

A Logical Framework with Dependently Typed Records*

Thierry Coquand

Chalmers Tekniska Högskola, Sweden

Randy Pollack

Edinburgh University, U.K.

Makoto Takeyama

*National Institute of Advanced Industrial Science and
Technology, Japan*

1. Introduction

Our long term goal is a system to formally represent complex structured mathematical objects, and proofs and computation on such objects; e.g. a foundational computer algebra system. Our approach is informed by the long development of module systems for functional programming based on dependent record types as signatures [22]. For our logical purposes, however, we want a dependently typed base language. In this paper we propose an extension of Martin-Löf's logical framework [25, 21] with dependently typed records, and present the semantic foundation and the typechecking algorithm of our system. Some of the work is formally checked in Coq [8].¹ We have also implemented and experimented with several related systems. Our proposal combines a semantic foundation, provably sound typechecking, good expressiveness (e.g. subtyping, sharing) and first-class higher-order modules.

The development of functional programming modules has addressed many aspects of the problem, such as use of *manifest* or *transparent* fields to control the information available in a signature, signature *strengthening*, type abstraction, sharing and subtyping [19, 15, 20]. The problem of modularity is not, however, closed, with much current research into first-class higher-order modules, recursive modules and mixins.

There has also been work on dependently typed records over dependent base languages. A first practical implementation is described in [3], however without semantic foundation. An original extension of Martin-Löf's logical framework is given in [6], however it lacks manifest fields to express sharing, and lacks metamathematical analysis. A general approach to adding modules on top of a Pure Type System

*Version of February 11, 2005 (639); a previous version appears in TLCA'03.

Address for correspondence: Randy Pollack, Computer Science, King's Buildings, Edinburgh EH9 3JZ, U.K. email rap@inf.ed.ac.uk

¹The formal development covers sections 2-4, is slightly different than that in this paper, and uses several unproved assumptions. It is available from <http://homepages.inf.ed.ac.uk/rap/export/LogicalFramework.v8>.

is described in [10, 7], but these modules are not first-class. [27] reviews much of this work, and gives an interpretation of dependently typed records in a type theory with inductive-recursive definition; however this approach does not address subtyping, and is infeasible in practice. An approach in the system NuPr1 is given in [18]. Related work on equality in logical frameworks is found in [16, 29, 14].

For many examples of the kinds of modular constructions we want our system to support we point the reader to [6, 5, 3, 20, 15, 10, 27].

General approach We begin with a model; then a variety of concrete expression languages may be interpreted in that model. This simplifies understanding of, and proofs about the expression languages and their typechecking. Although the expression language may have many typechecking rules, they can be checked individually for soundness w.r.t. the model.

Our starting model is quite close to a fragment of the PER model of NuPr1 [1]. However, unlike the closed terms of [1] we use open terms, for decidability of (definitional) equality and of type-checking. A further idea is needed, suggested by the work of Miquel [24] on semantics of subtyping: this is a general form of η -expansions (also see [13]). The main result says this η -expansion interacts well with the PER semantics: two terms are related by the PER interpreting a type iff their η -expanded forms are β -convertible.

Outline of the paper We describe stepwise our PER model, first for a core logical framework, then with singleton types and then with record types. In each case, we present a possible syntax for expressions and judgements, for which our model provides a realisability semantics, and for which we can prove termination of type-checking. We end by suggesting possible extensions and further work. An appendix collects the typechecking rules for the systems discussed in this paper.

2. Model for a Logical Framework

We present a model for a logical framework in which, up to “ η -expansion”, well typed objects are normalising and have a decidable notion of equality. This model will be used in later sections to interpret more concrete frameworks.

2.1. Values

Objects and types Let x, y , range over an infinite decidable set of *identifiers*, \mathcal{I} . We will be informal about variables and binding. The *objects* are usual *untyped* λ -terms.

$$M, N ::= x \mid M M \mid \lambda x.M$$

Let \mathcal{O} be the set of objects. The equality on objects is β -conversion, which we write as \simeq . We write $M[N/x]$ for capture avoiding substitution of N for x in M .

We define (weakly) *normalisable* as usual, and say that a term is *neutral* iff it is normalisable and of the form

$$\nu ::= x \mid \nu M.$$

Every normal object application is neutral.

We consider also the category of syntactic *types*

$$A, B ::= El\ M \mid \text{fun } A\ x.B \mid \star$$

Capture avoiding substitution, $A[N/x]$ is defined on types. When context makes clear what variable is being replaced, we may write $A[N]$ for $A[N/x]$. We write $A_1 \rightarrow A_2$ for $\text{fun } A_1\ x.A_2$ when x is not free in A_2 .

2.2. Categorical Judgement: Type

We will interpret types as partial equivalence relations (per) over objects. A *per* on set D is a binary relation \mathcal{A} on D which is symmetric and transitive. We may write $u_1 = u_2 : \mathcal{A}$ for $\mathcal{A}(u_1, u_2)$ and $u : \mathcal{A}$ for $u = u : \mathcal{A}$. Write $[\mathcal{A}]$ for the set of all $u \in D$ such that $u : \mathcal{A}$, and $\text{per}(D)$ for the set of all pers on D . If $\mathcal{A} \in \text{per}(D)$ then $Fam(\mathcal{A})$ is the set of all functions $\mathcal{F} : [\mathcal{A}] \rightarrow \text{per}(D)$ such that $\mathcal{F}(u_1) = \mathcal{F}(u_2)$ (extensionally equal) whenever $u_1 = u_2 : \mathcal{A}$.

If $\mathcal{A} \in \text{per}(\mathcal{O})$ and $\mathcal{F} \in Fam(\mathcal{A})$ we can form $\Pi(\mathcal{A}, \mathcal{F}) \in \text{per}(\mathcal{O})$, defined by $w_1 = w_2 : \Pi(\mathcal{A}, \mathcal{F})$ iff $u_1 = u_2 : \mathcal{A}$ implies $w_1\ u_1 = w_2\ u_2 : \mathcal{F}(u_1)$. Notice that we could have written $w_1\ u_1 = w_2\ u_2 : \mathcal{F}(u_2)$ as well, since $\mathcal{F}(u_1) = \mathcal{F}(u_2)$ in this case.

Being a type We define inductively a relation “=” of *intensional equality* on the set of types. (It will turn out to be a per.) Simultaneously for each A and B such that $A = B$ we define pers on objects, written \overline{A} and \overline{B} . For generality this definition is parameterised by the interpretations of \star and El .

We say that $\mathcal{A} \in \text{per}(\mathcal{O})$ is *saturated* iff

- $\nu : \mathcal{A}$ for every neutral ν ,
- if $u : \mathcal{A}$ then u is normalisable, and
- if $u_1 = u_2 : \mathcal{A}$ then $u_1 \simeq u_2$.

E.g. the relation “ M_1 and M_2 are neutral and equal” is saturated.

Definition 2.1. Let $\overline{\mathcal{A}} \in \text{per}(\mathcal{O})$ be saturated, and $\mathcal{E} \in Fam(\overline{\mathcal{A}})$ s.t. $\mathcal{E}(M)$ is saturated for $M : \overline{\mathcal{A}}$. The clauses defining intensional type equality are as follows.

- $\star = \star$.
- $El\ M = El\ N$ whenever $M = N : \overline{\mathcal{A}}$. $\overline{El\ M}$ is $\mathcal{E}(M)$.
- $\text{fun } A_1\ x_1.B_1 = \text{fun } A_2\ x_2.B_2$ whenever
 - $A_1 = A_2$,
 - $M_1 = M_2 : \overline{A_1} \implies B_1[M_1] = B_2[M_2]$.

$$\overline{\text{fun } A\ x.B} \text{ is } \Pi(\overline{A}, M \mapsto \overline{B[M]}).$$

This definition respects α, β equality on syntactic types. It is *intensional* in the sense that $\overline{A} = \overline{B}$ (extensionally) does not imply $A = B$.

We sometimes write $A \in \mathbf{Type}$ for $A = A$, and $M : \overline{A}$ for $M = M : \overline{A}$. Writing \overline{A} always implies $A \in \mathbf{Type}$.

Lemma 2.2. • If $A = B$ then \overline{A} and \overline{B} are extensionally equal.

