Higher-Order Constraint Simplification In Dependent Type Theory

Jason Reed[⋆]

School of Computer Science
Carnegie Mellon University
Pittsburgh PA, USA

Abstract. Higher-order unification is undecidable, but has fragments which admit practical algorithms, which are used extensively in logical frameworks. For example, it is decidable whether unification problems in the *pattern fragment* identified by Dale Miller are solvable, and they enjoy unique most general unifiers when they are. However, the restrictions that the pattern fragment imposes exclude many useful applications and encodings.

One way to proceed is to use instead a more general *constraint simplification* algorithm that works on the parts of a unification problem that are in the pattern fragment, postponing problematic parts in the hope that later substitutions will bring them back into the pattern fragment. Such an algorithm either finds a most general solution, determines that the problem does not have a solution, or else reports a set of remaining constraints on which no further work can be done.

While some constraint simplification algorithms have been proposed, their theory turns out to be surprisingly subtle — especially in the presence of dependent types, which complicate otherwise simple invariants that all equations in a unification problem are well-typed — and has, to our knowledge, not been investigated, leading to some problems with termination and completeness of implementations. This paper describes and proves correct a new, terminating constraint simplification algorithm for the dynamic pattern fragment of higher-order unification in a dependent type system.

1 Introduction

Unification is the task of solving equations over terms in some language, a problem of widespread utility in programming languages and logical frameworks: it is used for type inference and reconstruction algorithms, for the execution of programs in logic programming languages, and for reasoning about the totality of functions defined by pattern-matching clauses.

[⋆] The work was supported by the Fundação para a Cincia e Tecnologia (FCT), Portugal, under a grant from the Information and Communications Technology Institute (ICTI) at CMU. Many thanks to Frank Pfenning and William Lovas for helpful discussions.

Type theories with features like higher-order function types and dependent types permit powerful representation techniques, but the design of unification algorithms for such languages is more complicated. Even restricting attention to the simply-typed λ-calculus, it is known that full higher-order unification is undecidable. However, useful decidable subsets of the problem have been carved out. A particularly well-behaved fragment arises from the notion of higher-order *pattern* [Mil91] identified by Dale Miller.

The pattern fragment requires that metavariables (those for which we seek solutions during unification) only appear, when they are of function type, applied to a sequence of distinct λ-bound variables. This restriction makes unification decidable and even guarantees the existence of a most general unifier when there is a unifier at all. In practice, however, the pattern fragment is more restrictive than appropriate for many applications.

An empirical study of the limitations of the pattern fragment and the usefulness of going beyond it can be found in Michaylov and Pfenning [MP92]. They nonetheless also observed that the majority of unification problems encountered in practice still come close to being in the pattern fragment in a 'dynamic' way. Though a unification problem as a whole is not in the pattern fragment, certain parts of it may still be, and we may fruitfully try to eagerly solve these currently tractable equations, and let the information we learn from them instantiate variables and simplify other equations.

A *constraint simplification* algorithm along these lines was suggested by Dowek et al. [DHKP96] as an extension to their algorithm for higher-order pattern unification presented in the same paper. However, no proof was given of the extension's correctness, and, as it happens, it fails to terminate on some inputs. (a counterexample to termination is given in Section 2.2) The way the algorithm can be coaxed into nontermination is rather subtle, and has to do with precisely how the algorithm postpones difficult work, specifically when it is uncertain how to resolve conflicting information about how metavariables of function type can depend on their arguments.

The contribution of this paper is a new algorithm for constraint simplification that achieves termination by using a different technique for postponing this uncertainty. This technique nonetheless emerges rather naturally out of existing ideas in the study of pattern unification, as do certain aspects of its correctness proof.

Another significant challenge to proving that the algorithm we propose is correct is that we work in the presence of dependent types. While we would ordinarily like to assume every equation is between two objects of the same type, solving equations over dependent types creates the possibility of equations arising between terms of different types, or which are between terms that are not well-typed at all. This is because as we compare two function applications, the earlier arguments affect the type of the later arguments, and if the former are not equal, the latter will not be of the same type. Conal Elliott [Ell90] dealt with these issues in his PhD thesis, (as did Pym [Pym90] independently at roughly the same time) but in a Huet-style pre-unification algorithm, by using a rather complex invariant that equations can be partially ordered to exhibit how the solvability of one guarantees the well-typedness of another.

