PAL$^+$: a lambda-free logical framework[*][†]

Zhaohui Luo
Department of Computer Science
University of Durham
South Rd, Durham DH1 3LE, U.K.
Email: Zhaohui.Luo@durham.ac.uk

June 6, 2000

Abstract

A lambda-free logical framework takes parameterisation and cut as the basic notions to provide a schematic mechanism for specification of type theories and a definitional mechanism for their use in practice.

The framework presented here, PAL$^+$, is in the spirit of de Bruijn's PAL for Automath [dB80]. PAL$^+$ is a logical framework for specification and implementation of type theories. As in Martin-Löf's logical framework [NPS90], computational rules can be introduced and are used to give meanings to the declared constants. However, PAL$^+$ only allows one to talk about the concepts that are intuitively in the object type theories: types and their objects, and families of types and families of objects of types. In particular, in PAL$^+$, one cannot represent families of families of entities, which could be done in other logical frameworks by means of lambda abstraction. Compared with PAL, PAL$^+$ allows one to represent parametric concepts such as families of types and families of non-parametric objects, which can be used by themselves as totalities as well as when they are fully instantiated.

We claim that PAL$^+$ is a correct meta-language for specifying type theories (eg, dependent type theories), as it has the advantage of exactly capturing the intuitive concepts in object type theories, and that its implementation reflects the actual use of type theories in practice. We shall also study the meta-theory of PAL$^+$ and discuss some experiments done in a prototype implementation.

1 Motivations

A lambda-free logical framework takes parameterisation and cut as the basic notions to provide a schematic mechanism for specification of type theories and a definitional mechanism for their use in practice. The reasons to consider lambda-free logical frameworks include:

---

[*]This work is based on some notes I wrote in 1997 of the same title, which I did not finish. The basic ideas are the same, though they are presented here in a more complete version with some new development.

[†]This work is partly supported by UK EPSRC grants GR/K79130 and GR/M75518.

- Parametric constants/definitions and the associated operations of instantiation (substitution or cut) are more basic and arguably simpler notions/mechanisms than that of lambda-abstraction as found in other logical frameworks such as Martin-Löf's logical framework [NPS90]. A parametrically defined entity represents a family of entities, rather than a functional operation.

- The user of a proof system based on a lambda-free framework does not have to understand the meta-level lambda-abstraction that can be used to represent concepts such as families of families of entities, which do not exist in object type theories. Rather, one only has to grasp concepts of the object type theory and the definitional mechanism.

- The introduction of lambda-free logical framework makes clear that a logical framework is a *meta-language* that provides the schematic mechanisms for specifying type theories and the definitional mechanism for pragmatic use. It is worth remarking that such mechanisms are necessary for any framework to be used in practice, with or without λ-abstraction.[1]

The logical framework presented here, PAL^+, is such a framework in the spirit of de Bruijn's PAL for Automath [dB80].

PAL^+ is a logical framework for specification and implementation of type theories. As in Martin-Löf's logical framework [NPS90], computational rules can be introduced and are used to give meanings to the declared constants. However, PAL^+ only allows one to talk about the concepts that are intuitively in the object type theories: types and their objects, and families of types and families of objects of types. In particular, in PAL^+, one cannot represent families of families of entities, which could be done in other logical frameworks by means of lambda abstraction. Compared with PAL, PAL^+ allows one to represent parametric concepts such as families of types and families of non-parametric objects, which can be used by themselves (as totalities) as well as when they are fully instantiated. An implementation of a proof development system based on PAL^+ can truly reflect the intended use of a type theory. After a type theory is specified (and implemented), the user is concerned only with the object type theory and uses the definitional mechanism for abbreviation.

The following section introduces the parameterisation mechanism in PAL^+ (in particular, parametric kinds and parametric abstractions) and show how it can be used in specification of type theories. The definitional mechanisms (global and local definitions) are introduced in Section 3, where we also discuss how parametric abstractions can be introduced as let-expressions (local definitions) while taking the latter as basic. In Section 4, we consider the meta-theoretic properties of the logical framework PAL^+. In Section 5, we

[1] When an object type theory has types of functions, say Π-types, there is usually a confusion between the object-level functions and the meta-level functional operations, if the latter exist in the meta-framework. For example, in systems like ALF [MN94], one tends to use the meta-level functional operations as functional programs while ignoring the object level functions.

briefly discuss some implementation issues and describe some simple experiments carried out so far in Callaghan's prototype implementation of PAL$^+$. In the Conclusion, besides discussing some future work, we shall briefly consider further extensions of the framework with other meta-theoretic features such as coercive subtyping.

2 PAL$^+$: the parameterisation mechanism

In PAL$^+$, we have (non-parametric) *kinds* of the form **Type** or $El(A)$ and their objects (types and objects of type A), and *parametric kinds* of the form $(\Delta)T$, where Δ is a non-empty context and T is a non-parametric kind, and their objects (*parametric objects*) representing families of types or families of non-parametric objects. The following gives a more formal description.

2.1 Judgement forms and notations

The judgement forms are:

- Γ **valid** — Γ is a valid context.
- $\Gamma \vdash T$ **kind** — T is a (non-parametric) kind in Γ.
- $\Gamma \vdash P$ **pkind** — P is a parametric kind in Γ.
- $\Gamma \vdash t : T$ — t is an object of kind T in Γ.
- $\Gamma \vdash p : (\Delta)T$ — p is a parametric object of parametric kind $(\Delta)T$ in Γ.

