Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 199 (2008) 33–47
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Logical Framework with Explicit Conversions

Herman Geuvers and Freek Wiedijk[1]

Department of Computer Science, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract

The type theory λP corresponds to the logical framework LF. In this paper we present λH, a variant of λP where convertibility is not implemented by means of the customary *conversion rule*, but instead type conversions are made explicit in the terms. This means that the time to type check a λH term is proportional to the size of the term itself.
We define an *erasure* map from λH to λP, and show that through this map the type theory λH corresponds exactly to λP: any λH judgment will be erased to a λP judgment, and conversely each λP judgment can be *lifted* to a λH judgment.
We also show a version of subject reduction: if two λH terms are provably convertible then their types are also provably convertible.

Keywords: Logical framework, type theory, λP, λH, convertibility, explicit type conversion

1 Introduction

1.1 Problem

This paper addresses the question of whether a formal proof should be allowed to contain the formal equivalent of the sentence '*this is left as an exercise to the reader.*' Consider the following 'proof':

Lemma. $A(3, 2^{65536} - 3) = A(4, 3)$, where $A(m, n)$ is the Ackermann function.

Proof. Compute the digits of $A(3, 2^{65536} - 3)$ and $A(4, 3)$ using the recursive definition of the Ackermann function (computing these strings of digits is left as an exercise to the reader). These are easily seen to be the same. Therefore the statement is true. □

Do we accept this as a proof of the Lemma? Probably most people would say that, yes, formally it is a proof, but that, no, it is not a good argument for the truth of

[1] Email: {herman,freek}@cs.ru.nl

doi:10.1016/j.entcs.2007.11.011

the statement, as there is no computing device in the known universe that will be able to tell you the digits of those numbers (in fact, those digits probably will not fit in the universe in the first place). But if you press them, they still would agree that, yes, it is a proof.

In a theorem prover based on type theory, like Coq [14], one can simply formalize this proof: define the function A (using higher order recursion) and say that 'reflexivity' proves $A(3, 2^{65536} - 3) = A(4, 3)$. Then the type checker will start computing the values of the expressions on both sides of the equation to see whether these are the same. Of course, this is unfeasible, though theoretically this is a well-formed proof term of this statement indeed.

Now, as a second example, consider *this* 'proof':

> **Lemma.** There are no positive integers a, b, c and n with $n \geq 3$ such that $a^n + b^n = c^n$.
>
> *Proof.* There exists a derivation [2] of this statement with a length less than 2^{65536} symbols (the reader can find it by summing up all possible derivations of length $< 2^{65536}$ symbols and seeing that one of them is a proof of the statement). Therefore the statement is true. □

Now we do not claim that this 'proof' is in any way interesting for its own sake. [3] The question just is: should this at all be accepted as a – like in the previous example, very inefficient – proof of Fermat's last theorem? We *do* know that a derivation of the statement exists.

In a type theoretical proof assistant like Coq, one can actually formalize this 'proof', by defining a proof search algorithm inside Coq. Using *reflection* we encode our favorite logic (first order logic, higher order logic or set theory) inside Coq and we write an algorithm (a typed λ-term) that searches all derivations of length less than 2^{65536} symbols for a proof of Fermat's theorem. This is a well-formed proof term and its type is the statement of Fermat's theorem. Of course, the proof assistant will not finish type-checking that term before the end of the universe. Again this is a proof *by computation left to the reader*.

The philosophical question is: is this second 'proof' again at least formally a proof, just like in the previous example? [4]

Finally, consider a third example:

> **Lemma.** The non-trivial zeroes of Riemann's $\zeta(s)$ function all lie on the complex line $\Re s = \frac{1}{2}$.
>
> *Proof.* There exists a derivation of this statement with a length less than 2^{65536} symbols (finding it is left as an exercise to the reader). Therefore the statement

[2] The formal system in which this derivation is constructed should of course be powerful enough to prove Fermat's last theorem, but the specific choice of formal system does not really matter.

[3] One should consider this example to be similar to 'Schrödingers cat' in physics. No one actually will kill cats to discover something about quantum mechanics. Similarly, no one will gain any knowledge of number theory from this example. It should just be considered to clarify the notion of what a proof is.

[4] If you say that this is *not* a real proof of the statement (instead of just a very inefficient one), then things become interesting. Proof terms that involve reflection can run 'searches' by doing reduction. They can do 'reflective proof search' to solve small subproblems, which is considered as very useful and common in type theory. But this is exactly the same kind of proof search that is happening here on a large scale.

is true. □

Suppose you *did* agree that our second example contained a valid but *very* inefficient proof, and suppose that at some point the Riemann hypothesis is proved. Are we then allowed to say that 'oh, but we already have published a proof of the Riemann hypothesis before' (pointing to the paper that you are currently reading)? [5]

At the TYPES meeting in Kloster Irsee in 1998, there was an interesting discussion after the talk by Barendregt, where he had explained and advocated how to use the $\beta\delta\iota$-reduction of type theory to add automation to Coq by letting it do calculations during its type check phase. This uses the technique of *reflection* (see e.g. [4,11] for examples and a discussion), where part of the object language is reflected in itself to make computations and reasoning on the meta-level possible within the system.