In our case, the argument is still not entirely trivial, but we achieve some simplification by choosing the typing invariant to be more straightforwardly that all equations are well-typed *modulo*, in a suitable sense, the solvability of all equations.

The remainder of the paper is organized as follows. Section 2 describes the language in which we study unification. Section 3 gives the constraint simplification algorithm itself, and Section 4 outlines the proof of its correctnes.

2 Language

2.1 The Basic Language

The setting in which we wish to study the unification problem is the dependent type theory LF [HHP93]. We take advantage of several useful developments in the study of logical frameworks, some comparatively recent and not yet in widespread use.

The first is restricting attention to only the *canonical forms* of terms, those that are fully η-expanded and β normal, made possible by the observation by Watkins [WCPW03] that not only is it straightfoward to impose this restriction in the grammar of the language, but also that one can define an operation of *hereditary substitution* directly on canonical forms, a substitution operation that hereditarily reducing any β-redices that just a single substitution might have created, so that the output is again canonical.

Second, we restrict attention further to only ever consider expressions that are *simply well-typed*, sometimes known as 'approximately well-typed' [Ell89], which means that they are well-typed after all dependencies in types erased, and every Π turned into a mere $\rightarrow$. This simplifies many definitions and arguments; for instance the argument that hereditary substitution always terminates depends only on simple times, so if we only ever have simply-well-typed terms, it is manifestly terminating and total.

Third, we employ the *contextual modal type theory* [NPP05] built on top of LF, which provides a convenient and logically motivated language for dealing with metavariables. In it metavariables are logically hypotheses of *categorical* judgments corresponding to the fact that we substitute a *closed* term for them. It also makes it easy to describe the common implementation practice of *lowering* function variables to base type by changing function application to a notion of substitution.

Finally, the presentation is in *spine form* [CP97], where the arguments to a function are grouped together into a 'spine', exposing the head of the term separately.

The syntax of the language is

$$\begin{array}{rcl}
\text{Kinds} & K & ::= \text{type} \mid \mathit{\Pi} x{:}A.K \\
\text{Types} & A, B, C & ::= a \cdot S \mid \mathit{\Pi} x{:}A.B \\
\text{Normal Terms} & M, N & ::= \lambda x.M \mid R \\
\text{Atomic Terms} & R & ::= H \cdot S \mid u[\sigma] \\
\text{Heads} & H & ::= c \mid x \\
\text{Spines} & S & ::= () \mid (M; S) \\
\text{Substitutions} & \sigma & ::= \cdot \mid \sigma, (y/x) \mid \sigma, (M/x) \\
\text{Modal Substitutions} & \theta & ::= \cdot \mid \theta, (R /\!\!/ u) \\
\text{Contexts} & \mathit{\Gamma}, \mathit{\Psi} & ::= \cdot \mid \mathit{\Gamma}, x : A \\
\text{Modal Contexts} & \mathit{\Delta} & ::= \cdot \mid \mathit{\Delta}, u :: (\mathit{\Psi} \vdash a \cdot S)
\end{array}$$

As usual, $A \to B$ abbreviates $\mathit{\Pi} x{:}A.B$ where x does not appear in B. We use X, Y in the sequel to uniformly denote expressions of any of these syntactic sorts.

The kind and type levels are quite standard: kinds are formed by $\mathit{\Pi}$-abstraction from the base kind 'type', and types are formed by $\mathit{\Pi}$-abstraction from some base type $a \cdot S$, where a is a constant type family, applied to some arguments S.

The term language has functions $\lambda x.M$ and applications of heads (either constants c or variables x) to spines S. The novelty coming from contextual modal type theory is the expression $u[\sigma]$, which is a *use* of a metavariable u, under a suspended explicit substitution σ. Instead of directly considering metavariables at function type, we follow the practice of *lowering* them to base type and representing what would have been their arguments as substitutions for a *local context* of variables $\mathit{\Psi}$. These substitutions σ contain both term-for-variable replacements (M/x) as well as variable-for-variable substitutions (y/x). The latter are instrumental in the definition of pattern substitutions below.

