A Modular Hierarchy of Logical Frameworks

Citation for the original published paper (version of record):

Adams, R. (2004). A Modular Hierarchy of Logical Frameworks. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 3085: 1-16. http://dx.doi.org/10.1007/978-3-540-24849-1_1

A Modular Hierarchy of Logical Frameworks

Robin Adams

University of Manchester
robin.adams@ma.man.ac.uk

Abstract. We present a method for defining logical frameworks as a collection of features which are defined and behave independently of one another. Each feature is a set of grammar clauses and rules of deduction such that the result of adding the feature to a framework is a conservative extension of the framework itself. We show how several existing logical frameworks can be so built, and how several much weaker frameworks defined in this manner are adequate for expressing a wide variety of object logics.

1 Introduction

Logical frameworks were invented because there were a large number of differing systems of logic, with no common language or environment for their investigation and implementation. However, we now find ourselves in the same situation with the frameworks themselves. There are many systems that are used as logical frameworks, and it is often difficult to compare them or share results between them. It is often much work to discover whether two frameworks can express the same class of object logics, or whether one is stronger or weaker than the other. If we are interested in metavariables, and we compare Pientka and Pfenning's work [1] with Jojgov's [2], it is difficult to see which differences are due to the different handling of metavariables, and which are due to differences in the underlying logical framework.

To redress this situation somewhat, I humbly present the first steps towards a common scheme within which a surprising number of different frameworks can be fitted. We take a modular approach to the design of logical frameworks, defining a framework by specifying a set of *features*, each of which is defined and behaves independently of the others. Together, all the frameworks that can be built from a given set of features form a *modular hierarchy* of logical frameworks.

We may give an informal definition of a feature thus:

> A *feature* F is a set of grammar clauses and rules of deduction such that, for any logical framework L, the result of adding F to L is a conservative extension of L.

(This cannot be made a formal definition, as we do not (yet) have a notion of "any logical framework".)

It is not surprising that features exist — one would expect, for example, that adding a definitional mechanism to a typing system should yield a conservative extension. Perhaps more surprising is the fact that such things as lambda-abstraction can be regarded as features. In fact, we shall show how a logical framework can be regarded as being nothing but a set of features. More precisely, we shall define a system that we call the *basic framework* **BF**, and a number of features that can be added to it, and we shall show how a number of existing frameworks can be built by selecting the appropriate features.

We shall also show that most of these features are unnecessary from the theoretical point of view — that is, a much smaller set of features suffices to express a wide variety of object logics. These 'unnecessary' features may well be desirable for implementation, of course.

It may be asked why we insist that our features always yield conservative extensions. This would seem to be severely limiting; in one's experience with typing systems, rarely are extensions conservative. For typing systems in general, this is true. But I would argue that logical frameworks are an exception. The fact that all the features presented here yield conservative extensions is evidence to this effect. And it would seem to be desirable when working with a logical framework — if we add a feature to widen the class of object logics expressible, for example, we still want the old object logics to behave as they did before.

We suggest that, if this work were taken further, it would be possible and desirable to define mechanisms such as metavariables or subtyping as features, and investigate their properties separately from one another and from any specific framework. If we did this for metavariables, for example, we would then know immediately what the properties of ELF with metavariables were, or Martin-Löf's Theory of Types with metavariables, or ...

2 Logical Frameworks

Let us begin by being more precise as to what we mean by a logical framework.

Broadly speaking, logical frameworks can be used in two distinct ways. The first is to define an object logic by means of a *signature*, a series of declarations of constants, equations, etc. The typable terms under that signature should then correspond to the terms, derivations, etc. of the object logic, using contexts to keep track of free variables and undischarged hypotheses. Examples include the Edinburgh Logical Framework [3] and Martin-Löf's Theory of Types [4]. We shall call a framework used in this way a *logic-modelling* framework.

The second is to use the logical framework as a *book-writing* system, as exemplified by the AUTOMATH family of systems [5]. The most important judgement form in such a framework is that which declares a book correct; the other judgement forms are only needed as auxiliaries for deriving this first form of judgement.

These two kinds of system behave in very similar ways. Any system of one kind can be used as a system of the other, by simply reading 'signature' for 'book', or vice versa. This is a striking fact, considering the difference in use. In

a system of the first kind, deriving that a signature is valid is just the first step in using an object logic; in a book-writing system, it is the only judgement form of importance. We shall take advantage of this similarity. Our features shall be written with logic-modelling frameworks in mind; it shall turn out that they are equally useful for building book-writing frameworks.

We consider a *logical framework* to consist of:

1. Disjoint, countably infinite sets of *variables* and *constants*.
2. A number of *syntactic classes* of *expressions*, defined in a BNF-style grammar by a set of *constructors*, each of which forms a member of one class from members of other classes, possibly binding variables.
3. Three syntactic classes that are distinguished as being the classes of *signature declarations*, *context declarations* and *judgement bodies*. Each signature declaration is specified to be either a declaration of a particular constant, or of none. Similarly, each context declaration is specified to be either a declaration of a particular variable or of none.
   We now define a *signature* to be a finite sequence of signature declarations, such that no two declarations are of the same constant. The *domain* of the signature Σ, $\mathrm{dom}\,\Sigma$, is then defined to be the sequence consisting of the constants declared in Σ, in order. Similarly, we define a *context* to be a finite sequence of context declarations, no two of the same variable, and we define its domain similarly.
   Finally, we define a *judgement* to be a string of one of two forms: either
   $$\Sigma \text{ sig}$$
   or
   $$\Gamma \vdash_\Sigma J$$
   where Σ is a signature, Γ a context, and J a judgement body.
4. A set of *defined operations and relations* on terms. Typically, these shall include one or more relations of *reducibility* and *convertibility*.
5. The final component of a logical framework is a set of *rules of deduction* which define the set of *derivable* judgements.

