

Pure Type Systems Formalized ^{*†}

James McKinna
jhm@dcs.ed.ac.uk

Robert Pollack
rap@dcs.ed.ac.uk

Laboratory for Foundations of Computer Science
The King's Buildings, University of Edinburgh, EH9 3JZ, U.K.

1 Introduction

This paper is about our hobby. For us, machine-checked mathematics is a passion, and constructive type theory (in the broadest sense) is the way to this objective. Efficient and correct type-checking programs are necessary, so a formal theory of type systems leading to verified type synthesis algorithms is a natural goal. For over a year the second author has been developing a machine-checked presentation of the elementary meta-theory of Pure Type Systems (PTS) [Bar91], (formerly called Generalized Type Systems (GTS)). This project was blocked until the first author collaborated with a fresh idea. Here we describe the state of this ongoing project, presenting a completely formal, machine checked development of this basic meta-theory, including the underlying language of (explicitly typed) lambda calculus. We discuss some of the choices involved in formalization, some of the difficulties encountered, and techniques to overcome these difficulties.

PTS, a “framework of purely functional type theories” has a beautiful meta-theory, developed informally in [Bar92, Ber90, GN91, vBJ93]. The cited papers are unusually clear and mathematical, and there is little doubt about the correctness of the results we will discuss, so why write a machine-checked development? The informal presentations leave many decisions unspecified and many facts unproved. They are far from the detail of representation needed to write a computer program for proofchecking, and the lemmas needed to prove correctness of such a program. Our long-term goal is to fill these gaps.

Another goal of the project is to develop a realistic example of formal mathematics. In mathematics and computer science we do not prove one big theorem and then throw away all the work leading up to that theorem; we want to build a body of formal knowledge that can continually be extended and developed. Representations and definitions must be suitable for the whole development, not specialized for a single theorem. The theory should be structured, like computer programs, by abstraction, providing “isolation of components” so that several parts of the theory can be worked on simultaneously, perhaps by several workers, and so that the inevitable wrong decisions in underlying representations can later be fixed without affecting too seriously a large theory that depends on them. We are not working from a completed informal

*This work was supported by the ESPRIT Basic Research Action on Logical Frameworks, and by grants from the British Science and Engineering Research Council.

†A version of this paper appears in the *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA '93*, Utrecht, March, 1993, Springer-Verlag, LNCS 664.

theory; there are still open questions such as the Expansion Postponement problem [Pol92]. In particular, the theory of type-checking and type-synthesis has recently made much progress, and involves several related systems with similar properties [vBJMP94]. While the basic informal theory is well understood, we suggest reformulations which clarify the presentation. On the other hand, in some ways type theory is not a realistic example of formal mathematics: it is especially suitable for formalization because the objects are inductively constructed, their properties are proved by induction over structure, and there is little equality reasoning, one of the weak points of intentional type theory.

A novelty in our presentation is the use of named variables instead of de Bruijn “nameless variables”. Most of the formalizations of type theory or lambda calculus that we know of (e.g. [Sha85, Alt93]) use de Bruijn indices to avoid formalizing the Curry-style renaming of variables to prevent capture. While de Bruijn notation is very elegant and suitable for formalization, there are reasons to formalize the theory with named variables. For one thing, implementations must use names at some level, whether internally or only for parsing and printing. In either case this use of names must be formally explained. Also interesting is the insight to be gained into what we mean by binding. It is common wisdom among researchers that de Bruijn representation “really is” what we mean by lambda terms, in the sense that there is no quotient of terms by alpha-conversion, i.e. intensional equality on de Bruijn terms corresponds with what we intend to mean by identity of terms. We feel that this perception is correct for bound variables, but not for free variables: the names of free variables matter, their order of occurrence does not. Thus, in de Bruijn representation, we must consider a quotient with respect to the order of free variables, and while this notion is formalizable (as is Curry style alpha-conversion), it is inconvenient, and shows that de Bruijn representation is not the perfect answer to the problem. We use a formulation suggested by Coquand [Coq91] in which explicit alpha-conversion of named variables is not necessary in the theory of reduction, conversion, and typing.

The LEGO Proof Development System [LP92] was used to check the work in an implementation of the Pure Calculus of Constructions extended with inductive types in the style of the Martin-Löf logical framework (or monomorphic set theory) [NPS90] (first implemented in ALF). LEGO is a refinement style proof checker, publicly available by ftp, with a User’s Manual [LP92] and a large collection of examples. Interesting examples formalized in LEGO include the Tarski fixedpoint theorem (Pollack), construction of the reals from the rationals and completion of a metric space [Jon91], the chinese remainder theorem [McK92], program specification and data refinement [Luo91], and the Schröder-Bernstein theorem (M. Hofmann, proof in [LP92]). Recently Thorsten Altenkirch has used LEGO’s inductive types to give a very elegant and informative proof of strong normalization for System F [Alt93].

Acknowledgement We especially thank Thierry Coquand for many inspiring conversations, and many ideas that appear in this work.

2 Some Basic Types

We use LEGO’s built-in library of impredicative definitions for the usual logical connectives and their properties, although inductive definitions would do just as well. We also use LEGO’s library of basic inductive types. These include a type of booleans, `BB`, containing `tt` and `ff`, with the usual classical boolean operators, conjunction `andd`, disjunction `orrr` and conditional

`if`, together with the lifting functions `is_tt` and `is_ff`, which convert booleans to (decidable) propositions. Also, a type of polymorphic lists, `LL`, with its induction principle `LLrec`, and many common operations such as `append`, `member` and `assoc`, and a type of polymorphic cartesian products, `PROD`, with pairing `Pr`, and projections `Fst` and `Snd`. An inductive type of natural numbers, `NN` is included in the library, and used to support induction on the length of terms.

We use an inductive equality relation, `Q`, although Leibniz equality is sufficient for our purposes.