Most people clearly considered this way of using Coq's convertibility check to be a *feature*. The only dissenting voice came from Martin-Löf, who did not like it at all and seemed to consider this to be a *bug*. We do not have an overview of todays opinions, but the community seems to be quite unanimous that this kind of 'automatic calculations by type checking' is a good thing.

In theorem provers based on type theory the main performance bottleneck is the convertibility check: if the calculated type of a term M is A, but it is used in a context where the type should be B, then the system needs to verify that $A =_{\beta\iota\delta} B$, where δ is the equality arising from definitional expansion (unfolding definitions) and ι is the equality arising from functions defined by (higher order primitive) recursion. In fact, the inefficiency of the convertibility check means that type correctness is in practice only semi-decidable. Although in theory it is decidable whether a term M has type A, in practice when it is not correct the system could be endlessly reducing and would not terminate in an acceptable time anymore. In the Coq proof assistant [14], this problem is less noticeable because there many definitions are 'opaque', which means that they cannot be unfolded. This causes the equality check to fail and the user first has to make the definition transparent and unfold it. The Automath system [9] from the seventies just gave up after having failed to establish convertibility after some given number of reduction steps. Type theoretic theorem provers apparently *search* for a 'convertibility proof'. This proof would have to be rediscovered every time the terms would be type checked, and it would not be stored in a 'convertibility proof term'.

The HOL system [5,7] does not have a conversion rule. In HOL, β-reduction is not automatically tried by the system, but is one of the derivation rules of the logic. Similarly δ- and ι-reductions are performed using the rules of the logic. If one considers a HOL 'proof term' that stores the HOL rules that have been used to obtain a certain theorem [3], then this proof term somehow documents the 'reduction information' that is not available in a proof term from the type theoretical/LF

[5] Note that the last two 'proofs' really just 'are' the bound 2^{65536} (it is the only interesting thing that these 'proofs' contain). Can we consider a number to be a proof? Somehow it does not seem to contain enough structure.

world.[6]

The goal of this paper is to investigate whether it is possible to have a system close to the systems from type theory, but in which the convertibility of types *is* explicitly stored in the proof terms (like it is done in HOL). The advantages of such a system are the following.

- In such a system the type checker will not need to do a convertibility check on its own. Instead the term will contain the information needed to establish the convertibility.
- Because of this, type checking a term will be cheap. If we consider the substitution operation and term identity checking to take unit time, the time to type check a term will be *linear* in the size of the term.
- In such a system the type of a term will be *unique*, instead of only being unique *up to conversion.*[7]
- When rechecking a term, there is no need to recheck (possibly complex) equalities, because the equality check is in the proof term.
- To check an equality, a reduction to normal form may be infeasible, like in $A(3, 2^{65536} - 3) = A(4, 3)$. But we may have another 'smarter' way to establish this equality which we can store in a proof term of feasible size.

In the system that we describe in this paper, checking a proof matches much more the image of 'following the proof with your little finger, and checking locally that everything is correct' than is the case with the standard type theoretical proof systems.

1.2 Approach

We define a system called λH[8]. The system is very close to λP, the pure type system that corresponds to the logical framework LF [6] (currently implemented in the Twelf system [12]). The main difference between λH and λP is that in the first the conversion rule has been made explicit, leading to a linear time type synthesis algorithm. Note that type synthesis in λP is not elementary recursive, as equality in simple typed λ-calculus is not (due to Statman [13]) and we can encode any equality problem between simple typed terms as a type synthesis problem in λP.

In λH conversions are made explicit in the terms. If H is a term that shows that A is convertible to A', which we will write as

$$\vdash H : A = A'$$

and if the term a has type A, then the term a^H (the conversion H applied to a) will have type A'.

[6] Of course, as the HOL logic does not have dependent types, this kind of reduction is much less important in the first place.

[7] So in such a system the type of a term will be $(\lambda x{:}A.B)\,a$, or it will be $B[x := a]$, but not *both*.

[8] The 'H' in the name of the system reflects the letter that we use for the convertibility proof terms. So λH is 'the logical framework with Hs', i.e., with convertibility proofs.

Note that in our system we have explicit 'equality judgments' just like in Martin-Löf style type theory [10]. However there is a significant difference. In Martin-Löf style type theory there are no terms that prove equalities. The equality judgments in such theories look like:

$$\vdash A = A' : B$$

and the equality is on the *left* of the colon. In contrast, in our system two terms that are provably equal do not need to have the same computed type, so there will not be a common type to the right of the colon. Instead we will have a proof term, and so our equality will be to the *right* of the colon.

The fact that two terms in our system that occur in an equality judgment do not need to have computed types that are syntactically equal, means that our judgmental equality is a version of *John-Major equality* [8]. John-Major equality is an inductively defined equality, parameterized by two types: $M =_{A,B} N : *$ if $M : A$ and $N : B$. It has only one constructor, $\mathsf{refl}_{A,M} : M =_{A,A} M$. So we can *write down* any equality between two terms, but an equality is provable only in case both terms have the same type.

