Phase Your Erasure

Adam Gundry and Conor McBride

February 16, 2013

1 Introduction

We are used to the *phase distinction* being an oil-and-water separation of syntactic categories: terms and types not even thinkably in the same places, ensuring that types can safely be erased from a well typed term to yield an untyped term with the same observable behaviour. Whilst we should applaud the result — isolating and eliminating a fragment of the language not required for execution — we should not confuse sufficiency with necessity. Full spectrum dependent type systems make no such syntactic separation, allowing terms in types and types in terms, but the absence of *typecase* ensures that, for example, Coq's *extraction* process can erase all the types, as well as proofs of propositions. The key to erasure is to ensure that computation in what remains does not depend on information no longer present. The oil-and-water approach makes it easy to ensure that term computation does not depend on types, which is the only dependency that matters for type erasure, but it is not the only way to cut a calculus into phases which are independent in ways which matter.

In this paper, we present an alternative approach, where a uniform syntax of terms and types is marked up with *phases* drawn from a preorder characterizing permitted dependencies. Every cut across the preorder induces an operation on terms from phases below the cut which erases components from above and preserves computational behaviour, inspired by the translation from Church-style to Curry-style System F. Moreover, we facilitate the possibility to use the same components in different ways at different phases. It is clear that to add at run time, we require the run time presence of both summands, but if we add in a type (perhaps for concatenation of length-indexed lists), it is fine to permit summands which will be erased, for the whole addition is in any case for the chop.

2 A Simple Starting Point

Let us introduce the idea of phases by adapting a minimal system, essentially the 1971 type theory of Per Martin-Löf. The basic syntax is just as follows.

$$
\begin{array}{rcll}
s, t, f, S, T & ::= & * & \text{the type of all types} \\
& | & (x : S) \to T & \text{dependent function types} \\
& | & \lambda x : S \mapsto t & \text{abstractions} \\
& | & f\ s & \text{applications} \\
& | & x & \text{variables}
\end{array}
$$

Contexts Γ assign types to distinct variables. Context validity $\Gamma\ \vdash$ and typing $\Gamma\ \vdash\ t : T$ are as follows.

$$
\frac{}{\vdash} \qquad \frac{\Gamma\ \vdash \quad \Gamma\ \vdash\ S : *}{\Gamma, x : S\ \vdash}\ x \notin \Gamma
$$

$$
\frac{\Gamma\ \vdash \quad x : S \in \Gamma}{\Gamma\ \vdash\ x\ :\ S} \qquad \frac{\Gamma\ \vdash\ s\ :\ S \qquad \Gamma\ \vdash\ S \equiv T\ :\ *}{\Gamma\ \vdash\ s\ :\ T}
$$

$$
\frac{\Gamma\ \vdash}{\Gamma\ \vdash\ *\ :\ *} \qquad \frac{\Gamma\ \vdash\ S\ :\ * \quad \Gamma, x : S\ \vdash\ T\ :\ *}{\Gamma\ \vdash\ (x : S) \to T\ :\ *}
$$

$$
\frac{\Gamma\ \vdash\ S\ :\ * \quad \Gamma, x : S\ \vdash\ t\ :\ T}{\Gamma\ \vdash\ \lambda x : S \mapsto t\ :\ (x : S) \to T} \qquad \frac{\Gamma\ \vdash\ f\ :\ (x : S) \to T \quad \Gamma\ \vdash\ s\ :\ S}{\Gamma\ \vdash\ f\ s\ :\ [x \leftarrow s]T}
$$

We silently identify terms up to renaming of bound variables, and we note that suitable renamings will always allow us to avoid variable capture. We write $[x \leftarrow s]T$ for the capture-avoiding substitution of s for x in T. Our formal metatheory uses a de Bruijn index representation to avoid problems of name choice, but names seem preferable for human consumption.

