Higher-Order Pattern Unification and Proof Irrelevance

Jason Reed

Carnegie Mellon University
Pittsburgh, PA 15213
jcreed@andrew.cmu.edu

Abstract. Extending the type theory of a logical framework with a proof irrelevance modality has several potential advantages, including the ability to represent subset types and invariants for proof compression ([Pfe01]). Although the extended theory is well-behaved, it is not yet completely clear how to modify the implementation of a logical framework to accommodate proof irrelevance. The unification algorithm in Twelf ([PS99]), in particular, works by a process of constraint simplification that depends on the notion of *pattern substitution*. Adapting this algorithm to work with proof irrelevance requires generalizing the definition of pattern. We propose such a definition, guided by work with proof irrelevance and strictness, and make progress towards proving its correctness.

1 Introduction

1.1 Higher-Order Pattern Unification

Our starting point is an algorithm for higher-order pattern unification using explicit substitutions due to Dowek et al. [DHKP96]. The difficulties of variable capture involved in higher-order unification are avoided by a reduction [DHK95] to first-order equational unification in a language with explicit substitutions [ACCL90]. This first-order algorithm specializes nicely for a decidable subset of unification problems, the *pattern fragment*. The strategy of constraint simplification for solving general unification problems — trying to solve equations which are in the pattern fragment and postponing constraints produced by those that aren't, which may later be resolved by substitutions arising from other equations — though not complete, has been found to work well in practice.

Informally, *pattern* is a term where all the variables (by which we mean metavariables amenable to substitution) occur above a sequence of distinct deBruijn indices, i.e. bound variables. If all variables have atomic type (and we can easily transform problems to have this property) then this means each variable must occur under a substitution whose range is a set of distinct deBruijn indices, a *pattern substitution*. Pattern substitutions are desirable because they have one-sided inverses, and so are injective. Equations of the form

$$X[\zeta] = b$$

therefore have at most one solution, written $b[\zeta]^{-1}$.

1.2 Proof Irrelevance

Constants are naturally injective; if we have a constant $c : \tau_1 \to \cdots \to \tau_n \to \tau$ of n arguments, 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 $1 \le i \le n$. There are times when this is not the desired behavior. Sometimes we would like to be able make certain arguments to a function 'irrelevant' when it comes to deciding equality. This can occur when some arguments to a function are meant to be thought of as witnesses of provability rather than pieces of data whose structure matters. This phenomenon is *proof irrelevance*. Generally, suppose we use the following signature in the dependent type system LF ([HHP93]):

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

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, $c\ a\ b \ne c\ a\ b'$ because not all of c's arguments are the same on both sides of the equation — b and b' differ. We would like to declare instead a $d : \Pi x{:}\, t. \Pi y \div (p\ x).u$, where the $\div$ symbol is supposed to be a promise that the argument y 'doesn't matter' in the output of d. We proceed to discuss some applications of proof irrelevance in more detail.

1.3 Encoding Subset Types

The example above is really a trivial case of the use of proof irrelevance to obtain adequate encodings of *subset types*, which can be useful when representing programming languages and logics in a logical framework. A subset type is a formal version of the common mathematical set-formation syntax $\{x \in X \mid \phi(x)\}$ for some predicate ϕ on X. As a typical example of its usefulness, suppose we wish to represent a lambda calculus which has a 'relevant' binder $\lambda^{\ge 1}$ which requires the variable it binds to occur at least once. We could try doing this by beginning with the usual untyped lambda calculus

$$
\begin{aligned}
&\mathsf{tm} : \mathsf{type} \\
&\mathsf{app} : \mathsf{tm} \to (\mathsf{tm} \to \mathsf{tm}) \\
&\mathsf{lam} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}
\end{aligned}
$$

and defining a predicate

$$\mathsf{occurs} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{type}$$

via the *logic program*

$$\begin{aligned}
&\mathsf{occurs_var} : \mathsf{occurs}(\lambda x.x)\\
&\mathsf{occurs_appl} : \mathsf{occurs}(\lambda x.\,\mathsf{app}\ \ M_1\ M_2) \leftarrow \mathsf{occurs}(\lambda x.M_1)\\
&\mathsf{occurs_appr} : \mathsf{occurs}(\lambda x.\,\mathsf{app}\ \ M_1\ M_2) \leftarrow \mathsf{occurs}(\lambda x.M_2)\\
&\mathsf{occurs_lam} : \mathsf{occurs}(\lambda x.\,\mathsf{lam}(M\ x)) \leftarrow (\Pi y{:}\,\mathsf{tm}\,.\,\mathsf{occurs}(\lambda x.M\ x\ y))
\end{aligned}$$

which captures the proposition that an open term uses its free variable, and declaring a constant

$$\mathsf{olam} : \Pi t : (\mathsf{tm} \to \mathsf{tm}).\Pi p : \mathsf{occurs}\,t.\,\mathsf{tm}$$