1.3 Related Work

Robin Adams has presented a version of pure type systems that have judgmental equalities in the style of Martin-Löf type theories [1]. However, he does not have terms in his system that represent the derivation of the equality judgments. Also, he does not represent the conversions themselves in the terms. Therefore in his system more terms are syntactically identical than in our system. Another difference is that he develops his system for all functional pure type systems, while we only have a system that corresponds to λP.

1.4 Contribution

We define a system λH in which type conversion is represented in the proof term. We show that this system corresponds exactly to the pure type system λP. We also show that this system has a property closely related to subject reduction.

The λH system is quite a bit more complicated than the λP system. It has 13 instead of 4 term constructors, and 15 instead of 7 derivation rules.

1.5 Outline

In Section 2 we define our system. In Section 3 we show that it corresponds to the λP system. In Section 4 we show that an analog of the subject reduction property of λP holds for our system. In Section 5 we define a weak reduction relation for our equality proof terms that is confluent and strongly normalizing and that satisfies subject reduction. Finally, in Section 7 we present a slight modification of our system, where we do not allow conversions to go through the ill-typed terms. Such a system corresponds more closely to a semantical view upon type theory.

2 The system λH

Definition 2.1 The λH expressions are given by the following grammar (the syntactic category $\mathcal{V}$ are the variable names):

$$\mathcal{T} ::= \Box \mid * \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \mathcal{T}\mathcal{T} \mid \mathcal{T}^{\mathcal{E}}$$
$$\mathcal{E} ::= \bar{\mathcal{T}} \mid \mathcal{E}^{\dagger} \mid \mathcal{E}\cdot\mathcal{E} \mid \beta(\mathcal{T}) \mid \iota(\mathcal{T}) \mid \{\mathcal{E}, [\mathcal{V}]\mathcal{E}\} \mid \langle\mathcal{E}, [\mathcal{V}]\mathcal{E}\rangle \mid \mathcal{E}\mathcal{E}$$

$$\mathcal{C} ::= \ \mid \mathcal{C}, \mathcal{V} : \mathcal{T}$$
$$\mathcal{J} ::= \mathcal{C} \vdash \mathcal{T} : \mathcal{T} \ \mid \ \mathcal{E} : \mathcal{T} = \mathcal{T}$$

The $\mathcal{T}$ are the terms of the system, the $\mathcal{E}$ are convertibility proofs, the $\mathcal{C}$ are the contexts, and the $\mathcal{J}$ are the judgments. The *sorts* are the special cases of $\mathcal{T}$ that are the elements of $\{\Box, *\}$.

Definition 2.2 We define the *erasure* operation recursively by:

$$\begin{aligned}
|x| &\equiv x\\
|\Box| &\equiv \Box\\
|*| &\equiv *\\
|\Pi x{:}A.B| &\equiv \Pi x{:}|A|.|B|\\
|\lambda x{:}A.b| &\equiv \lambda x{:}|A|.|b|\\
|Fa| &\equiv |F||a|\\
|a^H| &\equiv |a|
\end{aligned}$$

It maps λH terms to λP terms and is extended straightforwardly to contexts.

There are two kinds of judgments in λH: *equality judgments* and *typing judgments*. The first are of the form $H : a = b$, where H codes a proof of convertibility (through not necessarily well-typed terms) of a and b. The rules for equality judgments are independent of typing judgments. In the rules for the typing judgments, equality judgments appear as a side-condition (in the rule for conversion).

Definition 2.3 The rules that inductively generate the λH judgments are the following (in these rules s only ranges over sorts):

definitional equality

$$\frac{}{\bar{A} : A = A}$$

$$\frac{H : A = A'}{H^{\dagger} : A' = A}$$

$$\frac{H : A = A' \quad H' : A' = A''}{H \cdot H' : A = A''}$$

β-redex

$$\frac{}{\beta((\lambda x{:}A.b)\,a) : (\lambda x{:}A.b)\,a = b[x := a]}$$

erasing equality proofs

$$\frac{}{\iota(a) : a = |a|}$$

congruence rules

$$\frac{H : A = A' \quad H' : B = B'}{\{H, [x]H'\} : \Pi x{:}A.B = \Pi x{:}A'.B'}$$

$$\frac{H : A = A' \quad H' : B = B'}{\langle H, [x]H'\rangle : \lambda x{:}A.B = \lambda x{:}A'.B'}$$

$$\frac{H : F = F' \quad H' : a = a'}{HH' : Fa = F'a'}$$

start & weakening

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash b : B}{\Gamma, x : A \vdash b : B}$$

box & star

$$\vdash * : \Box$$

typed lambda terms

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x{:}A.B : s}$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash b : B : s}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash F : \Pi x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$$

conversion

$$\frac{\Gamma \vdash a : A \quad H : A = A'}{\Gamma \vdash a^H : A'}$$

We write $\Gamma \vdash A : B : C$ as an abbreviation of $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$. We write $A =_{\lambda H} A'$ if we have that $H : A = A'$ for some H. We write 'A is type correct in context Γ' if we have that $\Gamma \vdash A : B$ for some B. We write 'A is type correct' if it is type correct in some context. We write 'Γ is well-formed' if some derivable judgment has Γ as the context. Finally we will write derivability in λH as $\vdash_{\lambda H}$ to distinguish it from derivability in λP which is written $\vdash_{\lambda P}$. (If we omit the subscript, it will be apparent from the context which system is meant.)

