Proof Irrelevance in a Logical Framework
Extended Abstract

Jason Reed

April 6, 2002

1 Preliminaries

1.1 Logic and Representation

There is a close relationship between type systems and logics known as the Curry-Howard correspondence. In its most basic form, it comes very naturally from an (intuitionistic) understanding of what it means to be a proof of a complex proposition. If we know what counts as a proof of a proposition A, and what counts as a proof of B, then a proof of the proposition $A \wedge B$ ("A and B") ought to be a sort of package containing evidence for A on one side, and evidence for B on the other. Similarly, we might expect a proof of the proposition $A \Rightarrow B$ ("A implies B") to be a machine that somehow transforms a proof of A into a proof of B. With this explanation of the meaning of these two connectives, we have already touched some basic concepts of a programming language: organizing data into larger chunks, and applying functions to arguments to obtain some result.

Given the above discussion, it may not seem unlikely that the rules of logic for $\wedge$ and $\Rightarrow$ might have something to do with programming with simple data and functions, but it is somewhat surprising how exact a correspondence can be made. For this logic, and many others, there exists a statically typed programming language corresponding to it such that every type in the language corresponds to a proposition in the logic, and every well-typed program corresponds to a derivation of the proposition that corresponds to its type, and vice-versa. To summarize the correspondence so far:

$$\begin{array}{rcl} \text{Propositions} & \Leftrightarrow & \text{Types} \\ \text{Proofs} & \Leftrightarrow & \text{Programs} \\ A \wedge B & \Leftrightarrow & A \times B \\ A \Rightarrow B & \Leftrightarrow & A \to B \end{array}$$

In fact the language's notion of 'well-typed' is essentially a definition of a valid proof. For instance, a program which has type $A \to B$ must faithfully produce some piece of data of type B whenever it is given data of type A, just as a valid proof of $A \Rightarrow B$ is obliged to show that B holds given any proof of A.

1.2 Dependent Types

Armed with an understanding of how a propositional logic can be seen through programming-language glasses, it is natural to try to add quantification on the logic side and see what new types appear. For a concrete example, suppose that K is the set of books, $O(k)$ is the proposition that k is an old book for any $k \in K$, and $L(k)$ is the proposition that k is in the library, again for any $k \in K$. The statement of "every book in the library is old" in symbols is, of course,

$$\forall k \in K. L(k) \Rightarrow O(k)$$

What counts as a proof of this proposition? It should be a program f which takes a book $k \in K$, a proof that $L(k)$ holds, and produce a proof of $O(k)$. In other words, a function of two arguments, whose type we might try to write as $K \to (L\ k \to O\ k)$. The obvious problem with this definition, however, is that it doesn't explain where k comes from. The trouble is that $L\ k$, the type of proofs that k is in the library, *depends on* the book chosen from the type K. The correct typing of f therefore involves a new *dependent type constructor* Π to take the place of the $\forall$.

$$f : \Pi k{:}\, K.(L\ k \to O\ k)$$

The type $\Pi x{:}\, A.B$ is a generalization of the type $A \to B$ where the argument is given a name x that can be used somewhere in the result type B. If x does not appear in B at all, then $\Pi x{:}\, A.B$ has the same meaning as $A \to B$, and because of this most theories of dependent types include $A \to B$ only as a syntactic abbreviation of $\Pi x{:}\, A.B$.

There remains the question of exactly what status the predicates O and L have in the programming language. Clearly for any $k{:}\, K$ we must have that $O\ k$ and $L\ k$ are types, since they appear on the left and right side of an arrow. As is suggested by the notation chosen, we might reasonably treat them as functions which take some element of K, and return a type. One then speaks of O and L as *type families* indexed by the type K, which have the *kind* $K \to \mathsf{type}$. We can extend the table above as follows:

Propositions	$\Leftrightarrow$	Types
$\cdots$	$\Leftrightarrow$	$\cdots$
$\forall x{:}\, A.B$	$\Leftrightarrow$	$\Pi x{:}\, A.B$
$P(x)$	$\Leftrightarrow$	$P\ x$

We can also extend the use of dependency to type families; The symbol Π is typically overloaded to also be the *dependent kind constructor*. We could express the proposition that a certain book from the library is checked out by a type family C with the kind $\Pi k{:}\, K.(L(k) \to \mathsf{type})$. In more plain language, this kind says that given a book k, and given a proof that k is in the library's collection, there is a type of proofs of the proposition that k is checked out.

