Structural Subtyping and the Notion of Power Type

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

Introduction

Many statically and dynamically typed languages attempt to achieve flexibility in their type discipline by some notion of *subtyping*. Subtyping relaxes the requirement that functions take arguments of a given type, by allowing arguments of any subtype of that type to be given. Such functions can accept arguments of types which will only be defined in the future; the open-ended character of this situation is important in large systems where it is essential to be able to extend existing facilities without modifying them.

Some languages attempt to use subtyping as a uniform structuring principle, noticeably pure object-oriented languages such as Smalltalk. Other languages are fairly consistent in their treatment of subtyping for some specific data types; this is the case, for example, for Simula67 subclasses and Pascal subranges. More commonly, ad-hoc notions of subtyping are introduced to relax type rules that seem too restrictive; this is the case for inclusions of basic types and notions such as sub-modules and sub-interfaces.

The purpose of this paper is to present a type system where subtyping is an orthogonal concept that applies to all type constructions, including function types, abstract types, and interfaces. Expressive power and feasibility of typechecking are our major concerns. Expressive power alone might point us to viewing subtypes as arbitrary subsets, but the desire to perform typechecking leads us to a more restricted, *structural*, notion of subtyping.

A type can be intuitively regarded as defining a set of values: the set of values having that type. It is then natural to consider the notion of *subtype* as analogous to the notion of subset: a type A is a subtype of a type B if all the values of type A are also values of type B.

Types are not intended as arbitrary sets of values, but as sets whose elements share a common *structure* (or *behavior*), with the property that the structure of a value (or the structure of a description of it) can help determine its type. For example, a pair is obviously related, because of its structure, to a cartesian product type, while a prime number is not so obviously related to the hypothetical type of prime numbers. Typechecking is often easier (e.g., it does not require theorem proving) if the types are *structural*, in this sense.

Similarly, subtypes should not be intended as arbitrary subsets. An arbitrary subset of a type corresponds to an arbitrary predicate, which for typechecking purposes may be very hard to test. As candidates for subtypes, we should look for *structural* subsets, i.e., for subsets that are determined by the structure of values (or their descriptions).

As an example, a simple notion of structural subtyping can be defined on record types: two record types are in a subtyping relation if one of them has more fields than the other, while the common fields have compatible types. Given a record value, it is possible to infer its most general record type, and to verify that that record is a member of any given subtype of that type.

In the type system presented in the next sections, typing and subtyping are determined strictly by structure, hence we diverge from the common programming language practice of matching types *by name* according to the name given to them in type declarations. Since pure name matching is never used in languages, it may be useful to study structural matching just to understand situations where a mixture of name and structural matching is used. It is also conceivable to add name matching rules to type systems based on structural matching, but many complex issues arise and we do not explore these here. The main advantage of pure structural matching is that types (including abstract types) have meaning independently of "when" they are generated; for example, their values can be stored and retrieved in distinct programming sessions.

We shall show how to extend the basic subtyping relations, such as the one among records, to *all* type constructors in certain languages. This program can be carried out in many type systems, for example [Cardelli 84] applied it to first-order λ-calculus, [Cardelli Wegner 85] to second-order λ-calculus, and [Wand 87] to polymorphic λ-calculus with implicit typing. Here we apply it to a more general type system derived from intuitionistic type theory [Martin-Löf 73] [Cardelli 86].

A novel notion of *power types*, analogous to powersets, is introduced to model subtyping in such a system. Combined with type quantifiers, power types can express *bounded* type quantification, leading to parametric inheritance and partially-abstract data types.

One of our types will be the type Type. We adopt here the property Type:Type (Type has itself type Type). There are both theoretical and practical reasons for *not* wanting this property (while still admitting Type as a type), and we discuss this later. For now, we simply remark that the Type:Type property simplifies the exposition of the the kind of type systems we are interested in, and provides a starting point for studying various systems which do not admit this property.

It is not feasible to illustrate all the different situations that can be modeled in our type system; we just remark that most of the examples in [Burstall Lampson 84] and [Cardelli Wegner 85] can be directly translated. In this paper we concentrate on examples involving dependent types and power types.

Notation and basic rules

Our type system is defined by *type inference* rules, describing how the type of an expression can be inferred from the types of its subexpressions. These rules do not define a typechecking algorithm, but are sufficient to specify (a superset of) what a typechecker should do; they are much easier to understand and less technology-dependent than any given algorithm.

The type inference rules are introduced in groups, one group for each type operator. The rules which are not connected to any specific type operator are described in this section.