The following lemmas about λH are immediate:

Lemma 2.4 *Any subterm of a type correct term is type correct (in the appropriate context).*

Lemma 2.5 *If* $\Gamma \vdash A : B : C$ *then* C *is a sort.*

Lemma 2.6 *If* $\Gamma \vdash a : A$ *then either* $\Gamma \vdash A : s$ *for some sort* s*, or* $A \equiv \Box$.

Lemma 2.7 (uniqueness of types) *If* $\Gamma \vdash a : A$ *and* $\Gamma \vdash a : A'$ *then* $A \equiv A'$.

We now show that typing is in linear time by defining a type checking algorithm.

Definition 2.8 Define the function $\mathsf{type} : \mathcal{C} \times \mathcal{T} \to \mathcal{T} \cup \{\mathsf{false}\}$ simultaneously with the functions $\mathsf{wf} : \mathcal{C} \to \{\mathsf{true}, \mathsf{false}\}$ and $\mathsf{comp} : \mathcal{E} \times \mathcal{T} \to \mathcal{T} \cup \{\mathsf{false}\}$ as follows.

$$\begin{aligned}
\mathsf{type}_\Gamma(*) &= \mathsf{if\ wf}(\Gamma)\ \mathsf{then}\ \Box\ \mathsf{else\ false}\\
\mathsf{type}_\Gamma(\Box) &= \mathsf{false}\\
\mathsf{type}_\Gamma(x) &= \mathsf{if\ wf}(\Gamma) \wedge (x{:}A) \in \Gamma\ \mathsf{then}\ A\ \mathsf{else\ false}\\
\mathsf{type}_\Gamma(\Pi x{:}A.B) &= \mathsf{if\ type}_\Gamma(A) \equiv * \wedge \mathsf{type}_{\Gamma,x:A}(B) \in \{*, \Box\}\\
&\quad\ \mathsf{then\ type}_{\Gamma,x:A}(B)\ \mathsf{else\ false}\\
\mathsf{type}_\Gamma(\lambda x{:}A.M) &= \mathsf{if\ type}_\Gamma(A) \equiv * \wedge \mathsf{type}_{\Gamma,x:A}(M) \neq \Box\\
&\quad\ \mathsf{then}\ \Pi x{:}A.\mathsf{type}_{\Gamma,x:A}(M)\ \mathsf{else\ false}\\
\mathsf{type}_\Gamma(MN) &= \mathsf{if\ type}_\Gamma(M) \equiv \Pi x{:}\mathsf{type}_\Gamma(N).B\\
&\quad\ \mathsf{then}\ B[x := N]\ \mathsf{else\ false}\\
\mathsf{type}_\Gamma(M^H) &= \mathsf{if\ type}_\Gamma(M) \equiv A\ \mathsf{then\ comp}(H, A)\ \mathsf{else\ false}
\end{aligned}$$

$$\begin{aligned}
\mathsf{wf}(\langle - \rangle) &= \mathsf{true}\\
\mathsf{wf}(\Gamma, x{:}A) &= \mathsf{type}_\Gamma(A) \in \{*, \Box\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{comp}(\bar{A}, B) &= \mathsf{if}\ A \equiv B\ \mathsf{then}\ B\ \mathsf{else\ false}\\
\mathsf{comp}(H^\dagger, B) &= \mathsf{comp}^{-1}(H, B)\\
\mathsf{comp}(H \cdot H', B) &= \mathsf{comp}(H', \mathsf{comp}(H, B))\\
\mathsf{comp}(\iota(A), B) &= \mathsf{if}\ A \equiv B\ \mathsf{then}\ |A|\ \mathsf{else\ false}\\
\mathsf{comp}(\beta((\lambda x{:}A.M)N), B) &= \mathsf{if}\ B \equiv (\lambda x{:}A.M)N\ \mathsf{then}\ M[x := N]\ \mathsf{else\ false}
\end{aligned}$$

$$\begin{aligned}
\mathsf{comp}(\{H,[x]H'\},B) = {} & \mathsf{if}\ B \equiv \Pi y{:}A.C \\
& \mathsf{then}\ \Pi x{:}\mathsf{comp}(H,A).\mathsf{comp}(H',C[y:=x]) \\
& \mathsf{else\ false} \\
\mathsf{comp}(\langle H,[x]H'\rangle,B) = {} & \mathsf{if}\ B \equiv \lambda y{:}A.C \\
& \mathsf{then}\ \lambda x{:}\mathsf{comp}(H,A).\mathsf{comp}(H',C[y:=x]) \\
& \mathsf{else\ false} \\
\mathsf{comp}(HH',B) = {} & \mathsf{if}\ B \equiv AC \\
& \mathsf{then\ comp}(H,A)\mathsf{comp}(H',C)\ \mathsf{else\ false}
\end{aligned}$$