2.1 The Basic Framework BF

As is to be expected, **BF** is a very simple system. It allows: the declaration of variable and constant types; the declaration of variables and constants of a previously declared type; and the assertion that a variable or constant has the type with which it was declared, or is itself a type.

The grammar of **BF** is as follows:

$$\begin{aligned}
\text{Term} \quad & a ::= x \mid c \\
\text{Kind} \quad & A ::= \mathbf{Type} \mid \mathrm{El}(a) \\
\text{Signature Declaration} \quad & \delta ::= c : A \text{ of } c \\
\text{Context Declaration} \quad & \gamma ::= x : A \text{ of } x \\
\text{Judgement Body} \quad & J ::= \text{valid} \mid A \text{ kind} \mid a : A
\end{aligned}$$

The rules of deduction of **BF** are given in Figure 1.

$$\frac{}{\langle\rangle \text{ sig}} \qquad \frac{\vdash_\Sigma A \text{ kind}}{\Sigma, c : A \text{ sig}} \; (c \notin \text{dom}\, \Sigma)$$

$$\frac{\Sigma \text{ sig}}{\vdash_\Sigma \text{ valid}} \qquad \frac{\Gamma \vdash_\Sigma A \text{ kind}}{\Gamma, x : A \vdash_\Sigma \text{ valid}} \; (x \notin \text{dom}\, \Gamma)$$

$$\frac{\Gamma \vdash_\Sigma \text{ valid}}{\Gamma \vdash_\Sigma c : A} \; (c : A \in \Sigma) \qquad \frac{\Gamma \vdash_\Sigma \text{ valid}}{\Gamma \vdash_\Sigma x : A} \; (x : A \in \Gamma)$$

$$\frac{\Gamma \vdash_\Sigma \text{ valid}}{\Gamma \vdash_\Sigma \mathbf{Type} \text{ kind}} \qquad \frac{\Gamma \vdash_\Sigma a : \mathbf{Type}}{\Gamma \vdash_\Sigma \text{El}(a) \text{ kind}}$$

Fig. 1. The basic framework **BF**

3 Features and the Modular Hierarchy

A *feature* that *depends* on the logical framework L consists of any number of new entities: new syntactic classes, new constructors, new defined operations and relations and new rules of deduction. The new constructors may take arguments from new classes or those of L, bind new variables or those of L, and return expressions in new classes or those of L. In particular, they may create new signature declarations, context declarations and judgement bodies. Likewise, the new defined operations and relations should be defined on both old and new expressions, and the new rules of deduction may use both old and new judgement forms.

A feature may also introduce *redundancies*. A *redundancy* takes an old constructor and declares that it is to be replaced by a certain expression. That is, the constructor is no longer part of the grammar; wherever it appeared in a defined operation or relation or a rule of deduction, its place is to be taken by the given expression.

Now, if L' is any logical framework that extends L, we define the logical framework $L' + F$ in the obvious manner.

It should be noted that these rules of deduction are assumed to automatically extend themselves when future features are added. For example, if a feature contains the rule of deduction

$$\frac{\Gamma \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma M = M : A}$$

and we later introduce a new constructor for terms M, this rule is assumed to hold for the *new* terms M as well as the old.

(Formally defining features in such a way that this is possible requires explicitly defining classes of *meta-expressions* in the manner of [6]. We shall not go into such details here.)

Finally, we define:

Definition 1. *A feature F that depends on the set of features $\{F_1, F_2, \ldots\}$ is a feature that depends on the logical framework*

$$\mathbf{BF} + F_1 + F_2 + \cdots .$$

Thus, if F depends on $\{F_1, F_2, \ldots\}$, we can add F to any framework in the hierarchy that contains all of F_1, F_2, ... Note that we do not stipulate in this definition whether the set $\{F_1, F_2, \ldots\}$ is finite or infinite.

3.1 Parametrization

The first, and most important, of our features are those which allow the declaration of variables and constants with *parameters*. This mechanism is taken as fundamental by the systems of the AUTOMATH [5] family as well as PAL$^+$ [9], but can be seen as a subsystem of almost all logical frameworks. Parametrization provides a common core, above which the different forms of abstraction (λ-abstraction with typed or untyped domains, and with β- or $\beta\eta$-conversion, as well as PAL$^+$-style abstraction by let-definition) can be built as conservative extensions.

We define a series of features: **SPar** (1), **SPar** (2), **SPar** (3), ..., and also **LPar** (1), **LPar** (2), **LPar** (3), ... These extend one another in the manner shown in Figure 2.

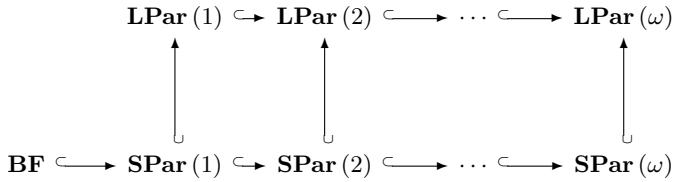

Fig. 2. The initial fragment of the modular hierarchy

BF already allows the declaration of constants in kinds: $c_1 : A$.

LPar (1) allows the declaration of constants in *first-order kinds*: $c_2 : (x_1 : A_1, \ldots, x_n : A_n)A$. This declaration indicates that c is a constant that takes *parameters* x_1 of kind A_1, ..., x_n of kind A_n, and returns a term $c_2[x_1, \ldots, x_n]$ of kind A.

