Typechecking Dependent Types and Subtypes

Luca Cardelli

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

1. Introduction

This paper is pragmatic in nature and describes some techniques and heuristics used in a prototype typechecker for dependent types and subtypes. These are to be used in the implementation of a language based on such features. Some care has been taken in formalizing the type system, but not yet in formalizing the algorithms involved.

In this paper we adopt the Type:Type rule (i.e. Type has type Type) [Cardelli 86]. This rule leads to undecidable type systems, but since we deal with undecidability anyway (because of recursive types) the presence or absence of this rule is a minor point in our discussion of typechecking algorithms. On the other hand, the presence of this rule simplifies the presentation of the type system.

Eventually, we want to disallow the Type:Type rule and obtain a *stratified* type system, maybe along the lines of [Martin-Löf 86]. Here we present the *unstratified* type system, which has some intrinsic interest.

2. Dependent types

Our basic language has universal and existential quantifiers [Martin-Löf 73], and the type of all types. Here is the syntax of expressions, where x and y are (distinct) identifiers, and a, b, A, and B are expressions:

x	identifiers
Type	the type of all types
All(x: A) B	dependent function types
fun(x: A) b	(dependent) functions
a(b)	function application
Some(x: A) B	dependent pair types
pair(x: A = a) b: B	(dependent) pairs
bind x,y = a in b	pair inspection

The scoping rules (i.e. the definitions of free and bound variables) are as follows. The binding occurrences of x in All, fun, Some and pair bind the occurrences of x in B and b, but not in A and a. The binding occurrences of x and y in bind bind the occurrences of x and y in b, but not in a. Then, α-conversion (i.e. renaming of bound variables) and substitution (b{x←a} means substituting a for all the free occurrences of x in b) are defined as usual. Terms are identified up to α-conversion.

Computation rules are given by the following conversion rules (the necessary type assumptions have been omitted, for brevity), together with the rules needed to make conversion into a congruence relation.

β (fun(x:A) b)(a) ↔ b{x←a}
η fun(x:A) b(x) ↔ b
σ bind x,y = (pair(z: A = a) b: B) in c ↔ c{x←a}{y←b{z←a}}
π pair(z: A = (bind x,y = c in x)) (bind x',y' = c in y'): B ↔ c

where x is not a free variable of b in η, and z is not a free variable of c in π.

The type rules are given in the form of a type inference system. One *conversion* rule accounts for computation during typechecking (following the notation in [Harper 87], S is a signature providing types for constants; E is an environment associating types to variables; $\vdash_S E\ env$ means that E is well formed; $E \vdash_S a:A$ means that we can deduce that a has type A in the signature S and environment E):

Conversion $\dfrac{E \vdash_S a:A \quad E \vdash_S B:Type \quad A \leftrightarrow B}{E \vdash_S a:B}$

The basic typing rules follow the syntactic structure:

Assumption

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

Type Formation

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

All Formation

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

All Introduction

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

All Elimination

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

Some Formation

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

Some Introduction

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

Some Elimination

$$\frac{E \vdash_S c : \text{Some}(x{:}A)\ B \qquad E, x{:}A, y{:}B \vdash_S d{:}C\{z \leftarrow \text{pair}(x{:}A{=}x)y{:}B\}}{E \vdash_S \text{bind}\ x,y{=}c\ \text{in}\ d\ :\ C\{z \leftarrow c\}}$$

3. Typechecking dependent types

The above inference system can be used as the basis for a typechecking algorithm. This particular type system turns out to be undecidable, because of the Type:Type rule. This means that the typechecking process might diverge. However, examples leading to divergence are extremely hard to reproduce (being based on Girard's paradox [Girard 71]), and do not have any impact on programming practice.

A typechecking algorithm computes a type for a term e given an environment E defining the type of all the free variables in the term. In the above type system, typechecking is fairly straightforward since the terms are rich in type information. We can distinguish three activities in typechecking: inference, matching and reduction, which are intertwined.

