Phase Distinctions in Type Theory

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301
January 3, 1988

Introduction

Type systems were originally introduced in programming languages to provide a degree of *static* checking, achieved through typechecking. As type systems become more complex and typechecking more sophisticated, the attribute *static* becomes less appropriate. The situation is better described by thinking that the execution of a program is carried out in two *phases*: a typechecking phase (*compile-time*) and an execution phase (*run time*).[1]

Normally one would expect the very notion of *type system* to enforce this phase distinction. This is true for all simple type systems found in programming languages (e.g. Pascal), which are usually variations on the type system of first-order typed λ-calculus. This phase distinction remains when generalizing to second-order typed λ-calculus, which can be used to model parametric polymorphism (e.g. in ML [Milner 84]) and abstract types [Mitchell 85].

In all these languages, phases can be distinguished syntactically: there are separate syntactic sorts of type expressions and value expressions. Phase distinction are however lost when moving to languages like Pebble [Burstall 84a] based on *dependent types* [Martin-Löf 73]. A dependent type is a type which may *depend* upon the value of some expression. This dependency directly causes phase mixing, unless one is careful to distinguish between *compile-time values* (e.g. given by constant expressions in Pascal), and *run-time values*.

[1]In compilers, typechecking is usually done at *compile-time*, while execution is done at *run-time*. In interpreters, both activities are normally done at *run-time* (i.e., there is no static checking). Notice however that one can build compilers for untyped languages, interpreters performing static typechecking, and even interactive compilers which alternate the two activities.

The loss of phase distinctions manifests itself as the inability to perform compilation, since compilers are based on a syntactic translation phase which strips out type information, followed by a type-free execution phase.

In this paper we present a *phase-free* system based on dependent types, and then we modify it to obtain a *phased* system, which can be used as the basis of a compilable programming language.

A phase-free type system

The type of a type (e.g. Type) or of a type operator (e.g. Type→Type) is called a *kind*. Systems which admit Type:Type effectively erase all kind distinctions, and are called kind-free. Kind-free type systems make it hard to distinguish between execution phases: assume A:Type and B:A; is B a compile-time or a run-time entity? We could have A=Int and B=3, or we could have A=Type (using Type:Type) and B=Int. Hence we cannot decide when to evaluate B, i.e. we cannot decide whether to generate machine code for B to be executed at run time.

Building a compiler for a language with dependent types requires designing a *phased* type system. This will be achieved (1) by introducing kind distinctions (hence rejecting Type:Type), and (2) by further restricting dependent types to prevent phase mixing. Of course, we want to retain as much of the power of dependent types as possible; in particular, we do not want to fall all the way back to second-order λ-calculus.

In this section we consider a very expressive kind-free type system, which will serve as a utopic goal; later we will introduce restrictions.

Here we have only one main kind of typing judgements:

$$E \vdash_S a : A$$

meaning that in the signature S (a function from constants to type terms) and in the environment E (a function from variables to type terms), we can deduce that a has type A. We also need two auxiliary judgements for building well-formed signatures and environments:

$\vdash S \quad \text{sig}$

$\vdash_S E \ \ \text{env}$

meaning that S is a well-formed signature, and that E is a well-formed environment over the signature S. (The distinction between signatures and environments [Harper 87] emphasized the distinction between "built-in" language features and "user-definable" ones).

Signature Construction

$$\frac{}{\vdash \emptyset \;\; \mathrm{sig}}$$

$$\frac{\vdash S \; \mathrm{sig} \qquad \emptyset \vdash_S A{:}\mathrm{Type} \qquad c \notin \mathrm{Dom}(S)}{\vdash S,\; c{:}A \;\; \mathrm{sig}}$$

Environment Construction

$$\frac{\vdash S \;\; \mathrm{sig}}{\vdash_S \emptyset \;\; \mathrm{env}}$$

$$\frac{\vdash_S E \; \mathrm{env} \qquad E \vdash_S A{:}\mathrm{Type} \qquad x \notin \mathrm{Dom}(E)}{\vdash_S E,\; x{:}A \;\; \mathrm{env}}$$

