Extending Higher-Order Unification to Support Proof Irrelevance

Jason Reed
Carnegie Mellon University

September 11, 2003
What is Proof Irrelevance?

- The idea that all proofs of a proposition are equal.
- (The term appears in the literature occasionally meaning ‘irrelevance everywhere’, of all proof equality becoming trivial, especially in proofs of the form ‘X and Y imply proof irrelevance’ — this is not what we are talking about)
- “Intensionality, Extensionality and Proof Irrelevance in Modal Type Theory” [Pfenning ’01] treats irrelevance as a modality.
- Compare with fact that both logic “linear everywhere” and logic with linear and intuitionistic variables are possible.
Outline

I. Motivation

II. Type Theory

III. Unification

IV. Patterns
What good is Proof Irrelevance?

- A couple examples, using the dependent type theory LF [Harper, Honsell, Plotkin ’93] as a starting point.
- Examples shaped and motivated throughout by the design choices of twelf, [Pfenning, Schürmann ’99] an implementation of LF and associated algorithms.
- Motivation #1: adequate encodings
- Motivation #2: proof compaction
Motivation #1: Adequate Encodings

- Desirable property for an encoding of a theory into a logic like LF is **adequacy**, existence of a **compositional bijection** between object-language terms and (canonical) LF objects.

- Compositional, i.e. substitution commutes with translation.

- Proof irrelevance as a modality makes adequate encodings of certain concepts much easier.
Adequate Encodings (2)

- Take the standard encoding of the untyped \( \lambda \)-calculus:

\[
\begin{align*}
\text{tm} : \text{type} & \quad \text{lam} : (\text{tm} \to \text{tm}) \to \text{tm} \\
\text{app} : \text{tm} \to \text{tm} \to \text{tm}
\end{align*}
\]

- How to get ‘strict lambda calculus’, each \( \lambda \) var to occur at least once? (Historical footnote: Church’s original calculus like this)

- Easy to code up a definition of occurrence:

\[
\begin{align*}
\text{occurs} : (\text{tm} \to \text{tm}) \to \text{type} \\
\text{occurs}\_\text{app1} : \text{occurs} (\lambda x.\text{app} (\text{M} x) (\text{N} x)) & \leftrightarrow \text{occurs} (\lambda x. (\text{M} x)) \\
\text{occurs}\_\text{app2} : \text{occurs} (\lambda x.\text{app} (\text{M} x) (\text{N} x)) & \leftrightarrow \text{occurs} (\lambda x. (\text{N} x)) \\
\text{occurs}\_\text{var} : \text{occurs} (\lambda x. x)
\end{align*}
\]

- So \( \text{occurs} (\lambda x.\text{M} x) \) type of proofs that \( x \) occurs in \( \text{M} \)
Adequate Encodings (3)

• We would try $\text{lam} : \Pi t : (tm \to tm).(\text{occurs } t) \to tm$ but it doesn’t work right.

• Generally lots of proofs that $x$ occurs, as many as occurrences!

• $\text{lam } t \ P_1 \neq \text{lam } t \ P_2$ for $P_1 \neq P_2$

• **Failure of adequacy!**

• Don’t want to care about which proof of occurrence.

• That is, we want an ‘irrelevant arrow’. We’ll write brackets around the argument to suggest:

$$lam : \Pi t : (tm \to tm).[(\text{occurs } t)] \to tm$$

• Need $\text{lam } t \ [P_1] = \text{lam } t \ [P_2]$ for any proofs $P_1, P_2$ to recover adequacy.
Motivation #2: Proof Compaction

- Domain: Proof-Carrying Code [Necula, Lee ’96]
- Problem: proofs are big — There’s a market for ways of making them smaller.
- Maybe we can omit subterms that can be recovered by the consumer?
- This is realistic; big proofs of undecidable properties can have lots of space-consuming little subproofs of (efficiently) decidable properties.
- Assert the existence of the little subproofs, let the consumer reconstruct them.
Proof Compaction (2)