An *inference* step consists in selecting a type rule which conforms to the structure of the term, checking that the assumptions of that rule are verified, and constructing the type of the term, as prescribed by the rule.

In checking that the assumptions are verified, we may have to *match* two types, when a type rule has two occurrences of the same meta-variable. In order to match two types, we *reduce* them both to normal form (if possible) and compare the results up to α-conversion. The matching routine is also responsible for η-expansions and π-expansions, when needed.

Reductions to normal form are performed as prescribed by the conversion rules, but since the conversion rules are typed, the reduction process may require further inference and matching. The reduction step might fail to terminate because the existence of normal forms is not guaranteed.

This completes our sketch of the basic typechecking algorithm.

4. Redundant type parameters

Our expressions contain much redundant type information. From a pragmatic point of view, some of this redundancy is welcomed. For example, it is good documentation practice to express the type of all the parameters and results of a function, even when some of this information can be inferred.

However, some type information seems just to make programs more cumbersome and harder to read, especially in languages with explicit polymorphism [Reynolds 85] where one must provide type parameters to polymorphic functions in order to instantiate them to the desired type.

Some languages (noticeably ML [Milner 84]) have attempted to provide the

largest possible amount of type inference, through implicit polymorphic typing: type quantifiers are always assumed to operated at the top level, and are not allowed in the syntax. This approach, although appealing and largely successful, has some difficulties. First, it may encourage cryptic programming styles, when too much type information is omitted. Second, it has problems in dealing with side effects which are ultimately due to the fact the untyped polymorphic programs are ambiguous with respect to their possible explicit typings (regarding the position of type quantifiers in such typings). Finally, some programs can be typed with explicit typing which are not typeable with implicit typing, namely the ones requiring type quantifiers nested within other type operators.

We are now going to describe a compromise between implicit and explicit typing. We adopt an explicit polymorphism framework, and we require all function parameters to be typed (in a language implementation of these ideas we would also require function results to be typed). However, we then allow one to omit applications of type parameters, when such omitted parameters are followed by other parameters which determine them uniquely. The missing information is then recovered using unification techniques, similar to the ones in the ML typechecker [Milner 78].

(One of the major problems of the unstratified system, from a pragmatic point of view, is that it is not possible to decide whether a given subexpression is a type, in the following sense. Let f be fun(A:Type) fun(a:A) a; it would appear that a does not have type Type, and in the application f(Int)(3), a is in fact bound to a non-type. But in the application f(Type)(Int), a is bound to a type. The following discussion applies strictly to a stratified systems, and only heuristically to unstratified systems.)

Suppose we have a term:

```
f : All(A: Type) All (a: A) A
```

Then, whenever we apply f to an argument a, we first have to supply the type A of a. Such a type parameter is *redundant* as it could be easily deduce from the type of a, i.e. we want to write:

```
f(a)          (instead of   f(A)(a))
```

The general situation looks as follows. Given a term f whose type has the form: ...All(A: Type) ... All(a: B(A)) C, and given an application of f of the form f...(X)...(x)..., where X:Type corresponds to A, and x:B(X) corresponds to a, then the parameter X may be redundant (note that X is not redundant in f...(X)).

Type parameters can only be omitted when their omission does not create ambiguities as to which parameters are missing. Since typechecking normally proceeds left-to-right, we only allow the omission of the final part of a sequence of type parameters. In a term of the form $f...(X_1)...(X_n)(x)...$, where $X_1...X_n$ have type Type while x does not have type Type, only subsequences of the form $(X_i)...(X_n)$ can be omitted.

Example: consider the following (schematic) definitions:

```
T = All(A: Type) All(a: B(A)) C(A)(a)
    where B(X) ≠ Type for any X
f: T = fun(A: Type) fun(a: B(A)) c
f(b)          where b:B(D)
```

When examining f(b) from left to right, first f is assigned the type T. Then a type parameter is expected; b is typechecked but it has type B(D) ≠ Type, hence there is a missing type parameter, and f is assigned the type All(a: B(α)) C(α)(a), where α (a new type variable) replaces the missing type parameter. Then a parameter of type B(α) is expected, and one of type B(D) was found. To make sure they are compatible, they are (simplified and) matched by simple unification (first-order unification is sufficient as α:Type cannot appear in function position). In this case the unification succeeds with α = D, hence D was the missing type parameter, and the type of f(b) is C(D)(b).

