{\Gamma}$ (where σ_1, τ_1, and Γ are as defined in NAll) is true every time NAll is used to reduce a subtyping problem to the subtyping problem derived from the right template above the line ($\Gamma, \alpha<:\tau_1 \vdash \sigma_2 <: \tau_2$), then the algorithm will terminate.*

Proof. Obvious from the results of this section. □

A.3.3 Subset restrictions to guarantee termination

When applying *NAll* on a subtyping problem, we shall require that we first consider the subtyping problem derived from $\Gamma \vdash \tau_1 <: \sigma_1$ (the left template above the line in *NAll*). Only if the algorithm terminates successfully on this problem do we consider the subtyping problem derived from $\Gamma, \alpha<:\tau_1 \vdash \sigma_2 <: \tau_2$ (the right template above the line in *NAll*). Hence, we can assume that $\Gamma \vdash \tau_1 <: \sigma_1$ when the algorithm is applied on the subtyping problem derived from $\Gamma, \alpha<:\tau_1 \vdash \sigma_2 <: \tau_2$. Assuming this, we have the following corollary to Theorem A.3.3.

Corollary A.3.4 *For every τ, σ that are bounds of polymorphic types and for every list of assumptions Γ, if $\Gamma \vdash \tau <: \sigma \Rightarrow size(\tau)_{\Gamma} \leq size(\sigma)_{\Gamma}$, then Curien and Ghelli's algorithm will terminate.*

Lemma A.3.5 *If $size(\tau_1)_{\Gamma} = size(\tau_2)_{\Gamma}$, then, for any σ, $size(\sigma)_{\Gamma,\alpha<:\tau_1} = size(\sigma)_{\Gamma,\alpha<:\tau_2}$.*

Proof. This follows trivially from Lemma A.3.2. $size(\sigma)_{\Gamma,\alpha<:\tau_1} = size(\sigma)_{\Gamma} + n(size(\tau_1)_{\Gamma} - 1) = size(\sigma)_{\Gamma} + n(size(\tau_2)_{\Gamma} - 1) = size(\sigma)_{\Gamma,\alpha<:\tau_2}$. □

We are now ready to present another key result, Theorem A.3.6, that if the subset restrictions mentioned in Section A.1 are satisfied, then the condition of Corollary A.3.4 will be satisfied. A straightforward consequence of this is that if these subset restrictions are met, then Curien and Ghelli's algorithm will terminate, and hence completely bounded quantification is decidable.

Theorem A.3.6 *For all types τ, σ that do not contain Top, and for any list of assumptions Γ, if $\Gamma \vdash \tau <: \sigma$, then $size(\tau)_\Gamma = size(\sigma)_\Gamma$.*

Proof. If $\Gamma \vdash \tau <: \sigma$, then there must be a proof $\Gamma^1 \vdash \tau^1 <: \sigma^1, \Gamma^2 \vdash \tau^2 <: \sigma^2, \ldots, \Gamma^n \vdash \tau^n <: \sigma^n$ where $\Gamma^n = \Gamma$, $\tau^n = \tau$, and $\sigma^n = \sigma$, and each $\Gamma^i \vdash \tau^i <: \sigma^i$ is either of the form of *NRefl*, or obtained from earlier steps in the proof using one of the rules *NVar*, *NArrow*, or *NAll*. Note that *NTop* will not be used in such a proof since *Top* does not occur in σ and τ.

We prove by induction that for all i $(1 \leq i \leq n)$, $size(\tau^i)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$. The induction hypothesis is that for all j $(1 \leq j < i)$, $size(\tau^j)_{\Gamma^j} = size(\sigma^j)_{\Gamma^j}$. Assuming the induction hypothesis, we prove $size(\tau^i)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$. There are four cases to consider:

1. $\Gamma^i \vdash \tau^i <: \sigma^i$ is of the form of *NRefl*. *i.e.*, $\tau^i = \sigma^i$. Therefore, $size(\tau^i)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$.

2. $\Gamma^i \vdash \tau^i <: \sigma^i$ is derived using *NVar* from an earlier step of the form $\Gamma^i \vdash \Gamma^i(\tau^i) <: \sigma^i$. In this case, τ^i is a variable bounded in Γ^i. Therefore, $size(\tau^i)_{\Gamma^i} = size(\Gamma^i(\tau^i))_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$.