3 The Presentation of a PTS

A PTS is a 5-tuple $(PP, VV, SS, Ax, Rule)$ where

- `PP` is an infinite set of *parameters*, ranged over by p, q . Parameters are the global, or free, variables.
- `VV` is an infinite set of *variables*, ranged over by x, v, u . Variables are the local, or bound, variables.
- `SS`, a set of *sorts*, ranged over by s . Sorts are the constants.
- `Ax` $\subseteq SS \times SS$, a set of *axioms* of the form `Ax(s:s)`
- `Rule` $\subseteq SS \times SS \times SS$, a set of *rules* of the form `Rule(s1, s2, s3)`

The parameters, variables, and sorts are used in term construction; the axioms and rules parameterize an inductively defined typing judgement. We now formalize this presentation.

Assume there is a type of parameters, `PP`, that is infinite and has a decidable equivalence relation. In fact we also assume that the decidable equivalence relation on `PP` is the same as intensional equality; this extra assumption is not necessary but it vastly simplifies our formal development.

```
[PP : Prop];                               (* Parameters *)
[PPeq : PP->PP->BB];
[PPeq_decides_Q : {p,q:PP}iff (is_tt (PPeq p q)) (Q p q)];
[PPs = LL|PP];                               (* lists of parameters *)
[PPinf : {l:PPs}ex[p:PP] is_ff (member PPeq p l)];
```

`PPinf` is a “local gensym” operation. It says that for every list of parameters, `l`, there is a parameter, `p`, that is not a member of `l`. `member` is defined with respect to some decidable equality, in this case `PPeq`.

These are not mathematical principles we are assuming, but part of the presentation of the language of a PTS. Having developed some theory of PTS in `LEGO`, we may `Discharge` these assumptions, making the whole theory parametric in such a type of parameters. The assumption that `PPeq` is equivalent to `Q` means that we may instantiate `PP` with, for example the type of natural numbers and its inductively definable decidable equality, or with the type of lists of characters, but not with the type of integers defined as a quotient over pairs of naturals, because in this latter type the intended equality, definable by induction, is not equivalent to `Q`.

Also, assume a type of variables, `VV`, with similar properties, `VVeq`, `VVeq_decides_Q`, `VVinf`; and a type of sorts, `SS`, which has decidable equality, `SSeq`, `SSeq_decides_Q`, but need not be infinite.

What remains to complete a presentation is the axioms and rules that parameterize the typing judgement. They are just relations.

```
[ax : SS->SS->Prop];
[rl : SS->SS->SS->Prop];
```

We usually intend **ax** and **rl** to be decidable, but the elementary theory developed in this paper does not use such an assumption. If we are interested in decidability of typechecking or algorithms for type synthesis, even stronger assumptions about decidability are needed [Pol92, vBJMP94].

4 Terms

The syntax of terms of PTS (**PP**, **VV**, **SS**, **Ax**, **Rule**) is informally given by

```
atoms   $\alpha$  ::=  $p$  |  $x$  |  $s$ 
terms   $M$  ::=  $\alpha$  |  $[x:M]M$  |  $\{x:M\}M$  |  $MM$   atoms, lambda, pi, application
```

The type of terms is formalized as an inductive type.

```
[Trm:Prop];
[sort:SS->Trm];           (* Trm constructors (introduction rules) *)
[var:VV->Trm];
[par:PP->Trm];
[pi:VV->Trm->Trm->Trm];
[llda:VV->Trm->Trm->Trm];
[app:Trm->Trm->Trm];
```

Every term is a finitely branching well-founded tree. In particular, the **lda** and **pi** constructors do *not* have type $\text{Trm} \rightarrow (\text{VV} \rightarrow \text{Trm}) \rightarrow \text{Trm}$, which would create well-founded but infinitely branching terms. The induction principle for **Trm** is

```
[Trec:{C:Trm->Prop}      (* Trm induction (elimination) principle *)
  {TSORT:{s:SS}C (sort s)}
  {TVAR:{n:VV}C (var n)}
  {TPAR:{n:PP}C (par n)}
  {TPI:{n:VV}{A,B:Trm}(C A)->(C B)->C (pi n A B)}
  {TLDA:{n:VV}{A,B:Trm}(C A)->(C B)->C (lda n A B)}
  {TAPP:{M,N:Trm}(C M)->(C N)->C (app M N)}
  {t:Trm}C t];
```

There is a boolean-valued structural equality function, **Trm_eq**, inductively definable on terms. Because **PPeq**, **VVeq**, and **SSeq** are equivalent to **Q**, **Trm_eq** is also provably equivalent to **Q**.

Substitution For the machinery on terms, we need two kinds of substitution, both defined by primitive recursion over term structure using the induction principle **Trec**. Substitution of a term for a parameter is entirely textual, not preventing capture. Since parameters have no binding instances in terms (we may view them as being globally bound by the context in a judgement), there is no hiding of a parameter name by a binder.

```

[psub [M:Trm] [n:PP] : Trm->Trm =
  Trec ([_:Trm]Trm)
    ([s:SS]sort s)
    ([v:VV]var v)
    ([p:PP]if (PPEq n p) M (par p))
    ([v:VV] [_,_],l,r:Trm]pi v l r)
    ([v:VV] [_,_],l,r:Trm]lda v l r)
    ([_ ,_,l,r:Trm]app l r)];

```

Substitution of a term for a variable does respect variable binders that hide their bound instances from substitution, but does not prevent capture.

```

[vsub [M:Trm] [n:VV] : Trm->Trm =
  Trec ([_:Trm]Trm)
    ([s:SS]sort s)
    ([v:VV]if (VVEq n v) M (var v))
    ([p:PP]par p)
    ([v:VV] [_,or,nl,nr:Trm]pi v nl (if (VVEq n v) or nr))
    ([v:VV] [_,or,nl,nr:Trm]lda v nl (if (VVEq n v) or nr))
    ([_ ,_,l,r:Trm]app l r)];

```

Both of these will be used only in safe ways in the type theory and the theory of reduction and conversion, so as to prevent unintended capture of variables.