We also have the corresponding equality judgement forms.

Notation We shall write $\Gamma \vdash K$ for '$\Gamma \vdash K$ **kind** or $\Gamma \vdash K$ **pkind**'. Similar notational conventions apply for the substitution forms such as $[k/x]K$.

Notation A context is a sequence of entries which are of the form $x{:}K$ (or of the form $F[\Delta] = t : T$, when PAL$^+$ is extended with global definitions). A context Δ of the form $x_1{:}K_1, ..., x_n{:}K_n$ is often abbreviated as $\bar{x}{:}\bar{K}$ and we also use Δ_i to stand for $x_1{:}K_1, ..., x_i{:}K_i$. For $\Delta \equiv x_1{:}K_1, ..., x_n{:}K_n$ and $\Delta' \equiv x_1{:}K'_1, ..., x_n{:}K'_n$, we write $\Gamma \vdash \Delta = \Delta'$ for the sequence of judgements $\Gamma, \Delta_{i-1} \vdash K_i = K'_i$ $(i = 1, ..., n)$.

We say that a context is *not empty* (or *non-empty*) if it contains one or more entries of the form $x{:}K$.

Notation We shall write $\Gamma \vdash \bar{a} : \bar{K}$ for the sequence of judgements $\Gamma \vdash a_1 : K_1$, $\Gamma \vdash a_2 : [a_1/x_1]K_2$, ..., $\Gamma \vdash a_n : [a_{n-1}/x_{n-1}]...[a_1/x_1]K_n$ and similarly the notation $\Gamma \vdash \bar{a} = \bar{b} : \bar{K}$ stands for the sequence of judgements $\Gamma \vdash a_1 = b_1 : K_1$, $\Gamma \vdash a_2 = b_2 : [a_1/x_1]K_2$, ..., $\Gamma \vdash a_n = b_n : [a_{n-1}/x_{n-1}]...[a_1/x_1]K_n$.

Contexts and assumptions

$$\frac{}{\langle\rangle\ \mathbf{valid}} \qquad \frac{\Gamma \vdash K \quad x \notin FV(\Gamma)}{\Gamma, x{:}K\ \mathbf{valid}} \qquad \frac{\Gamma, x{:}K, \Gamma'\ \mathbf{valid}}{\Gamma, x{:}K, \Gamma' \vdash x : K}$$

General equality rules

$$\frac{\Gamma \vdash K}{\Gamma \vdash K = K} \qquad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \qquad \frac{\Gamma \vdash K = K' \quad \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$\frac{\Gamma \vdash k : K}{\Gamma \vdash k = k : K} \qquad \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k : K} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' : K}$$

Equality typing rules

$$\frac{\Gamma \vdash k : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k : K'} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k = k' : K'}$$

The kind Type

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash \mathbf{Type}\ \mathbf{kind}} \qquad \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash El(A)\ \mathbf{kind}} \qquad \frac{\Gamma \vdash A = B : \mathbf{Type}}{\Gamma \vdash El(A) = El(B)}$$

Figure 1: The basic rules of PAL^+.

2.2 Basic rules

The basic rules are given in Figure 1 and Figure 2, the latter of which contain the substitution rules. Formally, without considering the difference between non-parametric and parametric kinds, these rules are essentially the general inference rules of the logical framework LF (see Figure 9.1 in Chapter 9 of [Luo94]).[2]

The kind **Type** represents the conceptual universe of types to be introduced, and for each type A of kind **Type**, the kind $El(A)$ is the kind of objects of type A.

2.3 Parameterisation

Parametric objects represent families of types or families of non-parametric objects. They can either be used as a totality or when they are fully instantiated.

[2]We should point out that LF, a typed version of Martin-Löf's logical framework, is different from the Edinburgh Logical Framework (ELF) [HHP87]. Though quite similar formally, the intended ways of use are very different.

Substitution rules

$$\frac{\Gamma, x{:}K, \Gamma' \ \textbf{valid} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \ \textbf{valid}}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K'} \quad \frac{\Gamma, x{:}K, \Gamma' \vdash K' \quad \Gamma \vdash k = k' : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k'/x]K'}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' : [k/x]K'} \quad \frac{\Gamma, x{:}K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]k' = [k_2/x]k' : [k_1/x]K'}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash K' = K'' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k/x]K''} \quad \frac{\Gamma, x{:}K, \Gamma' \vdash k' = k'' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' = [k/x]k'' : [k/x]K'}$$

Figure 2: The substitution rules of PAL^+.

Formation rules for parametric kinds

$$\frac{\Gamma, \Delta \vdash T \ \textbf{kind} \quad \Delta \ \textit{is not empty}}{\Gamma \vdash (\Delta)T \ \textbf{pkind}}$$

$$\frac{\Gamma \vdash \Delta = \Delta' \quad \Gamma, \Delta \vdash T \ \textbf{kind} \quad \Gamma, \Delta' \vdash T' \ \textbf{kind} \quad \Gamma, \Delta \vdash T = T'}{\Gamma \vdash (\Delta)T = (\Delta')T'}$$

Instantiation rules for parametric objects

$$\frac{\Gamma \vdash f : (\bar{x}{:}\bar{K})T \quad \Gamma \vdash \bar{a} : \bar{K}}{\Gamma \vdash f[\bar{a}] : [\bar{a}/\bar{x}]T} \quad \frac{\Gamma \vdash f = f' : (\bar{x}{:}\bar{K})T \quad \Gamma \vdash \bar{a} = \bar{b} : \bar{K}}{\Gamma \vdash f[\bar{a}] = f'[\bar{b}] : [\bar{a}/\bar{x}]T}$$

Figure 3: Rules for parametric kinds in PAL^+.