3. $\Gamma^i \vdash \tau^i <: \sigma^i$ is derived using *NArrow* from earlier steps $\Gamma^k \vdash \tau^k <: \sigma^k$ and $\Gamma^l \vdash \tau^l <: \sigma^l$. Then $\Gamma^i = \Gamma^k = \Gamma^l$, $\tau^i = \sigma^k \to \tau^l$, and $\sigma^i = \tau^k \to \sigma^l$. Therefore, $size(\tau^i)_{\Gamma^i} = size(\sigma^k)_{\Gamma^i} + size(\tau^l)_{\Gamma^i} = size(\sigma^k)_{\Gamma^k} + size(\tau^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^k} + size(\sigma^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^i} + size(\sigma^l)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$.

4. $\Gamma^i \vdash \tau^i <: \sigma^i$ is derived using *NAll* from earlier steps $\Gamma^k \vdash \tau^k <: \sigma^k$ and $\Gamma^l \vdash \tau^l <: \sigma^l$. Then $\Gamma^i = \Gamma^k$, $\Gamma^l = \Gamma^i, \alpha <: \tau^k$, $\tau^i = \forall \alpha <: \sigma^k . \tau^l$, and $\sigma^i = \forall \alpha <: \tau^k . \sigma^l$ for some variable α. Therefore, $size(\tau^i)_{\Gamma^i} = size(\sigma^k)_{\Gamma^i} + size(\tau^l)_{\Gamma^i, \alpha <: \sigma^k} = size(\sigma^k)_{\Gamma^k} + size(\tau^l)_{\Gamma^i, \alpha <: \tau^k} = size(\sigma^k)_{\Gamma^k} + size(\tau^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^k} + size(\sigma^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^i} + size(\sigma^l)_{\Gamma^i, \alpha <: \tau^k} = size(\sigma^i)_{\Gamma^i}$. $\Box$

A.4 Allowing unbounded quantification

We can relax our restriction on the use of *Top* to allow unbounded quantification and still retain decidability of subtype checking. Furthermore, any variable bounded by *Top* (directly or indirectly) may be used as a bound for another variable. We refer to these variables as "*Top*-bounded".

With this relaxation, there are two kinds of bounds that we can write: (1) Types that do not contain either *Top* or *Top*-bounded variables; and (2) Types that are either *Top* or a *Top*-bounded variable.

To show that this will not cause problems, we redefine *size*(*Top*) for all *Top*'s that occur as bounds of variables to be a number L that is larger than the *size* of any bounds of the first kind mentioned above. With this redefinition of *size*, it is quite easy to see that Corollary A.3.4 continues to hold even when the bounds τ and σ are from this relaxed domain.

There are four cases to consider where τ and σ may each be either of the two kinds mentioned above. We consider each case separately:

1. τ and σ are both types not containing *Top* or *Top*-bounded variables: This case has been handled in Theorem A.3.6.
2. τ and σ are both types that are either *Top* or a *Top*-bounded variable: Then $size(\tau) = size(\sigma) = L$.
3. τ is a type not containing *Top* or *Top*-bounded variables while σ is either *Top* or a *Top*-bounded variable: Then $size(\sigma) = L > size(\tau)$.
4. τ is either *Top* or a *Top*-bounded variable while σ is a type not containing *Top* or *Top*-bounded variables: In this case, τ cannot be a subtype of σ, so we need not consider it any further.

A.5 Record and union types

We have shown that one can achieve a decidable type system for a fairly unconstricted subset of $F_{<:}$. An immediate extension to consider is the addition of record and union types. However, it turns out that adding either record or union types to the type system (along with their associated inference rule) makes the subtyping problem undecidable.

A.5.1 Record types

We denote the record type with fields $l_1 \ldots l_n$ having types $\tau_1 \ldots \tau_n$ respectively as $\{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}$. The inference rule for records is :

$$(NRec) \qquad \frac{\Gamma \vdash \sigma_1 <: \tau_1 \qquad \ldots \qquad \Gamma \vdash \sigma_n <: \tau_n}{\Gamma \vdash \{l_1\colon \sigma_1, \ldots, l_n\colon \sigma_n, l_{n+1}\colon \sigma_{n+1}, \ldots\} <: \{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}}$$

Essentially one can either add extra fields to a record type or specialize the types of existing fields to get a subtype.