There is a frequently used abbreviation for substituting a parameter for a variable

```

[alpha [p:PP] [v:VV] = vsub (par p) v];

```

This is not alpha conversion in the usual sense.

Free and Bound Occurrences The list of parameters occurring in a term is computed by primitive recursion over term structure, and the boolean judgement whether or not a given parameter occurs in a given term is computed by the `member` function on this list of parameters.

```

[params : Trm->PPs =
  Trec ([_:Trm]PPs)
    ([_:SS]NIL|PP)
    ([_:VV]NIL|PP)
    ([p:PP]unit p)
    ([_:VV] [_,_]:Trm] [l,r:PPs]append l r)
    ([_:VV] [_,_]:Trm] [l,r:PPs]append l r)
    ([_ ,_:Trm] [l,r:PPs]append l r)];
[poccur [p:PP] [A:Trm] : BB = member PPEq p (params A)];

```

Similarly `sorts` and `soccur` are defined.

We do not compute the list of variables occurring free in a term, but instead define inductively a notion `Vclosed` of *closed term* (Table 1). In fact, only the `Vclosed` terms are considered to be well formed for the theory of reduction in the same way that only typable terms will be considered well formed for the type theory in later sections. Thus `Vclosed` is used as an induction principle over well formed terms.

`Vclosed` is provably equivalent to having no free variables, although the proof is not as trivial as might be expected. (Thanks to Thierry Coquand for the crucial step in this proof.)

```

[Vclosed : Trm->Prop];

[Vclosed_sort : {s:SS} Vclosed (sort s)];

[Vclosed_par : {p:PP} Vclosed (par p)];

[Vclosed_pi : {n|VV}{A,B|Trm}{p|PP}
  {premA:Vclosed A} {premB:Vclosed (alpha p n B)}
  (*****
   Vclosed (pi n A B)];

[Vclosed_lda : {n|VV}{A,B|Trm}{p|PP}
  {premA:Vclosed A} {premB:Vclosed (alpha p n B)}
  (*****
   Vclosed (lda n A B)];

[Vclosed_app : {A,B|Trm}
  {premA:Vclosed A} {premB:Vclosed B}
  (*****
   Vclosed (app A B)];

```

Table 1: The inductive property `Vclosed`

5 Reduction and Conversion

Table 2 shows the reduction and conversion relations. `par_red1` is the *one-step parallel reduction* used in the Tait–Martin-Löf proof of the Church-Rosser property for β . The interesting point in these rules is how reduction under binders is handled: to go under a binder, replace the free variable occurrences by a suitably fresh parameter, operate on the closed subterm, and undo the “closing up”. For example, consider informally one-step β -reduction of untyped lambda calculus. In the style of our formalization the ξ rule is

$$\xi \quad \frac{[q/x]M \rightarrow [q/y]N}{\lambda x.M \rightarrow \lambda y.N} \quad q \notin M, q \notin N$$

where $[q/x]M$ is our `(alpha q x M)`. Here is an instance of this rule, contracting the underlined redex.

$$\frac{(\lambda v.\lambda x.v) q \rightarrow \lambda x.q}{\lambda x.((\lambda v.\lambda x.v) x) \rightarrow \lambda y.\lambda x.y}$$

After removing the outer λx , replacing its bound instances by a fresh parameter, q , and contracting the closed weak-head redex thus obtained¹, we must re-bind the hole now occupied by q . According to the rule ξ , we require a variable, y , and a term, N , such that $[q/y]N = \lambda x.q$. Such a pair is $y, \lambda x.y$ (the one we have used above), as is $z, \lambda x.z$ for any $z \neq x$. However $x, \lambda x.x$ will not do, for `vsub` will not substitute q for x under the binder λx .

¹There is no possibility of variable capture when contracting a closed weak-head redex.

```

[par_red1 : Trm->Trm->Prop]                                (* parallel 1-step reduction *)

[par_red1_refl : refl par_red1];

[par_red1_beta:
  {U:Trm}{A,A'|Trm}{premA:par_red1 A A'}
  {u,v|VV}{B,B'|Trm}{p|PP}
  {nocCB:is_ff (poccur p B)}{nocCB':is_ff (poccur p B')}
  {premB:par_red1 (alpha p u B) (alpha p v B')}
  (*****
   par_red1 (app (lda u U B) A) (vsub A' v B'))];

[par_red1_pi:
  {A,A'|Trm}{premA:par_red1 A A'}
  {u,v|VV}{B,B'|Trm}{p|PP}
  {nocCB:is_ff (poccur p B)}{nocCB':is_ff (poccur p B')}
  {premB:par_red1 (alpha p u B) (alpha p v B')}
  (*****
   par_red1 (pi u A B) (pi v A' B'))];

[par_red1_lda:
  {A,A'|Trm}{premA:par_red1 A A'}
  {u,v|VV}{B,B'|Trm}{p|PP}
  {nocCB:is_ff (poccur p B)}{nocCB':is_ff (poccur p B')}
  {premB:par_red1 (alpha p u B) (alpha p v B')}
  (*****
   par_red1 (lda u A B) (lda v A' B'))];

[par_red1_app:
  {A,A'|Trm}{premA:par_red1 A A'}
  {B,B'|Trm}{premB:par_red1 B B'}
  (*****
   par_red1 (app A B) (app A' B'))];

[par_redn : Trm->Trm->Prop];                                (* parallel n-step reduction *)
[par_redn_red1 : {A,B|Trm}(par_red1 A B)->par_redn A B];
[par_redn_trans : trans par_redn];

[conv : Trm->Trm->Prop];                                    (* conversion *)
[reflConv : refl conv];
[symConv : sym conv];
[transConv : {A,D,B|Trm}(conv A D)->(Vclosed D)->(conv D B)->conv A B];
[rednConv : {A,B|Trm}(par_redn A B)->conv A B];

```

Table 2: The 1-Step Reduction, Many-step Reduction, and Conversion Relations

Parallel n -step reduction, `par_redn` (Table 2), is the transitive closure of `par_red1`. One must take care in defining conversion, `conv`, or the second Church-Rosser property will fail. In particular, the diamond property only holds for `Vclosed` terms. While reduction preserves `Vclosed`, expansion does not, so the transitivity rule for conversion, `transConv` in Table 2, must guarantee that the intermediate term is `Vclosed`.