Given

$$\frac{\vdash_S E \; \mathrm{env} \qquad c{:}A \in S}{E \vdash_S c{:}A}$$

Assumption

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

Type Formation

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

All Formation

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

All Introduction

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

All Elimination

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

Rec Formation

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

Conversion

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S B{:}\mathrm{Type} \qquad A =_{\beta\eta\mu} B}{E \vdash_S a{:}B}$$

This small set of rules centered on *universal* types (the All quantifier) captures many programming concepts. In the following examples, we work in a signature including common basic types, values, value operators and conditionals (sugared in the usual way). In this language we can define type operators, such as the (infix) function-space operator and an operator generating endomorphism types (the construct let x:A = a introduces a new variable x, of given type A and value a):

```
let → : All(A:Type) All(B:Type) Type =
      fun(A:Type) fun(B:Type) All(x:A) B

let Endo: Type→Type =
      fun(A:Type) A→A
```

We have ordinary (first-order, monomorphic) functions:

```
let succ: Endo(Int)  =
       fun(a:Int) a + 1;
```

and higher-order monomorphic functions:

```
let shift: Endo(Endo(Int)) =
      fun(f: Endo(Int)) fun(a: Int) f(a-1)
```

Then we have polymorphic functions:

```
let PolyEndo: Type =
      All(A:Type) Endo(A)

let id: PolyEndo =
      fun(A:Type) fun(a:A) a

let twice: Endo(PolyEndo) =
      fun(f: PolyEndo) fun(A:Type) fun(a: A) f(A)(f(A)(a))
```

On top of this we can express types dependent on values; Prop(n) is the type of propositional functions of n variables, such that Prop(0) = Bool and Prop(n+1) = Bool→Prop(n):

```
let Prop: All(n: Nat) Type =
      rec(f: All(n: Nat) Type)
            fun(n: Nat) case n of 0 ⇒ Bool; succ(m) ⇒ Bool→f(m);

let and2: Prop(2) =
      fun(a: Bool) fun(b: Bool) a ∧ b;
let and3: Prop(3) =
      fun(a: Bool) fun(b: Bool) fun(c: Bool) a ∧ b ∧ c;
```

This is the SASL tautology function [Turner 76], testing whether a propositional function of an arbitrary number of variables is a tautology:

```
let Taut: All(n: Nat) Prop(n)→Bool =
    rec(T: All(n: Nat) Prop(n)→Bool)
        fun(n: Nat) fun(p: Prop(n))
            case n of 0 ⇒ p; succ(m) ⇒ T(m)(p(true)) ∧ T(m)(p(false));

Taut(3)(and3);
```

Much of this flexibility comes from a combination of dependent types and the Type:Type property. Both will have to be restricted to obtain a phased system.

A phased type system

In this section we introduce phase distinctions in the phase-free type system. The key idea is to distinguish three levels: values, type families (types and type operators) and kinds. Our main judgement ($E \vdash_S a{:}A$) now becomes three:

$E \vdash_S K\ \text{kind}$	K is a kind
$E \vdash_S A{::}K$	A is a type family (possessing a kind)
$E \vdash_S a{:}A$	a is a value (possessing a type)

If $E \vdash_S a{:}A$, $E \vdash_S A{::}K$ and $E \vdash_S K\ \text{kind}$ then we say that a is a *(first-class) value*, A is a *type* and K is a *class*. Note that we may have $E \vdash_S A{::}K$ and $E \vdash_S K\ \text{kind}$ where A is not a type (e.g. it is a type operator) and therefore K is not a class (in the sense that its elements are not types). Whether or not A is a type in $E \vdash_S A{::}K$, we say that A is a *second-class value*; it can be the object of non-trivial computations but lives one level higher than first-class values.

Our aim is to associate first-class values with run-time computations, and all the rest with compile-time computations. In particular, we aim to look at types as second-class values, and as a first step we replace the rule $E \vdash_S \text{Type:Type}$ with the rule $E \vdash_S \text{Type kind}$.