As we saw earlier, the problem with the original unrestricted system was that one could have subtypes that were structurally much more complicated than the supertype and one could exploit this in creating subtyping subproblems that grew infinitely in their complexity. We prevented this finally by restricting the use of *Top*, which is what allowed a subtype to be more complicated than the supertype in the first place. However, record types provide another way of allowing a subtype to be more complex than the supertype, namely by adding extra fields. So the empty record type provides an entity conceptually similar to *Top*. We now show that one can reproduce the non-terminating example mentioned in Pierce [Pie92], in spite of the restrictions on *Top* with the use of records.

Let $\neg\tau = \tau \rightarrow b$ where b is some simple type. Note that $\neg\sigma <: \neg\tau$ iff $\tau <: \sigma$.

Let

$$\theta = \{a \colon \forall\alpha <: \{\}.\neg\{a \colon \forall\beta <: \alpha.\neg\beta\}\}$$

Now consider the subtyping problem

$$\alpha_0 <: \theta \vdash \alpha_0 <: \{a \colon \forall\alpha_1 <: \alpha_0.\neg\alpha_1\}$$

This causes the following sequence of subproblems to be generated infinitely:

$$\begin{array}{rclcl}
\alpha_0 <: \theta & \vdash & \alpha_0 & <: & \{a \colon \forall\alpha_1 <: \alpha_0.\neg\alpha_1\} \\
\alpha_0 <: \theta & \vdash & \{a \colon \forall\alpha_1 <: \{\}.\neg\{a \colon \forall\alpha_2 <: \alpha_1.\neg\alpha_2\}\} & <: & \{a \colon \forall\alpha_1 <: \alpha_0.\neg\alpha_1\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \neg\{a \colon \forall\alpha_2 <: \alpha_1.\neg\alpha_2\} & <: & \neg\alpha_1 \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \alpha_1 & <: & \{a \colon \forall\alpha_2 <: \alpha_1.\neg\alpha_2\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \alpha_0 & <: & \{a \colon \forall\alpha_2 <: \alpha_1.\neg\alpha_2\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \{a \colon \forall\alpha_2 <: \{\}.\neg\{a \colon \forall\alpha_3 <: \alpha_2.\neg\alpha_3\}\} & <: & \{a \colon \forall\alpha_2 <: \alpha_1.\neg\alpha_2\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0, \alpha_2 <: \alpha_1 & \vdash & \neg\{a \colon \forall\alpha_3 <: \alpha_2.\neg\alpha_3\} & <: & \neg\alpha_2
\end{array}$$

and so on.

This pattern of non-termination is practically identical to the one displayed in [Pie92].

A.5.2 Union types

Union types pose problems similar to that of record types.

Union types are written as $\{l_1 \colon \tau_1 + \ldots + l_n \colon \tau_n\}$, with the usual meaning. The inference rule for union types is fairly straightforward.

$$(NUnion) \qquad \frac{\Gamma \vdash \sigma_1 <: \tau_1 \qquad \ldots \qquad \Gamma \vdash \sigma_n <: \tau_n}{\Gamma \vdash \{l_1 \colon \sigma_1 + \ldots + l_n \colon \sigma_n\} <: \{l_1 \colon \tau_1 + \ldots + l_n \colon \tau_n + \ldots\}}$$

The subtype is allowed to be a union over a subset of specializations of the types in the supertype.

Since the subtype can be a union of a fewer types than the supertype, one can have types with unions in contravariant positions thereby resulting in subtypes that have more complex structures than the supertype. Again, one can exploit this to generate an example of a subtyping problem that doesn't terminate. Behavior identical to that displayed in the previous example on record types arises for the subtyping problem

$$\alpha_0 <: \theta \vdash \alpha_0 <: \neg\{a : \neg(\forall \alpha_1 <: \alpha_0 . \neg \alpha_1)\}$$

where

$$\theta = \neg\{a : \neg(\forall \alpha <: \neg\{\}.\{a : \neg(\forall \beta <: \alpha . \neg \beta)\})\}$$