- The relation $A = B$ is a per on types.

2.3. Normalisation and Decidability

We define an operation of η -expansion at type A (written $\eta\{A\}$) such that $M : \overline{A}$ implies $\eta\{A\}M$ is normalising.

Definition 2.3. $\eta\{-\}$ is defined by recursion on the syntactic class of types.²

$$\begin{aligned} \eta\{\star\}u &= u \\ \eta\{El M\}u &= u \\ \eta\{\text{fun } A x.B\}u &= \lambda z.\eta\{B[\widehat{z}]\}(u \widehat{z}) \quad \text{where } \widehat{z} = \eta\{A\}z \end{aligned}$$

where z is not free in $\eta\{A\}$ or B .

The *base types* are those of the form \star or $El M$. If, for instance, A is a base type, we have

$$\eta\{(A \rightarrow A) \rightarrow A\}u = \lambda z.u (\lambda x.z x).$$

We emphasise that we do not have η -conversion at the level of objects, but in the type system to be presented later there is no observable difference between η -convertible expressions.

Theorem 2.4. Let $A \in \mathbf{Type}$.

1. $\eta\{A\}\nu : \overline{A}$, where ν is neutral.
2. If $M : \overline{A}$ then $\eta\{A\}M$ is normalisable.
3. If $M : \overline{A}$ then $M = \eta\{A\}M : \overline{A}$.
4. If $M_1 = M_2 : \overline{A}$ then $\eta\{A\}M_1 \simeq \eta\{A\}M_2$.

Proof:

This proof is checked in Coq [8]. The four parts are proved simultaneously by induction on the proof that $A = B$ for some B . It is direct if A is a base type.

Function types Consider now a **Type** of the form $\text{fun } A y.B$. We have

$$\eta\{\text{fun } A y.B\}t \simeq \lambda x.\eta\{B[\eta\{A\}x]\}(t (\eta\{A\}x)) \quad (1)$$

²The number of funs is reduced on each recursive call.

1. Assuming that t is neutral and that $M_1 = M_2 : \bar{A}$, we must show

$$\eta\{\text{fun } A y.B\} t M_1 = \eta\{\text{fun } A y.B\} t M_2 : \overline{B[M_1]}.$$

Writing \widehat{M}_i for $\eta\{A\}M_i$ ($i = 1, 2$), and using (1), we need

$$\eta\{B[\widehat{M}_1]\} (t \widehat{M}_1) = \eta\{B[\widehat{M}_2]\} (t \widehat{M}_2) : \overline{B[M_1]}.$$

By induction hypothesis, $\widehat{M}_1 \simeq \widehat{M}_2$, so it suffices to show

$$\eta\{B[\widehat{M}_1]\} (t \widehat{M}_1) : \overline{B[M_1]}.$$

By IH, $M_1 = \widehat{M}_1 : \bar{A}$ (so $B[\widehat{M}_1] = B[M_1]$) and \widehat{M}_1 is normalisable. Thus $t \widehat{M}_1$ is neutral, and by IH $\eta\{B[\widehat{M}_1]\} (t \widehat{M}_1) : \overline{B[M_1]}$, so we are done.

2. Assume $t : \overline{\text{fun } A y.B}$; then $\eta\{A\}x : \bar{A}$ by induction hypothesis, hence $t (\eta\{A\}x) : \overline{B[\eta\{A\}x]}$. By induction hypothesis

$$\eta\{B[\eta\{A\}x]\} (t (\eta\{A\}x))$$

is normalisable, hence $\eta\{\text{fun } A y.B\} t$ is normalisable.

3. Assume $t : \overline{\text{fun } A y.B}$ and $M_1 = M_2 : \bar{A}$. We need

$$t M_1 = \eta\{\text{fun } A y.B\} t M_2 : \overline{B[M_1]}.$$

Writing \widehat{M}_2 for $\eta\{A\}M_2$, and using (1), we need

$$t M_1 = \eta\{B[\widehat{M}_2]\} (t \widehat{M}_2) : \overline{B[M_1]}.$$

By IH, $M_1 = \widehat{M}_2 : \bar{A}$, so $t M_1 = t \widehat{M}_2 : \overline{B[M_1]}$ and $\overline{B[M_1]} = \overline{B[\widehat{M}_2]}$. From this it suffices to prove

$$t \widehat{M}_2 = \eta\{B[\widehat{M}_2]\} (t \widehat{M}_2) : \overline{B[\widehat{M}_2]},$$

which follows by IH.

4. If $t_1 = t_2 : \overline{\text{fun } A y.B}$ we have by induction $\eta\{A\}x : \bar{A}$ and hence $t_1 (\eta\{A\}x) = t_2 (\eta\{A\}x) : \overline{B[\eta\{A\}x]}$. Hence, by induction

$$\eta\{B[\eta\{A\}x]\} (t_1 (\eta\{A\}x)) \simeq \eta\{B[\eta\{A\}x]\} (t_2 (\eta\{A\}x))$$

which implies $\eta\{\text{fun } A y.B\} t_1 \simeq \eta\{\text{fun } A y.B\} t_2$ as expected. \square

Combining parts 2, 3 and 4 of this theorem, we have:

Corollary 2.5. Let $M_1 : \bar{A}$ and $M_2 : \bar{A}$. Then $M_1 = M_2 : \bar{A}$ is equivalent to $\eta\{A\}M_1 \simeq \eta\{A\}M_2$, which is decidable.

Remark on eta. Using $\beta\eta$ -conversion for equality on objects (instead of β -conversion) would simplify the statements of theorem 2.4 and corollary 2.5, since our operator $\eta\{-\}$ is no longer needed. This simplification explains why [9] can model a type theory with one universe and η -conversion. Such a simplification however does not seem possible in presence of singleton types, or even unit types, a difficulty noticed in [16].

2.4. Hypothetical Judgements

Contexts We consider contexts

$$C ::= \nabla \mid C, x:A \quad (\nabla \text{ is the empty context.})$$

We say $x:A$ in C iff $C = C', x':A'$ and $x:A = x':A'$ or $x:A$ in C' . We also say $x \in C$ if $x:A$ in C for some A . In writing $C, x:A$ we assume $x \notin C$.

Environments An environment, ρ , is a function of type $\mathcal{I} \rightarrow \mathcal{O}$. ρ_0 is the identity environment, $\rho_0(x) = x$. $M\rho$ (resp. $A\rho$) is the simultaneous substitution of $\rho(x)$ for all free occurrences of x in M (resp. A). The composition of environments is defined by $(\rho_1 \circ \rho_2)x = (\rho_1 x)\rho_2$. Notice $M(\rho_1 \circ \rho_2) \simeq (M\rho_1)\rho_2$. We write $(\rho, x=M)$ for the *update* of ρ , defined by

$$(\rho, x=M)(x) = M, \quad (\rho, x=M)(y) = \rho(y) \text{ if } y \neq x.$$

Lemma 2.6. For A a type and ρ an environment, $(\eta\{A\})\rho \simeq \eta\{A\rho\}$. Hence for any object M , $(\eta\{A\}M)\rho \simeq \eta\{A\rho\}(M\rho)$.

Definition 2.7. We inductively define a judgement of form $\rho_1 = \rho_2 : C$ by the rules:

$$\frac{}{\rho_1 = \rho_2 : \nabla} \quad \frac{\rho_1 = \rho_2 : C \quad A\rho_1 \in \mathbf{Type} \quad \rho_1 x = \rho_2 x : \overline{A\rho_1}}{\rho_1 = \rho_2 : C, x:A}$$

We may write $\rho : C$ for $\rho = \rho : C$.

Lemma 2.8. If $\rho_1 = \rho_2 : C$ and $y \notin C$ then for any M , $\rho_1 = (\rho_2, y=M) : C$.