This encoding, however, is not *adequate* in the sense that there are generally multiple LF terms which represent a given object-language term. This is because there are potentially many proofs that a variable occurs bound — in fact there will be one proof per occurrence of the variable. Therefore we want to equate all terms using olam that differ only in which occurrence proof they use. That is, we should declare instead

$$\mathsf{olam} : \Pi t : (\mathsf{tm} \to \mathsf{tm}).\Pi p \div \mathsf{occurs}\,t.\,\mathsf{tm}$$

and the new encoding is adequate. Though the revised olam still requires the existence of a proof p that $\mathsf{occurs}\,t$ holds, it doesn't care *which* proof in a certain sense, which is guaranteed by the type system.

1.4 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 [NL97] 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 subterms. The question to be addressed is, when can a subterm be safely omitted? Twelf already has facilities for providing sufficient conditions for *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 subterm under an irrelevant application with another preserves the whole term being 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 occurs under an irrelevant application.

1.5 Extending Unification

The task before us is to modify the pattern unification algorithm to work in a language with proof irrelevance, and in particular to find the right notion of pattern. A similar situation arises with the notion of strict definitions [PS98], which depends on the definition of pattern spines. In the absence of irrelevance, a pattern spine is again a list of distinct bound variables. In [Ree02] we found a modification of the definition of pattern spine which satisfies the same critical lemma as the original, again an injectivity property. The changes made were adding an 'irrelevant cons' to the syntax of spines and requiring distinct bound variables at 'relevant' positions and allowing any term at 'irrelevant' positions. We imitate these changes in a definition of pattern substitution, and observe that one-sided inverses again exist, for an appropriate notion of equality of substitutions. The theory here is simply-typed throughout, though we expect an extension to full dependent types to be reasonably straightforward.

2 Syntax

We define the $\lambda\sigma^\iota$-calculus, an extension of the typed $\lambda\sigma$-calculus with term variables [ACCL90] to include proof irrelevance in the sense of [Pfe01]. For uniformity of syntax, we have a sort of modalities μ with which we annotate function types, function abstractions, context and substitution conses, and applications (written $\diamond$). The modality $\mathbf{r}$ refers to the usual ('relevant') notion from λ-calculus and $\mathbf{i}$ gives the 'irrelevant' version.

Modalities	μ	$::= \mathbf{r} \mid \mathbf{i}$
Types	A, B	$::= \alpha \mid A \rightarrow^\mu B$
Contexts	Γ	$::= \bullet \mid A \cdot^\mu \Gamma$
Terms	a, b	$::= 1 \mid \mathcal{X} \mid (a \diamond^\mu b) \mid \lambda^\mu_A a \mid a[s]$
Substitutions	s, t	$::= \mathbf{id} \mid \uparrow \mid a \cdot^\mu s \mid s \circ t$

3 Typing

There are two typing judgments,

Term Typing	$\Gamma \vdash a :^\mu A$	a has type A for modality μ in the context Γ.
Substitution Typing	$\Gamma \vdash s : \Gamma'$	s is a substitution for the variables of Γ' in Γ.

Note that the term typing judgment is also annotated with a modality. We abbreviate $:^\mathbf{r}$ by $:$ and $:^\mathbf{i}$ by $\div$.

The meaning of the irrelevant typing judgment is given by the context operation $—^\oplus$, defined recursively by

$$\bullet^\oplus = \bullet$$

$$(A \cdot^\mu \Gamma)^\oplus = A \cdot^\mathbf{r} (\Gamma^\oplus)$$

and the rule

$$\frac{\Gamma^\oplus \vdash a : A}{\Gamma \vdash a \div A}$$

$\Gamma^\oplus$ is the context Γ with all irrelevant assumptions promoted to 'real' assumptions, so the typing rule allows us to conclude that $a \div A$ if we can derive $a : A$ when we are allowed to use even irrelevant assumptions.