LPar (2) allows the parameters themselves to have parameters: $c_3 : (x_1 : (x_{11} : A_{11}, \ldots, x_{1k_1} : A_{1k_1})A_1, \ldots, x_n : (x_{n1} : A_{n1}, \ldots, x_{nk_n} : A_{nk_n})A_n)A$. **LPar** (3) allows these second-order parameters to have parameters, and so on. Similarly for declaration of variables.

We also define the feature **LPar** (ω) to be the union of all these features, allowing any level of parametrization.

The sequence of features **SPar** (n) is similar; the only difference is that, in **SPar** (n), every parameter must be in a *small* kind; that is, each A_i, A_{ij},

... above must be of the form $\mathrm{El}(a)$; it cannot be **Type**. (In **SPar** (n), A itself, the rightmost kind, can be **Type** in the declaration of a constant, but not in a declaration of a variable.)

The full details of these features are as follows:

Parameters in Small Kinds, SPar (n)

Grammar Before we can introduce the new grammar constructors, we need to make a few definitions.

We define an m-th order *pure context* by recursion on m as follows. An m-th order pure context is a string of the form

$$(x_1 : (\Delta_1)\,\mathrm{El}(a_1), \ldots, x_k : (\Delta_k)\,\mathrm{El}(a_k))$$

where each x_i is a variable, all distinct, Δ_i a pure context of order $< m$, and a_i a term. Its *domain* is $(x_1, \ldots, x_k)$.

We define an *abstraction* to be a string of the form

$$[\boldsymbol{x}]M$$

where $\boldsymbol{x}$ is a sequence of distinct variables, and M a term. We take each member of $\boldsymbol{x}$ to be bound within M in this abstraction, and we define free and bound variables and identify all our expressions up to α-conversion in the usual manner. We write $\hat{}M, \hat{}N, \ldots$ for arbitrary abstractions. It is important to note that these are *not* first-class objects of every framework that contains **SPar** (n).

Now, we add the following clause to the grammar:

$$z[\hat{}\boldsymbol{M}]$$

is a term, where z is a variable or constant, and $\hat{}\boldsymbol{M}$ a sequence of abstractions.. This clause subsumes the grammar of **BF**, for $x()$ and $c()$ are terms when x is a **0**-ary variable and c a **0**-ary constant.

We also allow declarations of the form

$$c : (\Delta)A$$

in the signature, where c is a constant, Δ a pure context of order $\leq n$, and A a kind; and those of the form

$$x : (\Delta)\,\mathrm{El}(a)$$

in the context, where x is a variable, Δ a pure context of order $\leq n$, and a a term. Again, these subsume those of **BF**.

Defined Operations We define the operation of *instantiation* as follows. This operation takes the place of substitution; we cannot substitute for a variable of kind $(\Delta)A$, as we have no first-class objects in such a kind — indeed, we have no such kind yet. But it is possible define a term

$$\{\hat{}M_1/x_1, \ldots, \hat{}M_m/x_m\}M$$

where M is a term, and, for $i = 1, \ldots, m$, $\hat{}M_i \equiv [\boldsymbol{y_i}]M_i$ is an abstraction, and x_i a variable, in such a way that first-class abstractions are never needed. They can, later, be added as a conservative extension. It may aid the understanding of the definition of instantiation to note that, once abstractions are added,

$$\{\hat{}M_1/x_1, \ldots, \hat{}M_m/x_m\} \text{ is the normal form of } [\hat{}M_1/x_1, \ldots, \hat{}M_m/x_m]$$

The definition is as follows:

$$\begin{aligned} \{\hat{}\boldsymbol{M}/\boldsymbol{x}\}z(\hat{}\boldsymbol{N}) &\equiv z(\{\hat{}\boldsymbol{M}/\boldsymbol{x}\}\hat{}\boldsymbol{N}) \\ &\qquad \text{(if } z \text{ is a constant or a variable not in } \boldsymbol{x}) \\ \{\hat{}\boldsymbol{M}/\boldsymbol{x}\}x_i(\hat{}\boldsymbol{N}) &\equiv \{\{\hat{}\boldsymbol{M}/\boldsymbol{x}\}\hat{}\boldsymbol{N}/\boldsymbol{y_i}\}M_i \end{aligned}$$

We also need a defined judgement form. If $\hat{}\boldsymbol{M}$ is an abstraction sequence, and Δ a pure context, we define a set of judgements

$$\Gamma \Vdash_\Sigma \hat{}\boldsymbol{M} :: \Delta$$

(read: under signature Σ and context Γ, $\hat{}\boldsymbol{M}$ *satisfies* Δ). The definition is as follows. Let

$$\Delta \equiv x_1 : (\Delta_1)\,\mathrm{El}(a_1), \ldots, x_m : (\Delta_m)\,\mathrm{El}(a_m)$$

and let

$$\hat{}M_i \equiv [\boldsymbol{y_i}]M_i\,.$$

We take $\Gamma \Vdash_\Sigma \hat{}\boldsymbol{M} :: \Delta$ to be defined only when $\boldsymbol{y_i} \equiv \Delta_i$ for all i.