The principal typing judgments are

$$\mathit{\Delta}; \mathit{\Gamma} \vdash M \Leftarrow A \qquad \mathit{\Delta}; \mathit{\Gamma} \vdash R \Rightarrow A \qquad \mathit{\Delta}; \mathit{\Gamma} \vdash S : A > C$$

pronounced 'M checks at type A', 'R synthesizes its type A', and 'S, if a head of type A is applied to it, yields an expression of type C', each in a context $\mathit{\Gamma}$ and modal context $\mathit{\Delta}$. We typically leave the $\mathit{\Delta}$ implicit when it is clear from context. Some of the more salient typing rules are as follows. Those omitted are as in standard presentations of LF.

$$\frac{\mathit{\Gamma}, x : A \vdash M \Leftarrow B}{\mathit{\Gamma} \vdash \lambda x.M \Leftarrow \mathit{\Pi} x{:}A.B}$$

$$\frac{x : A \in \mathit{\Gamma} \qquad \mathit{\Gamma} \vdash S : A > C}{\mathit{\Gamma} \vdash x \cdot S \Rightarrow C} \qquad \frac{\mathit{\Gamma} \vdash R \Rightarrow a \cdot S' \qquad S = S'}{\mathit{\Gamma} \vdash R \Leftarrow a \cdot S}$$

$$\frac{u :: (\mathit{\Psi} \vdash a \cdot S) \in \mathit{\Delta} \qquad \mathit{\Gamma} \vdash \sigma : \mathit{\Gamma}'}{\mathit{\Gamma} \vdash u[\sigma] \Rightarrow a \cdot [\sigma]S}$$

$$\frac{}{\mathit{\Gamma} \vdash \cdot : \cdot} \qquad \frac{\mathit{\Gamma} \vdash M \Leftarrow [\sigma]A' \qquad \mathit{\Gamma} \vdash \sigma : \mathit{\Gamma}'}{\mathit{\Gamma} \vdash (M/x).\sigma : (\mathit{\Gamma}', x : A')}$$

$$\frac{y : A \in \Gamma \qquad A = [\sigma]A' \qquad \Gamma \vdash \sigma : \Gamma'}{\Gamma \vdash (y/x).\sigma : (\Gamma', x : A')}$$

The notation $[\sigma]X$ indicates the operation applying all the individual substitutions in σ to the expression X (in constrast to $u[\sigma]$, where σ is simply remains inert as a suspended substitution, waiting for u to be instantiated)

The definition of carrying out single substitutions is given by the functions $[M/x]X$ and $[M \mid S]$, the latter giving essentially the β-reduction of the function M against its arguments S, defined as follow:

$$[M/x](x \cdot S) = [M \mid [M/x]S]$$

$$[M/x](H \cdot S) = H \cdot ([M/x]S) \qquad (\text{if } x \neq H)$$

$$[M/x]() = () \qquad [M/x](N; S) = ([M/x]N; [M/x]S)$$

$$[M/x](\lambda y.N) = \lambda y.[M/x]N$$

$$[M/x](u[\sigma]) = u[[M/x]\sigma]$$

$$[M/x]((x/y).\sigma) = (M/y).([M/x]\sigma)$$

$$[M/x]((z/y).\sigma) = (z/y).([M/x]\sigma) \qquad (x \neq z)$$

$$[M/x]((M'/y).\sigma) = ([M/x]M'/y).([M/x]\sigma)$$

$$[\lambda x.M \mid (N; S)] = [[N/x]M \mid S]$$

$$[R \mid ()] = R$$

Note that since all expressions are assumed to be simply-well-typed, both of these functions are total.

We also must specify when *modal substitutions* are well-formed, and how they operate. These are the substitutions of closed expressions for modal variables. For our purposes, we allow variables in the modal context to have types depending one another even when the graph of dependencies has cycles. This may seem rather exotic, but it is justifiable by thinking of the entire context Δ as assigning simple types first of all, and then refining these declarations with dependent types once all the variables are in scope. This approach has the advantage of eliminating the need for reasoning about reordering of the modal context, and also directly reflects the typical implementation of unification, which uses (intrinsically unordered) imperative reference cells for unification variables, whose types can indeed in practice be cyclically dependent during unification.

It is worth noting that when the algorithm succeeds and returns a set of solutions, the variables that are still free may still have cyclically dependent types. If the intended application of unification prohibits this (such as the abstraction phase of type reconstruction in a logical framework) then one can simply check for cycles and report an error if they are still present.