2.3.1 Parametric kinds and instantiation

The rules for parametric kinds of the form $(\Delta)T$ are given in Figure 3. Note that a parametric entity of a parametric kind cannot be used by partial instantiation. Only when given appropriate indexing objects $\bar{a}$, can a parametric object f, ie, an object of a parametric kind, be instantiated into $f[\bar{a}]$.

Besides variables of a parametric kind, parametric objects also include parametric constants (introduced when specifying an object type theory) and parametric abstraction.

2.3.2 Parametric abstraction

A parametric abstraction of the form $[\Delta]t$ represents either a family of types or non-parametric objects, indexed by (sequences of) objects of Δ. The vari-

Introduction rules

$$\frac{\Gamma, \Delta \vdash t : T \quad \Gamma, \Delta \vdash T \textbf{ kind}}{\Gamma \vdash [\Delta]t : (\Delta)T} \qquad \frac{\Gamma \vdash \Delta = \Delta' \quad \Gamma, \Delta \vdash t = t' : T}{\Gamma \vdash [\Delta]t = [\Delta']t' : (\Delta)T}$$

Equality rules

$$(P_\beta) \quad \frac{\Gamma, \bar{x}{:}\bar{K} \vdash t : T \quad \Gamma \vdash \bar{a} : \bar{K}}{\Gamma \vdash ([\bar{x}{:}\bar{K}]t)[\bar{a}] = [\bar{a}/\bar{x}]t : [\bar{a}/\bar{x}]T} \qquad (P_\eta) \quad \frac{\Gamma \vdash f : (\bar{x}{:}\bar{K})T \quad \bar{x} \notin FV(f)}{\Gamma \vdash [\bar{x}{:}\bar{K}]f[\bar{x}] = f : (\bar{x}{:}\bar{K})T}$$

Figure 4: Rules for parametric abstractions.

ables in Δ are bound variables; for example, if $\Delta \equiv x_1{:}K_1, ..., x_n{:}K_n$, then the variables x_i are bound in $[\Delta]t$.

The rules for parametric abstraction are given in Figure 4, where $(P_\beta)(P_\eta)$ are the equality rules for parametric abstraction, analogous to the $\beta\eta$-rules for lambda abstractions. Note that t in $[\Delta]t$ can only be a type or an object of a type (and cannot be itself a parametric object.) Just as other parametric objects, parametric abstraction can be used by itself or when it is fully instantiated (and cannot be used when it is partially instantiated.)

2.4 Specifications of type theories in PAL$^+$

As in Martin-Löf's type theory [NPS90], we specify type theories in the logical framework PAL$^+$. One of the points is that we can specify type theories with the simpler logical framework without arbitrary lambda-abstraction. For example, all of the types in UTT [Luo94] can be specified, including the impredicative universe of logical propositions, the inductive types and inductive families covered by the inductive schemata, and predicative universes.

In general, a specification of a type theory in PAL$^+$ will consist of a collection of declarations of new constants (either non-parametric or parametric) and a collection of associated computational equality rules. Like other parametric objects, a parametric constant cannot be used by partial instantiation.

Given a parametric kind $(\Delta)T$ (or just a kind T), one may introduce a (parametric) constant f by declaring the following:

$$f : T \quad \textit{or} \quad f : (\Delta)T,$$

which has the effect of introducing the following rules, respectively:

$$\frac{\Gamma \textbf{ valid}}{\Gamma \vdash f : T} \qquad \frac{\Gamma \textbf{ valid}}{\Gamma \vdash f : (\Delta)T}$$

One can introduce computation rules for such parametric constants. Computation rules can be introduced by asserting

$$a = b : T \;\; \textit{for } k_i : K_i \;\; (i = 1, ..., n),$$

which extends the type theory with the following rule:

$$\frac{\Gamma \vdash T\ \mathbf{kind} \quad \Gamma \vdash k_i : K_i\ (i = 1, ..., n)}{\Gamma \vdash a = b : T}$$

Note that computational equality can only be introduced between two objects of a type, or between two types.[3]

In the following, we give several examples of introducing type constructors and their associated operators as constants. We omit El to write A for $El(A)$ in the examples.