The function comp^{-1} is defined totally similar to comp, with two exceptions:

$$\begin{aligned}
\mathsf{comp}^{-1}(\iota(A),B) = {} & \mathsf{if}\ |A| \equiv B\ \mathsf{then}\ A\ \mathsf{else\ false} \\
\mathsf{comp}^{-1}(\beta((\lambda x{:}A.M)N),B) = {} & \mathsf{if}\ B \equiv M[x:=N]\ \mathsf{then}\ (\lambda x{:}A.M)N\ \mathsf{else\ false}
\end{aligned}$$

Proposition 2.9 (type checking) *$\Gamma \vdash_{\lambda H} a : A$ if and only if $\mathsf{type}(\Gamma,a) \equiv A$. Moreover, the time for* type *to compute an answer can be made linear in the length of the inputs.*

Proof. One first proves the fact that, $H : B = C$ if and only if $\mathsf{comp}(H,B) = C$. Then $\Gamma \vdash_{\lambda H} a : A$ implies $\mathsf{type}(\Gamma,a) \equiv A$ is proved by induction on the derivation, simultaneously with 'Γ is well formed' implies $\mathsf{wf}(\Gamma) = \mathsf{true}$. The other way around, one proves simultaneously that $\mathsf{type}(\Gamma,a) \equiv A$ implies $\Gamma \vdash_{\lambda H} a : A$ and that $\mathsf{wf}(\Gamma) = \mathsf{true}$ implies 'Γ is well formed' (by induction over the length of the input: $\mathrm{lth}(\Gamma,a)$, resp. $\mathrm{lth}(\Gamma)$).
The time for computing $\mathsf{comp}(H,A)$ is clearly linear in the size of the equational proof term H. The function type as defined above is not linear in its inputs. To make sure that type computes a type in linear time, one has to collect the 'side conditions' $\mathsf{wf}(\Gamma)$ properly to avoid checking the well-foundedness of the (local) context for every variable separately. This is quite folklore, e.g. see [2], so for matters of exposition we don't give the details here.

3 Correspondence to λP

Lemma 3.1 *If $A =_{\lambda H} A'$ then $|A| =_\beta |A'|$.*

Proof. By induction on the derivation of $A =_{\lambda H} A'$.

Proposition 3.2 ('from λH to λP') *If $\Gamma \vdash_{\lambda H} a : A$ then $|\Gamma| \vdash_{\lambda P} |a| : |A|$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda H} a : A$, using the previous Lemma in the conversion rule.

Lemma 3.3 *For $A, A' \in \mathcal{T}$,*

(i) *if $|A| \equiv |A'|$, then $A =_{\lambda H} A'$,*

(ii) *if $|A| =_\beta |A'|$, then $A =_{\lambda H} A'$.*

Proof.

(i) If $|A| \equiv |A'|$, then $\iota(A) \cdot \overline{\iota A'} : A = A'$.

(ii) If $|A| =_\beta |A'|$, we first prove that $|A| =_{\lambda H} |A'|$, by induction on the proof (in the equational theory of the λ-calculus) of $|A| =_\beta |A'|$. Then we conclude by using that $\iota(A) : A = |A|$. We do some cases

- $A \equiv \Pi x{:}B.C$ and $A' \equiv \Pi x{:}B'.C'$ and $|\Pi x{:}B.C| =_\beta |\Pi x{:}B'.C'|$ was derived from $|B| =_\beta |B'|$ and $|C| =_\beta |C'|$. By IH, $H_0 : |B| = |B'|$ and $H_1 : |C| = |C'|$ for some H_0, H_1, so $\{H_0, [x]H_1\} : |\Pi x{:}B.C| = |\Pi x{:}B'.C'|$.
- $A \equiv (\lambda x{:}B.M)P$, $A' \equiv M'[P'/x]$ with $|(\lambda x{:}B.M)P| \to_\beta |M'[P'/x]|$. Then $\beta((\lambda x{:}B.M)P) : (\lambda x{:}B.M)P = M[P/x]$ and we are done by two applications of (i) (using $|M[P/x]| \equiv |M'[P'/x]|$).

Proposition 3.4 ('from λP to λH') *Let Γ be a λP-context and a, A be λP-terms such that $\Gamma \vdash_{\lambda P} a : A$. Then the following two properties hold.*

(i) *There is a correct λH-context Γ' such that $|\Gamma'| \equiv \Gamma$.*

(ii) *For all λH-contexts Γ' for which $|\Gamma'| \equiv \Gamma$, there are λH-terms a', A' such that $\Gamma' \vdash_{\lambda H} a' : A'$ and $|a'| \equiv a$, $|A'| \equiv A$.*

Proof. Simultaneously by induction on the λP derivation, distinguishing cases according to the last applied rule. We treat four cases and abbreviate 'induction hypothesis' to IH.