One might think that it is now sufficient to identify compile-time terms with the class of terms τ such that either $E \vdash_S \tau{::}K$ or $E \vdash_S \tau\ \text{kind}$. This fails because of the [All Elimination] rule, whose conclusion includes $B\{x \leftarrow a\}$. Here a (presumably run-time) value term is substituted inside a (presumably compile-time) type term, and the result is a type or a kind where phase mixing has occurred.

Our goal is then to preserve as much of the unphased type system as possible, while enforcing the following informal requirement:

Phase Distinction Requirement
If A is a compile-time term and B is a subterm of A,
then B must also be a compile-time term.

Note that the tautology function is ruled out by this requirement; the parameter n is used both as a run-time value to terminate recursion, and as a part of the type Prop(n).

To meet the phase distinction requirement, we proceed as follows. We build a type system based on the three-level hierarchy of values, type families and kinds. The kind-free operators have now to be replaced by a number of *kinded* operators. For example, we will have recursion at the value level and at the type level. Similarly, we will have functions from kinds to kinds, from kinds to types, from types to kinds, and from types to types. Finally, we will notice that respecting the phase distinction requirement will make some of these kinded operators useless, and they will be removed.

Signature Construction

$$\frac{}{\vdash \emptyset \ \text{sig}}$$

$$\frac{\vdash S \text{ sig} \qquad c \notin \text{Dom}(S)}{\vdash (S,\ c \ \text{kind}) \ \text{sig}}$$

$$\frac{\vdash S \text{ sig} \qquad \emptyset \vdash_S K \text{ kind} \qquad c \notin \text{Dom}(S)}{\vdash (S,\ c{::}K) \ \text{sig}}$$

$$\frac{\vdash S \text{ sig} \qquad \emptyset \vdash_S A{::}\text{Type} \qquad c \notin \text{Dom}(S)}{\vdash \ S,\ c{:}A \ \text{sig}}$$

Environment Construction

$$\frac{\vdash \ S \ \text{sig}}{\vdash_S \emptyset \ \text{env}}$$

$$\frac{\vdash_S E \text{ env} \qquad E \vdash_S K \text{ kind} \qquad X \notin \text{Dom}(E)}{\vdash_S (E,\ X{::}K) \ \text{env}}$$

$$\frac{\vdash_S E \text{ env} \qquad E \vdash_S A{::}\text{Type} \qquad x \notin \text{Dom}(E)}{\vdash_S (E,\ x{:}A) \ \text{env}}$$

Given

$$\frac{\vdash_S E \text{ env} \qquad c \text{ kind} \in S}{E \vdash_S \ c \ \text{kind}}$$

$$\frac{\vdash_S E \text{ env} \qquad c{::}K \in S}{E \vdash_S \ c{::}K}$$

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

Assumption

$$\frac{\vdash_S E \text{ env} \qquad X{::}K \in E}{E \vdash_S X{::}K}$$

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

Type Formation

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

All Formation

$$\frac{E \vdash_S K \text{ kind} \qquad E, X{::}K \vdash_S L \text{ kind}}{E \vdash_S \text{All}_{KK}(X{::}K)\ L \text{ kind}}$$

(abandoned)

$$\frac{E \vdash_S A{::}\text{Type} \qquad E, x{:}A \vdash_S L \text{ kind}}{E \vdash_S \text{All}_{TK}(x{:}A)\ L \text{ kind}} \qquad (x \notin L)$$

$$\frac{E \vdash_S K \text{ kind} \qquad E, X{::}K \vdash_S B{::}\text{Type}}{E \vdash_S \text{All}_{KT}(X{::}K)\ B :: \text{Type}}$$

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

All Introduction

$$\frac{E \vdash_S K \text{ kind} \qquad E, X{::}K \vdash_S B{::}L}{E \vdash_S \text{fun}_{KK}(X{::}K)\ B :: \text{All}_{KK}(X{::}K)\ L}$$

(abandoned)

$$\frac{E \vdash_S A{::}\text{Type} \qquad E, x{:}A \vdash_S B{::}L}{E \vdash_S \text{fun}_{TK}(x{:}A)\ B :: \text{All}_{TK}(x{:}A)\ L} \qquad (x \notin L)$$

$$\frac{E \vdash_S K \text{ kind} \qquad E, X{::}K \vdash_S b{:}B}{E \vdash_S \text{fun}_{KT}(X{::}K)\ b : \text{All}_{KT}(X{::}K)\ B}$$