The Church-Rosser Theorem is now proved by the argument of Tait and Martin-Löf.

```
[CommonReduct [t|Type] [R,S:t->t->Prop]
  = [b,c:t]Ex [d:t]and (S b d) (R c d)];
[DiamondProperty [t|Type] [P:t->Prop] [R:t->t->Prop]
  = {a,b,c|t}(P a)->(R a b)->(R a c)->CommonReduct R R b c];
```

```
Goal DiamondProperty Vclosed par_red1;
Goal DiamondProperty Vclosed par_redn;
Goal {A,B|Trm}(conv A B)->(Vclosed A)->(Vclosed B)->
  Ex[C:Trm] and (par_redn A C) (par_redn B C);
```

In these proofs there is no need define or reason about any notion of α -conversion. The proofs are by structural induction, but use a technique we will describe in Section 9.2.

6 Contexts

The type of global bindings, $[p:A]$, that occur in contexts, is a cartesian product of `PP` by `Trm`. We also give the two projections of global bindings, and a defined structural equality that is provably equivalent to `Q`.

```
[GB : Prop = PROD|PP|Trm]; (* type of global bindings .. *)
[Gb : PP->Trm->GB = Pr|PP|Trm]; (* .. its constructor .. *)
[namOf = Fst][typOf = Snd]; (* .. its destructors .. *)
[GBeq [b,c:GB]
  = andd (PPEq (namOf b) (namOf c)) (Trm_eq (typOf b) (typOf c))];
```

Contexts are lists of global bindings.

```
[Cxt = LL|GB];
[nilCxt = NIL|GB];
```

Occurrences Occurrence of a global binding in a context is defined using the `member` function and the defined structural equality `GBeq`.

```
[GBoccur [b:GB][G:Cxt] : BB = member GBeq b G];
```

Now we define the list of parameters “bound” by a context, and, using the `member` function as before, the boolean relation deciding whether or not a given parameter is bound by a given context.

```
(* names with binding instances in a context *)
[globalNames : Cxt->PPs
```

AX	$\bullet \vdash c : s$	$\text{Ax}(c:s)$
START	$\frac{\Gamma \vdash A : s}{\Gamma[p:A] \vdash p : A}$	$p \notin \Gamma$
VWEAK	$\frac{\Gamma \vdash q : C \quad \Gamma \vdash A : s}{\Gamma[p:A] \vdash q : C}$	$p \notin \Gamma$
SWEAK	$\frac{\Gamma \vdash s : C \quad \Gamma \vdash A : s}{\Gamma[p:A] \vdash s : C}$	$p \notin \Gamma$
PI	$\frac{\Gamma \vdash A : s_1 \quad \Gamma[p:A] \vdash [p/x]B : s_2}{\Gamma \vdash \{x:A\}B : s_3}$	$p \notin B, \text{Rule}(s_1, s_2, s_3)$
LDA	$\frac{\Gamma[p:A] \vdash [p/x]M : [p/y]B \quad \Gamma \vdash \{y:A\}B : s}{\Gamma \vdash [x:A]M : \{y:A\}B}$	$p \notin M, p \notin B$
APP	$\frac{\Gamma \vdash M : \{x:A\}B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$	
TCONV	$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A \simeq B$

Table 3: The Informal Typing Rules of PTS

```

= LLrec ([_:Cxt]PPs)
  (NIL|PP) (* nil *)
  ([b:GB] [_:Cxt] [rest:PPs]CONS (namOf b) rest)]; (* cons *)
[Poccur [p:PP] [G:Cxt] : BB = member PPeq p (globalNames G)];

```

Subcontexts The subcontext relation is defined by containment of the respective occurrence predicates.

```
[subCxt [G,H:Cxt] = {b:GB}(is_tt (GBoccur b G))->is_tt (GBoccur b H)];
```

This is exactly the definition used informally in [Bar92, GN91, vBJ93]. Imagine expressing this property in a representation using de Bruijn indices for global variables.

7 The Typing Judgement

The typing judgement has the shape $\Gamma \vdash M : A$, meaning that in context Γ , term M has type A . An informal presentation of the rules inductively defining this relation is shown in Table 3. First observe that the handling of parameters and variables in the PI and LDA rules is similar to that in the rules of Table 2: to operate under a binder, locally extend the context replacing the newly freed variable by a new parameter, do the operation, then forget the parameter and bind the variable again. Natural deduction always has this aspect of locally extending and discharging the context, and implementors know that assumption/discharge is related to variable names. Huet’s Constructive Engine [Hue89], for example, “weaves” back and forth between named

global variables and de Bruijn indices for local binding in a manner similar to that of Table 3. Even an implementation that uses only one class of named variables may have to change names in the PI and LDA rules because locally bound names may be reused, but global names must be unique. In this regard, notice that the LDA rule allows “alpha converting” in the conclusion. Informal presentations [Bar92, Ber90, GN91] suggest the rule

$$\text{LDA}' \quad \frac{\Gamma[p:A] \vdash [p/x]M : [p/x]B \quad \Gamma \vdash \{x:A\}B : s}{\Gamma \vdash [x:A]M : \{x:A\}B} \quad p \notin M, p \notin B$$

To see why we use the rule LDA instead of LDA', consider deriving the judgement

$$[A:*][P:A \rightarrow *] \vdash [x:A][x:Px]x : \{x:A\}\{y:Px\}Px$$

in the Pure Calculus of Constructions. While both systems, using LDA, and using LDA', derive this judgement, they do so with different derivations, as the system with LDA' must use the rule TCONV to alpha-convert x to y in the type part of the judgement. In particular, the system with LDA' has different derivations than a presentation using de Bruijn indices.