Philosophies of judgmental equality vary widely, from Haskell core's miserly α-equivalence with explicit coercions between provably equal types, via equality up to β-conversion as in Pure Type Systems, to the generous equality of Martin-Löf's Extensional Type Theory which silently admits any provable equation. On this occasion, we seek to remain as neutral as possible, insisting that the following are admissible:

$$
\frac{\Gamma\ \vdash\ t\ :\ T}{\Gamma\ \vdash\ t \equiv t\ :\ T} \qquad \frac{\Gamma\ \vdash\ s \equiv t\ :\ T}{\Gamma\ \vdash\ s\ :\ T \quad \Gamma\ \vdash\ t\ :\ T}
$$

and that $T[(\lambda x : S \mapsto t)\ s] \equiv T[[x \leftarrow s]t]\ :\ *$ wherever a β-redex may occur in a type. The latter can be achieved either by admitting a β-rule for judgmental equality (which is what we do for a type theory), or by somehow excluding β-redexes from types (which is what we do for a Haskell kernel).

It is well known that this simple system is vulnerable to set theoretic paradoxes and thus neither strongly normalizing, nor consistent as a logic. It does, however, enjoy the basic metatheoretic property that computation by β-reduction preserves type. Intuitively, this is because it is always possible to

rearrange appeals to conversion so that any well typed β-redex arises from some

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S \vdash t : T}}{\Gamma \vdash \lambda x{:}S \mapsto t : (x{:}S) \to T} \quad \dfrac{\vdots}{\Gamma \vdash s : S}}{\Gamma \vdash (\lambda x{:}S \mapsto t)\, s : [x \leftarrow s]T}$$

and we can substitute all appeals to the variable rule for x in the typing $t : T$ by the derivation of $s : S$, yielding a derivation of

$$\Gamma \vdash [x \leftarrow s]t \ : \ [x \leftarrow s]T$$

Our plan is to refine this basic system by adding *phase* annotations, without disturbing the basic structure of the type preservation argument. We should also ensure that we do nothing to prejudice the standard approaches to ramification which refine the basic system to a consistent logic with a hierarchy of sorts.

3 Phases on Colons

Our first attempt at a phased refinement simply annotates every colon with a phase θ, ϕ, ψ drawn from a preorder $(\Phi, \preceq)$. There is a distinguished phase $* \in \Phi$ which is the lowest phase at which types are formed. Every place where we assign a thing (hypothetical or actual) a type, we also assign it a phase, and we insist that variables can move only upward through the phase order.

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash S :^{*} *}{\Gamma, x{:}^{\theta} S \vdash}\, x \notin \Gamma$$

$$\frac{\Gamma \vdash \quad x{:}^{\theta} S \in \Gamma}{\Gamma \vdash x :^{\phi} S}\, \theta \preceq \phi \qquad \frac{\Gamma \vdash s :^{\phi} S \qquad \Gamma \vdash S \equiv T :^{*} *}{\Gamma \vdash s :^{\phi} T}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash * :^{\phi} *}\, * \preceq \phi \qquad \frac{\Gamma \vdash S :^{*} * \quad \Gamma, x{:}^{\theta} S \vdash T :^{\phi} *}{\Gamma \vdash (x{:}^{\theta} S) \to T :^{\phi} *}\, * \preceq \phi$$

$$\frac{\Gamma \vdash S :^{*} * \quad \Gamma, x{:}^{\theta} S \vdash t :^{\phi} T}{\Gamma \vdash \lambda x{:}^{\theta} S \mapsto t :^{\phi} (x{:}^{\theta} S) \to T} \qquad \frac{\Gamma \vdash f :^{\phi} (x{:}^{\theta} S) \to T \quad \Gamma \vdash s :^{\theta} S}{\Gamma \vdash f\, s :^{\phi} [x \leftarrow s]T}$$

The typing rules have the property that they bound their conclusion phase ϕ only from below, so we can always shunt typings upward. An easy induction on derivations delivers the following admissible *phase subsumption* rule

$$\frac{\Gamma \vdash t :^{\phi} T}{\Gamma \vdash t :^{\psi} T}\, \phi \preceq \psi$$

with transitivity of $\preceq$ critical to preserve the side condition in the variable and set-formation cases. The application rule checks that the argument to a function

is typable at the phase demanded by the type of the function, and should that function be an abstraction, phase subsumption ensures that the argument can be used wherever the bound variable is permitted.