- But what if the consumer reconstructs a different proof of the same fact?
- Coordinating reconstruction algorithms at both ends possible, but a headache
- Instead use irrelevant subproof requirements in the signature.
- This permits the receiver to safely reconstruct any valid subproof.
- There’s a result that states that after replacing an irrelevant subterm with another of the same type, the whole term is still well-typed.
- Not true in ordinary LF because of dependent types.
- Another win: avoid constructing intermediate proof terms
Extending LF Type Theory

• Normally, we can check applications for equality with the rule

\[
\frac{\Gamma \vdash M = M' : \Pi x : A.B \quad \Gamma \vdash N = N' : A}{\Gamma \vdash M \, N = M' \, N' : \{N/x\}B}
\]

• For irrelevant functions, we want the arguments not to matter. So we have:

\[
\frac{\Gamma \vdash M = M' : \Pi x : [A].B \quad \Gamma \vdash N = N' : [A]}{\Gamma \vdash M \, [N] = M' \, [N'] : \{N/x\}B}
\]

and say that any two objects at [A] are equal.

• (Just as \( A \to B \) abbreviates \( \Pi x : A.B \) where \( x \) doesn’t occur in \( B \), we’ll say \( [A] \to B \) means \( \Pi x : [A].B \))
Extending LF (2)

- Naturally, we get terms at irrelevant-\(\Pi\) type from irrelevant lambdas:

\[
\Gamma, x : [A] \vdash M : B \\
\Gamma \vdash \lambda x : [A]. M : \Pi x : [A]. B
\]

- Forces us to consider what \textbf{irrelevant hypotheses} mean.

- Answer: \(x : [A]\) assumes that some object at type \(A\) exists, but we are not allowed to analyze its structure, only use the bare fact that its type is inhabited.

- Knee-jerk reaction to a new kind of hypothesis: what kind of objects can we substitute for it?

- New typing judgment: \(\Gamma \vdash M : [A]\). Think “\(M\) is an irrelevant object at type \(A\)” or “\(M\) is an inhabitation witness for type \(A\)”
Irrelevance Rules

- Defining inference rule: ([Γ'] just means \( x_1 : [A_1], \ldots x_n : [A_n] \))

\[
\begin{align*}
\Gamma, \Gamma' \vdash M : A \\
\hline
\Gamma, [\Gamma'] \vdash A : \text{type} \\
\hline
\Gamma, [\Gamma'] \vdash M : [A]
\end{align*}
\]

Note hypothesis rule is still merely \( \Gamma, x : A \vdash x : A \) not anything that would allow \( \Gamma, x : [A] \vdash x : A \). (\( \Gamma, x : [A] \vdash x : [A] \) is admissible)

- \( x : [A] \) is a weaker hypothesis than \( x : A \), and \( M : [A] \) is a weaker assertion than \( M : A \); When judging \( M : [A] \) one gets to use irrelevant hypotheses ‘unbracketed’.

- \( \Gamma \vdash M : A \) implies \( \Gamma \vdash M : [A] \).

- See the tech report for why \( \Gamma, [\Gamma'] \vdash A : \text{type} \) needed.
Higher-Order Pattern Unification

- How twelf, for instance, thinks of unification. Used for type reconstruction, logic programming queries.

- **Higher-order**: allow variables to be of function type.

- Restricted to the pattern fragment [Miller ’91], because we want unification to be **decidable** and have unique **most general unifiers**.

- The fact that type reconstruction relies on unification is a big motivation for this: don’t want type-checking to be undecidable or have an ambiguous answer.

- [Dowek, Hardin, Kirchner, Pfenning ’96] worked out an algorithm for this case; we extended it to cover LF with irrelevance.

- Just few interesting corner cases — see paper for details
Unification

- Stepping back a bit, a unification problem looks like

\[ \exists U_1 \ldots \exists U_n. M_1 \doteq N_1 \land \cdots \land M_n \doteq N_n \]

- Find terms for \( U_1, \ldots, U_n \) so all equations satisfied, or determine that no such exist.

- Must allow open (allowing \( \exists \)-quantified variables to occur) instantiations, or else immediate undecidability! For instance, \( \exists U. \exists V. U \doteq c \ V \). Answer: \( U \leftarrow c \ V \)

- Otherwise, exists closed term at \( V \)’s type? Undecidable.
**Unification (2)**

- Irrelevance means that equations that look straightforward are actually trivial in the same way as the above one.