If a missing type parameter remains undetermined, a free type variable may appear in the final type of the expression. This can be interpreted as an error condition, or it can be interpreted as a yet-undetermined situation which may become fully determined later on.

5. Recursion

Recursion is one of the features we have to add to our language to make it more like a programming languages. This is both for expressive power (the language without Type:Type and without recursion can only express total functions, although a surprising variety of them), and for programming

convenience (it is easier to program recursively than iteratively). We add expressions of the form:

rec(x: A) b recursion

which allow the definition of recursive values (typically functions) and recursive types (when A is Type). The conversion rule is (again, omitting the type assumptions):

μ rec(x:A) a = a{x← rec(x:A) a}

The only type rule we need for recursion is:

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

The introduction of recursion causes the type system to become undecidable, even if we omit the Type:Type rule. Unlike Type:Type undecidability, rec undecidability is pervasive. This is very disconcerting since one of the most intuitive properties of ordinary compilers is termination. Hence, we have to improve our typechecker to take divergence into account and to try and avoid it when possible.

Since a rec term reduces to a term containing a copy of itself, reduction of terms containing rec never produces normal forms. This means that (according to an obvious extension of our old algorithm based on inference, matching and reduction) all the typechecking involving recursive types will diverge.

It turns out that virtually all recursive type declarations which occur in practical programs are *regular* (i.e. the infinite expansion of such recursively defined types have a finite number of distinct subtrees). It is then natural to build a simple loop recognizer into the typechecking algorithm, which causes termination when analyzing regular recursion. (One could even try to impose syntactic restrictions which enforce regular type definitions, but we fear that this might be too restrictive. Much of the appeal of dependent type systems is in the ability to perform non-trivial computations on types, even if such computations always produce regular types.)

First we have to abandon the idea of reducing type terms to normal form during matching, because this will diverge too often. Instead, we adopt a more "incremental" approach interleaving reduction to *head* normal form with matching of the heads. This will still diverge in exactly the same situations, but now we can introduce new techniques.

The most straightforward technique consists in maintaining a pair of stacks (called the *left* and *right* stack) during typechecking. Matching now takes a *left* term and a *right* term, and handles recursion so that subterms of left terms always appear as left terms, and similarly for right terms.

First, we regard rec terms as terms in head normal form, so that the reduction process will not try to expand them. Whenever we have to match a (e.g. left) rec term against another (e.g. right) term, we search the left and right stacks, respectively, for occurrences of such terms at the same depth (using a pointer equality test). If found, the matching succeeds. Otherwise, the terms are pushed onto the left and right stacks respectively, the rec term is expanded, and the matching continues. Symmetrically for a right rec term.

This simple scheme seems to be extremely well behaved in programming situations. The cases in which it fails are (1) matching types which are intrinsically non-regular (they virtually never arise), and (2) matching recursive regular types which are expressed in non-straightforward ways.

Here is an example of the second class: consider the following two definitions of recursive parametric lists (using records and variants, discussed in the next section):

```
List1 = fun(A:Type) rec(B:Type) [nil: {}, cons: {head: A, tail: B}]

List2 = rec(B:Type→Type) fun(A:Type)  [nil: {}, cons: {head: A, tail: B(A)}]
```

It is not clear whether List1 and List2 should be considered the same type, but assume we expect them to match. Matching List1 against itself, or List2 against itself succeeds, because of pointer equality tests at the beginning of matching. Matching List1 against a copy of itself, succeeds, because the same subterms are encountered after unrolling (according to the stack technique). Matching List2 against a copy of itself diverges, because the unrolling of the recursion creates new terms (because of the application B(A)) and simple pointer equality fails; replacing pointer equality by more sophisticated tests might still succeed here, and slow down typechecking considerably. Finally, successfully matching List1

against List2 seems very hard, because the recursions are out of step and this causes divergency in the algorithm.