Proof:

By induction on the proof of $\rho_1 = \rho_2 : C$. Consider a proof ending with

$$\frac{\rho_1 = \rho_2 : C \quad A\rho_1 = A\rho_1 \quad \rho_1 x = \rho_2 x : \overline{A\rho_1}}{\rho_1 = \rho_2 : C, x:A}$$

It suffices to show $\rho_1 = (\rho_2, y=M) : C$ and $\rho_1 x = (\rho_2, y=M)x : \overline{A\rho_1}$. The former holds by IH. The latter holds because $y \notin (C, x:A)$, so $y \neq x$. \square

Hypothetical judgements We define simultaneously three judgement forms, C valid, $A_1 = A_2 [C]$ and $M_1 = M_2 : A [C]$. (We may write $A \text{ type } [C]$ for $A = A [C]$, and $M : A [C]$ for $M = M : A [C]$.)

$$\frac{}{\nabla \text{ valid}} \quad \frac{x \notin C \quad A \text{ type } [C]}{C, x:A \text{ valid}} \quad (2)$$

$$\frac{C \text{ valid} \quad \forall \rho_1, \rho_2 . \rho_1 = \rho_2 : C \implies A_1 \rho_1 = A_2 \rho_2}{A_1 = A_2 [C]} \quad (3)$$

$$\frac{A \text{ type } [C] \quad \forall \rho_1, \rho_2 . \rho_1 = \rho_2 : C \implies M_1 \rho_1 = M_2 \rho_2 : \overline{A \rho_1}}{M_1 = M_2 : A [C]}$$

For the well-formedness of rule (3), note that the first premise guarantees that $\overline{A \rho_1}$ is defined.

Lemma 2.9. Let C be valid.

- The following relations are pers

$$\begin{aligned} \rho_1, \rho_2 &\mapsto \rho_1 = \rho_2 : C, \\ A_1, A_2 &\mapsto A_1 = A_2 [C], \\ M_1, M_2 &\mapsto M_1 = M_2 : A [C] \end{aligned}$$

- Let $\rho_1 = \rho_2 : C$, $y \notin C$. For any M and N , $(\rho_1, y=M) = (\rho_2, y=N) : C$.

Theorem 2.10. The following implications hold. We write them as rules (consider also rules (2)) to invite comparison with the rules of Martin–Löf’s logical framework.

type formation and type equality

$$\frac{C \text{ valid}}{\star \text{ type } [C]} \quad \frac{M = N : \star [C]}{El M = El N [C]} \quad \frac{A_1 = A_2 [C] \quad B_1 = B_2 [C, x:A_1]}{\text{fun } A_1 x.B_1 = \text{fun } A_2 x.B_2 [C]} \quad (4)$$

objects

$$\frac{C, x:A \text{ valid}}{x : A [C, x:A]} \quad \frac{M_1 = M_2 : B [C, x:A]}{\lambda x.M_1 = \lambda x.M_2 : \text{fun } A x.B [C]}$$

$$\frac{M_1 = M_2 : \text{fun } A x.B [C] \quad N_1 = N_2 : A [C]}{M_1 N_1 = M_2 N_2 : B[N_1] [C]}$$

type conversion

$$\frac{M = N : A [C] \quad A = B [C]}{M = N : B [C]}$$

weakening

$$\frac{B_1 = B_2 [C] \quad C, x:A \text{ valid}}{B_1 = B_2 [C, x:A]} \quad \frac{M = N : B [C] \quad C, x:A \text{ valid}}{M = N : B [C, x:A]}$$

substitution

$$\frac{B_1 = B_2 [C, x:A] \quad N_1 = N_2 : A [C]}{B_1[N_1] = B_2[N_2] [C]} \quad \frac{M_1 = M_2 : B [C, x:A] \quad N_1 = N_2 : A [C]}{M_1[N_1] = M_2[N_2] : B_1[N_1] [C]}$$

Proof:

All parts are proved directly from the meanings of the definitions using the lemmas 2.2, 2.8 and 2.9. However there are a lot of details to be handled. We have checked this theorem in Coq [8]. \square

Decidability We give conditions for $M_1 = M_2 : A [C]$ to be decidable.

Definition 2.11. Eta-expansion of environment ρ at context C is defined by recursion on the structure of C .

$$\begin{aligned} \eta\{\nabla\}\rho &= \rho \\ \eta\{C, x:A\}\rho &= (\rho', x=\eta\{A\rho'\}(\rho x)) \quad \text{where } \rho' = \eta\{C\}\rho \end{aligned}$$

Write ρ_C for $\eta\{C\}\rho_0$, where ρ_0 is the identity environment.

Lemma 2.12. 1. $\rho_C \circ \rho = \eta\{C\}\rho$.

2. If C valid then $\rho_C : C$.

3. If C valid and $\rho : C$ then $\rho = \eta\{C\}\rho : C$.

4. If $M_1 : A [C]$ and $M_2 : A [C]$ then $M_1 = M_2 : A [C]$ is equivalent to $M_1\rho_C = M_2\rho_C : \overline{A\rho_C}$.

Proof:

1. By induction on C . For the induction step it suffices to prove $(\rho_{C,y:A} \circ \rho)y = (\eta\{C, y:A\}\rho)y$, which follows by computation using IH and lemma 2.6.

2. By induction on the proof of C valid. Consider the case where this proof ends with

$$\frac{x \notin C \quad A \text{ type } [C]}{C, x:A \text{ valid}}$$

In order to conclude $\rho_{C,x:A} : C, x:A$ we need $\rho_{C,x:A} : C$, $A\rho_{C,x:A} \in \mathbf{Type}$ and $\rho_{C,x:A}(x) : \overline{A\rho_{C,x:A}}$. Inverting the premise $A \text{ type } [C]$ we see that C valid, hence $\rho_C : C$ by IH. Thus, by lemma 2.8 we satisfy the first requirement:

$$\rho_C = \rho_{C,x:A} : C. \tag{5}$$

From $A \text{ type } [C]$ and equation (5) we have $A\rho_C = A\rho_{C,x:A}$, meeting the second requirement. Using lemma 2.2, we also have $\overline{A\rho_C} = \overline{A\rho_{C,x:A}}$ (extensionally). Thus, for the third requirement it suffices to show $\rho_{C,x:A}(x) : \overline{A\rho_C}$, which follows from

$$\rho_{C,x:A}(x) = \eta\{A(\eta\{C\}\rho_0)\}(\rho_0 x) = \eta\{A\rho_C\}x$$

because theorem 2.4 shows $\eta\{A\rho_C\}x : \overline{A\rho_C}$.

3. By induction on the proof of $\rho : C$. Consider the case where this proof ends with

$$\frac{\rho = \rho : C \quad A\rho = A\rho \quad \rho x = \rho x : \overline{A\rho}}{\rho = \rho : C, x:A}$$

To conclude $\rho = \eta\{C, x:A\}\rho : C, x:A$ we need $\rho = \eta\{C, x:A\}\rho : C$ and $\rho x = \eta\{A(\eta\{C\}\rho)\}(\rho x) : \overline{A\rho}$. The former follows by IH and lemma 2.8. For the latter, note that $A\rho = A(\eta\{C\}\rho)$ follows from IH and $C, x:A$ valid. By lemma 2.2, we have $\overline{A\rho} = \overline{A(\eta\{C\}\rho)}$ (extensionally), and it suffices to show $\rho x = \eta\{A(\eta\{C\}\rho)\}(\rho x) : \overline{A(\eta\{C\}\rho)}$. This follows from theorem 2.4 and the premise $\rho x = \rho x : \overline{A\rho}$.

4. Assume $M_1 : A [C]$ and $M_2 : A [C]$. It is immediate from part 2 that $M_1 = M_2 : A [C]$ implies $M_1 \rho_C = M_2 \rho_C : \overline{A \rho_C}$.

For the other direction, assume $M_1 \rho_C = M_2 \rho_C : \overline{A \rho_C}$. Then $(\eta\{A\}M_1)\rho_C \simeq (\eta\{A\}M_2)\rho_C$ by theorem 2.4 and lemma 2.6, so also

$$(\eta\{A\}M_1)(\rho_C \circ \rho) \simeq (\eta\{A\}M_2)(\rho_C \circ \rho). \quad (6)$$

Using $M_1 : A [C]$ and $M_2 : A [C]$ (hence also A type $[C]$), it suffices to show that for any $\rho : C$, $M_1 \rho = M_2 \rho : \overline{A \rho}$. Since $\rho = \rho_C \circ \rho : C$ by parts 1 and 3, we have $A \rho = A(\rho_C \circ \rho)$ and $M_i \rho = M_i(\rho_C \circ \rho) : \overline{A \rho}$. Further, $M_1(\rho_C \circ \rho) = M_2(\rho_C \circ \rho) : \overline{A(\rho_C \circ \rho)}$ by corollary 2.5 using equation 6, and we are done. \square

From this lemma and theorem 2.4, we have:

Corollary 2.13. Let $M_1 : A [C]$ and $M_2 : A [C]$. Then $M_1 = M_2 : A [C]$ is equivalent to $(\eta\{A\}M_1)\rho_C \simeq (\eta\{A\}M_2)\rho_C$, which is decidable.