The typing rule for modal substitutions is

$$\frac{\begin{array}{ccc} u_i :: (\Psi_i \vdash A_i) \in \Delta & & \theta A_i = A \\ \theta = (R_1 /\!\!/ u_1) \ldots (R_n /\!\!/ u_n) & \Delta', \theta\Psi_i \vdash R_i \Rightarrow A & (\forall i \in 1 \ldots n) \end{array}}{\Delta' \vdash \theta : \Delta}$$

which requires all terms R_i to have the type declared in Δ, after θ has been applied to it. Carrying out the substitution θ before we even know it is fully well-typed is meaningful because at least we know by prior assumption that it is simply well-typed.

The operation of modal substitution θX is defined similarly to ordinary substitution above, in that it is simply homomorphic on nearly all cases, the exception being when we come to a modal variable. We define $\theta(u[\sigma])$ to be $[\theta\sigma]R$ when $(R/u) \in \theta$.

The important feature of modal substitutions is that they are only constructed out of modal, not ordinary, hypotheses; and so they represent appropriately the role of closed instances of metavariables. The substitution principles for ordinary and modal substitutions are as usual, for example

Lemma 1. *For any judgment J, if $\Gamma \vdash M \Leftarrow A$ and $\Gamma, x : A, \Gamma' \vdash J$, then $\Gamma, [M/x]\Gamma' \vdash [M/x]J$.*

Lemma 2. *For any judgment J, if $\Delta' \vdash \theta \Leftarrow \Delta$ and $\Delta; \Gamma \vdash J$, then $\Delta'; \theta\Gamma \vdash \theta J$.*

2.2 Extensions

There are two further extensions we propose to this language. The first and more important one is reifying information about variable dependency with placeholders that stand for 'dead' variables. It can be motivated by considering the counterexample to the termination of the existing constraint simplification algorithm of Dowek et al. [DHKP96]

Placeholders Suppose o is a base type. Let Δ consist of the metavariables u, v, w all of type $(z : o \vdash o)$. Let f be a constant of type $o \to o$. Without setting up unification formally yet, consider the pair of equations

$$(\lambda x.\lambda y.u[x] \doteq \lambda x.\lambda y.f\ v[w[y]] \wedge (\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.u[w[y]])$$

and suppose we are trying to find solutions to u, v, w that satisfy both of them.

Examine the first equation in particular. We notice that the function $\lambda x.\lambda y.u[x]$ on the left cannot use its second argument at all, for u does not receive it as an argument. Therefore neither can the right side of this equation (once v and w are instantiated) mention y. However, since v is applied to an expression that itself has the variable w in it, we do not know whether v or w projects away its argument, but we know at least one of them must.

Notice that u's instantiation must nonetheless be of the form $f(U'\ z)$ for some $U' : o \to o$. We might hope that we are making progress by creating a new variable $u' :: (z : o \vdash o)$ and carrying out the substitution $u \leftarrow f(u'[z])$ — in fact, this is precisely what the algorithm suggested in [DHKP96] does. But if we do, we only arrive at the pair of equations

$$(\lambda x.\lambda y.f\ (u'[x]) \doteq \lambda x.\lambda y.f\ v[w[y]]) \wedge (\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.f\ u'[w[y]])$$

which, after stripping the identical constants f from the first equation, leads only to

$$(\lambda x.\lambda y.u'[x] \doteq \lambda x.\lambda y.v[w[y]]) \wedge (\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.f\ u'[w[y]])$$

Swapping the two equations and changing the names of the unification variables, this is identical to the pair we started with, and the algorithm may loop forever.

Our approach to fixing this problem is instead to directly embody the intuition that the occurrence of the bound variable y is something that 'cannot survive' the eventual instantiation of both v and w. This idea can be found implicitly in Tobias Nipkow's algorithm [Nip93] for higher-order pattern unification over simple types. He makes use of the 'deBruijn index $-\infty$' after computing inverses of substitutions to stand for variables that do not occur in the range of the substitution. Although Nipkow says it 'smells of a hack,' we aim to show that its use can be theoretically justified.

We therefore introduce an explicit placeholder, written $\_$, for an expression that occurs somewhere in an argument to a unification variable, but for which we mean to require that every solution will project it away:

$$\text{Normal Terms } M, N ::= \cdots \mid \_$$

The definition of hereditary substitution is extended by saying $[M/x](\_) = \_$ and $[\_ \mid S] = \_$.

Armed with this we can transform $\lambda x.\lambda y.u[x] \doteq \lambda x.\lambda y.f\ v[w[y]]$ by instantiating $u \leftarrow f\ v[w[\_]]$, which leads to the first equation being turned into $\lambda x.\lambda y.f\ v[w[\_]] \doteq \lambda x.\lambda y.f\ v[w[y]]$ and the second equation becoming

$$\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.f\ v[w[\_]]$$

It will turn out that we can reason about this latter equation using a form of the occurs-check, and correctly reject it as unsolvable.