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

All Elimination

$$\frac{E \vdash_S A{::}K \qquad E \vdash_S B :: \text{All}_{KK}(X{::}K)\ L}{E \vdash_S B({}_{KK}A) :: L\{X \leftarrow A\}}$$

(abandoned)

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S B :: \text{All}_{TK}(x{:}A)\ L}{E \vdash_S B({}_{TK}a) :: L} \qquad x \notin L$$

$$\frac{E \vdash_S A{::}K \qquad E \vdash_S b : \text{All}_{KT}(X{::}K)\ B}{E \vdash_S b({}_{KT}A) : B\{X \leftarrow A\}}$$

$$\frac{E \vdash_S a{:}A \qquad E \vdash_S b : \text{All}_{TT}(x{:}A)\ B}{E \vdash_S b({}_{TT}a) : B} \qquad x \notin B$$

Rec Formation

$$\frac{E \vdash_S K \text{ kind} \qquad E, X{::}K \vdash_S A{::}K}{E \vdash_S \text{rec}_K(X{::}K)\ A :: K}$$

$$\frac{E \vdash_S A::K \qquad E,\ x:A \vdash_S a:A}{E \vdash_S rec_T(x:A)\ a : A}$$

Conversion

$$\frac{E \vdash_S A::K \qquad E \vdash_S L \text{ kind} \qquad K =_{\beta\eta\mu} L}{E \vdash_S A::L}$$

$$\frac{E \vdash_S a:A \qquad E \vdash_S B::Type \qquad A =_{\beta\eta\mu} B}{E \vdash_S a:B}$$

The [All_{KT} Formation] rule embeds an important design decision. Consider the special case which results in the conclusion $E \vdash_S All_{KT}(X::Type)B :: Type$; here we build a type by predicating X over *all* types, including the one we are defining; such definitions are called *impredicative*. We could turn that definition into a predicative one simply by concluding $E \vdash_S All_{KT}(X::Type)B$ kind.

This choice between the predicative and the impredicative rules has an interesting impact on the language. The impredicative definition admits polymorphic functions as first-class (run-time) objects, i.e. one can produce a (first-class) value f such that $f : All_{KT}(X::Type)B$, and pass it as a parameter to other run-time functions. Instead, the predicative definition admits polymorphic functions only as second-class (compile-time) objects $f :: All_{KT}(X::Type)B$. Such polymorphic functions could not be passed as parameters to run-time functions, and would have to be first applied to a type to produce a run-time value.

These two rules distinguish between what we might call *impredicative polymorphism*, like in second-order λ-calculus, versus *predicative polymorphism*, like in Ada generic procedures.

Next we examine the [All_{TK} Elimination] rule. Here we have a direct violation of the phase separation requirement, since the compile-time term B(a) contains a run-time term. Hence we have to abandon this rule and, as a consequence, all the other All_{TK} rules.

This means that types cannot depend on run-time values. Notice however that we still have the All_{KK} rules where types depend on compile-time values, The [Given] rules can then provide compile-time versions of run-time values, for example we will work in a signature with Int_K kind, Int_T::Type, $0_K::Int_K$, $0_T: Int_T$, etc., hence distinguishing between compile-time and run-time integers.

In general, all the run-time expressions which are of interest at compile-time can be *lifted* to the type-family level. This is only admissible for *pure* expressions which have a substitution of equals for equals property, i.e. not involving side-effects. In languages like Pascal, there is a notion of *constant expressions* which can be evaluated at compile time;

they are normally restricted to simple arithmetic and logical operations. Here we have generalized the language of constant expressions to include typed lambda abstraction, application and recursion (because of the latter we do not require compile-time computations to terminate). Pascal also allows constant expressions (whose value is known at compile time) to be mixed with run-time computations; to reflect this situation we could add rules to *lower* certain K-level normal forms to the T level.