$$\Gamma \Vdash_\Sigma \hat{}\boldsymbol{M} :: \Delta$$

is the following set of judgements:

$$\begin{gathered} \Gamma, \Delta_1 \vdash_\Sigma M_1 : \mathrm{El}(a_1) \\ \Gamma, \{\hat{}M_1/x_1\}\Delta_2 \vdash_\Sigma M_2 : \mathrm{El}(\{\hat{}M_1/x_1\}a_2) \\ \Gamma, \{\hat{}M_1/x_1, \hat{}M_2/x_2\}\Delta_3 \vdash_\Sigma M_3 : \mathrm{El}(\{\hat{}M_1/x_1, \hat{}M_2/x_2\}a_3) \\ \vdots \\ \Gamma, \{\hat{}M_1/x_1, \ldots, \hat{}M_{m-1}/x_{m-1}\}\Delta_m \vdash_\Sigma M_m : \mathrm{El}(\{\hat{}M_1/x_1, \ldots, \hat{}M_{m-1}/x_{m-1}\}a_m) \end{gathered}$$

In the case $m = 0$, we take $\Gamma \Vdash_\Sigma \hat{}\boldsymbol{M} :: \Delta$ (i.e. $\Gamma \Vdash_\Sigma \langle\rangle :: \langle\rangle$) to be the single judgement

$$\Gamma \vdash_\Sigma \mathrm{valid}\,.$$

Rules of Deduction The rules of deduction in $\mathbf{SPar}\,(n)$ are as follows:

$$\frac{\Delta \vdash_\Sigma \mathrm{valid}}{\Sigma, c : (\Delta)\mathbf{Type}\ \mathrm{sig}}\ (c \notin \mathrm{dom}\,\Sigma) \qquad \frac{\Delta \vdash_\Sigma a : \mathbf{Type}}{\Sigma, c : (\Delta)\,\mathrm{El}(a)\ \mathrm{sig}}\ (c \notin \mathrm{dom}\,\Sigma)$$

$$\frac{\Gamma, \Delta \vdash_\Sigma a : \mathbf{Type}}{\Gamma, x : (\Delta)\,\mathrm{El}(a) \vdash_\Sigma \mathrm{valid}}\ (x \notin \mathrm{dom}\,\Gamma)$$

$$\frac{\Gamma \Vdash_\Sigma \hat{\boldsymbol{M}} :: \Delta}{\Gamma \vdash_\Sigma c[\hat{\boldsymbol{M}}] : \{\hat{\boldsymbol{M}}/\operatorname{dom}\Delta\}A}\,(c : (\Delta)A \in \Sigma)$$

$$\frac{\Gamma \Vdash_\Sigma \hat{\boldsymbol{M}} :: \Delta}{\Gamma \vdash_\Sigma x[\hat{\boldsymbol{M}}] : \mathrm{El}(\{\hat{\boldsymbol{M}}/\operatorname{dom}\Delta\}a)}\,(x : (\Delta)\,\mathrm{El}(a) \in \Gamma)$$

Finally, **SPar** (ω) is defined to be the union of all the features **SPar** (n).

Parameters in Large Kinds, LPar (n) The features **LPar** (n) and **LPar** (ω) are defined in exactly the same manner as **SPar** (n) and **SPar** (ω), with only two differences. The first is the definition of pure context, which now allows **Type** to appear:

An *m-th order pure context* is a string of the form

$$(x_1 : (\Delta_1)A_1, \ldots, x_k : (\Delta_k)A_k)$$

where each x_i is a variable, all distinct, Δ_i is a pure context of order $< m$, and A_i is either **Type** or $\mathrm{El}(a_i)$ for some term a_i.

The second is that large kinds are permitted in context declarations as well as signature declarations; that is, we allow declarations of the form $x : (\Delta)A$ in the context, where x is a variable, Δ a pure context of appropriate order, and A either **Type** or $\mathrm{El}(a)$ for some term a.

3.2 Lambda Abstraction

We can now, if we wish, build in traditional λ-abstraction. It should be noted that this does not change the class of object theories that can be expressed by the framework.

We can make these abstractions typed or untyped (i.e. explicitly include the domain or not), and we can choose to use β or $\beta\eta$-conversion. These two choices lead to four features that can be added to a framework. We shall denote them $\lambda^{\mathrm{t}}_\beta$, $\lambda^{\mathrm{ut}}_\beta$, $\lambda^{\mathrm{t}}_{\beta\eta}$, $\lambda^{\mathrm{ut}}_{\beta\eta}$. We shall give here the details of $\lambda^{\mathrm{t}}_\beta$; the others are very similar.

We shall describe here a feature $\lambda^{\mathrm{t}}_\beta$ to be built on top of **BF** + **LPar** (ω). It would be easy to change the details to give a feature that could be added to **BF** + **LPar** (n), **BF** + **SPar** (n), or **BF** + **SPar** (ω).

We add the following clauses to the grammar:

$$\begin{array}{ll} \text{Term} & M ::= \cdots \mid [x : A]M \mid M[M] \\ \text{Kind} & A ::= \cdots \mid (x : A)A \end{array}$$

There are two redundancies in the feature $\lambda^{\mathrm{t}}_\beta$. The first: let c be a constant, where Let

$$c : (x_1 : (\Delta_1)A_1, \ldots, x_m : (\Delta_m)A_m)A$$

be in the signature, where

$$\Delta_i \equiv (x_{i1} : (\Delta_{i1})A_{i1}, \ldots, x_{ik_i} : (\Delta_{ik_i})A_{ik_i})A_i\,.$$

Then we identify the term

$$c[[\boldsymbol{x_1}]M_1, \ldots, [\boldsymbol{x_m}]M_m]$$

with the base term

$$\begin{aligned} &c[[x_{11} : (\Delta_{11})A_{11}] \cdots [x_{1k_1} : (\Delta_{1k_1})A_{1k_1}]M_1] \cdots \\ &\quad [[x_{m1} : (\Delta_{m1})A_{m1}] \cdots [x_{mk_m} : (\Delta_{mk_m})A_{mk_m}]M_m] \end{aligned}$$

The second is a similar redundancy for terms beginning with a variable.