A *cut* is given by an upward-closed subset Ψ of Φ.

$$\frac{\phi \preceq \psi \quad \phi \in \Psi}{\psi \in \Psi}$$

Any such cut induces an erasure operation for terms typable at phases in its complement $\Phi \backslash \Psi$. The target language for the erasure is the untyped λ-calculus if $* \in \Psi$ and once more our typed syntax otherwise.

$$\begin{array}{rcll}
\lfloor x \rfloor^{\Psi} & = & x & \\
\lfloor * \rfloor^{\Psi} & = & * & \\
\lfloor (x :^{\theta} S) \to T \rfloor^{\Psi} & = & (x :^{\theta} \lfloor S \rfloor^{\Psi}) \to \lfloor T \rfloor^{\Psi} & \\
\lfloor \lambda x :^{\theta} S \mapsto t \rfloor^{\Psi} & = & \lambda x :^{\theta} \lfloor S \rfloor^{\Psi} \mapsto \lfloor t \rfloor^{\Psi} & \text{if } \theta \notin \Psi, * \notin \Psi \\
 & = & \lambda x \mapsto \lfloor t \rfloor^{\Psi} & \text{if } \theta \notin \Psi, * \in \Psi \\
 & = & \lfloor t \rfloor^{\Psi} & \text{if } \theta \in \Psi \\
\lfloor f\ s \rfloor^{\Psi} & = & \lfloor f \rfloor^{\Psi}\ \lfloor s \rfloor^{\Psi} & \text{if } f : (x :^{\theta} S) \to T, \theta \notin \Psi \\
 & = & \lfloor f \rfloor^{\Psi} & \text{if } f : (x :^{\theta} S) \to T, \theta \in \Psi
\end{array}$$

That is, $\lfloor - \rfloor^{\Psi}$ erases the subterms checked at phases in Ψ. Erased abstractions bind variables at phases in Ψ, so by upward closure of Ψ all their uses will be erased too!

Claim. If $\Gamma \vdash f\,\vec{s} :^{\phi} (x :^{\theta} S) \to T$, $\phi, \theta \notin \Psi$ and $\lfloor f\,\vec{s} \rfloor^{\Psi} = \lambda x \mapsto t'$, possibly with a type annotation, then there exist some S and t such that $f\,\vec{s} \triangleright_{\beta}^{*} \lambda x :^{\theta} S \mapsto t$ and $\lfloor t \rfloor^{=} t'$.

Claim. If $\Gamma \vdash s :^{\phi} S$, $\phi \notin \Psi$ and $\lfloor s \rfloor^{\Psi} \triangleright_{\beta} t'$, then there exists some t such that $s \triangleright_{\beta}^{+} t$.

Thought. It should be possible to extend the simple system to one which has implicit generalization and specialization for phases in Ψ, so that we get a (hopelessly undecidable) type system for erased terms. Any repositioning of the cut should result in a map between derivations exchanging implicit generalization-and-specialization with explicit abstraction-and-application.

4 Different Usage at Different Phases

The story so far gives a reasonable account of phase preorders for which subsumption is admissible and erasure is safe. However, it lacks a certain flexibility. We might have a two phase system $\bullet \preceq *$ where $\bullet$ is a 'dynamic' phase of run time values on which types may depend. We can erase $*$-phase subterms from $\bullet$-checked terms, so our two quantifiers $(x :^{*} S) \to T$ and $(x :^{\bullet} S) \to T$ behave respectively like parametric dependent intersections and non-parametric dependent products[1]. We might imagine having some notion of 'natural numbers',

[1] viewing functions from S as S-adic tuples

with

$$\begin{array}{rcl} \mathsf{Nat} & :^{*} & * \\ \mathsf{zero} & :^{\bullet} & \mathsf{Nat} \\ \mathsf{suc} & :^{\bullet} & (n :^{\bullet} \mathsf{Nat}) \to \mathsf{Nat} \end{array}$$

We can use these numbers dynamically, and we can form types which mention dynamic numbers, but we do not have a useful notion of 'static number' because, gallingly, although phase subsumption allows

$$\vdash \mathsf{zero} :^{*} \mathsf{Nat} \qquad \vdash \mathsf{suc} :^{*} (n :^{\bullet} \mathsf{Nat}) \to \mathsf{Nat}$$

we do not have

$$n :^{*} \mathsf{Nat} \not\vdash \mathsf{suc}\, n :^{*} \mathsf{Nat}$$

That is, we can make suc static, but we may still apply it only to dynamic numbers. If only we could apply static suc to static numbers and dynamic suc to dynamic numbers!