Finally, note that in the [All_{TT} Elimination] rule (and similarly in the abandoned [All_{TK} Elimination]) we must impose $x \notin B$ to prevent phase mixing in $B\{x{\leftarrow}a\}$. This induces $x \notin B$ in the other All_{TT} rules, although it is shown in parentheses there because it would not directly cause phase mixing in those contexts.

Let us see how we can express our previous examples in this phased framework. We use the convention that the *TT* and *T* subscripts (like in All_{TT} and Int_T) are omitted. First we define function spaces, again, but this time the definition will only work at the *TT* level:

```
let → :: All_KK(A::Type) All_KK(B::Type) Type =
    fun_KK(A::Type) fun_KK(B::Type) All_TT(x:A) B

let Endo :: All_KK(A::Type) Type =
    fun_KK(A::Type) A→A
```

We can say, e.g., Int→Int but not, e.g., Type→Type (for which we must use All_{KK}(A::Type) Type). Monomorphic function definitions are unchanged (because we drop the *TT* subscripts):

```
let succ: Endo(Int)  =
    fun(a:Int) a + 1;

let shift: Endo(Endo(Int)) =
    fun(f:Endo(Int)) fun(a: Int) f(a-1)
```

For polymorphic functions we use the *KT* operators; we have chosen the impredicative polymorphism rule which allows us to write a run-time twice function:

```
let PolyEndo:: Type =
    All_KT(A::Type) Endo(A)
```

```
let id: PolyEndo =
    fun_KT(A::Type) fun(a:A) a

let twice: Endo(PolyEndo) =
    fun(f: PolyEndo) fun_KT(A::Type) fun(a: A) f(A)(f(A)(a))
```

Compare the above definitions with what we would get with the predicative rule:

```
let PolyEndo kind =
    All_KT(A::Type) Endo(A)

let id:: PolyEndo =
    fun_KT(A::Type) fun(a:A) a

let twice:: All_KK(F:: PolyEndo) PolyEndo =
    fun_KK(F:: PolyEndo) fun_KT(A::Type) fun(a: A) F(A)(F(A)(a))
```

For types dependent on values we should use the *TK* operators, but since these have been abolished, we use *KK* operators and we lift the basic types and values from *T* to *K*:

```
let Prop:: All_KK(N:: Nat_K) Type =
    rec_K(F:: All_KK(N:: Nat_K) Type)
        fun_KK(N:: Nat_K) case_K N of 0_K ⇒ Bool; succ_K(M) ⇒ Bool→F(M);

let and2: Prop(2_K) =
    fun(a: Bool) fun(b: Bool) a ∧ b;
```

The tautology function was chosen as an example of something which is not expressible in the phased system.

We can now safely assume that judgements of the form $E \vdash_S K$ kind or $E \vdash_S A::K$ represent compile-time terms, while judgements of the form $E \vdash_S a:A$ represent run-time terms. More precisely, we can define an *erase* function which converts terms a such that $E \vdash_S a:A$ into untyped λ-terms. This process describes the task of the compiler when presented with a value-level expression:

$\mathrm{Erase}(c) = c$

$\mathrm{Erase}(x) = x$

$\mathrm{Erase}(\mathrm{fun}_{KT}(X::K)\ b) = \mathrm{Erase}(b)$

$\mathrm{Erase}(\mathrm{fun}_{TT}(x:A)\ b) = \mathrm{fun}(x)\ \mathrm{Erase}(b)$

$\mathrm{Erase}(b({}_{KT}A)) = \mathrm{Erase}(b)$

$\mathrm{Erase}(b({}_{TT}a)) = \mathrm{Erase}(b)\ (\mathrm{Erase}(a))$

$\mathrm{Erase}(\mathrm{rec}_{T}(x:A)\ a) = \mathrm{rec}(x)\ \mathrm{Erase}(a)$

This concludes the analysis of our basic type system. In the next sections we study some extensions.