Following the notation in [Harper Honsell Plotkin 87], S denotes *signatures* associating types to constants, E denotes *environments* associating types to variables, and the *judgement* $E \vdash_S a{:}A$ is read "we can deduce that the expression a has type A, in a signature S and an environment E".

The judgement $E \vdash_S A \leftrightarrow B$ is read "expressions A and B are definitionally (syntactically) equivalent". The exact rules for definitional equivalence depend on the language, but they should always guarantee that $\leftrightarrow$ is an equivalence relation which is also a congruence over the terms of the language.

The notation E,x:A stands for the environment E extended with the *assumption* that the variable x has type A. The notation S,k:A stands for the signature S extended with a constant k of type A. Finally, $B\{x \leftarrow a\}$ stands for the result of substituting the term a for the free occurrences of the variable x in the term B.

We need some well-formedness rules for signatures and environments: the judgement $\vdash S\ \text{sig}$ asserts that S is well formed, and the judgement $\vdash_S E\ \text{env}$ asserts that E is well formed in the signature S. Hence we have four basic judgements, to be defined by inference rules:

$\vdash S\ \text{sig}$	S is a well-formed signature
$\vdash_S E\ \text{env}$	E is a well-formed environment
$E \vdash_S a{:}A$	a has type A
$E \vdash_S A \leftrightarrow B$	A and B are equivalent

We do not need a subtyping judgement, since it is obtained as a special case of the typing judgement.

We briefly describe how the first two judgements are defined. The empty signature $\varnothing$ is well-formed; if S is well-formed and A is a type (i.e., $\varnothing \vdash_S A{:}\text{Type}$) then S,k:A is well-formed, provided k is not already defined in S. The empty environment $\varnothing$ is well-formed in any well-formed signature; if E is a well-formed environment in the signature S, and A is a type (i.e., $E \vdash_S A{:}Type$), then $E,x{:}A$ is well-formed in S, provided x is not already defined in E.

We always identify terms up to renaming of bound variables.

Our first typing rule is also the only one which applies to all terms, regardless of their syntactic shape. It asserts that if a value has a type A, and A can be shown to be equivalent to a type B, then that value also has type B (the horizontal line is read "implies").

Conversion

$$\frac{E \vdash_S a{:}A \quad E \vdash_S A \leftrightarrow B}{E \vdash_S a{:}B}$$

This rule says that equivalent types may replace each other as the type of a term. The precise definition of equivalence shall be discussed as we introduce type constructors. For the kind of type systems we are considering, equivalence normally includes equality up to typed lambda-conversion; hence the Conversion rule may imply that arbitrary computations can be carried out at the type level.

Each syntactic construct has its type rules. Here are the rule for the simplest constructs: constants and variables. The other constructs are treated in the next sections.

Given

$$\frac{\vdash_S E\ env \qquad k{:}A \in S}{E \vdash_S k{:}A}$$

Assumption

$$\frac{\vdash_S E\ env \qquad x{:}A \in E}{E \vdash_S x{:}A}$$

That is, if a constant (resp., variable) has a type in an signature (environment) then we can trivially deduce that that constant (variable) has that type.

Power types

Subtypes are intuitively analogous to subsets; *power types* can then be intuitively understood as powersets. If A is a type, then Power(A) is the type whose elements are all the subtypes of A; if B has type Power(A) then B is a subtype of A (written $B \subseteq A$, as an abbreviation of B:Power(A)). Our first task is to give formation, introduction, elimination and subtyping rules for Power (a similar task has to be carried out for all type constructors).

Power Formation

$$\frac{E \vdash_S A{:}Type}{E \vdash_S Power(A) : Type}$$

Power Introduction

$$\frac{E \vdash_S A{:}Type}{E \vdash_S A \subseteq A}$$

Power Elimination

$$\frac{E \vdash_S a{:}A \quad E \vdash_S A \subseteq B}{E \vdash_S a{:}B}$$

Power Subtyping

$$\frac{E \vdash_S A \subseteq B}{E \vdash_S Power(A) \subseteq Power(B)}$$

The formation rule asserts that if A is a type then Power(A) is also a type. (In general, formation rules prescribe how to construct a legal type.)

The introduction rule asserts that if A is a type, then A is a subtype of itself (i.e., A has type Power(A)). (In general, introduction rules prescribe how to create an object whose type is given by a type constructor.)

The elimination rule asserts that if a has type A and A is a subtype of B, then a has also type B. (In general, elimination rules prescribe how to use an object whose type is given by a type constructor.)