While $=$ will continue to use below to mean strict syntactic equality of two expressions up to α-varying bound variables, it will be convenient to also define $\equiv$ by saying that $X \equiv X'$ if $X = X'$ and also X, X' contain no occurrences of $\_$.

Free Variables The other extension is a notion of variables m which are *modal* like metavariables, but unlike them they are not *contextual.* To say that they are modal means that they are declared in the modal contextΔ and are allowed to be used in an R in a modal substitution containing (R/u). To say that they are not contextual means that instead of being used under a substitution for local variables, they take a spine of arguments, just as ordinary variables do.

$$\begin{array}{rl} \text{Heads } H & ::= \cdots \mid m \\ \text{Modal Contexts } \Delta & ::= \cdots \mid \Delta, m :: A \end{array}$$

The role of these variables is to represent *free* variables in a unfication problem whose type may involve metavariables, but which is not meant to be instantiated during the course of unification. These arise naturally from wanting to use unification for type reconstruction in a dependent type theory. For example, if we encoded $n \times p$ matrices, and the operation of matrix transposition $M^{\top}$ and a theorem claiming that if $M_1^{\top} = M_2$, then $M_2^{\top} = M_1$, we might write something like

$$\begin{array}{l} \mathit{matrix} : \mathit{nat} \rightarrow \mathit{nat} \rightarrow \text{type} \\ \mathit{transpose} : \mathit{matrix}\ N\ P \rightarrow \mathit{matrix}\ P\ N \rightarrow \text{type} \\ \mathit{thm} : \mathit{transpose}\ M_1\ M_2 \rightarrow \mathit{transpose}\ M_2\ M_1 \rightarrow \text{type} \end{array}$$

in which the free variables N, P, M_1, M_2 are understood as implicitly Π-bound, and whose types are to be determined by type reconstruction via unification. Our knowledge of the *type* of M_1 can be represented as $\mathit{matrix}\ u[\cdot]\ v[\cdot]$ for metavariables $u :: (\cdot \vdash \mathit{nat}), v :: (\cdot \vdash \mathit{nat})$, for unification will determine what u and v must be, but M_1 itself is not open for instantiation, and is represented therefore as a free variable.

The reason free variables take spines instead of substitutions so that they may be conveniently compared for equality: equality on spines is, as is typical for canonical-forms-only presentations of LF, a simple matter of running down the syntax and ensuring everything is literally identical up to α-conversion. For substitutions, because of the presence of variable-for-variable replacement (y/x), equality is complicated by the fact that such a replacement can also be represented as a term-for-variable replacement $(\eta^* y/x)$ where $\eta^* y$ is the full η-expansion of y. For this reason we avoid in the sequel ever posing the question of whether two substitutions are considered equal.

3 Unification

We now define the task of unification, and the constraint simplification algorithm for it. A unification problem is written as $\Delta \vdash P$ where

$$\begin{array}{rl} \text{Equation Sets } P & ::= \top \mid P \wedge Q \\ \text{Equations } Q & ::= M \doteq M' \mid R \doteq R' \mid S \doteq S' \mid u \doteq R \mid u \leftarrow R \end{array}$$

The intended interpretation of $\Delta \vdash P$ is a query whether there exist instantiations of all the metavariables in Δ that satisfy the conjunction of the equations in P. The ‘equation’ $u \leftarrow R$ indicates that we have found an instantiation for u, and that it is R. It differs from the use of $u \doteq R$, in that in the latter, u may have other occurrences in R, preventing us from carrying out an instantiation. We often write just P instead of $\Delta \vdash P$ when it is clear from context.

The *active metavariables* of P is the set of metavariables in P such that $u \leftarrow R \notin P$. A modal substitution $\theta = (R_1 /\!\!/ u_1) \ldots (R_n /\!\!/ u_n)$ is *ground* if there are no occurrences of metavariables in the R_i. Free variables m are still allowed, and in fact will guarantee that unification problems are never trivially unsolvable because the types of their metavariables are uninhabited, because we always have the option of making up a free variable of the same type. A *solution* to P is a ground modal substitution θ for all the metavariables in Δ such that

1. For every equation $X \doteq X' \in P$ we have $\theta X \equiv \theta X'$
2. For every $u \leftarrow R \in P$ we have $(\theta R /\!\!/ u) \in \theta$
3. All the terms in θ are free of $\_$.