Example 2.1 *Examples of parametric constants include those for introducing the type of natural numbers (I omit El):*

$$\begin{array}{rcl} N & : & \mathbf{Type} \\ 0 & : & N \\ s & : & (x{:}N)\ N \\ R & : & (C{:}(x{:}N)\mathbf{Type},\ c{:}C[0],\ f{:}(x{:}N, y{:}C[x])C[s[x]],\ z{:}N)\ C[z] \end{array}$$

The corresponding computation rules are:

$$\begin{array}{rcl} R[C, c, f, 0] & = & c\ :\ C[0] \\ R[C, c, f, s[x]] & = & f[x, R[C, c, f, x]]\ :\ C[s[x]] \end{array}$$

Example 2.2 *The* Π*-types,* $\Pi[A, B]$ *for a type* A *and a family of types* B*, can be introduced as follows:*

$$\begin{array}{rcl} \Pi & : & (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type})\ \mathbf{Type} \\ \lambda & : & (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type}, f{:}(x{:}A)B[x])\ \Pi[A, B] \\ \mathcal{E}_\Pi & : & (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type}, C{:}(F{:}\Pi[A, B])\mathbf{Type}, \\ & & \quad f{:}(g{:}(x{:}A)B[x])C[\lambda[A, B, g]], \\ & & \quad z{:}\Pi[A, B]) \\ & & C[z] \end{array}$$

The corresponding computation rule is:

$$\mathcal{E}_\Pi[A, B, C, f, \lambda[A, B, g]] \quad = \quad f[g]\ :\ C[\lambda[A, B, g]]$$

We can define the application operator

$$\mathbf{app}\ :\ (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type}, F{:}\Pi[A, B], x{:}A)\ B[x]$$

as follows:

$$\mathbf{app}[A, B, F, a] \quad =_{\mathrm{df}} \quad \mathcal{E}_\Pi[A, B, [G{:}\Pi[A, B]]B[a], [g{:}(x{:}A)B[x]]g[a], F]$$

[3] This confirms to the restriction considered in [Luo99], where LF is used to specify type theories.

Note that parametric abstraction is used in the above definition. We can show the expected equalities hold. For example, we can show that the usual β-equality

$$\mathbf{app}[A, B, \lambda[A, B, g], a] = g[a].$$

Furthermore, for propositional equality $=_{\Pi[A,B]}$ (eg, the Leibniz equality), we can show that logical η-rule holds, i.e., the following logical proposition is provable:

$$\lambda[A, B, [x{:}A]\mathbf{app}[A, B, F, x]] =_{\Pi[A,B]} F.$$

Example 2.3 *The family of types of vectors of objects of type A can be introduced as follows, where N is the type of natural numbers as introduced above.*

$$\begin{array}{rcl}
Vec & : & (A{:}\mathbf{Type}, x{:}N)\ \mathbf{Type} \\
nil & : & (A{:}\mathbf{Type})\ Vec[A, 0] \\
cons & : & (A{:}\mathbf{Type}, n{:}N, a{:}A, l{:}Vec[A, n])\ Vec[A, succ[n]] \\
\mathcal{E}_V & : & (A{:}\mathbf{Type},\ C{:}(n{:}N, v{:}Vec[A, n])\mathbf{Type}, \\
& & \quad c{:}C[0, nil[A]],\ f{:}(n{:}N, x{:}A, v{:}Vec[A, n], y{:}C[n, v])C[succ[n], cons[A, x, v]], \\
& & \quad n{:}N,\ v{:}Vec[A, n]) \\
& & C[n, v],
\end{array}$$

The corresponding computation rules are:

$$\begin{array}{l}
\mathcal{E}_V[A, C, c, f, 0, nil[A]] \ = \ c \ : \ C[0, nil[A]], \\
\mathcal{E}_V[A, C, c, f, succ[n], cons[A, n, a, v]] \\
\quad = \ f[n, a, v, \mathcal{E}_V[A, C, c, f, n, v]] \ : \ C[succ[n], cons[A, n, a, v]].
\end{array}$$

Example 2.4 *The W-types, $W[A, B]$ for a type A and a family of types B, can be introduced as follows:*

$$\begin{array}{rcl}
W & : & (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type})\ \mathbf{Type} \\
sup & : & (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type}, x{:}A, y{:}(v{:}B[x])W[A, B])\ W[A, B] \\
\mathcal{E}_W & : & (A{:}\mathbf{Type}, B{:}(x{:}A)\mathbf{Type}, C{:}(w{:}W[A, B])\mathbf{Type}, \\
& & \quad f{:}(x{:}A, y{:}(v{:}B[x])W[A, B], g{:}(v{:}B[x])C[y[v]])C[sup[A, B, x, y]], \\
& & \quad z{:}W[A, B]) \\
& & C[z]
\end{array}$$

The corresponding computation rule is:

$$\begin{array}{l}
\mathcal{E}_W[A, B, C, f, sup[A, B, x, y]] \\
\quad = \ f[x, y, [v{:}B[x]]\mathcal{E}_W[A, B, C, f, y[v]]] \ : \ C[sup[A, B, x, y]].
\end{array}$$

Special cases of W-types include the type of ordinals etc. Note that parametric abstraction is used in the computation rule above.