Finally, the subtyping rule asserts that if A is a subtype of B, then Power(A) is a subtype of Power(B), i.e., Power(A) has type Power(Power(B)). (In general, subtyping rules determine the subtypes of a given type constructor.)

The Power operator can be understood simply as a way of introducing subtyping into the system, but we shall see that new and interesting types can be expressed when combining Power with type quantifiers.

Variant types

Variant types are unordered, n-ary disjoint unions of types; the component types are indexed by distinct *tags* (or *labels*). An element of a variant type is a pair of a tag and a value of the type determined by that tag.

Variant Formation

$$\frac{E \vdash_S A_1\text{: Type} \quad ... \quad E \vdash_S A_n\text{: Type}}{E \vdash_S [t_1{:}A_1 \ ... \ t_n{:}A_n]\text{: Type}}$$

Variant Introduction

$$\frac{E \vdash_S a_i : A_i}{E \vdash_S [t_i{=}a_i] \text{ as } [t_1{:}A_1 \ ... \ t_n{:}A_n]: [t_1{:}A_1 \ ... \ t_n{:}A_n]} \qquad i \in 1..n$$

Variant Elimination (where $C = [t_1{:}A_1 \ ... \ t_n{:}A_n]$)

$$\frac{\begin{array}{c} E \vdash_S c{:}C \qquad E,z{:}C \vdash_S B : \text{Type} \\ E,x_i{:}A_i \vdash_S b_i{:}B\{z \leftarrow [t_i{=}x_i] \text{ as } C\} \qquad i \in 1..n \end{array}}{E \vdash_S \text{case}(z{:}C{=}c)B \mid [t_1{=}x_1]\ b_1 \ ... \mid [t_n{=}x_n]\ b_n : B\{z \leftarrow c\}}$$

Variant Subtyping

$$\frac{E \vdash_S A_1 \subseteq B_1 \quad ... \quad E \vdash_S A_n \subseteq B_n \quad ... \quad E \vdash_S B_m\text{: Type}}{E \vdash_S [t_1{:}A_1 \ ... \ t_n{:}A_n] \subseteq [t_1{:}B_1 \ ... \ t_n{:}B_n \ ... \ t_m{:}B_m]}$$

The formation rule constructs a variant type out of n (distinct) tags t_1 ... t_n and n types A_1 ... A_n. We identify variant types up to permutation of the tagged type components.

The introduction rule constructs a variant object $[t_i{=}a_i]$ as $[t_1{:}A_1 \ ... \ t_n{:}A_n]$ by labeling a term a_i with a tag t_i and specifying its full disjoint union type $[t_1{:}A_1 \ ... \ t_n{:}A_n]$ so that there is no ambiguity (the latter would normally be omitted if it can be inferred).

The elimination rule describes the typing of the case expression, which is used to inspect variant objects. If c has the form $[t_i{=}a_i]$ as C, then the branch b_i is executed with x_i bound to a_i. The type B is the result type of the case statement; it may have a free variable z which is bound to c. Normally the result type does not depend on the value being discriminated on, so case(z:C=c)B can be abbreviated as case c. We identify case expressions up to permutation of their branches.

The subtyping rule says that a variant of n components is also a variant of n+k components, if the corresponding components are in the subtyping relation.

The ↔ relation is extended with the typed computation rules for variant and case expressions.

Examples

It is a simple excercise to derive booleans and conditionals from variants.

Many examples will use *enumeration types*, which are a special case of variant types. If we assume a trivial type Ok, with a single constant ok:Ok, then we can take the enumeration type "[A, B, C]" as an abbreviation for "[A: Ok, B: Ok, C: Ok]", and "[A] as [A, B, C]" as an abbreviation for "[A = ok] as [A, B, C]".

Throughout the paper, we use *electronic video components* as sources of examples. The main parameters of a video component are the kind of signal it can receive and, in the case of a video cassette recorder, the tape format it can accept. These parameters are expressed as enumeration types. (All definitions have the form "def x:A = a" where x is a variable, A is a term of type Type and a is a term of type A; the type A may be omitted.)

```
def Signal: Type = [Ntsc, Pal, Secam]
def Format: Type = [Vhs, Beta, EightMm]
```

As a simple example of subtyping, we can define the type of European video signals:

```
def EuropeanSignal: Type = [Pal, Secam]
```

According to the variant subtyping rule, we have EuropeanSignal ⊆ Signal.

Record types

Record types have a natural subtyping relation, coming from object-oriented programming. Given a record type A, a subtype B of A can be formed by adding fields to A.