Appendix B

Syntax of the RAPIDE type system

type_expression ::=
  interface_type_expression
  | function_type_expression
  | type_constructor_application
  | '(' type_expression ')'
  | identifier
  | *FRAMEWORK_type_expression*

type_declaration ::=
  type identifier **is** type_expression ';'

interface_type_expression ::=
  interface
  { interface_declarative_region }
  end interface

interface_declarative_region ::=
  public_declarative_region
  | private_declarative_region
  | external_declarative_region
  | *FRAMEWORK_declarative_region*

public_declarative_region ::=
public
interface_declaration_list

private_declarative_region ::=
private
interface_declaration_list

external_declarative_region ::=
extern
interface_declaration_list

interface_declaration_list ::=
{ interface_declaration }
[*FRAMEWORK_interface_constraint*]

interface_declaration ::=
name_declaration
| interface_derivation_declaration

name_declaration ::=
type_name_declaration
| type_constructor_name_declaration
| object_name_declaration
| *FRAMEWORK_name_declaration*

type_name_declaration ::=
type identifier ';'
| **type** identifier '<:' type_expression ';'
| **type** identifier **is** type_expression ';'

type_constructor_name_declaration ::=
type identifier '(' [formal_parameter_declaration_list] ')' ';'
| **type** identifier '(' [formal_parameter_declaration_list] ')' '<:' type_expression ';'
| **type** identifier '(' [formal_parameter_declaration_list] ')' **is** type_expression ';'

object_name_declaration ::=
identifier_list ':' type_expression ';'

identifier_list ::=
identifier { ',' identifier }

interface_derivation_declaration ::=
include type_expression
[include_modifier identifier_list]
[**replace** replacement_list] ';'

include_modifier ::=
only
| **except**

replacement_list ::=
replacement { ',' replacement }

replacement ::=
identifier **to** identifier

function_type_expression ::=
function '(' [formal_parameter_declaration_list] ')' [**return** type_expression]
[*FRAMEWORK_function_constraint*]

formal_parameter_declaration_list ::=
formal_parameter_declaration { ';' formal_parameter_declaration }

formal_parameter_declaration ::=
formal_type_parameter_declaration
| formal_object_parameter_declaration

formal_type_parameter_declaration ::=
type identifier ['<:' type_expression]

formal_object_parameter_declaration ::=
identifier_list ':' type_expression [**is** *FRAMEWORK_object_expression*]

type_constructor_declaration ::=
  type identifier '(' [formal_parameter_declaration_list] ')'
    [*FRAMEWORK_type_constructor_constraint*]
   is type_expression ';'

type_constructor_expression ::=
  identifier
 | *FRAMEWORK_type_constructor_expression*

type_constructor_application ::=
  type_constructor_expression '(' [actual_parameter_list] ')'
 | *FRAMEWORK_type_constructor_declaration*

actual_parameter_list ::=
  actual_parameter { ',' actual_parameter }

actual_parameter ::=
  type_expression
 | *FRAMEWORK_object_expression*

Appendix C

Typing rules for the RAPIDE type calculus

C.1 Types

$$
\begin{array}{rll}
T ::= & a & (BAS) \\
| & T_1 \rightarrow T_2 & (FUN) \\
| & \{\ell_1 : T_1, \ldots, \ell_n : T_n\} & (RCD) \\
| & \forall t <: F(t).T & (UNV) \\
| & \exists t <: F(t).T & (EXS_1) \\
| & \exists t = F(t).T & (EXS_2) \\
| & \texttt{Top} & (TOP) \\
| & \texttt{fst}(e) & (EXS\text{-}E_1)
\end{array}
$$

C.2 Terms

$$
\begin{array}{rll}
e ::= & x & (VAR) \\
| & \lambda x{:}T.e & (FUN\!-\!I) \\
| & e_1 e_2 & (FUN\!-\!E) \\
| & \{\ell_1 = e_1, \ldots, \ell_n = e_n\} & (RCD\!-\!I) \\
| & e.\ell & (RCD\!-\!E) \\
| & \Lambda t <: F(t).e & (UNV\!-\!I) \\
| & eT & (UNV\!-\!E) \\
| & \langle \exists t <: F(t) = T, e \rangle & (EXS_1\!-\!I) \\
| & \langle \exists t = F(t) = T, e \rangle & (EXS_2\!-\!I) \\
| & \mathtt{snd}(e) & (EXS\!-\!E_2)
\end{array}
$$

C.3 Typing rules

$(TY\!-\!VAR)$
$$\Gamma, x{:}T \vdash x : T$$

$(TY\!-\!FUN\!-\!I)$
$$\frac{\Gamma, x{:}S \vdash e : T}{\Gamma \vdash \lambda x{:}S.e : S \to T}$$