We write $\theta \models P$ if θ is a solution of P.

Let $\vec{u}$ be a subset of the metavariables in P. A *$\vec{u}$-solution* to P is a ground modal substitution for $\vec{u}$ that is a restriction of a solution of P to the variables $\vec{u}$. We write $\theta \models_{\vec{u}} P$ in this case, and the set of all such solutions is written $Sol(P, \vec{u})$.

3.1 Algorithm

A state of the unification algorithm is either a set of equations in context $\Delta \vdash P$ or the constant $\bot$, standing for failure. We say for the sake of uniformity that $Sol(\bot, \vec{u}) = \emptyset$.

A *pattern substitution* is a substitution σ that consists of only distinct bound variables and underscores. Formally:

$$\frac{}{\vdash \cdot\ \mathsf{pattern}} \qquad \frac{\vdash \sigma\ \mathsf{pattern} \qquad y \notin \operatorname{rng}\sigma}{\vdash \sigma, (y/x)\ \mathsf{pattern}} \qquad \frac{\vdash \sigma\ \mathsf{pattern}}{\vdash \sigma, (\_/x)\ \mathsf{pattern}}$$

A *strong pattern substitution* is one that has no underscores. We use ρ to denote a pattern substitution, and ξ to denote a strong pattern substitution.

We define the following auxiliary functions: ξ^{-1}_{Γ} computes the inverse of a strong pattern substitution whose codomain is the context Γ.

$$\begin{aligned} \xi^{-1}_{\cdot} &= \cdot \\ \xi^{-1}_{\Gamma,x} &= \xi^{-1}_{\Gamma}, \begin{cases} (y/x) & \text{if } (x/y) \in \xi \\ (\_/x) & \text{otherwise.} \end{cases} \end{aligned}$$

$\xi \cap \mathrm{id}$ replaces non-identity substitutions in ξ with $\_$.

$$\begin{aligned} \cdot \cap \mathrm{id} &= \cdot \\ (\xi, (x/x)) \cap \mathrm{id} &= (\xi \cap \mathrm{id}), (x/x) \\ (\xi, (x/z)) \cap \mathrm{id} &= (\xi \cap \mathrm{id}), (\_/z) \ \ (\text{if } x \neq z) \end{aligned}$$

We say $\rho \setminus x = (x_1/x_1) \ldots (x_n/x_n)$ if $(\_/x) \in \rho$, and $x_1, \ldots, x_n$ are the variables in $\operatorname{dom}(\rho) \setminus \{x\}$. We say $(\Gamma \vdash a \cdot S) \setminus x = \Gamma' \vdash a \cdot S$ if the variable x does not occur free in S or in any type in Γ. and Γ' is Γ with that variable removed.

We use $\hat{X}\{Y\}$ to refer to an expression of syntactic class X with a hole in it, where the hole has been replaced by the expression Y, which may refer to variables bound in X. Further, $\hat{X}_{rig}\{Y\}$ refers to a *rigid* context in which Y occurs, that is, Y's occurrence is not within the arguments σ of some metavariable occurrence $u[\sigma]$. Similarly $\hat{X}_{srig}\{Y\}$ refers to a *strongly rigid context* in which Y occurs, that is, not within an argument to a metavariable, nor within an argument to a bound variable x.

The algorithm consists of repeatedly applying the following transition rules:

1. **Decomposition**.

$$\begin{array}{l}
(\lambda x.M \doteq \lambda x.M') \wedge P \mapsto (M \doteq M') \wedge P \\
(H \cdot S \doteq H \cdot S') \wedge P \mapsto (S \doteq S') \wedge P \\
(H \cdot S \doteq H' \cdot S') \wedge P \mapsto \bot \qquad (\text{if } H \neq H') \\
((M; S) \doteq (M'; S')) \wedge P \mapsto (M \doteq M') \wedge (S \doteq S') \wedge P \\
() \wedge P \mapsto P \\
\hat{Q}_{rig}\{\_\} \wedge P \mapsto \bot
\end{array}$$

2. **Inversion**.

$$(u[\xi] \doteq R) \wedge P \mapsto (u \doteq [\xi^{-1}]R) \wedge P$$

3. **Occurs-Check**.