Normally (e.g., in Simula67) this is done by specifying only the additional fields of B and its dependency on A. This only provides for single inheritance (when the subtyping hierarchy on record types forms a tree) and has name-matching instead of structure matching at the base of type compatibility.

To achieve structural subtyping we assume that the fields of records and record types are unordered (and uniquely identified by their tags), and that type matching is performed field-wise by matching type components indexed by the same tag. This gives us a form of multiple inheritance since a record type can be a subtype of many (possibly incompatible) record types, namely all those types which structurally have fewer fields. This is formalized in the following rules.

Record Formation

$$\frac{E \vdash_S A_1\text{: Type} \quad \dots \quad E \vdash_S A_n\text{: Type}}{E \vdash_S \{t_1{:}A_1 \dots t_n{:}A_n\}\text{: Type}}$$

Record Introduction

$$\frac{E \vdash_S a_1\text{: } A_1 \quad \dots \quad E \vdash_S a_n\text{: } A_n}{E \vdash_S \{t_1{=}a_1 \dots t_n{=}a_n\} : \{t_1{:}A_1 \dots t_n{:}A_n\}}$$

Record Elimination

$$\frac{E \vdash_S r\text{: } \{t_1{:}A_1 \dots t_n{:}A_n\}}{E \vdash_S r.t_i\text{: } A_i} \qquad i \in 1..n$$

Record Subtyping

$$\frac{E \vdash_S A_1 \subseteq B_1 \quad \dots \quad E \vdash_S A_n \subseteq B_n \quad \dots \quad E \vdash_S A_m\text{: Type}}{E \vdash_S \{t_1{:}A_1 \dots t_n{:}A_n \dots t_m{:}A_m\} \subseteq \{t_1{:}B_1 \dots t_n{:}B_n\}}$$

The formation rule says how to make a record type out of n (distinct) tags $t_1 \dots t_n$ and n types $A_1 \dots A_n$. The introduction rule determines the type of a record object. The elimination rule describes how to extract a record field, with the usual *dot* notation. The subtyping rule says that a subtype of a record type can be obtained by adding fields, or by weakening the existing component types. We identify records and record types up to reordering of their tagged components.

It may seem surprising that a record with three components is a subtype (intuitively, subset) of a record with two components, instead of vice-versa. To make sense of this, one must think in terms of the set of values of a type. Every record with three fields *is* (i.e., can be used as) a record with two fields (and not vice-versa). Hence there are *more* records with two fields than records with three fields. Hence a set of records with three fields is a subset of a corresponding set of records with two fields.

The ↔ relation is extended with the typed computation rules for records and record selections.

Examples

We build a multiple inheritance hierarchy of video components. A generic video component has a signal type as its only attribute. A *video camera* has a signal type and a zoom capability. A *video recorder* has a signal type and a tape format. Finally, a *cam-corder* is both a camera and a recorder, and has the attributes of both.

```
def VideoComponent: Type =
   {signal: Signal}

def VideoCamera: Type =
   {signal: Signal,
    zoom: Boolean}

def VideoRecorder: Type =
   {signal: Signal,
    format: Format}

def CamCorder: Type =
   {signal: Signal,
    format: Format,
    zoom: Boolean}
```

Now we define a particular video component (a 25- inch tv set) that has Ntsc as its signal.

```
def myTv =
   {signal = [Ntsc] as Signal,
    inches = 25}
```

By subtyping, myTv is a VideoComponent. Note that no type of tv sets has been defined yet; this can be done now, and we obtain myTv: TvSet and TvSet⊆VideoComponent. Structural typing and subtyping permits this kind of after-the-fact definitions.

```
def TvSet: Type =
   {signal: Signal, inches: Integer}
```

Here is another video component which has type CamCorder and (by subtyping) VideoCamera, VideoRecorder, VideoComponent, and many others (but not TvSet).

```
def myCamCorder: CamCorder =
  {signal = [Ntsc] as Signal,
   format = [EightMm] as Format,
   zoom = True}
```

Dependent function types

The type All(x:A)B is the type of all functions mapping an element x of type A into an element of type B, where B may *depend* on (have free occurrences of) the variable x. In the non-dependent case, when x does not occur in B, we reduce to an ordinary function space which can be written as A→B. In case that A=Type, the dependent function type reduces to universal type quantification, written ∀x. B, which can model parametric polymorphism [Reynolds 85]. Full dependent types are strictly more powerful than these two special cases, since we can have types depending on values, as we shall see in the examples.