Dependent types are in fact very common 'behind the scenes' in the definitions and reasoning of ordinary mathematics. As a more realistic example, we treat the type of the determinant operator in linear algebra. First we must establish what type a matrix has; we know that for every positive integers m, n we can consider the $m \times n$ matrices, so we write

$$\mathsf{mat} : (\Pi m{:}\,\mathsf{int}\,.\Pi n{:}\,\mathsf{int}\,.\,\mathsf{type})$$

to indicate the type family of matrices indexed by two integers m and n. We obtain a particular type by applying mat to particular integers. For instance, $\mathsf{mat}\ 3\ 4$ is the type of 3×4 matrices.

Now the determinant only exists for a square matrix, which is to say, for every n there is a determinant operator on $n \times n$ matrices. The type of det, then, is

$$\det : (\Pi n{:}\,\mathsf{int}\,.\Pi A{:}\,(\mathsf{mat}\ n\ n).\,\mathsf{real})$$

This type says that for each n which has type type, and each A which has type $\mathsf{mat}\ n\ n$ (which depends on the choice made for n) the expression $\det n\ A$ has type real. One can also check that the $n \times n$ identity matrix is

$$I : (\Pi n{:}\,\mathsf{int}\,.\,\mathsf{mat}\ n\ n)$$

("For every n an integer, $I\ n$ is an $n \times n$ matrix") and matrix transposition can be typed as

$$\mathsf{transp} : (\Pi m{:}\,\mathsf{int}\,.\Pi n{:}\,\mathsf{int}\,.\Pi A{:}\,(\mathsf{mat}\ m\ n).\,\mathsf{mat}\ n\ m)$$

("For every m, n integers and A an $m \times n$ matrix, $\mathsf{transp}\ m\ n\ A$ is an $n \times m$ matrix") and so on.

1.3 The Twelf System

Twelf is a *logical framework*, a system suitable for the formal description of programming languages and logics, and the automatic verification of properties of such descriptions. It is based on the type theory of LF, which is a λ-calculus with dependent types as described above. We use the word *language* by itself in what follows to mean either a programming language or logic.

A language can be represented in LF by declaring a *signature* of type families and constants which describe how the syntactic elements of the language are created and organized. A signature is essentially a formalization of the sort of preliminary assumptions made in each example above, e.g., "suppose K is a type, and O is a type family indexed by K." There are powerful and elegant techniques for representing common structures in languages in the presence such signatures. In particular, bound variables in programs and hypothetical reasoning in logic can both be rendered by using higher-order function types — that is, by declaring constants in the signature which take a function as an argument. Assertions about a programming language as a whole — for example, that every well-typed expression evaluates to a value — can be encoded as type families, and their validity can be automatically checked by the Twelf system.

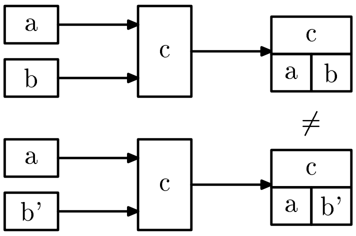
2 Proof Irrelevance

2.1 Equality

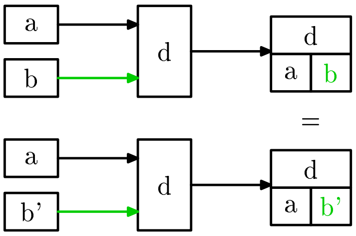
The two kinds of *judgments* in the theory of LF define typing of terms, and equality between terms. One of the consequences of the equality rules is that for some function $c : \tau_1 \to \cdots \to \tau_n \to \tau$ of n arguments declared in the signature, the terms $c\ M_1 \cdots M_n$ and $c\ M_1' \cdots M_n'$ are equal if only if $M_i = M_i'$ for all $1 \leq i \leq n$. There are times when this is not the desired behavior; sometimes we would like to be able make certain arguments to a declared function *irrelevant* when it comes to deciding equality. In particular, this is a common situation when some arguments to a function are meant to be thought of as *witnesses of provability* rather than *proofs* whose structure matters. Generally, suppose we use the following signature:

$$\begin{aligned}
&t, u : \mathsf{type}\\
&p : \Pi x\colon t.\, \mathsf{type}\\
&c : \Pi x\colon t.\Pi y\colon (p\ x).u\\
&a : t\\
&b, b' : p\ a.
\end{aligned}$$

Intuitively, we have an object a of type t, and some predicate p on objects of type t. The function c takes two arguments: an argument x of type t, and an argument y of type $p\ x$ (which might be thought of as a proof that p holds of x) and returns an object of type u. As the preceding discussion describes, $c\ a\ b \neq c\ a\ b'$ because not all of c's arguments are the same on both sides of the equation — b and b' differ.