Existential Types

Existential types can be used to model abstract types [Mitchell 85] and modules [Burstall 84b, MacQueen 86]. Unlike other presentations, our *pair* objects are heavily typed to make their existential types unambiguous (some of this type information would be omitted in practice). In pair(x:A=a) b:B, the left and right components of the pair are a and b respectively; x is a variable bound to a which may occur free both in b and B; A is the type of a, and B is the type of b. The type of the pair will then be Some(x:A) B.

In ordinary dependent pairs, the scope of x only includes B, so that the *type* of the right component may depend on the *value* of the left component. We extend the scope of x to b, for convenience; when generalizing from pairs to tuples (e.g. using the syntax (x:A=a; y:B=b; z:C=c)) it is natural to have variables declared at each stage available in later stages, achieving cascaded declarations.

Also note that we can define let x:A = a in b:B simply as rht(pair(x:A=a) b:B). No matter what scoping rules we chose for pairs, the typing rule for let should come from the existential elimination rule, which is more general than the rule obtained from the usual λ-encoding of let (making use of the assumption $E \vdash_S b\{x\leftarrow A\}:B\{x\leftarrow A\}$, instead of $E, x:A \vdash_S b:B$).

Some Formation

$$\frac{E \vdash_S K \text{ kind} \qquad E, X::K \vdash_S L \text{ kind}}{E \vdash_S \mathrm{Some}_{KK}(X::K)\ L \text{ kind}}$$

(abandoned)

$$\frac{E \vdash_S A::\mathrm{Type} \qquad E, x:A \vdash_S L \text{ kind}}{E \vdash_S \mathrm{Some}_{TK}(x:A)\ L \text{ kind}} \qquad (x \notin L)$$

$$\frac{E \vdash_S K \text{ kind} \qquad E, X::K \vdash_S B::\mathrm{Type}}{E \vdash_S \mathrm{Some}_{KT}(X::K)B :: \mathrm{Type}}$$

$$\frac{E \vdash_S A::\mathrm{Type} \qquad E, x:A \vdash_S B::\mathrm{Type}}{E \vdash_S \mathrm{Some}_{TT}(x:A)\ B :: \mathrm{Type}} \qquad (x \notin B)$$

Some Introduction

$$\frac{E \vdash_S A::K \qquad E \vdash_S B\{X\leftarrow A\}::L\{X\leftarrow A\}}{E \vdash_S \mathrm{pair}_{KK}(X::K=A)\ B::L\ ::\ \mathrm{Some}_{KK}(X::K)\ L}$$

(abandoned)

$$\frac{E \vdash_S a:A \qquad E \vdash_S B::L}{E \vdash_S \mathrm{pair}_{TK}(x:A=a)\ B::L\ ::\ \mathrm{Some}_{TK}(x:A)\ L} \qquad (x \notin B,L)$$

$$\frac{E \vdash_S A::K \qquad E \vdash_S b\{X\leftarrow A\}:B\{X\leftarrow A\}}{E \vdash_S \mathrm{pair}_{KT}(X::K=A)\ b:B\ :\ \mathrm{Some}_{KT}(X::K)\ B}$$

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

Some Elimination

$$\frac{E \vdash_S C :: \mathrm{Some}_{KK}(X::K)\ L}{E \vdash_S \mathrm{lft}_{KK}(C) :: K} \qquad \frac{E \vdash_S C :: \mathrm{Some}_{KK}(X::K)\ L}{E \vdash_S \mathrm{rht}_{KK}(C) :: L\{X\leftarrow \mathrm{lft}_{KK}(C)\}}$$

(abandoned)

$$\frac{E \vdash_S C :: \mathrm{Some}_{TK}(x:A)\ L}{E \vdash_S \mathrm{lft}_{TK}(C) : A} \qquad \frac{E \vdash_S C :: \mathrm{Some}_{TK}(x:A)\ L}{E \vdash_S \mathrm{rht}_{TK}(C) :: L} \qquad x \notin L$$

$$\frac{E \vdash_S c : \mathrm{Some}_{KT}(X::K)\ B}{E \vdash_S \mathrm{lft}_{KT}(c) :: K} \qquad \frac{E \vdash_S c : \mathrm{Some}_{KT}(X::K)\ B}{E \vdash_S \mathrm{rht}_{KT}(c) : B\{X\leftarrow \mathrm{lft}_{KT}(c)\}}$$

$$\frac{E \vdash_S c : \mathrm{Some}_{TT}(x:A)\ B}{E \vdash_S \mathrm{lft}_{TT}(c) : A} \qquad \frac{E \vdash_S c : \mathrm{Some}_{TT}(x:A)\ B}{E \vdash_S \mathrm{rht}_{TT}(c) : B} \qquad x \notin B$$