2.5. “Syntactic” Type Equality

We now define a syntactic relation of shape $C \vdash A_1 = A_2$ which is decidable and sound for the semantic relation.

Definition 2.14. The rules of syntactic type equality are as follows.

$$\frac{}{C \vdash \star = \star} \quad \frac{M_1 = M_2 : \star [C]}{C \vdash \text{El } M_1 = \text{El } M_2} \quad \frac{C \vdash A_1 = A_2 \quad C, x:A_1 \vdash B_1 = B_2}{C \vdash \text{fun } A_1 x.B_1 = \text{fun } A_2 x.B_2}$$

Lemma 2.15. 1. If A_1 type $[C]$ and A_2 type $[C]$ then $C \vdash A_1 = A_2$ is decidable.

2. If C valid and $C \vdash A_1 = A_2$ then $A_1 = A_2 [C]$.

Proof:

1. by corollary 2.13. 2. is direct using the derived rules (4). \square

3. A Logical Framework in Syntax

We give concrete syntax for a core LF, including rules for typechecking that interpret expressions in the model. This interpretation is semantically sound.

3.1. Expressions and Judgements

The syntax of expressions and expression contexts is defined by

$$\begin{aligned} e & ::= z \mid e e \mid [z:e]e \mid * \mid \text{El } e \mid \{z:e\}e \mid e \rightarrow e \\ \Gamma & ::= \blacktriangledown \mid \Gamma, x:e \end{aligned}$$

(As usual, we omit \blacktriangledown , the notation for the empty context, in concrete syntax.)

For simplicity and definiteness in this paper, expressions and contexts are taken concretely. Not even α -renaming is assumed. There is no substitution defined on expressions; that is delegated to the interpretation. Thus, given an implementation of pure lambda terms, our typechecking rules are directly implementable as presented below. However, more conventional languages can also be interpreted by our model.

Two judgement forms are defined simultaneously,

- $C \vdash e \Rightarrow A$, meaning that expression e is interpreted in C as type A .
- $C \vdash e \Rightarrow M : A$, meaning that expression e is interpreted in C as object M which has type A .

while a third can be defined afterwards.

- $\Gamma \Rightarrow C$, meaning that Γ is interpreted as the valid context C .

This organization is typical for implementations of type theory such as Lego [26] and Coq [8] that maintain a “current checked environment” representing the mathematics developed so far, since it avoids repeatedly rechecking that current context [23].

3.2. Typechecking

The rules are: type formation

$$\frac{}{C \vdash * \Rightarrow *} \quad \frac{C \vdash e \Rightarrow M : *}{C \vdash \text{El } e \Rightarrow \text{El } M} \quad \frac{C \vdash e_1 \Rightarrow A \quad C \vdash e_2 \Rightarrow B}{C \vdash e_1 \rightarrow e_2 \Rightarrow A \rightarrow B}$$

$$\frac{C \vdash e_1 \Rightarrow A \quad x \notin C \quad C, x:A \vdash e_2 \Rightarrow B}{C \vdash \{x:e_1\}e_2 \Rightarrow \text{fun } A x.B}$$

objects

$$\frac{x:A \text{ in } C}{C \vdash x \Rightarrow x : A} \quad \frac{C \vdash e_1 \Rightarrow A \quad x \notin C \quad C, x:A \vdash e_2 \Rightarrow M : B}{C \vdash [x:e_1]e_2 \Rightarrow \lambda x.M : \text{fun } A x.B} \quad (7)$$

$$\frac{C \vdash e_1 \Rightarrow M_1 : \text{fun } A_1 x.B \quad C \vdash e_2 \Rightarrow M_2 : A_2 \quad C \vdash A_1 = A_2}{C \vdash e_1 e_2 \Rightarrow M_1 M_2 : B[M_2]} \quad (8)$$

validity

$$\frac{}{\blacktriangledown \Rightarrow \nabla} \quad \frac{\Gamma \Rightarrow C \quad C \vdash e \Rightarrow A \quad x \notin C}{\Gamma, x:e \Rightarrow C, x:A}$$

In these rules, as in the rest of the paper, semantic values are taken up to α, β -equality as usual. However, as mentioned above, expressions are taken concretely. Thus some “good” expressions do not typecheck for reasons of variable clash. For example $[x:*][x:\text{El } x]x$ is rejected, while $[x:*][y:\text{El } x]y$ is accepted. The reason we include $e_1 \rightarrow e_2$ as an expression, rather than as an abbreviation, is so that nested arrows do not fail due to variable clash.

η -conversion in the expression language. η -convertible expressions are indistinguishable by typechecking. For example, consider $x:\{a:*\}\text{El } a$ and its η -expansion, $[a:*](x \ a)$. The only way these expressions might be compared during the typechecking of an expression is by the third premise of the rule for applications, (8), which calls definition 2.14. Since x and $[a:*](x \ a)$ are objects, they can only occur in types inside the El constructor, hence will both be η -expanded by the premise of the second rule of definition 2.14 (via corollary 2.13). These η -expansions are β -convertible, so considered equal.

3.3. Correctness and Termination of Typechecking

Theorem 3.1. • If $C \vdash a \Rightarrow A$ and C valid then A type $[C]$.

- If $C \vdash e \Rightarrow M : A$ and C valid then $M : A [C]$.

Proof:

By simultaneous induction on derivations. Straightforward using the definition of `valid` (rule (2)), theorem 2.10 and lemma 2.15. We do two cases.

Pi type The derivation ends with

$$\frac{C \vdash a \Rightarrow A \quad x \notin C \quad C, x:A \vdash b \Rightarrow B}{C \vdash \{x:a\}b \Rightarrow \text{fun } A \ x.B}$$

Assume C valid. By first IH, A type $[C]$, so $C, x:A$ valid. Hence by second IH, B type $[C, x:A]$. By theorem 2.10, $\text{fun } A \ x.B$ type $[C]$, as required.

Application The derivation ends with

$$\frac{C \vdash e_1 \Rightarrow M_1 : \text{fun } A_1 \ x.B \quad C \vdash e_2 \Rightarrow M_2 : A_2 \quad C \vdash A_1 = A_2}{C \vdash e_1 e_2 \Rightarrow M_1 M_2 : B[M_2]}$$

Assume C valid. By IH, $M_1 : \text{fun } A_1 \ x.B [C]$ and $M_2 : A_2 [C]$. By lemma 2.15, $A_1 = A_2 [C]$. $M_1 M_2 : B[M_2] [C]$ follows from theorem 2.10. \square

Corollary 3.2. If $\Gamma \Rightarrow C$ then C valid.

Corollary 3.3. 1. If C valid then $\exists A . C \vdash e \Rightarrow A$ and $\exists M, A . C \vdash e \Rightarrow M : A$ are decidable.

2. $\exists C . \Gamma \Rightarrow C$ is decidable.

3. $\exists C, A . \Gamma \Rightarrow C \wedge C \vdash e \Rightarrow A$ and $\exists C, M, A . \Gamma \Rightarrow C \wedge C \vdash e \Rightarrow M : A$ are decidable.

Proof:

The typechecking rules are deterministic and syntax directed. The only possible source of non-termination is the side condition $C \vdash A_1 = A_2$ of the rule for application.

For 1., the theorem shows that C is valid in any occurrence of the side condition $C \vdash A_1 = A_2$ in a derivation starting with a valid context. 2. follows from 1. by induction on the derivation of $\Gamma \Rightarrow C$. \square

4. Singleton Types and Definitions

We extend the semantics with singleton types in order to interpret local and global definitions (this section) and *manifest fields* in signatures (section 5). In this section we do not need subtyping for singletons. In this paper we do not consider singletons in the expression language itself. For other work on singletons see [2, 11, 28, 17, 12, 29].