$(TY\!-\!FUN\!-\!E)$
$$\frac{\Gamma \vdash f : S \to T \qquad \Gamma \vdash e : S}{\Gamma \vdash fe : T}$$

$(TY\!-\!RCD\!-\!I)$
$$\frac{\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\}}$$

$(TY\!-\!RCD\!-\!E)$
$$\frac{\Gamma \vdash e : \{\ell_1{:}T_1, \ldots, \ell_n{:}T_n\}}{\Gamma \vdash e.\ell_i : T_i} \quad i \in \{1 \ldots n\}$$

$(TY-UNV-I)$ $$\frac{\Gamma, t{<:}F(t) \vdash e : T}{\Gamma \vdash \Lambda t{<:}F(t).e : \forall t{<:}F(t).T}$$

$(TY-UNV-E)$ $$\frac{\Gamma \vdash e : \forall t{<:}F(t).T \qquad \Gamma \vdash S <: F(S)}{\Gamma \vdash eS : [S/t]T}$$

$(TY-EXS_1-I)$ $$\frac{\Gamma \vdash S <: F(S) \qquad \Gamma \vdash [S/t]e : [S/t]T}{\Gamma \vdash \langle \exists t{<:}F(t) = S, e\rangle : \exists t{<:}F(t).T}$$

$(TY-EXS_1-E_2)$ $$\frac{\Gamma \vdash e : \exists t{<:}F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

$(TY-EXS_2-I)$ $$\frac{\Gamma \vdash S = F(S) \qquad \Gamma \vdash [S/t]e : [S/t]T}{\Gamma \vdash \langle \exists t = F(t) = S, e\rangle : \exists t = F(t).T}$$

$(TY-EXS_2-E_2)$ $$\frac{\Gamma \vdash e : \exists t = F(t).T}{\Gamma \vdash \mathtt{snd}(e) : [\mathtt{fst}(e)/t]T}$$

$(TY-SUB)$ $$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e : T}$$

C.4 Subtyping rules

$(ST-REF)$ $$\frac{\Gamma \vdash T_1 \leftrightarrow T_2}{\Gamma \vdash T_1 <: T_2}$$

$(ST-TRA)$ $$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

$(ST-VAR_1)$ $$\Gamma, t{<:}T \vdash t <: T$$

$(ST\!-\!VAR_2)$
$$\Gamma, t = T \vdash t <: T$$

$(ST\!-\!FUN)$
$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

$(ST\!-\!RCD)$
$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \ldots \qquad \Gamma \vdash S_n <: T_n}{\Gamma \vdash \{\ell_1 : S_1, \ldots, \ell_n : S_n, \ldots\} <: \{\ell_1 : T_1, \ldots, \ell_n : T_n\}}$$

$(ST\!-\!UNV)$
$$\frac{\Gamma, t <: G(t) \vdash t <: F(t) \qquad \Gamma, t <: G(t) \vdash T_1 <: T_2}{\Gamma \vdash \forall t <: F(t).T_1 <: \forall t <: G(t).T_2}$$

$(ST\!-\!EXS_1)$
$$\frac{\Gamma, t <: F(t) \vdash t <: G(t) \qquad \Gamma, t <: F(t) \vdash T_1 <: T_2}{\Gamma \vdash \exists t <: F(t).T_1 <: \exists t <: G(t).T_2}$$

$(ST\!-\!EXS_2)$
$$\frac{\Gamma, t = F(t) \vdash t = G(t) \qquad \Gamma, t = F(t) \vdash T_1 <: T_2}{\Gamma \vdash \exists t = F(t).T_1 <: \exists t = G(t).T_2}$$

$(ST\!-\!TOP)$
$$\Gamma \vdash t <: \texttt{Top}$$

$(ST\!-\!EXS_1\!-\!E_1)$
$$\frac{\Gamma \vdash e : \exists t <: F(t).T}{\Gamma \vdash \texttt{fst}(e) <: F(\texttt{fst}(e))}$$

$(ST\!-\!EXS_2\!-\!E_1)$
$$\frac{\Gamma \vdash e : \exists t = F(t).T}{\Gamma \vdash \texttt{fst}(e) <: F(\texttt{fst}(e)) \Gamma \vdash F(\texttt{fst}(e)) <: \texttt{fst}(e)}$$