We define the relations of β-reduction, β-conversion, etc. on our classes of terms in the usual manner, based on the contraction

$$([x : (\Delta)A]M)[N] \rightsquigarrow_\beta [N/x]M\,.$$

The rules of deduction in $\lambda^{\mathrm{t}}_\beta$ are now:

$$\frac{\Gamma, x : A \vdash_\Sigma B \text{ kind}}{\Gamma \vdash_\Sigma (x : A)B \text{ kind}}$$

$$\frac{\vdash_\Sigma A \text{ kind}}{\Sigma, c : A \text{ sig}}\,(c \notin \operatorname{dom}\Sigma) \qquad \frac{\Gamma \vdash_\Sigma A \text{ kind}}{\Gamma, x : A \vdash_\Sigma \text{valid}}\,(x \notin \operatorname{dom}\Gamma)$$

$$\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma [x : A]M : (x : A)B} \qquad \frac{\Gamma \vdash_\Sigma M : (x : A)B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M[N] : [N/x]B}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma B \text{ kind}}{\Gamma \vdash_\Sigma M : B}\,(A =_\beta B)$$

3.3 Other Features

We present a summary of other features in Figures 3 and 4. Each of these features depends on $\mathbf{SPar}\,(\omega)$. It would be easy enough to write a version dependent on $\mathbf{SPar}\,(n)$ for some finite n.

3.4 Conservativity Results

The guiding principle behind the modular hierarchy is that the features are defined, and behave, independently of one another. The formal result that corresponds to this principle is:

Theorem 1. *If L is a logical framework in the hierarchy, and F a feature such that every feature on which F depends is present in L, then $L + F$ is a conservative extension of L.*

This theorem can be proven for the finitely many features we have presented in this paper. We prove that, if J is a judgement of L derivable in $L + F$, then J is derivable in L, by direct induction on the derivation of J in $L + F$. The only non-trivial cases are the conversion rules; these require the Church-Rosser property to be proven for the set of typable terms. This is never too demanding; even the case of $\beta\eta$-conversion can be handled using, for example, the techniques of [7], because the frameworks, as type systems, are very simple: there is a single predicative universe and no reflection.

Global Definition of Constants, **cdef** Depends on $\mathbf{SPar}\,(\omega)$.

$$\text{Signature Declaration } \gamma ::= \cdots \mid c^{\alpha}[\Delta^{\alpha}] := M : A$$

If $c[\Delta] := M : A$ is in the signature, the following is a reduction rule:

$$c[\hat{\boldsymbol{N}}] \rightsquigarrow_{\delta_c} \{\hat{\boldsymbol{N}}/\operatorname{dom}\Delta\}$$

$$\frac{\Delta \vdash_{\Sigma} M : A}{\Sigma, c[\Delta] := M : A \text{ sig}}\,(c \notin \operatorname{dom}\Sigma) \qquad \frac{\Gamma \vdash_{\Sigma} M : A \quad \Gamma \vdash_{\Sigma} B \text{ kind}}{\Gamma \vdash_{\Sigma} M : B}\,(\Gamma \vdash_{\Sigma} A =_{\delta_c} B)$$

$$\frac{\Gamma \Vdash_{\Sigma} \hat{\boldsymbol{N}} :: \Delta}{\Gamma \vdash_{\Sigma} c[\hat{\boldsymbol{N}}] : \{\hat{\boldsymbol{N}}/\operatorname{dom}\Delta\}A}\,(c[\Delta] := M : A \in \Sigma)$$

Global Definition of Variables, **vdef** Depends on $\mathbf{SPar}\,(\omega)$.

$$\text{Context Declaration } \delta ::= \cdots \mid x^{\alpha}[\Delta^{\alpha}] := M : A$$

If $x^{\alpha}[\Delta^{\alpha}] := M^{\beta} : A^{\beta}$ is in the context, the following is a reduction rule:

$$x[\hat{\boldsymbol{N}}] \rightsquigarrow_{\delta_v} \{\hat{\boldsymbol{N}}/\operatorname{dom}\Delta\}M$$

$$\frac{\Gamma, \Delta \vdash_{\Sigma} M : A}{\Gamma, x[\Delta] := M : A \vdash_{\Sigma} \text{valid}}\,(x \notin \operatorname{dom}\Gamma) \quad \frac{\Gamma \vdash_{\Sigma} M : A \quad \Gamma \vdash_{\Sigma} B \text{ kind}}{\Gamma \vdash_{\Sigma} M : B}\,(\Gamma \vdash_{\Sigma} a =_{\delta_v} b)$$

$$\frac{\Gamma \Vdash_{\Sigma} \hat{\boldsymbol{N}} :: \Delta}{\Gamma \vdash_{\Sigma} x[\hat{\boldsymbol{N}}] : \{\hat{\boldsymbol{N}}/\operatorname{dom}\Delta\}A}\,(x[\Delta] := M : A \in \Gamma)$$

Local Definitions, **let** Depends on **vdef**.