6. Records and Variants

Records (labeled, unordered cartesian products) and variants (labeled, unordered disjoint sums) are now introduced. They are useful on their own right, but they are introduced here mostly because they are the basic types on which subtyping is built.

We add expressions of the form (where t are *labels*):

$\{t_1:A_1, \ldots, t_n:A_n\}$	record types
$\{t_1=a_1, \ldots, t_n=a_n\}$	records
$a.t$	selection
$[t_1:A_1, \ldots, t_n:A_n]$	variant types
$[t=a]$	variants
case z=c giving B $\mid t_1(x_1:A_1)b_1 \ldots \mid t_n(x_n:A_n)b_n$	case

where all the t_i appearing in a record type, record, variant type or case are distinct. These expressions are identified up to reordering of record fields, record type components, case branches and variant type components.

The case expression above discriminates over the c term, a variant whose tag t_i determines the branch b_i to be executed, and whose contents are bound to x_i in that branch. The scoping of z is restricted to B, which is the result type.

Here are the typing rules for the above constructs.

Record Formation

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

Record Introduction

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

Record Elimination

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

Variant Formation

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

Variant Introduction

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

Variant Elimination

$$\frac{E \vdash_S A_1{:}\text{Type} \quad E \vdash_S A_n{:}\text{Type} \qquad E,x_1{:}A_1 \vdash_S b_1{:}B\{z \leftarrow [t_1{=}x_1]\} \quad \ldots \quad E,x_n{:}A_n \vdash_S b_n{:}B\{z \leftarrow [t_n{=}x_n]\}}{E \vdash_S \text{case } z{=}c \text{ giving } B \mid t_1(x_1{:}A_1)b_1 \ \ldots \ \mid t_n(x_n{:}A_n)b_n : B\{z \leftarrow c\}}$$

7. Subtypes

Subtypes are introduced by the Power operator: if B is a type, then Power(B) is intended to be the type of all subtypes of B. Hence, if A:Power(B) then A is a subtype of B; this is also written $A \subseteq B$.

$A \subseteq B$ abbreviates A:Power(B)

Power Formation

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

Power Introduction

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

Power Elimination

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

In addition to the formation, introduction and elimination rules, Power has rules relating it to each of the other type operators. Each of these additional rules determines the meaning of subtyping for that type operator.

Power Type

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

Power All

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

Power Some

$$\frac{E \vdash_S A \subseteq A' \qquad E,x:A \vdash_S B \subseteq B'}{E \vdash_S \text{Some}(x:A)\ B \subseteq \text{Some}(x:A')\ B'}$$

Power Record

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

Power Variant

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

Power Power

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

The [Power Type] and [Power Power] rules only apply to the unstratified system. The [Power Record] rule captures the essence of *multiple inheritance*; a record type (called a *class* in object oriented programming) is a subtype (*subclass*) of another record type if it has more components (*properties*), and if the common components are in subtype relation. The [Power All] rule (which generalizes the controvariant rule of ordinary function spaces with respect to inclusion) extends multiple inheritance from object types to higher-order types , and effectively integrates object oriented programming with functional programming.

8. Typechecking subtypes

Subtyping introduces a number of ambiguities in the type system; at the

moment we have heuristic solutions which appear to work well in practice. The worse kind of ambiguities are only present in the unstratified system.

One basic ambiguity regards the type of a type. If A is a type, should its type be Type or Power(A) (according to [Power Introduction])? If the algorithm gives one of the two by default, it should always be ready to change its interpretation when needed. For example, in determining the type of fun(B:Type) fun(A⊆B) fun(a:A) a, one of the steps requires A:Type, but the environment provides A:Power(B). One solution is to replace tests of the kind *type*(A)=Type normally used during typechecking by tests of the kind *type*(A)⊆Type, since Type ⊆Type, and Power(A) ⊆Type if A:Type.