Introduction rule for global definitions

$$\frac{\Gamma, \Delta \vdash T \ \mathbf{kind} \quad \Gamma, \Delta \vdash t : T \quad f \notin FV(\Gamma)}{\Gamma, f[\Delta] = t{:}T \ \mathbf{valid}}$$

Typing and equality rules for global definitions

$$\frac{\Gamma, f[\bar{x}{:}\bar{K}] = t{:}T, \Gamma' \ \mathbf{valid} \quad \Gamma \vdash \bar{a} : \bar{K}}{\Gamma, f[\bar{x}{:}\bar{K}] = t{:}T, \Gamma' \vdash f[\bar{a}] : [\bar{a}/\bar{x}]T} \quad \frac{\Gamma, f[\bar{x}{:}\bar{K}] = t{:}T, \Gamma' \ \mathbf{valid} \quad \Gamma \vdash \bar{a} : \bar{K}}{\Gamma, f[\bar{x}{:}\bar{K}] = t{:}T, \Gamma' \vdash f[\bar{a}] = [\bar{a}/\bar{x}]t : [\bar{a}/\bar{x}]T}$$

$$\frac{\Gamma, f[\Delta] = t{:}T, \Gamma' \ \mathbf{valid}}{\Gamma, f[\Delta] = t{:}T, \Gamma' \vdash f : (\Delta)T} \quad \frac{\Gamma, f[\Delta] = t{:}T, \Gamma' \ \mathbf{valid}}{\Gamma, f[\Delta] = t{:}T, \Gamma' \vdash f = [\Delta]t : (\Delta)T}$$

$$\frac{\Gamma, f[\bar{x}{:}\bar{K}] = g[\bar{x}] : T, \Gamma' \ \mathbf{valid} \quad \bar{x} \notin FV(g)}{\Gamma, f[\bar{x}{:}\bar{K}] = g[\bar{x}] : T, \Gamma' \vdash f = g : (\bar{x}{:}\bar{K})T}$$

Subsitution rules for global definitions

$$\frac{\Gamma, f[\Delta] = t{:}T, \Gamma' \ \mathbf{valid}}{\Gamma, [[\Delta]t/f]\Gamma' \ \mathbf{valid}} \quad \frac{\Gamma, f[\Delta] = t{:}T, \Gamma' \vdash J}{\Gamma, [[\Delta]t/f]\Gamma' \vdash [[\Delta]t/f]J}$$

Figure 5: Rules for global definitions.

3 Definitions for PAL$^+$

To use a specified object type theory, one needs a definitional mechanism. We can introduce in the logical framework PAL$^+$ both global definitions (of the form $f[\Delta] = t{:}T$ introduced into contexts, where variables in Δ are bound in t and T) and local definitions (let expressions of the form **let** $f[\Delta] = t$ **in** M, where f is bound in M and variables in Δ are bound in t). Parametric abstraction can be defined as a special form of let-expression and we shall discuss this in Section 3.3.

3.1 Global definitions

Global definitions can be introduced into contexts and used by means of the rules in Figure 5, where in the last rule, J is of the form K **kind**, K **pkind**, $k : K$, $K = K'$, or $k = k' : K$.

Remark Several remarks are in order.

- Note that, in the introduction rule for global definitions, we require T to be a kind, ie, it is equal to either **Type** or $El(A)$.[4] Hence the body of a

[4]Kinds or parametric kinds in the form of let-expressions (to be introduced later) are always equal to **Type**, $El(A)$, or $(\Delta)T$. This is clear by the distribution rules in Figure 7.

let-introduction rules

$$\frac{\Gamma, f[\Delta] = t{:}T \vdash K}{\Gamma \vdash \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K} \quad \frac{\Gamma, f[\Delta] = t{:}T \vdash M : K}{\Gamma \vdash \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ M : \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K}$$

Congruence rules for let-expressions

$$\frac{\Gamma \vdash \Delta = \Delta' \quad \Gamma, \Delta \vdash t = t' : T \quad \Gamma, f[\Delta] = t{:}T \vdash K = K'}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K) = (\mathbf{let}\ f[\Delta'] = t'\ \mathbf{in}\ K')}$$

$$\frac{\Gamma \vdash \Delta = \Delta' \quad \Gamma, \Delta \vdash t = t' : T \quad \Gamma, f[\Delta] = t{:}T \vdash M = M' : K}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ M) = (\mathbf{let}\ f[\Delta'] = t'\ \mathbf{in}\ M') : \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K}$$

Figure 6: Introduction and congruence rules for let-expressions.

global definition must be a type or an object of a type. Also, when Δ is empty, the rules specialises into those for (non-parametric) kinds.

- The definiendum of a global definition can either be used when it is fully applied, or as a totality.
- The last equality rule (the last rule in the second group in Figure 5) is an η-like rule for global definitions.
- The 'meaning' of a globally defined entity can be given directly by means of parametric abstraction (but see Section 3.3 for the possibility of taking let-expressions as basic.)

3.2 Local definitions

Local definitions, or let-expressions, can be introduced by the let-introduction rules in Figure 6. They abide by the congruence rules in Figure 6 and some distribution laws in Figure 7. Note that, in some (eg, the last two) distribution rules, we assume that variable capturing be avoided appropriately.