Another thing to notice about Table 3 is that we have replaced the usual weakening rule

$$\text{WEAK} \quad \frac{\Gamma \vdash M : C \quad \Gamma \vdash A : s}{\Gamma[p:A] \vdash M : C} \quad p \notin \Gamma$$

having an arbitrary term as subject, with vWEAK and sWEAK, restricting weakening to atomic subjects, and later show that WEAK is derivable in our system. This presentation improves the meta-theory (see Section 8) and makes the presentation more syntax-directed for type checking [Pol92].

Finally observe that instances of $[M/v]N$ in Table 3 only occur at the top level (not under a binder) and with M vclosed: it is safe to use vsub in this manner, even though vsub is not a correct substitution operation.

The Formal Typing Judgement The typing rules, formalized as the constructors of an inductive relation, **gts**, are shown in Table 4. To understand this formalization, consider the “genericity” of the informal rules, presented in Table 3. The rule START states that for any context Γ , term A , and parameter $p \notin \Gamma$, to derive $\Gamma[p:A] \vdash p : A$ it suffices to derive $\Gamma \vdash A : s$. What is the quantification of s in this explanation? The view that rules are constructors of derivations suggests that one must actually provide s and a derivation of $\Gamma \vdash A : s$, i.e. that the correct formalization is “for any context Γ , term A , parameter p , and sort s, \dots ”. This question is more interesting for the PI rule: must we actually supply p , is it sufficient to know the existence of such a p , or is there another way to characterize those parameters that may be used in an instance of the rule? Here we feel that p really doesn't matter, as long as it is “fresh enough”, i.e. doesn't occur in Γ or B . Again, we decide that the correct formalization actually requires a particular parameter p to be supplied. Another way to understand the choice we have made is that the meaning of the informal rules is as constructors of well-founded and finitely branching trees. It is also possible to accept a type of well-founded but infinitely branching trees, and the issue is discussed again in Section 9.2.

8 Some Basic Lemmas

Most of the basic results about **gts** are straightforward to prove by induction. For example

```

[gts : Cxt->Trm->Trm->Prop];

[Ax : {s1,s2|SS}{sc:ax s1 s2}
      gts nilCxt (sort s1) (sort s2)];

[Start : {G|Cxt}{A|Trm}{s|SS}{p|PP}{sc:is_ff (Poccur p G)}
         {prem:gts G A (sort s)}
         (*****)
         gts (CONS (Gb p A) G) (par p) A];

[vWeak : {G|Cxt}{D,A|Trm}{s|SS}{n,p|PP}{sc:is_ff (Poccur p G)}
         {l_prem:gts G (par n) D}
         {r_prem:gts G A (sort s)}
         (*****)
         gts (CONS (Gb p A) G) (par n) D];

[sWeak : {G|Cxt}{D,A|Trm}{t,s|SS}{p|PP}{sc:is_ff (Poccur p G)}
         {l_prem:gts G (sort t) D}
         {r_prem:gts G A (sort s)}
         (*****)
         gts (CONS (Gb p A) G) (sort t) D];

[Pi : {G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{p|PP}{n|VV}
      {sc:rl s1 s2 s3}{sc':is_ff (poccur p B)}
      {l_prem:gts G A (sort s1)}
      {r_prem:gts (CONS (Gb p A) G) (alpha p n B) (sort s2)}
      (*****)
      gts G (pi n A B) (sort s3)];

[Lda : {G|Cxt}{A,M,B|Trm}{s|SS}{p|PP}{n,m|VV}
      {sc:and (is_ff (poccur p M)) (is_ff (poccur p B))}
      {l_prem:gts (CONS (Gb p A) G) (alpha p n M) (alpha p m B)}
      {r_prem:gts G (pi m A B) (sort s)}
      (*****)
      gts G (lda n A M) (pi m A B)];

[App : {G|Cxt}{M,A,B,L|Trm}{n|VV}
      {l_prem:gts G M (pi n A B)}
      {r_prem:gts G L A}
      (*****)
      gts G (app M L) (vsub L n B)];

[tConv : {G|Cxt}{M,A,B|Trm}{s|SS}{sc:conv A B}
         {l_prem:gts G M A}
         {r_prem:gts G B (sort s)}
         (*****)
         gts G M B];

```

```

(* Free Variable Lemmas *)
Goal {G|Cxt}{M,A|Trm}(gts G M A)->and (Vclosed M) (Vclosed A);
Goal {G|Cxt}{M,A|Trm}(gts G M A)->
  {p|PP}(or (is_tt (poccur p M)) (is_tt (poccur p A)))->is_tt (Poccur p G);
(* Start Lemmas *)
Goal {G|Cxt}{M,A|Trm}(gts G M A)->
  {s1,s2:SS}(ax s1 s2)->(gts G (sort s1) (sort s2));
Goal {G|Cxt}{M,A|Trm}(gts G M A)->
  {b|GB}(is_tt (GBoccur b G))->gts G (par (namOf b)) (typOf b);

```

Generation Lemmas For each term constructor, there is one rule in Table 3 with that term constructor as its subject. In our presentation, only the rule `tConv` is not syntax-directed, giving a form of “uniqueness of generation” for judgements, up to conversion. This observation is formalized in the Generation Lemmas. For example, the Generation Lemma for sorts, whose types are generated by `AX`, and that for parameters, whose types are generated by `START` are

```

Goal {G|Cxt}{s|SS}{C|Trm}{d:gts G (sort s) C}
  Ex [s1:SS] and (ax s s1) (conv (sort s1) C);
Goal {G|Cxt}{C|Trm}{p|PP}{d:gts G (par p) C}
  Ex [B:Trm]and (is_tt (GBoccur (Gb p B) G)) (conv B C);

```

In the standard presentation of PTS, say [Bar92], with the general non-syntax-directed weakening rule `WEAK`, uniqueness of generation holds only up to conversion *and weakening*; then the generation lemmas depend on the Thinning Lemma, while in our presentation with atomic weakening, they are provable before the Thinning Lemma.