The remaining typing rules are

$$\frac{}{A \cdot^{\mathbf{r}} \Gamma \vdash 1 : A} \qquad \frac{}{A \cdot^{\mu} \Gamma \vdash \uparrow : \Gamma}$$

$$\frac{A \cdot^{\mu} \Gamma \vdash b : B}{\Gamma \vdash \lambda^{\mu}_{A}.b : A \rightarrow^{\mu} B} \qquad \frac{\Gamma \vdash a : A \rightarrow^{\mu} B \qquad \Gamma \vdash b :^{\mu} B}{\Gamma \vdash (a \diamond^{\mu} b) : B}$$

$$\frac{}{\Gamma \vdash \mathbf{id} : \Gamma}$$

$$\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma' \vdash a :^{\mu} A}{\Gamma \vdash a[s] :^{\mu} A}$$

$$\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma' \vdash t : \Gamma''}{\Gamma \vdash t \circ s : \Gamma''}$$

$$\frac{\Gamma \vdash s : \Gamma' \qquad \Gamma \vdash a :^{\mu} A}{\Gamma \vdash a \cdot^{\mu} s : A \cdot^{\mu} \Gamma'}$$

$$\frac{}{\Gamma_X \vdash X : T_X}$$

4 Reduction

We list a set of reduction rules which are a straightforward adaptation of those in $\lambda\sigma$, including those added in [ACCL90] (**Id**, **IdR**, **VarShift**, **Scons**) for critical pairs arising from the addition of term variables.

Beta	$(\lambda^{\mu}_{A}.a)b$	$\rightarrow (a[b \cdot^{\mu} \mathbf{id}])$
Eta	$\lambda^{\mu}_{A}.(a \diamond^{\mu} 1)$	$\rightarrow b$ if $a =_{\sigma} b[\uparrow]$
App	$(a \diamond^{\mu} b)[s]$	$\rightarrow (a[s] \diamond^{\mu} b[s])$
VarCons	$1[a \cdot^{\mu} s]$	$\rightarrow a$
Id	$a[\mathbf{id}]$	$\rightarrow a$
Abs	$(\lambda^{\mu}_{A}.a)[s]$	$\rightarrow \lambda^{\mu}_{A}.(a[1 \cdot^{\mu} (s \circ \uparrow)])$
Clos	$(a[s])[t]$	$\rightarrow a[s \circ t]$
IdL	$\mathbf{id} \circ s$	$\rightarrow s$
IdR	$s \circ \mathbf{id}$	$\rightarrow s$
ShiftCons	$\uparrow \circ (a \cdot^{\mu} s)$	$\rightarrow s$
AssEnv	$(s_1 \circ s_2) \circ s_3$	$\rightarrow s_1 \circ (s_2 \circ s_3)$
MapEnv	$(a \cdot^{\mu} s) \circ t$	$\rightarrow a[t] \cdot^{\mu} (s \circ t)$
VarShift	$1 \cdot^{\mu} \uparrow$	$\rightarrow \mathbf{id}$
Scons	$1[s] \cdot^{\mu} (\uparrow \circ s)$	$\rightarrow s$

It remains to work out what normalization and confluence properties this system enjoys. It seems likely that weak or strong normalization of any fragment should follow easily from the same property holding of the corresponding fragment in the underlying calculus without irrelevance by a simple erasure of modality information. However, to whatever extent the property would depend on the typability of a term, we might encounter difficulty dealing with full dependent types, since erasing irrelevance from a dependently typed term may not result in a well-typed term.

We might also consider adding a reduction rule

$$\textbf{Irrel} \qquad a \cdot^{\mathbf{i}} s \to a' \cdot^{\mathbf{i}} s$$

to capture the intended meaning of an irrelevant cons, but this clearly destroys any hope of weak normalization. We expect to handle the necessary quotienting-out of terms at irrelevant conses in a different way.

5 Pattern substitutions

We define a judgment $s\,\mathbf{pat}^{n\geq I}$, where s is a substitution, n a natural number, and I a list of natural numbers. Its intended meaning is that s is a pattern substitution using deBruijn indices from I which are no greater than n.

$$\frac{}{\uparrow^n \mathbf{pat}^{n\geq\cdot}}$$

$$\frac{s\,\mathbf{pat}^{n\geq I} \qquad m \leq n, m \notin I}{\mathtt{m} \cdot^{\mathbf{r}} s\,\mathbf{pat}^{n\geq I,m}} \qquad \frac{s\,\mathbf{pat}^{n\geq I}}{a \cdot^{\mathbf{i}} s\,\mathbf{pat}^{n\geq I}}$$

From this we can define *pattern terms* via a judgment $a\,\mathbf{pat}^{n\geq I}$, defined by the rules

$$\frac{s\,\mathbf{pat}^{n\geq I}}{X[s]\,\mathbf{pat}^{n\geq I}}$$

$$\frac{a\,\mathbf{pat}^{n\geq I} \qquad m \leq n, m \notin I}{a \diamond^{\mathbf{r}} \mathtt{m}\,\mathbf{pat}^{n\geq I,m}} \qquad \frac{a\,\mathbf{pat}^{n\geq I}}{a \diamond^{\mathbf{i}} b\,\mathbf{pat}^{n\geq I}}$$