Another ambiguity comes from transitivity of inclusion (the transitivity of ⊆ is a derived rule in the unstratified system). In testing A ⊆ B in situations where A ⊆ A', one option is to test A' ⊆ B.

A final ambiguity comes up in trying to determine whether Power(A) ⊆ Power(B) (terms like Power(Power(B)) are only allowed in the unstratified version of the system). One can either use [Power Power] and attempt to determine A ⊆ B, or use [Power Introduction] and attempt to determine Power(A) = B, or use transitivity of ⊆ and attempt to determine Power(A) ⊆ B. It is however safe to assume that situations of the kind Power(Power(B)) do not happen in practice.

When these ambiguities are resolved, another basic problem appears. So far we have expressed the typing and subtyping rules separately, but for typechecking purposes it is necessary to merge the typing and subtyping rules for the same type constructor into a single rule (or set of rules), so that the typechecker knows what to do for each constructor.

The basic technique for merging type rules with subtyping rules consists in relaxing all the assumptions of a given typing rule by [Power Elimination] and (using the subtyping rules) see what derived rule one obtains. For example, for [All Elimination] and [Record Elimination] one obtains the derived rules:

All Elimination'

$$\frac{E \vdash_S a:A' \quad E \vdash_S A' \subseteq A \quad E \vdash_S b: \mathrm{All}(x:A)\ B}{E \vdash_S b(a): B\{x \leftarrow a\}}$$

Record Elimination'

$$\frac{E \vdash_S r:B \quad E \vdash_S B \subseteq \{t:A\}}{E \vdash_S r.t: A}$$

As these derived rules show, in typechecking the system with subtypes the *matching* routine must sometimes be used along with an an *inclusion* routine. The latter is very similar in structure to the matching routine in the treatment of recursion, unification, etc., but follows the basic and derived subtyping rules. The recursive calls inside the subtyping routine may run into terms which are not types; subtyping then reverts to ordinary matching.

9. Conclusions

We have presented a type system based on dependent types and subtypes, and sketched some typechecking techniques which make it viable as a type system for practical programming languages. Its expressive power allows it to model a great variety of advanced programming concepts, like polymorphism, abstract types, parametric modules and multiple inheritance [Mitchell 85] [Burstall 84] [Cardelli 85].

A prototype typechecker has been built incorporating dependent types, subtypes, recursive types and limited type inference. Much has to be done to clean up and formalize the typechecking algorithm, but the current experiment is very encouraging. The kernel of the algorithm is based on a typed version of well-know lazy-evaluation techniques developed for the λ-calculus [Aiello 81]. The performance is comparable to unification-based polymorphic typecheckers. We believe side-effecting operations can be embedded in this framework, but they require stratification in the type system to keep impure behaviour confined at "run-time".

There are no known mathematical models for the type system presented here; the consistency question is open.

References

[Aiello 81] L.Aiello, G.Prini: **An Efficient Interpreter for the Lambda-Calculus**, Journal of Computer and System Sciences, 23,3 pp. 384-424, 1981.

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

[Cardelli 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, Technical Report No. CS-85-14, Brown University.

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

[Coquand 85] T.Coquand, G.Huet: **Constructions: a higher order proof system for mechanizing mathematics**, Technical report 401, INRIA, May 1985.

[Girard 71] J-Y.Girard: **Une extension de l'interprŽtation de Gšdel a l'analyse, et son application ˆ l'Žlimination des coupures dans l'analyse et la thŽorie des types**, *Proceedings of the second Scandinavian logic symposium*, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Harper 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.

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

[Martin-Löf 86] P.Martin-Löf, **The type structure of intuitionistic type theory**, lecture at the workshop on *Foundations of Logic and Functional Programming*, Trento, Italy, Dec. 1986.

[Milner 78] R.Milner: **A theory of type polymorphism in programming**, *Journal of Computer and System Science 17*, pp. 348-375, 1978.

[Milner 84] R.Milner: **A proposal for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp. 184-197. ACM, New York.

[Mitchell 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 185, Springer-Verlag, Berlin 1985, pp. 97-138.