C.5 Reduction rules

(RR–FUN) $$(\lambda x\colon S.e)e' \longrightarrow [e'/x]e$$

(RR–RCD) $$\{\ell_1 = e_1, \ldots, \ell_n = e_n\}.\ell_i \longrightarrow e_i$$

(RR–UNV) $$(\Lambda t{<:}F(t).e)S \longrightarrow [S/t]e, (if\, S <: F(S))$$

(RR–EXS) $$\texttt{snd}(\langle \exists t{<:}F(t) = S, e\rangle) \longrightarrow [S/t]e$$

(EXS–Rule)

$$\Gamma \vdash [S/t]e : [S/t]T \;\&\; \Gamma \vdash S <: F(S) \Longrightarrow \Gamma \vdash [S/t]e : [\texttt{fst}(\langle \exists t{<:}F(t) = S, e\rangle)/t]T$$

Bibliography

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, Sendai, Japan, 1994.

[App92] Apple Computer Inc. *Dylan, an object-oriented dynamic language*, Nov 1992.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[Car91] Luca Cardelli. Extensible records in a pure calculus of subtyping. Manuscript, 1991.

[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.

[CG91] P.-L. Curien and G. Ghelli. Coherence of subsumption. In *Mathematical Structures in Computer Science*. 1991.

[CGL92] Castagna, Ghelli, and Longo. A calculus for overloaded functions with subtyping. In *1992 ACM Conf. Lisp and Functional Programming*, pages 182–192, 1992.

[CHC90] William Cook, Walt Hill, and Peter Canning. Inheritance is not subtyping. In *Proc. 17-th ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.

[CM89] L. Cardelli and J.C. Mitchell. Operations on records. In *Math. Foundations of Prog. Lang. Semantics*, 1989. Also available as DEC SRC Technical Report 48, August 1989, 60 pages.

[Coo89a] W.R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[Coo89b] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[Coo92] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

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

[ES90] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.

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

[Ghe93] G. Ghelli. post to the TYPES electronic mailing list, Feb 1993.

[GP94] Castagna Giuseppe and Benjamin Pierce. Decidable bounded quantification. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

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

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[Kat92] Dinesh Katiyar. Subtyping F-bounded types. In *ANSA Workshop on F-bounded quantification, Cambridge*, Sept 1992. Position paper.

[Kee89] S.E. Keene. *Object-oriented programming in Common Lisp.* Addison-Wesley, 1989.

[Ken93] John. J. Kenney. Banking on X/Open. To appear as a Stanford University Technical Report, 1993.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, pages 138–150, January 1994. Also printed as Stanford CSL Tech Report CSL-TR-94-603.

[KS92] D. Katiyar and S. Sankar. Completely bounded quantification is decidable. In *ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[Lan66] P.J. Landin. The next 700 programming languages. *CACM*, 9:157–166, 1966.

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 214–223, October 1986.

[LVB+93] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 1993.

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.

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

[MHF93] J. C. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 26–38, 1993.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12-th ACM Symp. on Principles of Programming Languages,* 1985.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, CMU, Pittsburgh, PA, December 1991.

[Pie92] Benjamin Pierce. Bounded quantification is undecidable. In *Proc. 19-th ACM Symp. on Principles of Programming Languages*, pages 305–315, 1992.

[PT93a] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[PT93b] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[REx] The PAVG group, Stanford University. *The Rapide Executable Language Reference Manual*.

[Rit91] M. Rittri. Using types as search keys in function libraries. *J. Functional Programming*, 1(1):71–90, 1991.

[RPr] The PAVG group, Stanford University. *The Rapide Predefined Types Reference Manual.*

[RSp] The PAVG group, Stanford University. *The Rapide Specification Language Reference Manual.*

[RT91] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *J. Functional Programming*, 1(2):191–212, 1991.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. 1-st ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA*, 1987.

[TC93] W. Tracz and L. Coglianese. An adaptable software architecture for integrated avionics. In *Proceedings of NAECON 93*, pages 1161–1168, Dayton, Ohio, May 1993. IEEE.

[Tof89] M. Tofte. Four lectures on Standard ML. Technical Report ECS-LFCS-89-73, Lab. for Foundations of C.S., University of Edinburgh, 1989.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

[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.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994. Published earlier as Rice University Technical Report TR91-160.