9 The Thinning Lemma

The Thinning Lemma is important to our formulation because it shows that full weakening is derivable in our system from atomic weakening. First a definition

```
[Valid [G:Cxt] = Ex2[M,A:Trm] gts G M A];
```

```
(** Thinning Lemma **)
```

```
Goal {G,G'|Cxt}{M,A|Trm}(gts G M A)->(subCxt G G')->(Valid G')->gts G' M A;
```

When we come to prove the Thinning Lemma, a serious difficulty arises from our use of parameters. We temporarily revert to informal notation.

Naive attempt to prove the Thinning Lemma By induction on the derivation of $G \vdash M : A$. Consider the case for the `Pi` rule: the derivation ends with

$$\frac{G \vdash A : s_1 \quad G[p:A] \vdash [p/x]B : s_2}{G \vdash \{x:A\}B : s_3} \quad \text{Rule}(s_1, s_2, s_3)$$

left IH For any K , if G is a subcontext of K and K is Valid, then $K \vdash A : s_1$.

right IH For any K , if $G[p:A]$ is a subcontext of K and K is Valid, then $K \vdash [p/x]B : s_2$.

to show For any G' , if G is a subcontext of G' and G' is Valid, then $G' \vdash \{x:A\}B : s_3$.

So let G' be a valid extension of G . Using the Pi rule, we need to show $G' \vdash A : s_1$ and $G'[p:A] \vdash [p/x]B : s_2$. The first of these is proved by the left IH applied to G' . In order to use the right IH applied to $G'[p:A]$ to prove the second, we need to show $G'[p:A]$ is valid. However, this may be false, e.g. if p occurs in G' !

The left premiss of the LDA rule presents a similar problem. All other cases are straightforward.

Naive attempt to fix the problem Let us change the name of p in the right premiss. We prove a global renaming lemma to the effect that any injective renaming of parameters preserves judgements. (This is straightforward given the technology of renaming we will develop below.) However, this does not help us finish the naive proof of the Thinning Lemma, for the right IH still mentions p , not some fresh parameter q . That is, although we can turn the right subderivation $G[p:A] \vdash [p/x]B : s_2$ into a derivation of $G[q:A] \vdash [q/x]B : s_2$ for some fresh q , this is *not* a structural subderivation of the original proof of $G \vdash \{x:A\}B : s_3$, and is no help in structural induction.

9.1 Some correct proofs of the Thinning Lemma

We present three correct proofs of the Thinning Lemma, based on different analyses of what goes wrong in the naive proof. The two proofs in this section are, we think, possible to formalize, but a little heavy. Section 9.2 gives the more beautiful formalization we have chosen.

Changing the names of parameters. Once we say “by induction on the derivation ...” it is too late to change the names of any troublesome parameters, but one may fix the naive proof by changing the names of parameters in a derivation *before* the structural induction. First observe, by structural induction, that if d is a derivation of $G \vdash M : A$, G' is a Valid extension of G , and no parameter occurring in d is in $\text{globalNames}(G') \setminus \text{globalNames}(G)$, then $G' \vdash M : A$. Now for the Thinning Lemma, given G , G' , and a derivation d of $G \vdash M : A$, change parameters in d to produce a derivation d' of $G \vdash M : A$ such that no parameter occurring in d' is in $\text{globalNames}(G') \setminus \text{globalNames}(G)$. The previous observation completes the proof.

This proof takes the view that it is fortunate there are many derivations of each judgement, for we can find one suitable for our purposes among them. To formalize it we need technology for renaming parameters in derivations, which is heavier than the technology for renaming parameters in *judgements* that we will present below.

Induction on length. A different analysis of the failure of the naive proof shows that it is the use of *structural* induction that is at fault. Induction on the height of derivations appears to work, but we must show that renaming parameters in a derivation doesn't change its height; again reasoning about derivations rather than judgements. Induction on the length of the subject term M does not seem to work, as some rules have premisses whose subject is not shorter than that of the conclusion.

9.2 A Better Solution

We take the view that there are too many derivations of each judgement, but instead of giving up on structural induction, we present a different inductive definition of the same rela-

tion which is more suitable for the problem at hand. Consider an “alternative PTS” relation, $\text{apts}:\text{Cxt}\rightarrow\text{Trm}\rightarrow\text{Trm}\rightarrow\text{Prop}$, that differs from gts only in the right premiss of the Pi rule and the left premiss of the Lda rule:

```
[aPi : {G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{n|VV}{sc:rl s1 s2 s3}
  {l_prem:apts G A (sort s1)}
  {r_prem:{p|PP}{scG:is_ff (Poccur p G)}
    apts (CONS (Gb p A) G) (alpha p n B) (sort s2)}
  (*****
    apts G (pi n A B) (sort s3)];

[aLda : {G|Cxt}{A,M,B|Trm}{s|SS}{n,m|VV}
  {l_prem:{p|PP}{scG:is_ff (Poccur p G)}
    apts (CONS (Gb p A) G) (alpha p n M) (alpha p m B)}
  {r_prem:apts G (pi n A B) (sort s)}
  (*****
    apts G (lda n A M) (pi m A B)];
```

In these premisses we avoid the problem of choosing a particular parameter by requiring the premiss to hold for all parameters for which there is no reason it should not hold, that is, for all “sufficiently fresh” parameters. apts identifies all those derivations of a gts judgement that are inessentially different because of parameters occurring in the derivation but not in its conclusion. Notice that apts constructs well-founded but infinitely branching trees.