4.1. Singleton Types in the Model

The category of syntactic types is extended with a notation A/M for the singleton of object M at type A .

$$A, B ::= \text{El } M \mid \text{fun } A \ x.B \mid \star \mid A/M.$$

If $\mathcal{A} \in \text{per}(D)$ and $u : \mathcal{A}$ we form the singleton $\mathcal{A}/u \in \text{per}(D)$ which is defined by $u_1 = u_2 : \mathcal{A}/u$ iff $u_1 = u : \mathcal{A}$ and $u_2 = u : \mathcal{A}$. Definition 2.1 is extended with a new clause

- $A_1/M_1 = A_2/M_2$ iff $A_1 = A_2$ and $M_1 = M_2 : \overline{A_1}$.
 $M_1 = M_2 : A/N$ iff $M_1 = M_2 = N : \overline{A}$.

Lemma 2.2 holds of this extended definition.

Definition 2.3 is extended with the clause

$$\eta\{A/M\}u = \eta\{A\}M$$

where u is not free in $\eta\{A\}M$. Theorem 2.4 holds for $A/M \in \mathbf{Type}$.

The following implications hold, extending theorem 2.10.

$$\frac{M_1 = M_2 : A [C]}{M_1 = M_2 : A/M_1 [C]} \quad \frac{M : A/N [C]}{M = N : A [C]} \quad \frac{M_1 = M_2 : B [C, x:A/N]}{M_1[N] = M_2[N] : B[N] [C]} \quad (9)$$

Lemma 2.15 holds in the extended system.

4.2. Typechecking Definitions

Extend the syntax of expressions and expression contexts with local and global definitions respectively.

$$\begin{aligned} e & ::= z \mid ee \mid [z:e]e \mid \star \mid \text{El } e \mid \{z:e\}e \mid e \rightarrow e \mid [z=e]e \\ \Gamma & ::= \blacktriangledown \mid \Gamma, x:e \mid \Gamma, x=e \end{aligned}$$

The typechecking rule for variables (first rule (7)) works unchanged for global definitions because a variable x of type A/N η -expands to $\eta\{A\}N$. Thus global definitions are expanded only for equality testing. Heuristics to avoid some definition expansion are possible in pragmatic implementations.

We have new typechecking rules for local definitions in types and in objects

$$\frac{C \vdash e_1 \Rightarrow N : A \quad x \notin C \quad C, x:A/N \vdash e_2 \Rightarrow B}{C \vdash [x=e_1]e_2 \Rightarrow B[N]}$$

$$\frac{C \vdash e_1 \Rightarrow N : A \quad x \notin C \quad C, x:A/N \vdash e_2 \Rightarrow M : B}{C \vdash [x=e_1]e_2 \Rightarrow M[N] : B[N]}$$

and for validity with global definitions.

$$\frac{\Gamma \Rightarrow C \quad C \vdash e \Rightarrow N : A \quad x \notin C}{\Gamma, x=e \Rightarrow C, x:A/N}$$

Semantical soundness and termination of typechecking follow as expected, using rules (9).

Remark on incompleteness. This syntactic typechecking system is not as strong as one might hope. For example, the following judgement is not derivable:³

$$A : \star, a : El A \vdash ([x : El A] x) ([y = a] y) \Rightarrow (\lambda x. x) a : El A.$$

A “subsumption rule”

$$\frac{C \vdash e \Rightarrow M : A/N}{C \vdash e \Rightarrow M : A}$$

is adequate to fix the class of problems suggested by this example. This rule is semantically sound, as shown by the second rule in (9). However we have not studied the decidability of typechecking with this rule, nor the more general question of expressiveness of syntactic typechecking.

5. Records, Signatures and Subtyping

The model extends to both left associating and right associating signatures [27]. We will present the more standard right associating signatures. Convenient use of signatures requires subtyping of some kind. Here we describe *structural subtyping* for signatures. There are many interesting subtyping rules for singletons [2], but we include only what is needed for manifest signatures.

5.1. Signatures and Records in the Model

We use identifiers as field labels, but often write l , k , instead of x for labels to make an informal distinction. The category of syntactic types is extended with a unit type, $\mathbf{1}$ (which will serve as a top type, and as the empty signature), and with a signature extension constructor.

$$A, B, S ::= El M \mid \text{fun } A x. B \mid \star \mid A/M \mid \mathbf{1} \mid \langle l:A, x.S \rangle.^4$$

We say $l \in A$ iff $A = \langle k:A', x.B' \rangle$ and $l = k$ or $l \in B'$. In writing $\langle l:A, x.B \rangle$ we assume $l \notin B$.

To the category of objects we introduce $()$ (which serves as the empty record and the canonical element of $\mathbf{1}$) and objects for record extension and field projection.

$$M, N ::= x \mid M M \mid \lambda x. M \mid () \mid (l=M, M) \mid M.l$$

There are new reduction rules for *dot reduction* (record elimination)

$$(l=M_1, M_2).l \succ M_1, \quad (k=M_1, M_2).l \succ M_2.l \quad (l \neq k).$$

Clearly, every object has a unique dot-normal form, so \succ is Church-Rosser. Also \succ commutes with β , so by the Hindley-Rosen lemma [4], $\succ \cup \beta$ is Church-Rosser. Equality on objects is defined as the closure of $\succ \cup \beta$. This equality does not include surjective pairing or record field permutation, but because of generalised η -expansion, $\eta\{A\}$, the interpreted expression language does.

Definition 2.1 is extended with clauses for $\mathbf{1}$ and $\langle l:A, x.B \rangle$.

- $\mathbf{1} = \mathbf{1}$. $\bar{\mathbf{1}}$ is $\mathcal{O} \times \mathcal{O}$.

³Thanks to Jeff Polakow for this example.

⁴The notation $\langle l:A, x.S \rangle$ is analogous to $\{l \triangleright x:A, S\}$ originally proposed in [15].

- $\langle l_1:A_1, x_1.B_1 \rangle = \langle l_2:A_2, x_2.B_2 \rangle$ iff
 - $l_1 = l_2$
 - $A_1 = A_2$,
 - $M_1 = M_2 : \overline{A_1} \implies B_1[M_1] = B_2[M_2]$.

$$M_1 = M_2 : \overline{\langle l:A, x.B \rangle} \text{ iff } M_1.l = M_2.l : \overline{A} \wedge M_1 = M_2 : \overline{B[M_1.l]}.$$

Lemma 2.2 holds of this extended definition.

Definition 2.3 is extended with new clauses.

$$\begin{aligned} \eta\{\mathbf{1}\}u &= () \\ \eta\{\langle l:A, x.B \rangle\}u &= (l=\hat{u}, \eta\{B[\hat{u}]\}u) \quad \text{where } \hat{u} = \eta\{A\}(u.l) \end{aligned}$$

Theorem 2.4 holds for the extended notion of **Type**.

The following implications are derivable, extending theorem 2.10.

$$\begin{array}{c} \frac{C \text{ valid}}{\mathbf{1} \text{ type } [C]} \quad \frac{l_1 = l_2 \quad A_1 = A_2 [C] \quad B_1 = B_2 [C, x:A_1]}{\langle l_1:A_1, x.B_1 \rangle = \langle l_2:A_2, x.B_2 \rangle [C]} \\ \\ \frac{C \text{ valid}}{M = N : \mathbf{1} [C]} \quad \frac{M_A : A [C] \quad M_B : B[M_A] [C] \quad B \text{ type } [C, x:A]}{(l=M_A, M_B) : \langle l:A, x.B \rangle [C]} \\ \\ \frac{M : \langle l:A, x.B \rangle [C]}{M.l : A [C]} \quad \frac{M : \langle l:A, x.B \rangle [C] \quad l \neq k \quad k \in B}{M.k : B[M.l] [C]} \end{array}$$

5.2. Subtyping

For $A, B \in \mathbf{Type}$ we define the semantic notion *categorical subtype*, written $A \sqsubseteq B$, to be $\overline{A} \subseteq \overline{B}$ (extensionally subrelation); this is adequate for the claims in this paper.