Suppose we declare instead a $d : \Pi x\colon t.\Pi y \div (p\ x).u$, where the $\div$ symbol is supposed to be a promise that the argument y 'doesn't matter' in the output of d.



2.2 Proof Irrelevant Typing

The main typing judgment in LF has the form $\Gamma \vdash M : A$, where Γ is a list of typed variables $x_1 : A_1, \ldots, x_n : A_n$, M is an object, and A is a type. It is read as “in the context Γ, M has the type A.” The modifications to typing to accommodate reasoning about irrelevance are on the one hand extending contexts to allow also *irrelevant variables* $x \div A$ and on the other defining an additional *irrelevant typing judgment* $\Gamma \vdash M \div A$. A variable $x \div A$ can be thought of as the assumption that an object of type A exists, but whose structure cannot be inspected. The judgment $\Gamma \vdash M \div A$ says that, in the context Γ, M is a proof that there exists a term of type A. There is one inference rule for this judgment:

$$\frac{\Gamma^{\oplus} \vdash M : A \qquad \Gamma \vdash A : \mathsf{type}}{\Gamma \vdash M \div A}$$

where the *promotion* $\Gamma \mapsto \Gamma^{\oplus}$ is defined recursively by

$$\begin{array}{rcl} \cdot^{\oplus} & := & \cdot \\ (\Gamma, x : A)^{\oplus} & := & \Gamma^{\oplus}, x : A \\ (\Gamma, x \div A)^{\oplus} & := & \Gamma^{\oplus}, x : A \end{array}$$

The $\Gamma \vdash A : \mathsf{type}$ premise is necessary in the presence of dependent types for technical reasons. Ignoring it for the present, we can read the rule as saying that M is a proof that there is a term of type A if M would be a term of type A if we only supposed that all of our assumptions of the form $x \div A$ were strengthened to $x : A$. Since M may use irrelevant assumptions, whose structure cannot be inspected, M as a whole is only ‘irrelevantly at type A’, that is, $M \div A$, and M makes no promise to the outside world about its structure.

2.3 Irrelevant Equality

The corresponding equality judgment in LF is also typed; in other words, we do not ask if $\Gamma \vdash M = N$, but if $\Gamma \vdash M = N : A$ — if M and N are equal objects at type A. We add a new *irrelevant equality judgment* $\Gamma \vdash M = N \div A$, which is defined[1] by the rule

$$\frac{\Gamma \vdash M \div A \qquad \Gamma \vdash N \div A}{\Gamma \vdash M = N \div A}$$

We see immediately that, as desired, this means that equality at an irrelevant type is trivial: any two objects at the same irrelevant type are equated. The consequences of this definition become more clear when we consider the natural definition of the application equality rule. There is a rule in LF which can be used to show that two terms which are applications of a function to an argument are equal:

$$\frac{\Gamma \vdash M_1 = M_2 : \Pi x : A.B \qquad \Gamma \vdash N_1 = N_2 : A}{\Gamma \vdash M_1\ N_1 = M_2\ N_2 : B[N_1/x]}$$

For functions which take an *irrelevant argument* we add the rule

$$\frac{\Gamma \vdash M_1 = M_2 : \Pi x \div A.B \qquad \Gamma \vdash N_1 = N_2 \div A}{\Gamma \vdash M_1 \circ N_1 = M_2 \circ N_2 : B[N_1/x]}$$

where $\circ$ is the *irrelevant application* operator. We see in particular that if $M : \Pi x \div A.B$, then $M \circ N_1$ and $M \circ N_2$ for any $N_1, N_2 \div A$. This exactly captures the fact that M is allowed to require an argument of type A, but cannot observe its structure in any way.

2.4 Applications

2.4.1 Subset Types

As hinted at earlier, irrelevance can be used to adequately encode *subset types*. For example, suppose we wish to encode the type of composite numbers as a subset of the natural numbers. A standard representation of the naturals with addition and multiplication is

$$\begin{aligned}
&\mathsf{nat} : \mathsf{type}\\
&\mathsf{z} : \mathsf{nat}\\
&\mathsf{s} : \mathsf{nat} \to \mathsf{nat}\\
&\mathsf{sum} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}\\
&\mathsf{prod} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}
\end{aligned}$$

followed by declarations which define *logic programs* for addition and multplication such that $\mathsf{sum}\ A\ B\ C$ is the type of proofs that $A + B = C$ and $\mathsf{prod}\ A\ B\ C$ is the type of proofs that $A \times B = C$. Since a number x is composite if and only if there exist natural numbers $a, b \geq 2$ such that $ab = x$ (or equivalently,