All Formation

$$\frac{E \vdash_S A{:}Type \qquad E, x{:}A \vdash_S B{:}Type}{E \vdash_S All(x{:}A)B : Type}$$

All Introduction

$$\frac{E \vdash_S A{:}Type \qquad E, x{:}A \vdash_S b{:}B}{E \vdash_S fun(x{:}A)b : All(x{:}A)B}$$

All Elimination

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S b{:}\ All(x{:}A)B}{E \vdash_S b(a){:}\ B\{x \leftarrow a\}}$$

All Subtyping

$$\frac{E \vdash_S A_0 \subseteq A \qquad E, x{:}A_0 \vdash_S B \subseteq B_0}{E \vdash_S All(x{:}A)B \subseteq All(x{:}A_0)B_0}$$

The introduction rule concerns functions, and the elimination rule concerns applications; note that the type of the result of an application depends on the value of the parameter.

The subtyping rule is a generalization of the ordinary contravariance rule for function spaces. A (dependent) function space is a subtype of another one if the domains are in the *inverse* inclusion relation, and the ranges are in the direct inclusion relation with an assumption about the free variable x.

The subtyping rule extends subtyping to higher-order function spaces. This integrates at the type level functional programming, which is based on higher-order functions, with object-oriented programming, which is based on first-order subtyping. This rule also defines subtyping for parametric polymorphic types, since we have seen that these are a special case of dependent types.

Combining dependent function types with the Power operator, we can express interesting new types such as All(A⊆B)A→A (which is an abbreviation for All(A:Power(B))A→A). This is the type of all functions which given as first argument any subtype of B and as second argument an object of that subtype, return a result of that subtype. One such function is fun(A⊆B) fun(a:A)a, the polymorphic identity over subtypes of B. The prefix All(A⊆B) is called a *bounded universal quantifier*.

A reason for choosing Power as the subtyping primitive, instead of ⊆, is that it allows us to introduce uniform abbreviations (such as the prefix fun(A⊆B) for fun(A:Power(B))) in all binding positions, without needing special binders for subtyping. Note however that the expressiveness of Power goes beyond ⊆ and bounded quantification, for example when Power appears in the result type of a function.

The ↔ relation is extended with typed β and η reductions.

Examples

A video tape has a format and a length. Instead of defining the type of video tapes directly, we define a function which given a format type returns the type of tapes of that format:

```
def TapeOfFormat: All(F⊆Format) Type =
  fun(F⊆Format) {format: F, length: Integer}
```

The type of all video tapes is simply TapeOfFormat(Format) ↔ {format: Format, length: Integer}, but we can be more specific and define the type of eight-millimeter video tapes only:

```
def EightMmTape: Type =
  TapeOfFormat([EightMm])
```

Then we can define a particular video tape:

```
def myTape: EightMmTape =
  {format = [EightMm] as [EightMm],
   length = 120}
```

The following function produces a type (a subtype of Format) out of a value (a VideoRecorder). Note that the result type of this function could be Type, but Power(Format) is more precise.

```
def FormatOfVcr:
  All(Vcr⊆VideoRecorder) Vcr → Power(Format) =
  fun(Vcr⊆VideoRecorder) fun(vcr: Vcr)
    case (f:Format=vcr.format) Power(Format)
    | [Vhs=x]  [Vhs]
    | [Beta=x]  [Beta]
    | [EightMm=x]  [EightMm]
```

We can now compose TapeOfFormat and FormatOfVcr to obtain the type of tapes which are compatible with a given vcr.

```
def TapeForVcr:
  All(Vcr⊆VideoRecorder) Vcr → Type =
  fun(Vcr⊆VideoRecorder) fun(vcr: Vcr)
    TapeOfFormat(FormatOfVcr(Vcr)(vcr))
```

For example, myCamCorder requires eight-millimeter tapes. Note the subtyping relations.

```
TapeForVcr(CamCorder)(myCamCorder)
   ↔ {format: [EightMm], length: Integer}
```

Finally, we can write a procedure which given a vcr and a tape of the correct format, "inserts" the tape in the vcr, by returning the pair of them:

```
def insertTape:
  All(Vcr⊆VideoRecorder) All(vcr: Vcr)
   All(tape: TapeForVcr(Vcr)(vcr))
     {vcr: Vcr, tape: TapeForVcr(Vcr)(vcr)} =
  fun(Vcr⊆VideoRecorder) fun(vcr: Vcr)
   fun(tape: TapeForVcr(Vcr)(vcr))
     {vcr = vcr, tape = tape};
```