Now, with the rules for global definitions in Figure 5 and for local definitions in Figures 6 and 7, the following equality rules are admissible.[5]

$$(let^K_\emptyset) \qquad \frac{\Gamma \vdash \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K \quad f \notin FV(K)}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K) = K}$$

$$(let_\emptyset) \qquad \frac{\Gamma \vdash \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ M : K \quad f \notin FV(M)}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ M) = M : K}$$

[5] Thanks to Conor McBride for a discussion on this. He has also pointed out the possibility to consider some of these equality rules as basic while having the distribution laws as admissible.

let-distribution rules for kinds or parametric kinds

$$\frac{\Gamma, f[\Delta] = t{:}T \ \mathbf{valid}}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \mathbf{Type}) = \mathbf{Type}}$$

$$\frac{\Gamma, f[\Delta] = t{:}T \vdash A : \mathbf{Type}}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ El(A)) = El(\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ A)}$$

$$\frac{\Gamma, f[\Delta] = t{:}T \vdash (\Delta')T' \ \mathbf{pkind}}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ (\Delta')T') = (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \Delta')\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ T'}$$

let-distribution rules for objects

$$\frac{\Gamma, f[\Delta] = t{:}T \vdash z : K \quad z \not\equiv f \ \textit{is a constant or a variable}}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ z) = z : \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K}$$

$$\frac{\Gamma, f[\Delta] = t{:}T \vdash g[\bar{a}] : T'}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ g[\bar{a}]) = (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ g)[\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \bar{a}] : \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ T'}$$

let-distribution rules for let-expressions

$$\frac{\Gamma, f[\Delta] = t{:}T, f'[\Delta'] = t'{:}T' \vdash K}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \mathbf{let}\ f'[\Delta'] = t'\ \mathbf{in}\ K) = (\mathbf{let}\ f'[\Delta_1] = t_1\ \mathbf{in}\ K_1)}$$

where $\Delta_1 \equiv \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \Delta'$, $t_1 \equiv \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ t'$, and $K_1 \equiv \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ K$.

$$\frac{\Gamma, f[\Delta] = t{:}T, f'[\Delta'] = t'{:}T' \vdash M : K}{\Gamma \vdash (\mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \mathbf{let}\ f'[\Delta'] = t'\ \mathbf{in}\ M) = (\mathbf{let}\ f'[\Delta_1] = t_1\ \mathbf{in}\ M_1) : K_1}$$

where Δ_1 and t_1 are as above, $M_1 \equiv \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ M$, and $K_1 \equiv \mathbf{let}\ f[\Delta] = t\ \mathbf{in}\ \mathbf{let}\ f'[\Delta'] = t'\ \mathbf{in}\ K$.

Figure 7: Distribution rules for let-expressions.

$$(let_\beta)\qquad \frac{\Gamma, f[\bar{x}{:}\bar{K}] = t{:}T \ \textbf{valid} \quad \Gamma \vdash \bar{a} : \bar{K}}{\Gamma \vdash (\textbf{let}\ f[\bar{x}{:}\bar{K}] = t\ \textbf{in}\ f)[\bar{a}] = [\bar{a}/\bar{x}]t : [\bar{a}/\bar{x}]T}$$

$$(let'_\beta)\qquad \frac{\Gamma, \bar{x}{:}\bar{K} \vdash t : T \quad \Gamma \vdash \bar{a} : \bar{K} \quad f \notin FV(\Gamma, \bar{a})}{\Gamma \vdash (\textbf{let}\ f[\bar{x}{:}\bar{K}] = t\ \textbf{in}\ f[\bar{a}]) = [\bar{a}/\bar{x}]t : [\bar{a}/\bar{x}]T}$$

$$(let_\eta^K)\qquad \frac{\Gamma \vdash \textbf{let}\ f[\bar{x}{:}\bar{K}] = g[\bar{x}]\ \textbf{in}\ K \quad \Gamma \vdash g : (\bar{x}{:}\bar{K})T}{\Gamma \vdash \textbf{let}\ f[\bar{x}{:}\bar{K}] = g[\bar{x}]\ \textbf{in}\ K = [g/f]K}$$

$$(let_\eta)\qquad \frac{\Gamma \vdash (\textbf{let}\ f[\bar{x}{:}\bar{K}] = g[\bar{x}]\ \textbf{in}\ M) : K \quad \Gamma \vdash g : (\bar{x}{:}\bar{K})T}{\Gamma \vdash (\textbf{let}\ f[\bar{x}{:}\bar{K}] = g[\bar{x}]\ \textbf{in}\ M) = [g/f]M : K}$$

The following simple example shows how local definitions may be used.

Example 3.1 *The application operator for* Π*-types can be defined by means of local definitions as follows.*

$$\begin{array}{rcl}\textbf{app}[A,B,F,a] & = & \textbf{let}\ C[G{:}\Pi[A,B]] = B[a]\ \textbf{in} \\ & & \textbf{let}\ f[g{:}(x{:}A)B[x]] = g[a]\ \textbf{in}\ \mathcal{E}_\Pi[A,B,C,f,F]\end{array}$$

3.3 Parametric abstractions as let-expressions

Parametric abstractions, directly introduced as basic in the current presentation of PAL$^+$, can be defined as special cases of let-expressions. In other words, we may take let-expressions (and global definitions) as basic while taking parametric abstractions as a defined notion as follows.

$$[\Delta]t =_{\mathrm{df}} \textbf{let}\ f[\Delta] = t\ \textbf{in}\ f,$$

where the bound variable f is chosen as fresh (and hence not clashed with other variables). Then we can show that the rules for parametric abstraction all hold given the above definition. Note that the rules for global definitions in Figure 5 remain the same.

On the other hand, one might want to take 'parametric abstraction' and its application as basic and define let-expressions as, eg,

$$\textbf{let}\ f[\Delta] = t\ \textbf{in}\ M =_{\mathrm{df}} [[\Delta]t/f]M.$$

We remark that let-expressions are slightly more general than parametric abstractions as they can be used for all expressions including parametric kinds. Furthermore, local definitions are useful in any proof development or programming environment. In this sense, it seems better to take let-expressions as basic while taking parametric abstractions as a defined notion.