Haskell approaches this problem by having two distinct sucs, with the static one being a 'promoted' version of the dynamic one, taking static arguments. It is troublesome to combine promotion with phase subsumption if you can really tell the difference between static and dynamic versions of constructors: we would have two ways to make a static zero, three ways to make suc zero, and so on.

We should rather have one version of each thing, but characterize its mode of usage at each phase. With this in mind, we require a binary operator $-/-$, pronounced 'for', on phases, and we modify the application rule as follows

$$\frac{\Gamma \vdash f :^{\phi} (x :^{\theta} S) \to T \quad \Gamma \vdash s :^{\theta/\phi} S}{\Gamma \vdash f\ s :^{\phi} [x \leftarrow s]T}$$

so that we may then define

θ/ϕ	$\bullet$	$*$
$\bullet$	$\bullet$	$*$
$*$	$*$	$*$

and now,

$$n :^{*} \mathsf{Nat} \vdash \mathsf{suc}\, n :^{*} \mathsf{Nat}$$

because $\bullet/* = *$.

But hold on a minute! We are surely not free to define $-/-$ any which way we choose. For a start, we shall need *right monotonicity*

$$\frac{\phi \preceq \psi}{\theta/\phi \preceq \theta/\psi}$$

if we want to retain phase subsumption by induction on derivations. Moreover, it is not obviously a good idea to modify the application rule without modifying the abstraction rule correspondingly, to ensure that substituting the bound variable in a β-step yields a well typed reduct. Nor is it clear why the type of an

application is well formed in the first place, given that x is hypothesized at θ but instantiated by term s at θ/ϕ, and we neither have nor intend $\theta/\phi \preceq \theta$. We should certainly forgive you for detecting a whiff of fish, except that it is not fish, but a subtle kind of phase polymorphism. Every concrete abstraction really stands for a collection of abstractions where loosening guarantees allows loosening of requirements.

Let us take our $\bullet \preceq *$ as a concrete example. Every dynamic abstraction gives rise to a corresponding static abstraction. A *phase strengthening* rule[2]

$$\frac{\Gamma, x :^{\bullet} S, \Delta \vdash t :^{*} T}{\Gamma, x :^{*} S, \Delta \vdash t :^{*} T}$$

is admissible by a straightforward induction on derivations. In the variable case, we have either x at phase $*$ or some other variable at a phase $\theta \preceq *$. In the abstraction case, the additional hypothesis extends Δ. In the application case, both premises are checked at $*$, even if the function's type quantifies over $\bullet$. The other cases are just structural. Note that phase subsumption followed by phase strengthening now gets us

$$\frac{\Gamma, x :^{\bullet} S, \Delta \vdash t :^{\bullet} T}{\Gamma, x :^{*} S, \Delta \vdash t :^{*} T}$$

The point is that nowhere in the construction of $t :^{*} T$ do we ever check $x :^{\bullet} S$. The most we ever demand is $x :^{*} S$, so we are quite at liberty to weaken our assumption!

More generally, we can capture our potential phase demands by the following relation $- \sqsupseteq -$, pronounced 'involves'.

$$\frac{}{\phi \sqsupseteq \phi} \qquad \frac{}{\phi \sqsupseteq *} \qquad \frac{\phi \sqsupseteq \psi}{\phi \sqsupseteq \theta/\psi}$$

For our example system, we get

$$\bullet \sqsupseteq \bullet \qquad \bullet \sqsupseteq * \qquad * \not\sqsupseteq \bullet \qquad * \sqsupseteq *$$