As with universal types, the [Some_{KT} Formation] rule embeds an important design decision. Since existential types model abstract types, the impredicative version, concluding $E \vdash_S \mathrm{Some}_{KT}(X::\mathrm{Type})B :: \mathrm{Type}$ claims that abstract types are indeed types, and implementations of abstract types (packages) are first-class values [Donahue 85]. This means that we can, for example, select on the base of a run-time test an optimal implementation of an abstract type for a given task.

In the predicative view (where $E \vdash_S \mathrm{Some}_{KT}(X::\mathrm{Type})B\ \mathrm{kind}$) packages are second-class objects which cannot be manipulated at run time [MacQueen 84] (although it is still possible to extract run-time objects out of them).

Our phased system seems to achieve the flexibility of DL [MacQueen 86] in typechecking parametric modules, but in an impredicative framework where packages are values. (This point is still under investigation; phases do present some obstacles in expressing modular structures.)

As with function types, phase separation imposes a constraint in the elimination rule which makes the *TK* types practically useless. In this case the requirement $x \notin L$ means that one can replace any *TK* type with a *KT* type by swapping the pairs around.

Finally, we impose the $x \notin B$ constraint in [Some$_{TT}$ Elimination] to prevent phase mixing. This means that the *TT* pairs are just ordinary pairs of values, and can be abbreviated as <a,b> (if $x \notin b$).

Reference Types

Since we have a clear distinction between compile-time and run-time phases, we can introduce side-effects without fear that they will propagate to the type level during typechecking. Here the ref(a) term creates an updatable reference (of type Ref(A)) to the value a:A; deref(b) recovers the contents of a value b of reference type.

Ref Formation

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

Ref Introduction

$$\frac{E \vdash_S a:A}{E \vdash_S \mathrm{ref}(a):\mathrm{Ref}(A)}$$

Ref Elimination

$$\frac{E \vdash_S b:\mathrm{Ref}(A)}{E \vdash_S \mathrm{deref}(b):A}$$

Assignment

$$\frac{E \vdash_S b:\mathrm{Ref}(A) \qquad E \vdash_S a:A}{E \vdash_S b{:=}a : \ A}$$

Where we assume that an assignment returns the value of its right argument.

Conclusions

We have described the problems involved in harnessing the power of dependent types for compiled languages.

The proposed solution consists in layering the language in three levels (roughly: values, types and kinds) and introduce additional restrictions to distinguish between compile-time and run-time expressions (and their types). These distinctions facilitate some extensions of the language, e.g. to side-effectable data, which would otherwise be troublesome.

In a sense we have two languages in one. One is a run-time language to express computations, which can be compiled into efficient code evaluated *by-value* (applicative-order evaluation); this language can have imperative features. The other is a language to express type structures and compile-time computations; this is purely functional and is to be evaluated *by-name* (normal-order evaluation) to achieve proper symbolic type matching.

We have also briefly illustrated the implications of predicative and impredicative type systems with respect to polymorphism and abstract data types.

References

[Burstall 84a] R.M.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.

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

[Donahue 85] J.Donahue, A.Demers: **Data types are values**, *ACM TOPLAS*, 7(3), pp. 426-445, July 1985.

[Harper 87] R.Harper, F.Honsell, G.Plotkin: **A framework for defining logics** Proc. of the Second Logic in Computer Science Conference, Ithaca New York, June 1987.

[MacQueen 84] D.B.MacQueen: **Modules for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp 198-207. *ACM*, New York.

[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 ed. pp 73-118, North-Holland, 1973.

[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*.

[Turner 76] D.A.Turner: **SASL language manual**, Computer Laboratory, University of Kent, 1976.