[1]The actual premises are actually of the form $\Gamma^{\oplus} \vdash M = M : A$ rather than $\Gamma \vdash M \div A$ to make certain lemmas go through.

if there exist arbitrary natural numbers a, b such that $(a+2)(b+2) = x$) we might then try to define a type

$$\mathsf{comp} : \mathsf{type}$$

of composites and provide a constructor

$$\begin{aligned}\mathsf{comp\_rule} \quad : \quad & \Pi x : \mathsf{nat}\,.\Pi a \div \mathsf{nat}\,.\Pi b \div \mathsf{nat}\,. \\ & \Pi p \div \mathsf{prod}\,(\mathsf{s}\,\mathsf{s}\,a)\ (\mathsf{s}\,\mathsf{s}\,b)\ x.\,\mathsf{comp}\end{aligned}$$

which takes a natural x, natural numbers a, b, and a proof p that $(a+2)(b+2) = x$, and produces an object of type comp. By marking a, b, and p irrelevant we intend that there is exactly one object of type comp for each composite number; we would hope that $\mathsf{rcomp\_rule}\ \ 12\ 1\ 2\ p_1$ and $\mathsf{rcomp\_rule}\ \ 12\ 0\ 4\ p_2$ (where p_1, p_2 are proofs that $3 \times 4 = 12$ and $2 \times 6 = 12$, respectively) are simply different ways of exhibiting the compositeness of the number 12, and are therefore equated. Unforunately, it is not the case that $p_1 = p_2 \div A$ for any type A, since p_1 and p_2 have different types, since they are not merely different proofs, but proofs of different facts. In fact the signature given is not well-typed: The expression $\mathsf{prod}(\mathsf{s}\,\mathsf{s}\,a)\ (\mathsf{s}\,\mathsf{s}\,b)\ x$ is a type under the assumptions $a, b : \mathsf{nat}$, but what we really have available is merely $a, b \div \mathsf{nat}$. Both of these issues are related to the 'technical premise' $\Gamma \vdash A : \mathsf{type}$ of the irrelevant typing rule.

Thankfully the problem can be worked around, but it serves to illuminate the subtleties inherent in mixing the apparently simple idea of ignoring the structure of proofs with rich type systems. The solution is to declare a separate predicate of compositeness

$$\begin{aligned}\mathsf{is\_comp} \quad : \quad & \mathsf{nat} \to \mathsf{type} \\ \mathsf{is\_comp\_rule} \quad : \quad & \Pi x : \mathsf{nat}\,.\Pi a : \mathsf{nat}\,.\Pi b : \mathsf{nat}\,. \\ & \Pi p : \mathsf{prod}\,(\mathsf{s}\,\mathsf{s}\,a)\ (\mathsf{s}\,\mathsf{s}\,b)\ x.(\mathsf{is\_comp}\ x)\end{aligned}$$

and mark as irrelevant a proof of this predicate

$$\mathsf{comp\_rule} : \Pi x : \mathsf{nat}\,.\Pi p \div \mathsf{is\_comp}\ x.\,\mathsf{comp}$$

2.4.2 Subterm Omission in Proof-Carrying Code

In a language like Java, some measure of safety of running code is insured by running the code 'in a sandbox,' inside a trusted virtual machine. Proof-carrying code is another technique which aims to achieve the same (if not greater) safety properties without sacrificing runtime efficiency to emulating a virtual machine. The burden of a making a program safe falls instead on the author of the program, the *code producer*. The recipient of a program, the *code consumer* requires a *proof* that the program received satisfies some safety policy, and so the code producer must send with the program a formal certificate of safety which can be mechanically checked by the code consumer to actually prove that the program won't violate the policy.

Unfortunately, these certificates can sometimes be large, even on the order of the size of the size of the program being proved safe. Techniques for reducing proof size are desirable. Although the problem of finding a proof for a given proposition is typically undecidable, a proof may have many subproofs which could be easily and efficiently reconstructed by the code consumer. For instance, as part of a large proof that shows that a program always computes a certain mathematical function correctly, it might be necessary to show some trivial fact, say $3 + 4 = 7$. Now the formal correctness of this program depends on every last detail of the proof being correct, but there is no need to send a proof of $3 + 4 = 7$ across the network — there can simply be a blank spot in the proof with an instruction saying, "please check that in fact 3+4=7." The trade-off here is saving network bandwidth by perhaps spending more time reconstructing proofs on the code consumer end.