It is easy to check that whenever the derivation for $\Gamma \vdash t :^{\phi} T$ has a subderivation of $\Gamma, \Delta \vdash s :^{\psi} S$, we have $\phi \sqsupseteq \psi$. Correspondingly, we only ever demand $x :^{\psi} S$ when checking uses of x in t. We therefore insist that $-/-$ satisfies the following *loosening* property

$$\frac{\phi \sqsupseteq \psi \quad \theta \preceq \psi}{\theta/\phi \preceq \psi}$$

from which we may deduce the admissibility of a general *phase strengthening* rule

$$\frac{\Gamma, x :^{\theta} S, \Delta \vdash t :^{\psi} T}{\Gamma, x :^{\theta/\phi} S, \Delta \vdash t :^{\psi} T}\ \phi \sqsupseteq \psi$$

[2]strengthening in that we deduce looser contextual requirements for the same typing guarantee and hence acquire a more useful result

The proof is by induction on derivations. By construction of $- \sqsupseteq -$, the side condition of the inductive hypothesis always holds. The loosening property exactly delivers the result in the case of the variable rule for x.

Consequently, whenever we have

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}^{\theta} S \vdash t :^{\phi} T}}{\Gamma \vdash \lambda x{:}^{\theta} S \mapsto t :^{\phi} (x{:}^{\phi} S) \to T} \quad \dfrac{\vdots}{\Gamma \vdash s :^{\theta/\phi} S}}{\Gamma \vdash (\lambda x{:}^{\theta} S \mapsto t)\, s :^{\phi} [x \leftarrow s]T}$$

we may obtain

$$\dfrac{\dfrac{\vdots}{\Gamma, x{:}^{\theta/\phi} S \vdash t :^{\phi} T} \quad \dfrac{\vdots}{\Gamma \vdash s :^{\theta/\phi} S}}{\Gamma \vdash [x \leftarrow s]t :^{\phi} [x \leftarrow s]T}$$

and, for that matter,

$$\dfrac{\dfrac{\vdots}{\Gamma, x{:}^{\theta/\phi} S \vdash T :^{*} *} \quad \dfrac{\vdots}{\Gamma \vdash s :^{\theta/\phi} S}}{\Gamma \vdash [x \leftarrow s]T :^{*} *}$$

Remark. We need to check what happens with the erasure. It may become more nuanced.

5 Example Deployment: Indexed State Monad

Remark. This is part of our motivation, in search of a home in the narrative flow.

Consider a state monad for states of type S with operations

$$\begin{array}{rcl} \mathsf{get} & : & \mathsf{State}\, S \\ \mathsf{put} & : & S \to \mathsf{State}\, 1 \end{array}$$

We have no guarantee that any implementation of State will actually behave like a memory, as opposed to, say, a duplex communication channel for signals of type S. We might consider making precise our memorylike requirements by an Atkey-style indexing of the monad by the state value before and after a state manipulating process. In our simple system, we might take

$$\begin{array}{rcl} \mathsf{State} & : & S \to * \to S \to * \\ \mathsf{get} & : & (s{:}S) \to \mathsf{State}\, s\, ((s'{:}S) \times s' = s)\, s \\ \mathsf{put} & : & (s'{:}S) \to (s{:}S) \to \mathsf{State}\, s\, 1\, s' \end{array}$$

This choice certainly ensures that get returns (a copy of) the state without changing it, and that put makes the state take the given new value, whatever it was before. However, there is an obvious drawback: in order to obtain the state via get, it is necessary already to know it.

The trouble is that we are not distinguishing static from dynamic uses of states. If we work in the $\bullet \preceq *$ system, we may consider the more nuanced

$$\begin{array}{rcl} \mathsf{State} & :^{*} & S \to * \to S \to * \\ \mathsf{get} & :^{\bullet} & (s :^{*} S) \to \mathsf{State}\ s\ ((s' :^{\bullet} S) \wedge s' = s)\ s \\ \mathsf{put} & :^{\bullet} & (s' :^{\bullet} S) \to (s :^{*} S) \to \mathsf{State}\ s\ 1\ s' \end{array}$$

where $(s :^{\bullet} S) \wedge T$ is the type of dependent pairs with a dynamic first component and a static second. Correspondingly, it is not necessary to have *dynamic* knowledge of the state in order to acquire that knowledge via get. We track statically our best guess about the value of the state and can thus be precise about when we are (or are not) indifferent to its initial value.