$$\begin{array}{ll}\text{Term} & M ::= \cdots \mid \operatorname{let} x^{\alpha}[\Delta^{\alpha}] := M : A \operatorname{in} M \\ \text{Kind} & A ::= \cdots \mid \operatorname{let} x^{\alpha}[\Delta^{\alpha}] := M : A \operatorname{in} A\end{array}$$

$$\begin{array}{l}\operatorname{let} v[\Delta] = M : A \operatorname{in} N \rightsquigarrow_{\delta} \{[\operatorname{dom}\Delta]M/v\}N \\ \operatorname{let} v[\Delta] = M : A \operatorname{in} K \rightsquigarrow_{\delta} \{[\operatorname{dom}\Delta]M/v\}K\end{array}$$

$$\frac{\Gamma, v[\Delta] = M : A \vdash_{\Sigma} K \text{ kind}}{\Gamma \vdash_{\Sigma} \operatorname{let} v[\Delta] = M : A \operatorname{in} K \text{ kind}} \qquad \frac{\Gamma \vdash_{\Sigma} M : A \quad \Gamma \vdash_{\Sigma} B \text{ kind}}{\Gamma \vdash_{\Sigma} M : B}\,(A =_{\delta} B)$$

$$\frac{\Gamma, v[\Delta] = M : A \vdash_{\Sigma} N : K}{\Gamma \vdash_{\Sigma} \operatorname{let} v[\Delta] = M : A \operatorname{in} N : \operatorname{let} v[\Delta] = M : A \operatorname{in} K}$$

Fig. 3. Miscellaneous features

Judgemental Equality, **eq** Depends on $\mathbf{SPar}\,(\omega)$.

$$\begin{array}{ll}\text{Judgement body} & J ::= \cdots \mid M = M : A \mid A = A \\ \text{Signature declaration} & \delta ::= \cdots \mid (\Delta)(M = M : A)\end{array}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M = N : A}\,(M = N) \qquad \frac{\Gamma \vdash_\Sigma A \text{ kind} \quad \Gamma \vdash_\Sigma B \text{ kind}}{\Gamma \vdash_\Sigma A = B}$$

$$\frac{\Delta \vdash_\Sigma M : A \quad \Delta \vdash_\Sigma N : A}{\Sigma, (\Delta)(M = N : A)\ \text{sig}}$$

$$\frac{\Gamma \Vdash_\Sigma \hat{\boldsymbol{P}} :: \Delta}{\Gamma \vdash_\Sigma \{\hat{\boldsymbol{P}}/\operatorname{dom}\Delta\}M = \{\hat{\boldsymbol{P}}/\operatorname{dom}\Delta\}N : \{\hat{\boldsymbol{P}}/\operatorname{dom}\Delta\}A}\,((\Delta)(M = N : A) \in \Sigma)$$

$$\frac{\Gamma \vdash_\Sigma M = N : A}{\Gamma \vdash_\Sigma N = M : A} \qquad \frac{\Gamma \vdash_\Sigma M = N : A \quad \Gamma \vdash_\Sigma N = P : A}{\Gamma \vdash_\Sigma M = P : A}$$

$$\frac{\Gamma \vdash_\Sigma A = B}{\Gamma \vdash_\Sigma B = A} \qquad \frac{\Gamma \vdash_\Sigma A = B \quad \Gamma \vdash_\Sigma B = C}{\Gamma \vdash_\Sigma A = C}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma A = B}{\Gamma \vdash_\Sigma M : B} \qquad \frac{\Gamma \vdash_\Sigma M = N : A \quad \Gamma \vdash_\Sigma A = B}{\Gamma \vdash_\Sigma M = N : B}$$

Fig. 4. Miscellaneous features

4 Existing Logical Frameworks

We show here how several existing logical frameworks are equivalent to systems that are built out of the features we have introduced. As well as the frameworks we have already mentioned, we deal with Luo's frameworks LF [8].

$$\begin{aligned}\text{PAL} &= \mathbf{BF} + \mathbf{LPar}\,(1) + \mathbf{cdef} \\ \text{AUT-68} &\simeq \mathbf{BF} + \mathbf{SPar}\,(\omega) + \lambda^{\mathrm{t}}_{\beta} + \mathbf{LPar}\,(1) + \mathbf{cdef} \\ \text{AUT-QE} &\simeq \mathbf{BF} + \mathbf{LPar}\,(\omega) + \lambda^{\mathrm{t}}_{\beta} + \mathbf{cdef} \\ \text{ELF} &= \mathbf{BF} + \mathbf{SPar}\,(\omega) + \lambda^{\mathrm{t}}_{\beta} \\ \text{Martin-Löf's Theory of Types} &= \mathbf{BF} + \mathbf{LPar}\,(\omega) + \lambda^{\mathrm{ut}}_{\beta\eta} + \mathbf{eq} \\ \text{LF} &= \mathbf{BF} + \mathbf{LPar}\,(\omega) + \lambda^{\mathrm{t}}_{\beta\eta} + \mathbf{eq}\end{aligned}$$

(Note: in the second line, the version of $\lambda^{\mathrm{t}}_{\beta}$ included is built on top of $\mathbf{SPar}\,(\omega)$ only, not $\mathbf{LPar}\,(1)$. $AUT-68$ allows the declaration of constants with first-order parameters, but does not allow such lambda-abstractions to be formed.)

The notion of equivalence with which we are working is the possiblity of defining a translations between the members of the syntactic classes of the two frameworks, such that the translate of each rule of deduction of one is admissible

in the other, and which are inverses of one another up to the relevant notion of convertibility within each framework.

For the lines in which we have used an equality sign, such translations can be given; the correspondence between the existing framework and the one produced by the hierarchy is fairly close. For the first two 'AUT-' frameworks, the correspondence is not nearly as neat. There is a correspondence between the hierarchy framework and a *variant* of the AUTOMATH framework. This variant removes the distinction between, for example, the constant defined by

$$(\mathbf{0}, x, -, A), (x, c, \mathtt{PN}, B)$$

(defining c with *parameter* $x : A$ inside the kind B) and that defined by

$$(\mathbf{0}, c, \mathtt{PN}, [x : A]B)$$

(defining c with no parameters inside the kind $[x : A]B$). It also replaces AUTOMATH's system of declaring variables with a more orthodox system of contexts.