- Consider

\[ \exists U.c \ [k] \doteq c \ [U] \]  

\((*)\)

- If this were

\[ \exists U.c \ k \doteq c \ U \]

We’d just assign \( U \leftarrow k \).

- But in \((*)\), the equation holds no matter what \( U \) is set to; to get most general unifier, we **don’t** instantiate \( U \).
Unification (3)

- Sometimes we need to introduce new variables. Consider

\[ \exists U.(\lambda x. c [x] \doteq \lambda x. U) \]

Since \( U \) is quantified on the outside, it doesn’t make sense to say \( U \leftarrow c \ [x] \).

- But the argument to \( c \) here is irrelevant!

- We can introduce \( V \), instantiate \( U \leftarrow c \ [V] \) and the equation \( \lambda x. c \ [x] = \lambda x. c \ [V] \) holds, because of irrelevant application.

- In fact this is the most general unifier.

- Compare \( \exists U.(\lambda x. c \ x \doteq \lambda x. U) \), which fails.
Unification (4)

- $\exists U. U \doteq c U$ fails.
- $U$ appears rigid on the right, ‘occurs-check’ fails.
- $\exists U. U \doteq c[U]$? Introduce new variable $V$;
- $U \leftarrow c[V]$ gives $c[V] = c[c[V]]$. This is the most general unifier!
So...

- Why not replace every term $M \ [N]$ with $M \ [V]$ for fresh $V$? Def’n of equality lets us.

- Better yet, why not replace with $M \ *$, where $*$ is a magic new term such that $* = *$?

- Answer: We don’t just want to solve the question of unifiability, but unification. We mean to find actual unifiers, and provide as much inhabitation information as possible to potential algorithms downstream.

- Overagressive insertion of variables or placeholders suboptimal in this aspect.
Patterns

- When we come down to an equation like $U M_1 M_2 M_3 \equiv N$, things get hard. Got to build $N$ out of $M_i$, but $M_i$ may be messy.

- A **pattern** [Miller ’91] is where we restrict variables $U, V$, etc. to occur only applied to distinct local (i.e. once bound by $\lambda$) variables.

  Pattern: $\lambda x.\lambda y.\lambda z. U z x$

  Not: $\lambda x.\lambda y.\lambda z. U x x$

  Not: $\lambda x.\lambda y.\lambda z. U (c y)$

  Not: $\lambda x.\lambda y.\lambda z. U (V x y z)$

  $\exists U. U z x \equiv c z (x z) \Longrightarrow U \leftarrow \lambda z.\lambda x. c z (x z)$

  $\exists U. U x x \equiv x \Longrightarrow ??? U \leftarrow \lambda x_1.\lambda x_2. x_1? U \leftarrow \lambda x_1.\lambda x_2. x_2?$
Patterns (2)

- Pattern restriction makes unification decidable, and most general unifiers always exist. Current definition is sound with irrelevance, but we can squeeze more patterns out of it.

- Turns out we can allow irrelevant applications of any argument at all. Normal args must still be distinct bound variables.

- Pattern: $\lambda x.\lambda y. U \ y \ [c \ x \ y] \ x \ [V \ y \ y]$

- Pattern: $U \ [M]$ for any $M$. e.g.

  $\exists U. U \ [M] \doteq c \ [N] \implies U \leftarrow \lambda z: [A]. c \ [N]$

- Any substitution for $U$ that satisfies the eq’n must be equal to $\lambda z: [A]. c \ [N]$!
Unification with Irrelevance

- Turn all of these intuitions into an algorithm; technical details:
  - Soundness and completeness go as usual, showing that transition rules maintain unifiers.
  - Termination because they make the problem smaller according to the right metric.
  - Pattern unification with irrelevance is **decidable**, has unique **most general unifiers**.
  - Extensible to the so-called **dynamic pattern fragment** by postponing constraints.
- We have a **prototype implementation** based on **twelf**.
Summary

• **Proof irrelevance** as a modality is useful for expressing adequate encodings and guaranteeing the safety of flexible proof reconstruction.

• Known algorithm for **higher-order pattern unification** modified to work in a type theory with irrelevance.

• Known definition of **higher-order pattern** has a simple generalization to irrelevant arguments.

• Questions?