It is interesting to compare the side conditions of Pi with those of aPi . In Pi , we intuitively want to know $(\text{is_ff } (\text{Poccur } p \ G))$, but this is derivable from the right premiss so is not required as a side condition. In aPi , we cannot require the right premiss for all p , but only for those such that $(\text{is_ff } (\text{Poccur } p \ G))$, while the condition $(\text{is_ff } (\text{poccur } p \ B))$ is not required because of genericity.

Once we prove that apts and gts have the same derivable judgements, the Thinning Lemma becomes

```
{G,G'|Cxt}{M,A|Trm}(apts G M A)->(subCxt G G')->(Valid G')->apts G' M A;
```

which is straightforward to prove by structural induction on $(\text{apts } G \ M \ A)$.

9.2.1 apts is equivalent to gts

There are two implications to prove; one is straightforward. The interesting one is

```
Goal {G|Cxt}{M,A|Trm}(gts G M A)->apts G M A;
```

This is the essence of our solution to the problem of name clash with the parameters introduced by the right premiss of Pi and the left premiss of Lda . In order to prove it we introduce a concept of “renaming” and prove an induction loaded version of the goal.

Renamings A *renaming* is, informally, a finite function from parameters to parameters. They are represented formally by their graphs as lists of ordered pairs

```
[rp = PROD|PP|PP];
[Renaming = LL|rp];
```

`rho` and `sigma` range over renamings. Renamings are applied to parameters by `assoc`, and extended compositionally to `Trm`, `GB` and `Cxt`.

```
[renPar [rho:Renaming] [p:PP] : PP = assoc (PPEq p) p rho];
[renTrm [rho:Renaming] : Trm->Trm =
  Trec ([_:Trm]Trm)
    ([s:SS]sort s)
    ([v:VV]var v)
    ([p:PP]par (renPar rho p))
    ([v:VV] [_,,l,r:Trm]pi v l r)
    ([v:VV] [_,,l,r:Trm]lda v l r)
    ([_,_,l,r:Trm]app l r)];
[renGB [rho:Renaming] : GB->GB =
  GBrec ([_:GB]GB) ([p:PP][t:Trm](Gb (renPar rho p) (renTrm rho t)))]];
[renCxt [rho:Renaming] : Cxt->Cxt =
  LLrec ([_:Cxt]Cxt)
    (nilCxt)
    ([b:GB][_:Cxt][rest:Cxt]CONS (renGB rho b) rest)];
```

This is a “tricky” representation. First, if there is no pair (p,q) in the list `rho`, $(\text{assoc } (\text{PPEq } p) \text{ p } \text{rho})$ returns `p` (the second occurrence of `p` in this expression), so `renPar rho` is always a total function with finite support. Also, while there is no assumption that renamings are the graphs of functional or injective relations, *application* of a renaming is functional, because `assoc` only finds the first pair whose domain matches a given parameter. Conversely, consing a new pair to the front of a renaming will “shadow” any old pair with the same domain. Interestingly we do not have to formalize these observations.

We prove an induction-loaded form of the goal

```
Goal {G|Cxt}{M,A|Trm}(gts G M A)->
  {rho|Renaming}(Valid (renCxt rho G))->
  apts (renCxt rho G) (renTrm rho M) (renTrm rho A);
```

by induction on the derivation of `gts G M A`. (The desired goal is obtained using the identity renaming.) Reverting again to informal notation, consider the case where the derivation ends with the rule `Pi`

$$\frac{G \vdash A : s_1 \quad G[p:A] \vdash [p/x]B : s_2}{G \vdash \{x:A\}B : s_3} \quad p \notin B, \text{Rule}(s_1, s_2, s_3)$$

We have

left IH For any σ , if σG is Valid, then $\sigma G \vdash_{\text{apts}} \sigma A : s_1$.

right IH For any σ , if $\sigma(G[p:A])$ is Valid, then $\sigma(G[p:A]) \vdash_{\text{apts}} \sigma([p/x]B) : s_2$.

to show For any ρ , if ρG is Valid, then $\rho G \vdash_{\text{apts}} \rho(\{x:A\}B) : s_3$.

So choose ρ with ρG Valid. Using the `aPi` rule we need to show

left subgoal $\rho G \vdash_{\text{apts}} \rho A : s_1$

right subgoal for all q not occurring in ρG , $(\rho G)[q;\rho A] \vdash_{\text{apts}} [q/x]\rho B : s_2$

The left subgoal follows from the left IH.

For the right subgoal, choose a parameter q not occurring in ρG . Let $\sigma = (\text{CONS } (\text{Pr } p \ q) \ \text{rho})$ be the renaming that first changes p to q , then behaves like ρ . Notice $\sigma p = q$, and, because p does not occur in B , A , or G , $\sigma B = \rho B$, $\sigma A = \rho A$, and $\sigma G = \rho G$. Thus $\sigma(G[p:A]) = \rho(G)[q;\rho A]$ is Valid (using the left IH and the **Start** rule), and by the right IH, $(\rho G)[q;\rho A] \vdash_{\text{apts}} [q/x]\rho B : s_2$ as required.

The **aLda** case is similar, and all other cases are straightforward, except the **tConv** case, which requires that renaming preserves conversion. This is proved by similarly using an alternative inductive definition of reduction. ■

Notice that the reason we don't need any assumption that renamings are injective is that the assumption **Valid (renCxt rho G)** guarantees the parts of **rho** actually used are, in fact, injective.

10 Other Results

With the techniques of the last section, the three main results of the elementary theory of PTS are now straightforward.