4 Meta-theoretic properties of PAL^+

The basic system PAL^+, as presented in Section 2 (more precisely, the first three subsections of Section 2), is obviously a subsystem of LF as presented in [Luo94]. Therefore, some of the properties of LF can be directly transferred to PAL^+. For example, it is easy to show that PAL^+ has the strong normalisation property by translating PAL^+ into LF. However, it is not clear how some of the other properties (eg, Church-Rosser or Subject Reduction) can be transferred in a similar way.

We consider in this section the typed operational semantics for PAL^+ and show that the system has good meta-theoretic properties. Typed operational semantics (TOS) was developed for the type theory UTT in Goguen's thesis [Gog94], and a concise account of TOS for LF can be found in [Gog99].

4.1 A typed operational semantics for PAL^+

The typed operational semantics for PAL^+ has the following judgement forms:

- $\models \Gamma \to \Gamma'$ — context Γ has normal form Γ'.
- $\Gamma \models K \to K'$ — kind or parametric kind K is a well-typed and has normal form K' in context Γ.
- $\Gamma \models k \to k_0 \to k' : K$ — k, k_0, and k' are of kind K in Γ, and k has weak-head normal form k_0 and normal form k.

We shall use the following notations:

- For $\Delta \equiv \bar{x}{:}\bar{K}$ and $\Delta' \equiv \bar{x}'{:}\bar{K}'$ of the same length, we shall use the notation $\Gamma \models \Delta \to \Delta'$ to stand for the sequence of judgements $\Gamma \models K_1 \to K_1'$, $\Gamma, \Delta_1 \models K_2 \to K_2'$, ..., $\Gamma, \Delta_{n-1} \models K_n \to K_n'$.
- For $\bar{a}$, $\bar{b}$, $\bar{c}$, and $\bar{K}$ of the same length, we shall use $\Gamma \models \bar{a} \to \bar{b} \to \bar{c} : \bar{K}$ to stand for the sequence of judgements

$$\begin{aligned}
&\Gamma \models a_1 \to b_1 \to c_1 : K_1,\\
&\Gamma \models \delta_2 a_2 \to \delta_2 b_2 \to \delta_2 c_2 : \delta_2 K_2,\\
&...,\\
&\Gamma \models \delta_n a_n \to \delta_n b_n \to \delta_n c_n : \delta_n K_n,
\end{aligned}$$

  where δ_i $(i = 2, ..., n)$ is the substitution $[a_1, ..., a_{i-1}/x_1, ..., x_{i-1}]$.

The rules of TOS for PAL^+ are given in Figure 8.

4.2 Soundness and completeness

Let $\vdash^-$ stand for the (derivability of) judgements in the subsystem of PAL^+ *without* the substitution rules in Figure 2. We have the following theorems about completeness and soundness of the TOS for PAL^+, the proofs of which are omitted here.

Contexts

$$\frac{}{\models \langle\rangle \to \langle\rangle} \qquad \frac{\models \Gamma \to \Gamma' \quad \Gamma \models K \to K' \quad x \notin FV(\Gamma)}{\models \Gamma, x{:}K \to \Gamma', x{:}K'}$$

Kinds

$$\frac{\models \Gamma \to \Gamma'}{\Gamma \models \mathbf{Type} \to \mathbf{Type}} \qquad \frac{\Gamma \models A \to A_0 \to A' : \mathbf{Type}}{\Gamma \models El(A) \to El(A')}$$

Parametric kinds

$$\frac{\Gamma \models \Delta \to \Delta' \quad \Gamma, \Delta \models T \to T' \quad T \equiv \mathbf{Type} \text{ } \mathit{or} \text{ } El(A)}{\Gamma \models (\Delta)T \to (\Delta')T'}$$

Variables

$$\frac{\Gamma, x{:}K, \Gamma' \models K \to K'}{\Gamma, x{:}K, \Gamma' \models x \to x \to x : K'}$$

Instantiations

$$\frac{\begin{array}{c}\Gamma \models f \to f_0 \to f' : (\bar{x}{:}\bar{K})T \qquad f_0 \not\equiv [\Delta]t \\ \Gamma \models \bar{a} \to \bar{a}_0 \to \bar{a}' : \bar{K} \qquad \Gamma \models [\bar{a}/\bar{x}]T \to T'\end{array}}{\Gamma \models f[\bar{a}] \to f_0[\bar{a}] \to f'[\bar{a}'] : T'}$$

$$\frac{\begin{array}{c}\Gamma \models f \to [\bar{x}{:}\bar{K}_0]t \to f' : (\bar{x}{:}\bar{K})T \quad \Gamma \models \bar{a} \to \bar{a}_0 \to \bar{a}' : \bar{K} \\ \Gamma \models [\bar{a}/\bar{x}]t \to t_0 \to t' : T' \quad \Gamma \models [\bar{a}/\bar{x}]T \to T'\end{array}}{\Gamma \models f[\bar{a}] \to t_0 \to t' : T'}$$

Parametric abstractions

$$\frac{\Gamma \vdash \Delta \to \Delta' \quad \Gamma, \Delta \models t \to t_0 \to t' : T' \quad [\Delta']t' \text{ } \mathit{is\ not\ an\ } \eta\mathit{-redex}}{\Gamma \models [\Delta]t \to [\Delta]t \to [\Delta']t' : (\Delta')T'}$$

$$\frac{\Gamma \models \Delta \to \Delta' \quad \Gamma, \Delta \models t \to t_0 \to f[\bar{x}] : T' \quad \Gamma \models f \to f \to f : (\Delta')T'}{\Gamma \models [\Delta]t \to [\Delta]t \to f : (\Delta')T'}$$

Figure 8: TOS for PAL$^+$.