The notion of *hypothetical subtype* is defined by

$$A \sqsubseteq B [C] = A \text{ type } [C] \wedge B \text{ type } [C] \wedge \forall \rho . \rho : C \implies A\rho \sqsubseteq B\rho$$

Lemma 5.1. Analogous to the type conversion rule (theorem 2.10) we have.

$$\frac{M = N : A [C] \quad A \sqsubseteq B [C]}{M = N : B [C]}$$

Syntactic subtyping We define a relation of *syntactic subtyping*, which replaces syntactic type equality, definition 2.14.

Definition 5.2. The rules of syntactic subtyping are as follows.

$$\frac{}{C \vdash \star/M \sqsubseteq \star} \quad \frac{M_1 = M_2 : \star [C]}{C \vdash (El M_1)/M \sqsubseteq El M_2} \quad \frac{}{C \vdash A/M \sqsubseteq \mathbf{1}}$$

$$\frac{C, x:A_2 \vdash A_2/x \sqsubseteq A_1 \quad C, x:A_2 \vdash B_1/(Mx) \sqsubseteq B_2}{C \vdash (\text{fun } A_1 \ x.B_1)/M \sqsubseteq \text{fun } A_2 \ x.B_2}$$

$$\frac{C \vdash A/M \sqsubseteq B \quad M = N : B [C]}{C \vdash A/M \sqsubseteq B/N} \quad (10)$$

$$\frac{l:A'/N \text{ in } S/M \quad C \vdash A'/N \sqsubseteq A \quad C \vdash S/M \sqsubseteq B[N]}{C \vdash S/M \sqsubseteq \langle l:A, x.B \rangle} \quad (11)$$

where the relation $l:A \text{ in } S$ is defined by the following rules.

$$\frac{}{l:A/N \text{ in } \langle l:A/N, x.B \rangle/M} \quad \frac{A \text{ not singleton}}{l:A/M.l \text{ in } \langle l:A, x.B \rangle/M} \quad (12)$$

$$\frac{l:A/N \text{ in } B[M.l']/M \quad l \neq l'}{l:A/N \text{ in } \langle l':A', x.B \rangle/M}$$

This formulation computes the principal type of a record directly, rather than using an auxiliary operation of *signature strengthening*, as in [20, 10]. This is most clearly seen in the second rule (12).

Syntactic subtyping is decidable and sound.

Lemma 5.3. If $A_1 \text{ type}[C]$ and $A_2 \text{ type}[C]$ then $C \vdash A_1 \sqsubseteq A_2$ is decidable and implies $A_1 \sqsubseteq A_2[C]$.

5.3. Typechecking with Signatures, Records and Subtyping

Extend the syntax of expressions.

$$e ::= z \mid ee \mid [z:e]e \mid * \mid \mathbf{E}1 e \mid \{z:e\}e \mid e \rightarrow e \mid [z=e]e \mid \langle \rangle \mid \langle l:e, e \rangle \mid \langle l=e, e \rangle \mid () \mid (l=e, e) \mid e.l$$

$\langle \rangle$ is the empty signature; $\langle l:e, e \rangle$ (resp. $\langle l=e, e \rangle$) is *opaque* (resp. *manifest*) signature extension. $()$ is the empty record, $(l=e, e)$ is record extension and $e.l$ is record projection at label l .

In order to use the syntactic subtyping relation in typechecking, we replace the typechecking rule for applications, equation (8), by

$$\frac{C \vdash e_1 \Rightarrow M_1 : \text{fun } A_1 \ x.B \quad C \vdash e_2 \Rightarrow M_2 : A_2 \quad C \vdash A_2/M_2 \sqsubseteq A_1}{C \vdash e_1 e_2 \Rightarrow M_1 M_2 : B[M_2]} \quad (13)$$

Note how the third premise of this rule calls syntactic subtyping with a type-correct singleton LHS. This goes some way towards clarifying the rules of definition 5.2.

We have new typechecking rules for signature formation

$$\frac{}{C \vdash \langle \rangle \Rightarrow \mathbf{1}} \quad \frac{C \vdash e_1 \Rightarrow A \quad l \notin C \quad C, l:A \vdash e_2 \Rightarrow B}{C \vdash \langle l:e_1, e_2 \rangle \Rightarrow \langle l:A, l.B \rangle}$$

$$\frac{C \vdash e_1 \Rightarrow M : A \quad l \notin C \quad C, l:A/M \vdash e_2 \Rightarrow B}{C \vdash \langle l=e_1, e_2 \rangle \Rightarrow \langle l:A/M, l.B \rangle}$$

introduction

$$\frac{}{C \vdash () \Rightarrow () : \mathbf{1}} \quad \frac{C \vdash e_1 \Rightarrow M_1 : A \quad C \vdash e_2 \Rightarrow M_2 : B \quad x \notin C}{C \vdash (l=e_1, e_2) \Rightarrow (l=M_1, M_2) : \langle l:A/M_1, x.B \rangle}$$

and elimination.

$$\frac{C \vdash e \Rightarrow M : S \quad l : B/N \text{ in } S/M}{C \vdash e.l \Rightarrow N : B}$$

Semantical soundness and termination of typechecking follow as expected.

Signature strengthening Suppose $x : \langle a : * \rangle$ is declared. Can x be used where an expression of type $\langle a = x . a \rangle$ is expected? We have $C \vdash \langle a : * \rangle \Rightarrow \langle a : * \rangle$ and $C \vdash \langle a = x . a \rangle \Rightarrow \langle a : * /x.a \rangle$. By rule (13), the question reduces to deciding $C \vdash \langle a : * \rangle /x \sqsubseteq \langle a : * /x.a \rangle$. This is derivable by rules (11) and (12).

6. Type Families

The system of section 3 does not accept the expression $[a : *] \text{El}(a) \rightarrow *$ (the “type” of unary predicates over arbitrary type a) although for any particular $a : *$, $\text{El}(a) \rightarrow *$ is well typed. In order to allow such expressions to be typed, we extend the semantics with *type families*, to be typed by *kinds*.

Type Families and Kinds We consider the syntactic categories of *type families*⁵

$$F ::= A \mid \lambda x.F \mid F M$$

and of *kinds*

$$K ::= \square \mid \text{fun } A x.K$$

Capture avoiding substitution is defined on type families and kinds. We have β -reduction on type families which is clearly normalising and Church–Rosser. Type family equality is β -conversion.

Being a kind We define inductively a relation “=” of *intensional equality* on the set of kinds. (It will turn out to be a per.) Simultaneously for each K and L such that $K = L$ we define pers on type families, written \overline{K} and \overline{L} . The clauses are as follows.

- $\square = \square$. $F = G : \square$ iff $F = G$ (as types).
- $\text{fun } A_1 x_1.K_1 = \text{fun } A_2 x_2.K_2$ whenever
 - $A_1 = A_2$,
 - $M_1 = M_2 : \overline{A_1} \implies K_1[M_1] = K_2[M_2]$.

$$\overline{\text{fun } A_1 x_1.K_1} \text{ is } \Pi(\overline{A_1}, M \mapsto \overline{K_1[M]}).$$

This definition respects α, β equality on syntactic kinds. We sometimes write $K \in \mathbf{Kind}$ for $K = K$, $F = G : \overline{K}$ for $K = K \wedge F = G : \overline{K}$, and $F : \overline{K}$ for $F = F : \overline{K}$.

Lemma 6.1. • If $K = L$ then \overline{K} and \overline{L} are extensionally equal.

- The relation $K = L$ is a per on kinds.

⁵We could eliminate the application constructor of type families by keeping them in weak-head-normal form.

Hypothetical judgements We define simultaneously two new judgement forms, $K_1 = K_2 [C]$ and $F_1 = F_2 : K [C]$. (We may write $K \text{ kind } [C]$ for $K = K [C]$, and $F : K [C]$ for $F = F : K [C]$.)

$$\frac{C \text{ valid} \quad \forall \rho_1, \rho_2 . \rho_1 = \rho_2 : C \implies K_1 \rho_1 = K_2 \rho_2}{K_1 = K_2 [C]}$$

$$\frac{K \text{ kind } [C] \quad \forall \rho_1, \rho_2 . \rho_1 = \rho_2 : C \implies F_1 \rho_1 = F_2 \rho_2 : \overline{K} \rho_1}{F_1 = F_2 : K [C]}$$

Lemma 6.2. For C valid, the relations

$$\begin{aligned} K_1, K_2 &\mapsto K_1 = K_2 [C], \\ F_1, F_2 &\mapsto F_1 = F_2 : A [C] \end{aligned}$$

are pers.