In practice, proofs are frequently represented as terms in a type theory like LF, and checked with a tool like Twelf. In this case, the idea of omitting subproofs really means omitting sub*terms*. The question to be addressed is, when can a subterm be safely omitted? Twelf already has facilities for providing guarantees[2] of *termination* of predicates considered as logic programs. When we can show using these tools that searching for a proof of P (which is what is meant by "running the logic program P") always terminates, and we know a proof of P, then we can be sure that the code consumer can also find a proof of P. What we do not know is that the code consumer will find the *same* proof. It may seem like a desirable property of a type system that if we replace a subterm S of a term M with a different subterm S' of the same type, then M is still well-typed, but dependent types systems do not necessarily have this property exactly because of the dependence of types on terms. For example, in the signature

$$
\begin{aligned}
&a, z : \mathsf{type} \\
&b : a \to \mathsf{type} \\
&c : \Pi x{:}\, a.(b\ x) \\
&d : \Pi x{:}\, a.(b\ x) \to z \\
&k_1, k_2 : a
\end{aligned}
$$

we have the typing

$$\cdot \vdash (d\ k_1)(c\ k_1) : z$$

but not

$$\cdot \vdash (d\ k_1)(c\ k_2) : z$$

even though we have only changed one subterm of type a to another.

If we introduce proof irrelevance, however, then it can be shown that replacing one irrelevantly typed subterm with another preserves the whole term being

[2] Only sufficient, not necessary conditions for termination! Nobody is pretending to have solved the halting problem.

well-typed. Therefore, it is safe to omit a subproof of a large proof if we can show that the subproof can be decidably recovered (i.e. if the predicate can be shown to be terminating) and is proof irrelevant.

3 Contributions

To realize the above applications, we proposed to integrate the type theory of proof irrelevance with some of the algorithms of the Twelf system.

3.1 Strict Definitions

Abbreviations and definitions are important tools in informal as well as formal mathematics in making large proofs understandable. However, it is inefficient for proof-checking software to expand the meaning of every definition it encounters. If a defined function can be shown to be *injective* — if it has a different output for every possible input — then we need not expand its definition when checking whether two applications of arguments to the function are equal. If d is injective, then $d\ M_1 \cdots M_n = d\ M_1' \cdots M_n'$ only if $M_i = M_i'$ for all $1 \leq i \leq n$. It is undecidable whether a given defined function is injective, but there is a stronger condition, *strictness*, that implies injectivity. Experience suggests that many definitions in practice are in fact strict.

Intuitively, a defined function is strict if it "uses all its arguments substantially." The idea is that if every argument has a substantial use, then changing any argument must change the result, and so the function is injective. Since the concepts of strictness and irrelevance are both about whether arguments to functions matter (even though they make opposite guarantees in a sense — a strict function definitely *does* use its argument and an irrelevant function definitely doesn't) we would expect they would nontrivially interact. Indeed some care is required to define strict definitions in a type system with proof irrelevance, but we have arrived at what we believe is a reasonable definition, which satisfies the analogous lemma which guarantees injectivity of strict functions.

3.2 Unification

Solving *unification problems*, problems of the form

$$\exists \theta. M[\theta] \stackrel{?}{=} N[\theta]$$

or, in words, 'Does there exist a substitution for the free variables of M and N that makes them equal?' is a common issue in logic and type theory in general, and in logical frameworks in particular. Even the question of existence (much less finding such a substitution when it exists) is undecidable for LF. Twelf gets around this problem by only promising to be able to solve unification problems in the *pattern fragment*, a restriction on how variables the substitution affects can appear in the terms M and N. In practice, even problems which are not pattern can be solved by Twelf by progressively postponing the 'hard' (i.e. non-pattern) parts of the problem and gaining information about them from solutions to other (pattern) subproblems. The notion of *pattern substitution* which is central to this algorithm is very closely related to the definition of *pattern spine*, which is used in the formulation of strict definitions as above. We have not finished the theoretical work in generalizing Twelf's unification algorithm to irrelevance, but we hope to exploit this relationship to arrive at the right definitions.

3.3 Implementation

We have begun making changes to the current CVS version of Twelf to support proof irrelevance in concrete syntax input and output of terms, strictness checking, and typechecking. Since our theoretical understanding of unification is still not complete, the typechecking algorithm (which depends on unification to make sure that functions are applied to arguments of the right type among other things) is still not completely reliable, but it is producing promising first results. Some issues remain with the choice of concrete syntax as well. We intend to have a correct, usable implementation by the end of April 2002.