For example, it is legal to use myTape in myCamCorder (but typechecking would fail when trying to insert the wrong kind of tape):

```
insertTape(CamCorder)(myCamCorder)(myTape)
  ↔ {vcr = myCamCorder, tape = myTape}
```

In these examples, some apparently *dynamic* relations which depend on object values (like "being a tape of the correct format"), are *statically* captured by the type system. A word of caution is due here. Dependent types can capture "more" situations statically, but not arbitrary situations. More complex semantic relations may be inexpressible and typechecking may fail. The point here is to show something which is not normally typecheckable, but one should not infer that everything is typecheckable. In particular, types in this system are still *not* first-class values, since there are no operations for inspecting their structure.

Dependent pair types

The type Some(x:A)B is the type of all pairs consisting of a left component x of type A and a right component of type B, where B may *depend* on the value of the left component (since it may have free occurrences of x). In the non-dependent case, when x does not occur in B, we have a simple cartesian product, written A×B. In case that A=Type, we have existential type quantification [Mitchell Plotkin 85], written ∃x. B, and which can model abstract types and interfaces. For example, Some(A:Type)A×(A→Int) is the type of a package providing a constant of type A and an operation of type A→Int over a hidden representation type A. Combining dependent function and dependent pair types, one can give account of various parametric module mechanisms [MacQueen 86].

Some Formation

$$\frac{E \vdash_S A{:}Type \quad E, x{:}A \vdash_S B{:}Type}{E \vdash_S Some(x{:}A)B : Type}$$

Some Introduction

$$\frac{E \vdash_S a{:}A \quad E \vdash_S b\{x\leftarrow a\}{:}B\{x\leftarrow a\}}{E \vdash_S pair(x{:}A{=}a)b{:}B : \ Some(x{:}A)B}$$

Some Elimination

$$\frac{\begin{array}{c}E \vdash_S c : Some(x{:}A)B \quad E, z{:}Some(x{:}A)B \vdash_S C{:}Type \\ E,x{:}A,y{:}B \vdash_S d{:}C\{z\leftarrow pair(x{:}A{=}x)y{:}B\}\end{array}}{E \vdash_S bind\ x,y{=}c\ in\ d \ : \ C\{z\leftarrow c\}}$$

Some Subtyping

$$\frac{E \vdash_S A \subseteq A_0 \quad E,x{:}A \vdash_S B \subseteq B_0}{E \vdash_S Some(x{:}A)B \subseteq Some(x{:}A_0)B_0}$$

The introduction rule concerns pairs: pair(x:A=a)b:B is the pair with left component a of type A and right component b of type B (written <a,b> in simple non-dependent situations), where the variable x of type A is bound to a and may appear in b and B.

The elimination rule concerns splitting pairs into their components: bind x,y=c in d splits the pair c and binds the components to the variables x,y which can be used in the scope d. The left and right projections of a pair can be easily defined.

The subtyping rule defines subtyping for cartesian products and abstract types as special cases. Combining dependent pair types with Power we can express *partially* abstract types: for example the interface Some(A⊆B) A×(A→Int) is the type of a package in which the representation type A is unknown, except that A is known to be a subtype of B.

The ↔ relation is extended with (the typed versions of) the reductions:

bind x,y = pair(z=a) b in d ↔ d{x←a, y←b{z←a}}

pair(z=bind x,y = c in x) bind x,y = c in y ↔ c

where z does not occur free in c.

Examples

A vcr gift is a package consisting of three items: the type of vcr contained in the package, a vcr of that type, and a tape of the correct format.

```
def VcrGift: Type =
  Some(Vcr⊆VideoRecorder) Some(vcr: Vcr)
    TapeForVcr(Vcr)(vcr)
```

For example, here is how to wrap myCamCorder and myTape as a gift:

```
def gift: VcrGift =
  pair(Vcr⊆VideoRecorder = CamCorder)
   pair(vcr: Vcr = myCamCorder)
    myTape
```

Recursive types

Recursion can be introduced by an operator rec(x:A)a (the variable x occurring in a is recursively identified with a) with a reduction rule rec(x:A)a ↔ a{x←rec(x:A)a} (given the appropriate type assumptions). Note that this operator can be used for building recursive values, and also recursive types when A=Type, taking advantage of Type:Type.

Rec

$$\frac{E \vdash_S A{:}Type \quad E, x{:}A \vdash_S a{:}A}{E \vdash_S rec(x{:}A)a : A}$$

This rule says how to build a recursive value or type. We do not need any other rules, since we can fold or unfold the recursion whenever needed, according to the reduction rule.