Definition 1. *A $\lambda\sigma^\iota$-term is a $\lambda\sigma^\iota$-*pattern *if all of its subterms of the form $a = (X[s]\; b_1 \cdots b_n)$ are such that $a\,\mathbf{pat}^{n\geq I}$ for some n, I. A* pattern substitution *is a $\lambda\sigma^\iota$-substitution all of whose $\lambda\sigma^\iota$-normal forms s are such that $s\,\mathbf{pat}^{n\geq I}$ for some n, I.*

Note that in the absence of any confluence result we hedge our definition of of pattern substitutions by referring to all normal forms rather than *the* normal form of the substitution s. Since we do not have substitution variables, we hope that in fact the normal forms of substitutions are well-behaved, and this inelegance can be removed.

The proof in [DHKP96] that pattern spines have one-sided inverses is constructive, and so can be described algorithmically. We first give a direct description, and then sketch how to modify the algorithm. By unrolling the induction definition, a pattern substitution ξ is of the form

$$a_1 \cdot^{\mu_1} a_2 \cdot^{\mu_2} \cdots a_n \cdot^{\mu_n} \uparrow^m$$

Put

$$\bar{\xi} = b_1 \cdot^{\mathbf{r}} b_2 \cdot^{\mathbf{r}} \cdots b_m \cdot^{\mathbf{r}} \uparrow^n$$

where

$$b_i = \begin{cases} \mathtt{j} & \text{If } \exists j.\mu_j = \mathbf{r} \wedge a_j = \mathtt{i}; \\ \text{a fresh variable } X & \text{otherwise.} \end{cases}$$

Observe that

$$\xi \circ \bar{\xi} \to^* a_1[\bar{\xi}] \cdot^{\mu_1} a_2[\bar{\xi}] \cdot^{\mu_2} \cdots a_n[\bar{\xi}] \cdot^{\mu_n} \uparrow^n$$

For any i, if μ_i is $\mathbf{r}$, then by definition of $\bar{\xi}$ and pattern substitutions we have $a_i[\bar{\xi}] \to^* \mathtt{i}$. Otherwise, μ_i is $\mathbf{i}$, and we have some term $a_i[\bar{\xi}]$ occurring at an irrelevant position in the substitution ξ. The intended meaning of irrelevance is that this is just as good as any other term at the same type, in particular the deBruijn index $\mathtt{i}$. Therefore, in a certain sense (and determining the right way to formalize this is the subject of our current effort) we have

$$\xi \circ \bar{\xi} \approx \mathtt{1} \cdot^{\mu_1} \mathtt{2} \cdot^{\mu_2} \cdots \mathtt{n} \cdot^{\mu_n} \uparrow^n \to^* \mathbf{id}$$

To modify the algorithm, we add modality annotations to the existing rules in a straightforward way, and add two rules,

$$\begin{array}{ll} \textbf{NDI} & n[a \cdot^{\mathbf{i}} \xi]^{-1} = (n[\xi]^{-1})[\uparrow] \\ \textbf{UDI} & \uparrow^m \circ (a \cdot^{\mathbf{i}} \xi)^{-1} = (\uparrow^m \circ \xi^{-1}) \circ \uparrow \end{array}$$

which cause irrelevant conses to act as if they were deBruijn indices that 'never occur,' in the sense that they never match other indices during application of an inverse substitution to a term (**NDI** is like **ND**$\neq$ in [DHKP96]) and they never occur in the range of $\uparrow^m$ during composition of a substitution with an inverse substitution (**UDI** is like **UD1**).

6 Future Work

Computing the inverse is not the only operation on substitutions involved in the unification algorithm of [DHKP96]. We must also extend the definition of intersection $\xi \cap \zeta$ to transform equations of the form $X[\xi] = X[\zeta]$, and the pruning substitution $\xi|\zeta$ to correctly handle flexible occurrences of metavariables. Moreover it may be that the notion of flexible occurrence can be extended to include occurrences of metavariables anywhere under irrelevant application while maintaining the correctness of the overall algorithm. We intend to work out the appropriate extensions of these concepts towards a complete unification algorithm for higher-order patterns in the extended theory, and answer the basic questions of normalization and confluence mentioned above.

References

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[DHK95] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 366–374, San Diego, California, June 1995. IEEE Computer Society Press.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[NL97] George C. Necula and Peter Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[Pfe01] Frank Pfenning. Intensionality, extensionality and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, 2001.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Ree02] Jason Reed. Proof irrelevance and strict definitions in a logical framework. Unpublished Manuscript, 2002.