- (application)

$$\frac{\Gamma \vdash_{\lambda P} F : \Pi x{:}A.B \quad \Gamma \vdash_{\lambda P} a : A}{\Gamma \vdash_{\lambda P} Fa : B[x := a]}$$

The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. Furthermore, for Γ' any λH-context such that $|\Gamma'| \equiv \Gamma$, by IH, there are F', a', A', A'' and B' such that $\Gamma' \vdash_{\lambda H} F' : \Pi x{:}A'.B'$, $\Gamma' \vdash_{\lambda H} a' : A''$ and $|F'| \equiv F$, $|a'| \equiv a$, $|A'| \equiv |A''| \equiv A$ and $|B'| \equiv B$. By Lemma 3.3, we have $H : A'' = A'$ for some H, so $\Gamma' \vdash_{\lambda H} a'^H : A'$ and $\Gamma' \vdash Fa'^H : B'[a'^H/x]$. We are done, because $|Fa'^H| \equiv Fa$ and $|B'[a'^H/x]| \equiv B[a/x]$.

- (λ)

$$\frac{\Gamma, x{:}A \vdash_{\lambda P} M : B \quad \Gamma \vdash_{\lambda P} A : \star}{\Gamma \vdash_{\lambda P} \lambda x{:}A.M : \Pi x{:}A.B}$$

The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. Furthermore, for Γ' any λH-context such that $|\Gamma'| \equiv \Gamma$, by IH, there is an A' such that $\Gamma' \vdash_{\lambda H} A' : \star$ and $|A'| \equiv A$. So $\Gamma', x{:}A'$ is a correct λH-context. So, by IH there are M' and B' such that $\Gamma', x{:}A' \vdash_{\lambda H} M' : B'$ and $|M'| \equiv M$ and $|B'| \equiv B$. Hence, $\Gamma' \vdash \lambda x{:}A'.M' : \Pi x{:}A'.B'$ and we are done, because $|\lambda x{:}A'.M'| \equiv \lambda x{:}A.M$ and $|\Pi x{:}a'.B'| \equiv \Pi x{:}A.B$.

- (conversion)

$$\frac{\Gamma \vdash_{\lambda P} M : A \quad \Gamma \vdash_{\lambda P} B : s \quad A =_\beta B}{\Gamma \vdash_{\lambda P} M^H : B}$$

The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. Furthermore, for Γ' any λH-context such that $|\Gamma'| \equiv \Gamma$, by IH, there is are M', A', B' such that $\Gamma' \vdash_{\lambda H} M' : A'$, $\Gamma \vdash_{\lambda H} B' : s$ and $|A'| \equiv A$, $|B'| \equiv B$, $|M'| \equiv M$. So

$|A'| \equiv A =_\beta B \equiv |B'|$ and by Lemma 3.3, $H : A' = B'$ for some H. Now, $\Gamma' \vdash_{\lambda H} M' : B'$ by the conversion rule in λH and we are done.

- (weakening)

$$\frac{\Gamma \vdash_{\lambda P} A : \star \quad \Gamma \vdash_{\lambda P} M : B}{\Gamma, x{:}A \vdash_{\lambda P} M : B}$$

The IH states that there is an λH-context Γ' such that $|\Gamma'| \equiv \Gamma$. By IH, there is an A' such that $\Gamma' \vdash_{\lambda H} A' : \star$ and $|A'| \equiv A$. So $\Gamma', x{:}A'$ is a correct λH-context, proving part (1). Now, for any λH context $\Gamma', x{:}A'$ such that $|\Gamma', x{:}A'| \equiv \Gamma, x{:}A$, we know that $|\Gamma'| \equiv \Gamma$, so, by IH there are M' and B' such that $\Gamma' \vdash_{\lambda H} M' : B'$ and $|M'| \equiv M$ and $|B'| \equiv B$. As $\Gamma', x{:}A'$ is correct, we can weaken this to obtain $\Gamma', x{:}A' \vdash_{\lambda H} M' : B'$ and we are done.

Corollary 3.5 (conservativity of λP over λH) *Given a well-formed λH context Γ and λH type A in Γ,*

$$|\Gamma| \vdash_{\lambda P} M : |A| \Rightarrow \exists M'(\Gamma \vdash_{\lambda H} M' : A \wedge |M'| \equiv M)$$

Proof. The second part of the Proposition ensures that there are N and B such that $\Gamma \vdash_{\lambda H} N : B$ and $|N| \equiv M$ and $|B| \equiv |A|$. Then $B =_{\lambda H} A$, due to Lemma 3.3, say $H : B = A$. Then $\Gamma \vdash_{\lambda H} N^H : A$.

4 An analogue of subject reduction

The following proposition is the equivalent for λH of the subject reduction property of λP. The system λH does not have a notion of β-reduction, so the statement $a \rightarrow_\beta a'$ in the condition of the statement is replaced by $a =_{\lambda H} a'$. Also, we do not get that the type is conserved, it just is conserved up to convertibility (so if $a = a'$ and $a : A$ then we will not always get that $a' : A$, but just that $a' : A'$ for some A' with $A = A'$.)