Theorem 6.3. The following implications are derivable.

$$\frac{K \text{ type } [C, x:A]}{\text{fun } A \ x.K \text{ kind } [C]} \quad \frac{F : K [C, x:A]}{\lambda x.F : \text{fun } A \ x.K [C]} \quad \frac{F : \text{fun } A \ x.K [C] \quad N : A [C]}{F N : K[N] [C]}$$

Typechecking No new expressions are required, although distinct expression syntax for type family abstraction and application could be introduced if desired. We generalise the judgement form $C \vdash e \Rightarrow A$ to form $C \vdash e \Rightarrow F : K$, meaning that expression e is interpreted in C as type family F which has kind K . In the typechecking rules of section 3.2, replace every occurrence of $C \vdash e \Rightarrow A$ by $C \vdash e \Rightarrow A : \square$.

There are two new rules.

$$\frac{C \vdash e_1 \Rightarrow A \quad x \notin C \quad C, x:A \vdash e_2 \Rightarrow F : K}{C \vdash [x:e_1]e_2 \Rightarrow \lambda x.F : \text{fun } A \ x.K}$$

$$\frac{C \vdash e_1 \Rightarrow F : \text{fun } A_1 \ x.K \quad C \vdash e_2 \Rightarrow M : A_2 \quad C \vdash A_2/M_2 \sqsubseteq A_1}{C \vdash e_1 e_2 \Rightarrow F M : K[M]} \quad (14)$$

Soundness and decidability of typechecking are as expected.

7. Type Abbreviations

The local and global definitions explained in section 4 have definiens that are objects. It is also convenient to have names for types and type families. We treat this orthogonally to definitions.

Contexts and Environments We extend the syntax of contexts

$$C ::= \nabla \mid C, x:A \mid C, x=F:K$$

We extend definition 2.7 by the rule

$$\frac{\rho_1 = \rho_2 : C}{\rho_1 = \rho_2 : C, x=F:K}$$

and extend definition 2.11 for the new context constructor

$$\eta\{C, x=F:K\}\rho = \eta\{C\}\rho.$$

Typechecking No new expressions are required, although distinct expression syntax for type abbreviations could be introduced if desired. There are new typechecking rules, for global and local type abbreviations

$$\frac{x=F:K \text{ in } C}{C \vdash x \Rightarrow F : K} \quad \frac{C \vdash e_1 \Rightarrow F : K \quad x \notin C \quad C, x=F:K \vdash e_2 \Rightarrow F' : K'}{C \vdash [x=e_1]e_2 \Rightarrow F' : K'}$$

and for validity with type abbreviations.

$$\frac{\Gamma \Rightarrow C \quad C \vdash e \Rightarrow F : K \quad x \notin C}{\Gamma, x=e \Rightarrow C, x=F:K}$$

Notice that, unlike definitions (section 4.2), we expand type abbreviations when the definiendum is encountered in an expression.

8. Conclusion and Further Work

There is some way to go before our system is a usable modular logic. What we can say now is that the proposal in this paper is metamathematically well founded and directly implementable. It has most of the features of functional language module systems. Here are some of the issues we intend to work on.

The with notation. A convenient feature of many functional programming module systems is the ability to add new manifest constraints to existing signatures. For example if `grpSig` and `monSig` are the signatures of groups and monoids respectively, perhaps from an existing library, one wants to define

```
ringSig = <G:grpSig, M:monSig with crr=G.crr, ...>
```

where the `with` clause states that the group and monoid share the same carrier. We have formalised and implemented this in earlier versions of our system. The model we have given can already interpret this notation.

Inductive sets. For practical application we must be able to extend the framework with new inductively defined sets: booleans, natural numbers, lists, ... It is clear that definitions such as the naturals can be added to our model. Let N , 0 , S and rec be new objects. Add the expected computation rules for these new objects. Now choose \bar{x} such that $N = N : \bar{x}$ (definition 2.1 is parameterised by saturated \bar{x}) and define $El N$ appropriately.

Proof irrelevance. Adding a type `Prop` (analogous to \star) with a type family $Proof M$ (analogous to $El M$), where $\eta\{Proof M\} = \lambda u.()$, we get a model for a type theory with proof irrelevance and decidability of equality.

What type theory is it? Our typechecking rules have an unusual form. Can our model interpret the official rules of Martin-Löf's framework [25], or Luo's variation [21]?

Information hiding One important aspect we have not yet addressed is information hiding, so that a module can be replaced by any other having the same signature.

References

- [1] Allen, S.: *A Non-Type-Theoretic Semantics For Type-Theoretic Language*, Ph.D. Thesis, Cornell Univ., 1987, Report TR87-866.
- [2] Aspinall, D.: Subtyping with Singleton Types, *Proc. Computer Science Logic, CSL'94*, 933, 1995.
- [3] Augustsson, L.: Cayenne – a language with dependent types, *ICFP '98*, 34(1), ACM, June 1999.
- [4] Barendregt, H. P.: *The Lambda Calculus: Its Syntax and Semantics*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*, revised edition, North-Holland, 1984.
- [5] Betarte, G.: *Dependent Record Types and Formal Abstract Reasoning*, Ph.D. Thesis, Chalmers Univ. of Technology, 1998.
- [6] Betarte, G., Tasistro, A.: Extension of Martin-Löf's Type Theory with Record Types and Subtyping, *Twenty Five Years of Constructive Type Theory* (G. Sambin, J. Smith, Eds.), Oxford Univ. Press, 1998.
- [7] Chrzaszcz: Modules in Coq Are and Will Be Correct, *Types for Proofs and Programs, TYPES 2003*, 3085, Springer-Verlag, 2004.
- [8] Coq: The Coq Project, 2002, <http://coq.inria.fr/>.
- [9] Coquand, T.: An Algorithm for Testing Conversion in Type Theory, *Logical Frameworks* (G. Huet, G. Plotkin, Eds.), Camb. Univ. Press, 1991.
- [10] Courant, J.: *MC: A Module Calculus for Pure Type Systems*, Technical Report 1217, CNRS Université Paris Sud 8623: LRI, June 1999.
- [11] Courant, J.: Strong Normalization with Singleton Types, *Second Workshop on Intersection Types and Related Systems* (S. V. Bakel, Ed.), 70, Elsevier, 2002.
- [12] Crary, K.: Sound and Complete Elimination of Singleton Kinds, *Types in Compilation, TIC 2000*, 2071, Springer-Verlag, 2000.
- [13] Girard, J.-Y.: Locus Solum: From the Rules of Logic to the Logic of Rules, *Mathematical Structures in Computer Science*, **11**(3), 2001, 301–506.
- [14] Goguen, H.: A Syntactic Approach to Eta Equality in Type Theory, *Proceedings of POPL 2005*, 2005.
- [15] Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing, *POPL'94*, ACM Press, 1994.
- [16] Harper, R., Pfenning, F.: On Equivalence and Canonical Forms in the LF Type Theory, *ACM Trans. on Computational Logic*, 200?, (To appear).
- [17] Hayashi, S.: Singleton, Union and Intersection Types for Program Extraction, *Information and Computation*, **109**, 1994, 174–210.
- [18] Kopylov, A.: Dependent Intersection: A New Way of Defining Records in Type Theory, *Proceedings of the eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS-03)*, IEEE Computer Society, 2003.
- [19] Leroy, X.: Manifest types, modules, and separate compilation, *POPL'94*, ACM Press, 1994.
- [20] Leroy, X.: A syntactic theory of type generativity and sharing, *Journal of Functional Programming*, **6**(5), September 1996, 667–698.
- [21] Luo, Z.: *Computation and Reasoning: A Type Theory for Computer Science*, International Series of Monographs on Computer Science, Oxford Univ. Press, 1994.
- [22] MacQueen, D.: Using Dependent Types to Express Modular Structure, *POPL'86*, 1986.