The Substitution Lemma, a cut property

```
Goal {Gamma,Delta:Cxt}{N,A,M,B:Trm}{p:PP}
  [sub = psub N p][subGB [pA:GB] = Gb (namOf pA) (sub (typOf pA))]
  (gts Gamma N A)->(gts (append Delta (CONS (Gb p A) Gamma)) M B)->
  gts (append (map subGB Delta) Gamma) (sub M) (sub B);
```

Correctness of Types

```
{G|Cxt}{M,A|Trm}(gts G M A)->
  Ex [s:SS]or (is_tt (Trm_eq A (sort s))) (gts G A (sort s));
```

Subject Reduction Theorem, also called closure under reduction.

```
Goal {G|Cxt}{M,A|Trm}(gts G M A)->{M'|Trm}(par_redn M M')->gts G M' A;
```

The proof in [GN91, Bar92] goes by simultaneous induction on

- i If $G \vdash M : A$ and $M \rightarrow M'$ then $G \vdash M' : A$.
- ii If $G \vdash M : A$ and $G \rightarrow G'$ then $G' \vdash M : A$.

where \rightarrow is one-step-reduction, i.e. contraction of one β -redex in a term or context. (The reason for this simultaneous induction is that some of the typing rules move terms from the subject to the context, e.g. **Pi**.) This approach produces a large number of cases, all of which are trivial except for the case where the one redex contracted is the application constructed by the rule **App**. All of these case distinctions are inessential except to isolate the one non-trivial case. Thus, in place of one-step-reduction, we use a new relation of *non-overlapping reduction*, **no_red1**, that differs from **par_red1** only in the β rule

```
[no_red1_beta:{u:VV}{U,A,B:Trm}no_red1 (app (lda u U B) A) (vsub A u B)];
```

Further, the distinction between a redex in the term and a redex in the context is also inessential, so we extend `no_red1` compositionally to contexts, and then to pairs of a context and a term. We suggest that the correct meaning of *subject* of a PTS judgement $G \vdash M : A$ is the pair $\langle G, M \rangle$, and call this non-overlapping reduction on subjects

```
[red1Subj : Cxt->Trm->Cxt->Trm->Prop];
```

Now we prove

```
Goal {G|Cxt}{M,A|Trm}(gts G M A)->
      {G'|Cxt}{M'|Trm}(red1Subj G M G' M')->gts G' M' A;
```

by straightforward induction (with far fewer cases), and the usual Subject Reduction Theorem is an easy corollary.

10.1 Work in Progress

As mentioned in the Introduction, one of our long term goals is a completely verified proofchecker. The formalization reported here has been used to verify typechecking algorithms for subclasses of PTS [Pol92]. We are currently working with van Benthem Jutting on formalizing results leading to more general typechecking algorithms and the strengthening theorem [vBJMP94]. (Typechecking is not enough for real proofcheckers; e.g. LEGO uses strengthening in its `Discharge` tactic.) This project uses the basic theory of several systems similar to PTS, and by “cutting and pasting” existing proofs we have been able to effectively reuse some of the vast amount of work expended so far.

11 Conclusion

In doing this work of formalizing a well known body of mathematics, we spent a large amount of time solving mathematical problems, e.g. the Thinning Lemma. Another big problem was maintaining and organizing the formal knowledge, e.g. allowing two people to extend different parts of the data base at the same time, and finding the right lemma in the mass of checked material. We feel that better understanding of mathematical issues of formalization (e.g. names/namefree, intentional/extentional), and organization of formal development are the most useful areas to work on now for the long-term goal of formal mathematics.

Finally, it is not so easy to understand the relationship between some informal mathematics and a claimed formalization of it. Are you satisfied with our definition of reduction? It might be more satisfying if we also defined de Bruijn terms and their reduction, and proved a correspondence between the two representations, but this only changes the degree of the problem, not its nature. What about the choice between the typing rules `Lda` and `Lda'`? There may be no “right” answer, as we may have different ideas in mind informally. There is no such thing as certain truth, and formalization does not change this state of affairs.

References

- [Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*. Springer-Verlag, LNCS 664, March 1993.

- [Bar91] Henk Barendregt. Introduction to Generalised Type Systems. *J. Functional Programming*, 1(2):125–154, April 1991.
- [Bar92] Henk Barendregt. Lambda calculi with types. In Abramsky, Gabbai, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.
- [Ber90] Stefano Berardi. *Type Dependence and Constructive Mathematics*. PhD thesis, Dipartimento di Informatica, Torino, Italy, 1990.
- [Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
- [GN91] Herman Geuvers and Mark-Jan Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.
- [Hue89] Gérard Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.
- [Jon91] Claire Jones. Completing the rationals and metric spaces in LEGO. In *2nd Workshop of Logical Frameworks, Edinburgh*, pages 209–222, May 1991. Available by ftp.
- [LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User’s manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King’s Buildings, Edinburgh EH9 3JZ, May 1992. Updated version. See <http://www.dcs.ed.ac.uk/packages/lego/>
- [Luo91] Zhaohui Luo. Program specification and data refinement in type theory. In *TAPSOFT ’91 (Volume 1)*, number 493 in Lecture Notes in Computer Science, pages 143–168. Springer-Verlag, 1991.
- [McK92] James McKinna. *Deliverables: a Categorical Approach to Program Development in Type Theory*. PhD thesis, University of Edinburgh, 1992.
- [NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf’s Type Theory. An Introduction*. Oxford University Press, 1990.
- [Pol92] R. Pollack. Typechecking in Pure Type Systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, June 1992. Available by ftp.
- [Sha85] N. Shankar. A mechanical proof of the Church-Rosser theorem. Technical Report 45, Institute for Computing Science, University of Texas at Austin, March 1985.
- [vBJ93] L.S. van Benthem Jutting. Typing in Pure Type Systems. *Information and Computation*, 105(1):30–41, July 1993.
- [vBJMP94] L.S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES’93, Nijmegen, May 1993, Selected Papers*, volume 806 of *LNCS*, pages 19–61. Springer-Verlag, 1994.