Proposition 4.1 ('subject reduction') *If $\Gamma \vdash_{\lambda H} a : A : s$ and $\Gamma \vdash_{\lambda H} a' : A' : s'$ and $a =_{\lambda H} a'$ then $A =_{\lambda H} A'$ and $s \equiv s'$.*

Proof. From Proposition 3.2 we get that $|\Gamma| \vdash_{\lambda P} |a| : |A| : s$ and $|\Gamma| \vdash_{\lambda P} |a'| : |A'| : s'$ and $|a| =_\beta |a'|$. By subject reduction of λP and uniqueness of types in λP we get that $|A| =_\beta |A'|$ and $s \equiv s'$. From Lemma 3.3 we finally get that $A =_{\lambda H} A'$.

5 Conversion reduction

Definition 5.1 We define the *conversion reduction* relation $\twoheadrightarrow$ as the rewrite relation of the following rewrite rules:

$$A^{\overline{A'}} \rightarrow A$$
$$A^{H \cdot H'} \rightarrow (A^H)^{H'}$$
$$\bar{A}^\dagger \rightarrow \bar{A}$$

$$H^{\dagger\dagger} \to H$$
$$(H \cdot H')^{\dagger} \to H'^{\dagger} \cdot H^{\dagger}$$

We will now list some simple properties of conversion reduction (with some proofs omitted for space reasons):

Proposition 5.2 *Conversion reduction is confluent.*

Proposition 5.3 *Conversion reduction is strongly normalizing.*

(These two propositions even hold for terms that are not type correct.)

Proposition 5.4 (subject reduction for conversion reduction)
If $\Gamma \vdash_{\lambda H} a : A$ *and* $a \twoheadrightarrow a'$ *then* $\Gamma \vdash_{\lambda H} a' : A$.

Proof. By induction on the derivation of $\Gamma \vdash_{\lambda H} a : A$ one proves that, if $a \to a'$, then $\Gamma \vdash_{\lambda H} a' : A$, distinguishing cases according to the applied reduction step. (The $a \twoheadrightarrow a'$ case then follows immediately.)

Although this proposition is a subject reduction property, it is *not* related to the subject reduction property of λP, as it does not involve β-reduction.

Proposition 5.5 *If* $\Gamma \vdash_{\lambda H} a : A$ *and* $a \twoheadrightarrow a'$ *then* $a =_{\lambda H} a'$.

Proof. If $a \twoheadrightarrow a'$, then $|a| \equiv |a'|$ and hence $a =_{\lambda H} a'$ by Lemma 3.3.

Proposition 5.6 *A term that is in conversion reduction normal form does not contain the operations* $\bar{A}$ *and* $H \cdot H'$*, and it only contains the operation* $H^{\dagger}$ *in the combinations* $\beta(\ldots)^{\dagger}$ *and* $\iota(\ldots)^{\dagger}$.

This last proposition shows that we can do away with the $\bar{A}$ and $H \cdot H'$ operations in our system.

6 Discussion

Imagine a formalization of the 'proofs' of the Lemmas from Section 1.1 in Coq. This is doable and we obtain a well-formed proof-term. However, type checking each of these proofs is completely infeasible.

Now imagine a version of Coq that is built on top of the logical framework λH. When type checking a Coq 'proof' the system would need to store the reduction information in the explicit conversions in the λH proof term that it would build internally. Therefore that proof term would be impractically big. So in such a system the proof would not be considered to be a *real* proof, as the underlying λH proof object would be impossible to construct.

For this reason λH adequately represents both our unease with our 'proofs' of the Lemmas, as well as Martin-Löf's unease with Barendregt's talk in Kloster Irsee.

Note that this formalization of the 'proofs' of the Lemmas needs ι-reduction, so it is not possible in LF itself. Therefore for our argument one needs to imagine a version of Coq's type theory that has explicit conversions: a system that relates to the calculus of inductive constructions CiC, in the same way that the system λH relates to λP. We don't see any principal problem with extending our system λH to CiC.

7 Future work

An interesting thing to do now is to implement λH as the basis of an actual proof assistant, to see whether it is a practical system for doing actual proof checking. Part of such a system might be a term *lifter* that lifts proof terms from λP to λH, inserting the conversions that were needed to make the terms type check.

Another issue is whether it is possible to build such a system in a way that the bulk of the proof terms will not actually be stored in memory, but checked and discarded while it is being generated. This is the way that HOL checks its proofs. Barendregt calls this *ephemeral proofs.* So the question is whether it will be possible to have a practical λH implementation with ephemeral proof terms.