Type universe

Finally, we have to provide the rules for Type. We only need formation and subtyping:

Type Formation

$$\frac{\vdash_S E\ env}{E \vdash_S Type : Type}$$

Type Subtyping

$$\frac{E \vdash_S A{:}\mathrm{Type}}{E \vdash_S \mathrm{Power}(A) \subseteq \mathrm{Type}}$$

The formation rule says that Type is a member of itself. The subtyping rules says that if A is a type, then Power(A) is a subtype of Type (intuitively, the collection of subtypes of A is a subset of the collection of all types). In particular Power(Type) ⊆ Type, and we already had rules implying Power(Type) : Type and Type : Power(Type). The transitivity of ⊆ can be derived.

Discussion

We now discuss some of the troublesome aspects of our type system.

Undecidable typechecking

Our system is striving for expressiveness, which is achieved by providing a rich set of constructions at the type level. In fact, arbitrary computations, and in particular non-terminating computations, can be carried out at the type level because of the presence of a general recursion operator. (Even without general recursion, the Type:Type property is sufficient to express non-terminating computations [Girard 71].)

Recursion is necessary to express recursive types in the familiar ways. (Most such types can be expressed without recursion, but this leads to an unacceptably awkward programming style.) Hence one must choose between limiting the expressive power of the system, or living with possibly non-terminating typechecking. Without ruling out the viability of the former solution, we have chosen the latter, since virtually all common programming situations, and many uncommon ones, can be typechecked, using reasonably simple and efficient techniques [Cardelli 87].

The basic reason typechecking works in practice is that it is still fundamentally based on type structure. The situations in which typechecking diverges turn out to be either degenerate, or the result of using the full power of the system, beyond what is possible to typecheck in ordinary languages.

Kind and phase distinctions

The system presented here makes use of the controversial Type:Type property. This property presents theoretical problems [Meyer Reinhold 86] and practical implementation problems [Cardelli 87], sketched below.

The Type:Type property is perfectly acceptable in purely-applicative interpretive languages, such as languages for describing modular structure [Burstall 84]. None of the theoretical objections seem to apply to this class of applications, and the implementation problems are avoided by using interpretive techniques.

The basic problem of Type:Type is that it eliminates the distinctions between types and *kinds* (the types of types), hence introducing confusion into the system. Distinguishing between types and kinds is however not trivial; several interesting design decisions come up. For example, one has to decide whether All(A:Type) A→A is a type or a kind. In the former case we obtain proper polymorphic functions as values, in the latter case polymorphic functions must be fully instantiated before they can be used as values. Similarly, if Some(A:Type) A×(A→Int) is a type, we obtain proper abstract data types; if it is a kind, the instances of abstract types are second-class values, as it happens in most module systems.

Eliminating the distinctions between types and kinds through Type:Type, also indirectly confuses the distinctions between values and types. For example consider the program f = fun(A:Type) fun(x:A) x, and try and answer the question "is x a value?", or "is the result of f a value?". This cannot be determined, because the following applications are both legal: f(Integer)(3) and f(Type)(Integer).

This mixing of value and type levels becomes a considerable obstacle when considering compiled languages, or languages extended with imperative features, which must make a clear distinction between compile-time and run-time *phases*. In such cases, one should at least make a proper distinction between types and kinds, or otherwise *stratify* the system in order to eliminate Type:Type.

Unfortunately, the confusion between compile-time and run-time phases can be caused by dependent types alone, even without Type:Type. This is because of the All Elimination rule, where B{x←a} requires an arbitrary term (possibly a run-time value) to be substituted inside a (compile-time) type.

Hence, for compiled languages the main question becomes one of being able to make phase distinctions, rather than whether to abolish Type:Type.

Since we feel there is still much to be done in the way of introducing kind and phase distinctions, we have chosen to present a *kind-free* system, using the Type:Type property, which is simpler to present, and can be used as a paradigm for more refined systems.

Accidental matching

Since types match purely by structure, it is possible that some types match *by accident*. This may happen in a large programs, since the labels associated with records and variants have global scoping.

This is obviously a problem, but there are some natural solutions. If the values are really determined by their structure, then nothing bad can happen by accidental type matching. The only case in which accidental matching may lead to problems is when values maintain implicit invariants, not reflected in the type structure, which are violated by accidental type matching. But when values have hidden invariants, one should always build abstract types; then the accidental matching problem disappears (but see the discussion below on abstract types).