It is possible to make the correspondence in these two cases better; and it is also possible to tighten the other four, so that the translations are inverses up to identity (that is, α-conversion), not just convertibility. However, doing so requires a large number of features to be defined, with hair-splitting distinctions being made between them. It is not at all clear that the advantages are worth this cost.

To build PAL$^+$ in the hierarchy, there are two possibilities. Firstly, we could write a feature that introduces classes of α-ary terms and kinds for every arity α, in a similar manner to $\lambda^{\mathsf{t}}_{\beta}$, but the only such terms are the α-ary variables and constants. Then we could build on top of this a features similar to **vdef** and **let**, but allowing global and local definitions of any arity term and kind. Putting these three features on top of $\mathbf{BF} + \mathbf{LPar}\,(\omega) + \mathbf{eq}$ yields a framework equivalent to PAL$^+$.

Alternatively, we could build features similar to **vdef** and **let** on top of $\mathbf{BF}{+}\mathbf{LPar}\,(\omega){+}\mathbf{eq}{+}\lambda^{\mathsf{t}}_{\beta\eta}$, including a redundancy that identifies $[x_1 : A_1] \cdots [x_n : A_n]M$ with $\mathrm{let}\, v[x_1 : A_1, \ldots, x_n : A_n] = M : A \,\mathrm{in}\, v$, where A is an inferred kind for M.

5 Use of Frameworks

Note that all the existing frameworks we have considered (with the notable exception of PAL) use either $\mathbf{SPar}\,(\omega)$ or $\mathbf{LPar}\,(\omega)$. This is natural if one is approaching frameworks from the point of view of the lambda calculus; these are the easiest features to define as (say) PTSs. However, it is overkill. For:

Theorem 2. – *The grammar of propositional logic, and Hilbert-style rules of inference, are representable in* $\mathbf{BF} + \mathbf{SPar}\,(1)$.
– *The grammar of predicate logic, and natural deduction-style rules of inference, are representable in* $\mathbf{BF} + \mathbf{SPar}\,(2)$.

– *Martin-Löf's Theory of Sets is representable in* $\mathbf{BF} + \mathbf{LPar}\,(2) + \mathbf{eq}$.

We only have space here to partially justify a few of these claims. We shall show how to build an arbitrary first-order theory in $\mathbf{BF} + \mathbf{SPar}\,(2)$, and how W-types are built within $\mathbf{BF} + \mathbf{LPar}\,(2) + \mathbf{eq}$.

For a first-order theory in $\mathbf{BF} + \mathbf{SPar}\,(2)$, the signature consists of:

$$
\begin{aligned}
&\text{term} : \mathbf{Type}\\
&\quad F : (x_1 : \mathrm{El}(\text{term}),\ \ldots,\ x_n : \mathrm{El}(\text{term}))\,\mathrm{El}(\text{term})\\
&\qquad \text{for each } n\text{-ary function symbol } F \text{ in the language}\\
&\text{prop} : \mathbf{Type}\\
&\quad P : (x_1 : \mathrm{El}(\text{term}),\ \ldots,\ x_n : \mathrm{El}(\text{term}))\,\mathrm{El}(\text{prop})\\
&\qquad \text{for each } n\text{-ary predicate symbol } P \text{ in the language}\\
&\quad \to : (x : \mathrm{El}(\text{prop}),\ y : \mathrm{El}(\text{prop}))\,\mathrm{El}(\text{prop})\\
&\quad \forall : (p : (x : \mathrm{El}(\text{term}))\,\mathrm{El}(\text{prop}))\,\mathrm{El}(\text{prop})\\
&\mathrm{Prf} : (x : \mathrm{El}(\text{prop}))\mathbf{Type}\\
&\to I : (p, q : \mathrm{El}(\text{prop}),\ H : (x : \mathrm{El}(\mathrm{Prf}[p]))\,\mathrm{El}(\mathrm{Prf}[q]))\,\mathrm{El}(\mathrm{Prf}[\to [p, q]])\\
&\to E : (p, q : \mathrm{El}(\text{prop}),\ H_1 : \mathrm{El}(\mathrm{Prf}[\to [p, q]]),\ H_2 : \mathrm{El}(\mathrm{Prf}[p]))\,\mathrm{El}(\mathrm{Prf}[q])\\
&\forall I : (p : (x : \mathrm{El}(\text{term}))\,\mathrm{El}(\text{prop}),\ H : (x : \mathrm{El}(\text{term}))\,\mathrm{El}(\mathrm{Prf}[p[x]]))\,\mathrm{El}(\mathrm{Prf}[\forall [p]])\\
&\forall E : (p : (x : \mathrm{El}(\text{term}))\,\mathrm{El}(\text{prop}),\ t : \mathrm{El}(\text{term}),\ H : \mathrm{El}(\mathrm{Prf}[\forall [p]]))\,\mathrm{El}(\mathrm{Prf}[p[t]])
\end{aligned}
$$

Theorem 3. *1. There is a bijection ρ between the terms with free variables among $x_1, \ldots, x_n$ in the first-order language, and the terms M such that*

$$x_1 : \mathrm{El}(\text{term}), \ldots, x_n : \mathrm{El}(\text{term}) \vdash_\Sigma M : \mathrm{El}(\text{term})$$

2. There is a bijection σ between the formulas with free variables among x_1, $\ldots$, x_n in the first-order language, and the terms M such that

$$x_1 : \mathrm{El}(\text{term}), \ldots, x_n : \mathrm{El}(\text{term}) \vdash_\Sigma M : \mathrm{El}(\text{prop})$$