- [23] McKinna, J., Pollack, R.: Some Lambda Calculus and Type Theory Formalized, *Journal of Automated Reasoning*, **23**(3–4), November 1999.
- [24] Miquel, A.: The Implicit Calculus of Constructions, *5th Conf. on Typed Lambda Calculi and Applications, TLCA'01* (S. Abramsky, Ed.), 2044, Springer-Verlag, 2001.
- [25] Nordström, B., Petersson, K., Smith, J.: Martin-Löf's Type Theory, in: *Handbook of Logic in Computer Science* (Abramsky, Gabbai, Maibaum, Eds.), vol. 5, Oxford Univ. Press, 2001.
- [26] Pollack, et al.: The LEGO Proof Assistant, 2001, <http://www.dcs.ed.ac.uk/home/lego/>.
- [27] Pollack, R.: Dependently Typed Records in Type Theory, *Formal Aspects of Computing*, **13**, 2002, 386–402.
- [28] Stone, C.: *Singleton Kinds and Singleton Types*, Ph.D. Thesis, Carnegie Mellon University, 2000, Report CMU-CS-00-153.
- [29] Stone, C., Harper, R.: Extensional Equivalence and Singleton Types, *ACM Transactions on Computational Logic*, 200?, To appear.

A. Collected Rules for an Implementation

A.1. Collected typechecking rules with type families and type abbreviations

- type formation; core

$$\frac{}{C \vdash * \Rightarrow * : \square} \quad \frac{C \vdash e \Rightarrow M : \star}{C \vdash \mathbf{E1} e \Rightarrow \mathbf{El} M : \square} \quad \frac{C \vdash e_1 \Rightarrow A : \square \quad C \vdash e_2 \Rightarrow B : \square}{C \vdash e_1 \rightarrow e_2 \Rightarrow A \rightarrow B : \square}$$

$$\frac{C \vdash e_1 \Rightarrow A : \square \quad x \notin C \quad C, x:A \vdash e_2 \Rightarrow B : \square}{C \vdash \{x:e_1\}e_2 \Rightarrow \mathbf{fun} A x.B : \square}$$

signatures

$$\frac{}{C \vdash \langle \rangle \Rightarrow \mathbf{1} : \square} \quad \frac{C \vdash e_1 \Rightarrow A : \square \quad l \notin C \quad C, l:A \vdash e_2 \Rightarrow B : \square}{C \vdash \langle l:e_1, e_2 \rangle \Rightarrow \langle l:A, l.B \rangle : \square}$$

$$\frac{C \vdash e_1 \Rightarrow M : A \quad l \notin C \quad C, l:A/M \vdash e_2 \Rightarrow B : \square}{C \vdash \langle l=e_1, e_2 \rangle \Rightarrow \langle l:A/M, l.B \rangle : \square}$$

families

$$\frac{C \vdash e_1 \Rightarrow A \quad x \notin C \quad C, x:A \vdash e_2 \Rightarrow F : K}{C \vdash [x:e_1]e_2 \Rightarrow \lambda x.F : \mathbf{fun} A x.K}$$

$$\frac{C \vdash e_1 \Rightarrow F : \mathbf{fun} A_1 x.K \quad C \vdash e_2 \Rightarrow M : A_2 \quad C \vdash A_2/M_2 \sqsubseteq A_1}{C \vdash e_1 e_2 \Rightarrow F M : K[M]}$$

definitions and abbreviations

$$\frac{C \vdash e_1 \Rightarrow N : A \quad x \notin C \quad C, x:A/N \vdash e_2 \Rightarrow F : K}{C \vdash [x=e_1]e_2 \Rightarrow F[N] : K[N]}$$

$$\frac{x=F:K \text{ in } C}{C \vdash x \Rightarrow F : K} \quad \frac{C \vdash e_1 \Rightarrow F : K \quad x \notin C \quad C, x=F:K \vdash e_2 \Rightarrow F' : K'}{C \vdash [x=e_1]e_2 \Rightarrow F' : K'}$$

• objects

$$\frac{x:A \text{ in } C}{C \vdash x \Rightarrow x : A} \quad \frac{C \vdash e_1 \Rightarrow A : \square \quad x \notin C \quad C, x:A \vdash e_2 \Rightarrow M : B}{C \vdash [x:e_1]e_2 \Rightarrow \lambda x.M : \text{fun } A x.B}$$

$$\frac{C \vdash e_1 \Rightarrow M_1 : \text{fun } A_1 x.B \quad C \vdash e_2 \Rightarrow M_2 : A_2 \quad C \vdash A_2/M_2 \sqsubseteq A_1}{C \vdash e_1 e_2 \Rightarrow M_1 M_2 : B[M_2]}$$

$$\frac{C \vdash e_1 \Rightarrow N : A \quad x \notin C \quad C, x:A/N \vdash e_2 \Rightarrow M : B}{C \vdash [x=e_1]e_2 \Rightarrow M[N] : B[N]}$$

$$\frac{}{C \vdash () \Rightarrow () : \mathbf{1}} \quad \frac{C \vdash e_1 \Rightarrow M_1 : A \quad C \vdash e_2 \Rightarrow M_2 : B}{C \vdash (l=e_1, e_2) \Rightarrow (l=M_1, M_2) : \langle l:A/M_1, x.B \rangle}$$

$$\frac{C \vdash e \Rightarrow M : S \quad l:B/N \text{ in } S/M}{C \vdash e.l \Rightarrow N : B}$$

• validity

$$\frac{}{\blacktriangledown \Rightarrow \nabla} \quad \frac{\Gamma \Rightarrow C \quad C \vdash e \Rightarrow A : \square \quad x \notin C}{\Gamma, x:e \Rightarrow C, x:A}$$

$$\frac{\Gamma \Rightarrow C \quad C \vdash e \Rightarrow N : A \quad x \notin C}{\Gamma, x=e \Rightarrow C, x:A/N} \quad \frac{\Gamma \Rightarrow C \quad C \vdash e \Rightarrow F : K \quad x \notin C}{\Gamma, x=e \Rightarrow C, x=F:K}$$

A.2. Syntactic subtyping

$$\frac{}{C \vdash \star/M \sqsubseteq \star} \quad \frac{M_1 = M_2 : \star [C]}{C \vdash (El M_1)/M \sqsubseteq El M_2} \quad \frac{}{C \vdash A/M \sqsubseteq \mathbf{1}}$$

$$\frac{C, x:A_2 \vdash A_2/x \sqsubseteq A_1 \quad C, x:A_2 \vdash B_1/(Mx) \sqsubseteq B_2}{C \vdash (\text{fun } A_1 x.B_1)/M \sqsubseteq \text{fun } A_2 x.B_2}$$

$$\frac{C \vdash A/M \sqsubseteq B \quad M = N : B [C]}{C \vdash A/M \sqsubseteq B/N}$$

$$\frac{l:A'/N \text{ in } S/M \quad C \vdash A'/N \sqsubseteq A \quad C \vdash S/M \sqsubseteq B[N]}{C \vdash S/M \sqsubseteq \langle l:A, x.B \rangle}$$

where the relation $l:A \text{ in } S$ is defined by the following rules.

$$\frac{}{l:A/N \text{ in } \langle l:A/N, x.B \rangle/M} \quad \frac{A \text{ not singleton}}{l:A/M.l \text{ in } \langle l:A, x.B \rangle/M}$$

$$\frac{l:A/N \text{ in } B[M.l']/M \quad l \neq l'}{l:A/N \text{ in } \langle l':A', x.B \rangle/M}$$

A.3. Generalised eta expansion

$$\begin{aligned}
\eta\{\star\} u &= u \\
\eta\{El M\} u &= u \\
\eta\{\text{fun } A x.B\} u &= \lambda z.\eta\{B[\eta\{A\}z]\} (u (\eta\{A\}z)) \\
\eta\{A/M\} u &= \eta\{A\}M \\
\eta\{\mathbf{1}\} u &= () \\
\eta\{<l:A, x.B>\} u &= (l=\eta\{A\}(u.l), \eta\{B[\eta\{A\}(u.l)]\}u)
\end{aligned}$$

where z is not free in $\eta\{A\}$ or B .

$$\begin{aligned}
\eta\{\nabla\} \rho &= \rho \\
\eta\{C, x:A\} \rho &= (\eta\{C\} \rho, x=\eta\{A(\eta\{C\} \rho)\}(\rho x)) \\
\eta\{C, x=F:K\} \rho &= \eta\{C\} \rho.
\end{aligned}$$