It may also seem strange that to organize a *concrete* inheritance hierarchy one has to choose the syntactic name of labels very carefully, so that the desired hierarchy will follow. Most object-oriented languages have declarative mechanisms for specifying hierarchy; these could be added to our language as syntactic sugar.

A unique property of our system is that we can set up *abstract* inheritance hierarchies, of the form Some(A:Type) Some(B$\subseteq$A) Some(C$\subseteq$A) ... , which can be implemented in different ways, using different labels in the implementation types. (In the present system, abstract multiple inheritance hierarchies cannot be expressed, because this would require a quantifier of the form Some(D$\subseteq$B$\cap$C), using a general type intersection operator).

Abstract types

We have used the term *abstract type* for types of the form P = Some(A:Type) B, because this models the concept of having an unknown type A which supports a set of operations of signature B.

It should be pointed out that, unlike abstract types in second-order lambda calculus [Mitchell Plotkin 85] this notion of type abstraction does not prevent *impersonation*. That is, given a particular implementation p = pair(A:Type = C) b:B of P, the representation type C is visible, hence one can build an object of type A without using the operations in b.

This problems can be partially solved by scoping techniques; e.g., a function which must operate on arbitrary implementations of P cannot make assumptions about any particular C. A complete solution to the impersonation problem requires introducing additional concepts [MacQueen 86].

On the positive side, the fact that even abstract types are matched by structure means that redefining an abstract type does not create a "new" one. This is convenient in interactive systems, which often reload definitions, and in systems which store abstract type instances across programming sessions.

Conclusions

We have presented a type system supporting a uniform notion of subtyping in a very general framework based on dependent types. In the present form, it can be used for purely functional, interpreted languages. Adaptations to compiled, imperative languages are being investigated.

We have worked in a very syntactic fashion, basing our system on syntactic reductions and inferences. No semantic models seem to be known, because of the difficulty of mixing recursive types, contravariance of function types, Type:Type, and Power. (Several models are known for systems with various subsets of these features.) Stratified versions of this type system may be easier to model. However it is not quite trivial to stratify the type system while preserving desirable programming properties, like having first-class polymorphic functions, abstract type instances, and modules.

Elsewhere we have investigated typechecking techniques for this type system, and we have built a prototype typechecker which performs quite satisfactorily on a selection of interesting programming examples, although in principle it may diverge.

References

[Burstall 84] R.M.Burstall: **Programming with modules as typed functional programming**, *International Conference on 5th Generation Computing Systems*, Tokyo, Nov 1984.

[Burstall Lampson 84] R.M.Burstall, B.Lampson: **A kernel language for abstract data types and modules**, in *Sematics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Cardelli 84] L.Cardelli: **A semantics of multiple inheritance**, in *Sematics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Eds., Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[Cardelli Wegner 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, *Computing Surveys*, Vol 17 n. 4, pp 471-522, Dec1985.

[Cardelli 86] L. Cardelli: **A polymorphic λ-calculus with Type:Type**, Technical Report n.10, DEC Systems Research Center, May 1986.

[Cardelli 87] L. Cardelli: **Typechecking dependent types and subtypes**, *Proc. of the Workshop on Foundations of Logic and Functional Programming*, Trento, Italy, Dec 15-19 1986, to appear.

[Girard 71] J.-Y.Girard: **Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types**, *Proceedings of the second Scandinavian logic symposium*, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Harper Honsell Plotkin 87] R.Harper, F.Honsell, G.Plotkin, **A framework for defining logics**, *Proc. Symposium on Logic in Computer Science*, Ithaca NY, June 22-25 1987, IEEE Computer Society Press, 1987.

[MacQueen 86] D.B.MacQueen: **Using dependent types to express modular structure**, *Proc. POPL 1986*.

[Martin-Löf 73] P.Martin-Löf, **An intuitionistic theory of types: predicative part**, in *Logic Colloquium III*, F.Rose, J.Sheperdson Eds., pp 73-118, North-Holland, 1973.

[Meyer Reinhold 86] A.R.Meyer, M.B.Reinhold: **'Type' is not a type: preliminary report**, *Proc. POPL 1986*.

[Mitchell Plotkin 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, *Proc. POPL 1985*.

[Reynolds 85] J.C.Reynolds: **Three approaches to type structure**, *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science n.185, Springer-Verlag, Berlin 1985, pp. 97-138.

[Wand 87] M.Wand, **Complete type inference for simple objects**, *Proc. Symposium on Logic in Computer Science*, Ithaca NY, June 22-25 1987, IEEE Computer Society Press, 1987.