$$\begin{array}{l}
(u \doteq H \cdot \hat{S}\{u[\xi]\}) \wedge P \mapsto (u \doteq H \cdot \hat{S}\{\_\}) \wedge P \\
(u \doteq c \cdot \hat{S}_{srig}\{u[\sigma]\}) \wedge P \mapsto \bot
\end{array}$$

4. **Intersection**.

$$(u \doteq u[\xi]) \wedge P \mapsto \begin{cases} P & \text{if } \xi \cap \mathrm{id} = \xi \\ (u \doteq u[\xi \cap \mathrm{id}]) \wedge P & \text{otherwise} \end{cases}$$

5. **Pruning**.

$$\begin{array}{l}
(\Delta, u :: (\Gamma \vdash A) \vdash \hat{Q}_{rig}\{u[\rho]\} \wedge P) \mapsto \\
\qquad (\Delta, u :: (\Gamma \vdash A), v :: ((\Gamma \vdash A) \setminus x) \vdash (u \doteq v[\rho \setminus x]) \wedge \hat{Q}_{rig}\{u[\rho]\} \wedge P)
\end{array}$$

6. **Instantiation**.

$$(\Delta \vdash (u \doteq R) \wedge P) \mapsto ([R/u]\Delta \vdash (u \leftarrow R) \wedge [R/u]P) \qquad (u \notin FV(R))$$

The algorithm may nondeterministically choose any of these steps, with the restriction that after Pruning, it must immediately take an Instantiation step on the freshly created equation $u \doteq v[\rho \setminus x]$. This instantiation could have been incorporated into the definition of pruning, obviating such a side condition, but it is simpler for the proofs of correctness below to isolate the substitutions that instantiation carries out. Furthermore steps that would not cause the unification state to change at all (for example performing intersection twice in a row on the same equation) are forbidden.

Whenever the entire problem consists of assignments $u \leftarrow R$ (with no R containing $\_$) the algorithm succeeds, and reports the modal substitution induced by the assignments.

4 Correctness

The three facts we wish to show about the algorithm are that it terminates, that it preserves solutions, and that it preserves well-typedness of unification states.

Theorem 1. *The algorithm always terminates, resulting in one of*

- *A solved state, where only assignments $u \leftarrow R$ remain in P*
- *A stuck state, i.e. one on which no transition rule applies*
- *Failure $\bot$*

Proof. Consider the lexicographic order of

1. The number of active metavariables.
2. The total size of the local contexts of the active metavariables.
3. The total size of the terms in all the equations in the unification problem, with _ considered smaller than any other term.

All transitions that change the state at all decrease this metric. Most transitions decrease (3) and maintain (1) and (2). Pruning (with a required instantiation step following it as described above) replaces one metavariable with another one of a smaller context, decreasing (2) and maintaining (1). Instantiation reduces (1). ■

As the algorithm progresses, each transition rule neither creates nor destroys solutions.

Theorem 2. *If $P_0 \mapsto P_1$ then $Sol(P_0, \vec{u}) = Sol(P_1, \vec{u})$, where $\vec{u}$ is the set of metavariables of P_0.*

Proof. By case analysis on the transition. Details deferred to the appendix.

This has as an immediate consequence that if we reach a solved state $u_1 \leftarrow R_1 \wedge \cdots \wedge u_n \leftarrow R_n$, then $\theta = (R_1 /\!\!/ u_1) \ldots (R_n /\!\!/ u_n)$ is a most general unifier of the original problem, since every solution to the original problem is an instance of θ, because it is a solution of $u_1 \leftarrow R_1 \wedge \cdots \wedge u_n \leftarrow R_n$ by invariant.

4.1 Preservation of Types

Defining the typing invariant for the algorithm is possibly the most challenging part of the present work. There are two complicating factors: one is that we work on equations in a fairly arbitrary order despite the fact that typing of one equation depends on solvability of another, and two is reasoning about placeholder _. To handle the first issue, we define what it means to be well-typed *modulo* a unification problem P.

Typing Modulo We say $X \equiv_P X'$ ('X is equivalent to X' modulo P') if, for any ground θ that substitutes for the metavariables of X, X' that is a solution of P, we have $\theta X \equiv \theta X'$. This equivalence is only meant to be asked of X, X' that are underscore-free, although P may have underscores remaining in it. Clearly $\equiv_P$ it is a partial equivalence relation whose support is the set of underscore-free simply-typed expressions. For all typing judgments $\Gamma \vdash J$, we define $\Gamma \vdash_P J$ by the same rules as for $\Gamma \vdash J$, except replacing the occurrences of $=$ in them with $\equiv_P$.