7.1 Avoiding ill-typed terms

In the system λH, we have avoided the conversion rule by introducing proof terms that witness an equality (and that can be checked in linear time). But the conversion goes through $\mathcal{T}$, the set of 'pseudo-terms'. This is in line with most implementations of proof checkers, where equality checking is done by a separate algorithm that does not take typing into account. But what if we restrict equalities to conversions that pass through the well-typed terms only? This is more in line with a semantical intuition, where the ill-typed terms just do not exist. We can adapt the syntax of λH to cover this situation and we put the question whether this system is equivalent to λH. We call this new system λF.[9]

The system λF has the same terms and equality proofs as λH, but the judgments are different:

$$\mathcal{J} ::= \mathcal{C} \vdash \mathcal{T} : \mathcal{T} \mid \mathcal{C} \vdash \mathcal{E} : \mathcal{T} = \mathcal{T}$$

So an equality in λF is always stated and proved *within* a context, in which the terms are well-typed. The rules that inductively generate the λF judgments are the same as for λH, apart from the rules that involve equalities, which are as follows (in these rules s only ranges over sorts):

definitional equality

$$\frac{\Gamma \vdash A : B}{\Gamma \vdash \bar{A} : A = A}$$

$$\frac{\Gamma \vdash H : A = A'}{\Gamma \vdash H^{\dagger} : A' = A}$$

[9] The 'F' stands for 'fully well-typed'.

$$\frac{\Gamma \vdash H : A = A' \quad \Gamma \vdash H' : A' = A''}{\Gamma \vdash H \cdot H' : A = A''}$$

β-redex

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash b : B : s \quad \Gamma \vdash a : A}{\Gamma \vdash \beta((\lambda x{:}A.b)\, a) : (\lambda x{:}A.b)\, a = b[x := a]}$$

conversion

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash H : A = A'}{\Gamma \vdash a^H : A'}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash H : A = A'}{\Gamma \vdash \iota(a^H) : a = a^H}$$

congruence rules

$$\frac{\begin{array}{c}\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s \\ \Gamma \vdash A' : * \quad \Gamma, x' : A' \vdash B' : s \\ \Gamma \vdash H : A = A' \quad \Gamma, x : A \vdash H' : B = B'[x' := x^H]\end{array}}{\Gamma \vdash \{H, [x{:}A]H'\} : \Pi x{:}A.B = \Pi x'{:}A'.B'}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \quad \Gamma, x : A \vdash b : B : s \\ \Gamma \vdash A' : * \quad \Gamma, x : A' \vdash b' : B' : s \\ \Gamma \vdash H : A = A' \quad \Gamma, x : A \vdash H' : b = b'[x' := x^H]\end{array}}{\Gamma \vdash \langle H, [x{:}A]H' \rangle : \lambda x{:}A.b = \lambda x{:}A'.b'}$$

$$\frac{\begin{array}{c}\Gamma \vdash F : \Pi x{:}A.B \quad \Gamma \vdash a : A \\ \Gamma \vdash F' : \Pi x'{:}A'.B' \quad \Gamma \vdash a' : A' \\ \Gamma \vdash H : F = F' \quad \Gamma \vdash H' : a = a'\end{array}}{\Gamma \vdash HH' : Fa = F'a'}$$

(Note that the $\iota(\ldots)$ of λF just removes one conversion, in contrast to the $\iota(\ldots)$ of λH which removes all conversions at once. Removing all conversions generally leads to a term that is not well-typed, so that is not an option for λF where all terms have to be well-typed, even in the conversion proofs.)

Acknowledgement

Thanks to Thorsten Altenkirch for the suggestion to use a John-Major-like equality in our system. Thanks to the anonymous referees for valuable comments on a first version of this paper.

References

[1] R. Adams. Pure Type Systems with Judgemental Equality. *Journal of Functional Programming* 16 (2): 219-246, 2006.

[2] L.S. van Benthem Jutting, J. McKinna and R. Pollack, Checking Algorithms for Pure Type Systems, *Proceedings of the International Workshop on Types for Proofs and Programs*, eds. H. Barendregt, T. Nipkow, pp. 19–61, Springer LNCS 806, 1994.

[3] S. Berghofer. New features of the Isabelle theorem prover – proof terms and code generation, 2000. http://www4.in.tum.de/~berghofe/papers/TYPES2000_slides.ps.gz

[4] H. Geuvers, F. Wiedijk and J. Zwanenburg, Equational Reasoning via Partial Reflection, in *Theorem Proving for Higher Order Logics*, TPHOL 2000, Portland OR, USA, eds. M. Aagaard and J. Harrison, LNCS 1869, pp. 162–178.

[5] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.

[6] R. Harper, F. Honsell and G. Plotkin, A framework for defining logics, in *Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1987, pp. 194–204.

[7] J. Harrison. *The HOL Light manual (1.1)*, 2000. http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz

[8] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. http://www.dur.ac.uk/c.t.mcbride/thesis.ps.gz

[9] R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, Amsterdam, 1994.

[10] B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's Type Theory, An Introduction*. Oxford University Press, 1990. http://www.cs.chalmers.se/Cs/Research/Logic/book/book.ps

[11] M. Oostdijk and H. Geuvers, Proof by Computation in the Coq system, *Theoretical Computer Science* 272 (1-2), 2001, pp. 293–314.

[12] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems, in *Proceedings of the 16th International Conference on Automated Deduction* (CADE-16), ed. H. Ganzinger, LNAI 1632, 1999, pp. 202–206.

[13] R. Statman. The typed lambda-calculus is not elementary recursive, *TCS* 9(1), pp. 73–81, July 1979.

[14] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2002. ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual-all.ps.gz