3. Let ϕ, ψ_1, $\ldots$, ψ_m be formulas with free variables among $x_1, \ldots, x_n$. Then ϕ is provable from hypothese ψ_1, $\ldots$, ψ_m iff there is a term M such that

$$\begin{gathered}x_1 : \mathrm{El}(\text{term}), \ldots, x_n : \mathrm{El}(\text{term}), y_1 : \mathrm{El}(\mathrm{Prf}[\sigma(\psi_1)]), \ldots, y_m : \mathrm{El}(\mathrm{Prf}[\sigma(\psi_m)])\\ \vdash_\Sigma M : \mathrm{El}(\mathrm{Prf}[\sigma(\phi)])\end{gathered}$$

Notice that the correspondance between the entities of the object logic and the terms of the logical framework is a bijection up to *identity* (that is, α-conversion), not up to convertibility; indeed, in a framework whose only features are $\mathbf{SPar}\,(n)$ and $\mathbf{LPar}\,(n)$, there is no such thing as convertibility. This theorem is much easier to prove than most adequacy theorems, because the correspondence between the framework and the object logic is so much closer than in a traditional logical framework.

We now show how to build W-types within $\mathbf{BF} + \mathbf{LPar}\,(2) + \mathbf{eq}$. In the following, we shall suppress instances of El, and use η-contractions; e.g. we write $W[A, B]$ for $W[A, [x : A]B[x]]$.

$$
\begin{array}{r}
W : (A : \mathbf{Type},\ B : (A)\mathbf{Type})\mathbf{Type}, \\
\sup : (A : \mathbf{Type},\ B : (A)\mathbf{Type},\ a : A,\ b : (B[a])W[A, B])W[A, B] \\
E_W : (A : \mathbf{Type},\ B : (A)\mathbf{Type},\ C : (W[A, B])\mathbf{Type}, \\
\quad f : (x : A,\ y : (B[x])W[A, B], \\
\quad g : (v : B[x])C[y[v]])C[\sup[A, B, x, y]], \\
\quad z : W[A, B])C[z],
\end{array}
$$

$$
\begin{array}{l}
(A : \mathbf{Type},\ B : (A)\mathbf{Type},\ C : (W[A, B])\mathbf{Type}, \\
\ f : (x : A,\ y : (B[x])W[A, B],\ g : (v : B[x])C[y[v]])C[\sup[A, B, x, y]], \\
\ a : A,\ b : (B[a])W[A, B]) \\
\quad E_W[A, B, C, f, g, \sup[A, B, a, b]] = f[a, b, [v : B[x]]E_W[A, B, C, f, g, y[v]]] \\
\quad\ : C[\sup[A, B, a, b]]
\end{array}
$$

6 Conclusion

We have given a modular method for defining logical frameworks, and shown that it captures, up to a reasonable notion of equivalence, several existing logical frameworks. It has revealed common subsystems between these frameworks that may not otherwise have been found — it is doubtful, for example, that one would have discovered the fact that there is a system $\mathbf{BF} + \mathbf{SPar}\,(\omega)$ which can be conservatively embedded in both ELF and Martin-Löf's Theory of Types without this work. It has revealed much weaker frameworks than we are accustomed to using, that may prove advantageous for theoretical work, such as proving adequacy theorems. And, finally, it may yet provide a method for defining features in a generic manner such that they can be added to any logical framework, and their properties studied independently of any framework.

Future and Related Work

The only work of a similar nature of which I am aware is the Tinkertype system [10]. There are striking similarities between the two systems. However, I believe this work is different in character. Tinkertype's features cannot be defined separately, and do not behave independently; they certainly do not always yield conservative extensions. While this would not be a desideratum for type systems in general, as with which Tinkertype deals, I believe it is important for logical frameworks.

In the future, as well as the obvious matters of defining more features, capturing more aspects of logical frameworks, and exploring the properties of features independently of one another, it would be interesting to see if we could lay down general conditions C_1, C_2, ... on features, and prove results such as:

> Any feature with conditions C_1, C_2, ... yields a conservative extension of any framework composed solely of features that satisfy conditions C_1, C_2, ...

It would also be interesting to see if we could prove generalised adequacy results using the hierarchy, and give general definitions of semantics for an object theory and prove generalised soundness and completeness results.

References

1. Pientka, B., Pfenning, F.: Optimizing higher-order pattern unification (2003)
2. Jojgov, G.I.: Holes with binding power. In: Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Selected Papers. Volume 2646 of LNCS., Springer-Verlag Heidelberg (2003) 162 – 181
3. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987, New York, IEEE Computer Society Press (1987) 194–204
4. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory: an Introduction. Oxford University Press (1990)
5. Nederpelt, R.P., Geuvers, J.H., Vrijer, R.C.D., eds.: Selected Papers on AUTOMATH. Number 133 in Studies in Logic and the Foundations of Mathematics. North-Holland (1994)
6. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. Theoretical Computer Science **121** (1993) 279–308
7. Ghani, N.: Eta-expansions in dependent type theory — the calculus of constructions. In de Groote, P., Hindley, J.R., eds.: Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97), Nancy, France, Springer-Verlag LNCS 1210 (1997) 164–180
8. Luo, Z.: Computation and Reasoning: A Type Theory for Computer Science. Number 11 in International Series of Monographs on Computer Science. Oxford University Press (1994)
9. Luo, Z.: PAL$^+$: a lambda-free logical framework. Journal of Functional Programming **13** (2003) 317–338
10. Levin, M.Y., Pierce, B.C.: Tinkertype: A language for playing with formal systems. Technical report (2000)