Theorem 4.1 (Completeness) *The TOS is complete wrt to* PAL$^+$*:*

- *If* $\models \Gamma \to \Gamma'$, *then* $\vdash^- \Gamma$ **valid**.
- *If* $\Gamma \models K \to K'$, *then* $\Gamma \vdash^- K$ *and* $\Gamma \vdash^- K = K'$.
- *If* $\Gamma \models k \to k_0 \to k' : K$, *then* $\Gamma \vdash^- k : K$, $\Gamma \vdash^- k = k_0 : K$, $\Gamma \vdash^- k = k' : K$, *and* $\Gamma \vdash^- K = K$.

Theorem 4.2 (Soundness) *The TOS is sound wrt to* PAL$^+$*:*

- *If* Γ **valid**, *then* $\models \Gamma \to \Gamma'$ *for some* Γ'.
- *If* $\Gamma \vdash K$, *then* $\Gamma \models K \to K'$ *for some* K'.
- *If* $\Gamma \vdash K_1 = K_2$, *then* $\Gamma \models K_1 \to K'$ *and* $\Gamma \models K_2 \to K'$ *for some* K'.
- *If* $\Gamma \vdash k : K$, *then* $\Gamma \models K \to K'$ *and* $\Gamma \models k \to k_0 \to k' : K'$ *for some* K', k_0, *and* k'.
- *If* $\Gamma \vdash k_1 = k_2 : K$, *then* $\Gamma \models K \to K'$, $\Gamma \models k_1 \to k_{10} \to k' : K'$, *and* $\Gamma \models k_2 \to k_{20} \to k' : K'$, *for some* K', k', k_{10}, *and* k_{20}.

4.3 Properties of PAL$^+$ and discussions

With the above theorems for the TOS of PAL$^+$ and the relationship between untyped reduction and TOS, we can easily show that the system PAL$^+$ has nice meta-theoretic properties. These include admissibility results of the structural rules (eg, the substitution rules), and computational properties such as Strong Normalisation, Church-Rosser, Subject Reduction.

Remark When PAL$^+$ is extended with new constants of the object type theories, we remark that the techniques developed for TOS in [Gog94] can be used to prove the meta-theoretic results of the object type theories such as UTT.

Remark When PAL$^+$ is extended with global and local definitions (as in Section 3), work by Severi and Poll [SP94] on extending PTS with definitions may be extended. Another more radical approach is to take let-expressions as basic and consider the meta-theory directly; but this requires further work which we leave for the future.

5 Implementation and experiments

PAL$^+$ is developed partly as an underlying framework for implementing new proof development systems. We expect that the simplicity gained would benefit implementation as well as the user (eg, it is expected that the use of de Bruijn indices would be simplified, and the treatment of meta-variables may be dealt with using the simple method as proposed in [Luo97] and implemented in Plastic [CL00].) Paul Callaghan at Durham has implemented a prototype of PAL$^+$, based on his implementation of LF in the system Plastic. We have done some experiments on proof development (eg, about inductive types and universes) based on the prototype implementation. A better understanding of what we can gain in implementations requires further research and a real development of a proof system.

6 Conclusion and discussions

We have presented and studied PAL^+, a logical framework simpler than those based on lambda-abstraction. Further extensions of meta-features such as coercive subtyping (see eg, [Luo99]) may be considered. Among the related work, a comparison with Cartmell's notion contextual categories [Car78, Car86] may lead to further insight.

Acknowledgements I am grateful to Paul Callaghan, Conor McBride, Bengt Nordström, Randy Pollack, and an LFM'00 referee, who have read drafts of this paper and given useful suggestions. Callaghan's prototype implementation of PAL^+ has been extremely useful. I am also grateful to the discussions on this topic I have had with members of the Computer-Assisted Reasoning Group at Durham.

References

[Car78] J. Cartmell. *Generalized Algebraic Theories and Contextual Category.* PhD thesis, University of Oxford, 1978.

[Car86] J. Cartmell. Generalized algebraic theories and contextual category. *Annals of Pure and Applied Logic*, 32, 1986.

[CL00] P. C. Callaghan and Z. Luo. Plastic: an implementation of typed LF with coercive subtyping and universes. Submitted to J. of Automated Reasoning, special issue on Logical Frameworks, 2000.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[Gog94] H. Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.

[Gog99] H. Goguen. Soundness of typed operational semantics for the logical framework. *Typed Lambda Calculi and Applications (TLCA'99)*, 1999.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science. IEEE*, 1987.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[Luo97] Z. Luo. Meta-variables and existential judgements. Notes, August 1997.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proof and Programs, LNCS*, 1994.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[SP94] P. Severi and E. Poll. Pure type systems with definitions. *Proc of LFCS'94, LNCS 813*, 1994.