Generalizing to equivalence modulo P sometimes requires more subtle care about which parts of judgments are input and output. For example, the generalizations of the substitution principles for normal and atomic terms

Lemma 3. *If $\Gamma \vdash_P M \Rightarrow A$ and $\Gamma, x : B, \Gamma' \vdash_P J$ and $A \equiv_P B$, then $\Gamma, [M/x]\Gamma' \vdash_P [M/x]J$.*

Lemma 4. *If $\Gamma \vdash_P R \Rightarrow A$ and $u : (\Psi \vdash a \cdot S) \in \Delta$ and $a \cdot S \equiv_P A$ and $\Psi \vdash_P J$, then $[R/u]\Delta; [R/u]\Gamma' \vdash_P [R/u]J$.*

differ in that for atomic terms, we need to 'slacken' to account for a possible equivalence modulo P rather than exact equality between $a \cdot S$ and A, because the type A is an output of the process of synthesizing a type for R.

From the fact that unification preserves solutions comes the fact that equivalence and typing modulo P do not change when the unification algorithm acts on P.

Lemma 5. *If $P_0 \mapsto P_1$ and $A \equiv_{P_0} A'$, then $A \equiv_{P_1} A'$.*

Proof. Let θ be given such that $\theta \models P_1$. Suppose the set of metavariables of P_0 is $\vec{u}$: it may be smaller, but not bigger, than that of P_1. Thus $\theta\big|_{\vec{u}} \models_{\vec{u}} P_1$. By theorem 2, also $\theta\big|_{\vec{u}} \models_{\vec{u}} P_0$. By assumption that $A \equiv_{P_0} A'$, we have $\theta\big|_{\vec{u}}A = \theta\big|_{\vec{u}}A'$, and so $\theta A = \theta A'$, as required. ∎

Corollary 1. *Suppose $P_0 \mapsto P_1$ Then*

- *If $\Delta; \Gamma \vdash_{P_0} M \Leftarrow A$, then $\Delta; \Gamma \vdash_{P_1} M \Leftarrow A$.*
- *If $\Delta; \Gamma \vdash_{P_0} R \Rightarrow A$, then exists A' such that $\Delta; \Gamma \vdash_{P_1} R \Rightarrow A'$ and $A' \equiv_{P_0} A$.*
- *If $\Delta; \Gamma \vdash_{P_0} S : A > C$, then exists C' such that $\Delta; \Gamma \vdash_{P_1} S : A > C'$ and $C' \equiv_{P_0} C$.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash_{P_0} J$. Most cases are simple appeals to the induction hypothesis on all components. A more interesting case is when we deal with the following rule:

$$\frac{\Delta; \Gamma \vdash_{P_0} R \Rightarrow A \qquad A \equiv_{P_0} B}{\Delta; \Gamma \vdash_{P_0} R \Leftarrow B}$$

By induction hypothesis, there is an A' such that $\Delta; \Gamma \vdash_{P_1} R \Rightarrow A'$ and $A \equiv_{P_0} A'$. By transitivity, we have $A' \equiv_{P_0} B$, and by Lemma 5, $A' \equiv_{P_1} B$. So we can form a derivation

$$\frac{\Delta; \Gamma \vdash_{P_1} R \Rightarrow A' \qquad A' \equiv_{P_0} B}{\Delta; \Gamma \vdash_{P_1} R \Leftarrow B}$$

∎

Well-formedness of Unification States To type an expression X that has placeholders _ in it, we will say that X is well-typed if it is appropriately related to a well-typed expression that is underscore-free. Define $X' \sqsupseteq X$ (pronounced "X' is a completion of X") to mean X' arises by replacing every _ in X with some normal term, not necessarily the same term for every _.

This means that if, during unification, we take some well-formed X' and simply replace a normal subterm of it with _, the resulting term X will manifestly still be considered well-typed, because its immediately prior state $X' \sqsupseteq X$ is a witness to the fact that X is suitably related to a well-typed expression. This makes reasoning about the type preservation of the occurs-check and intersection transitions straightforward.

Completions only play a role in the theory, and need not appear at all in an implementation of the algorithm.

We can now define the judgments $\Delta \vdash_{P'} P\ \mathsf{wf}$ and $\Delta \vdash_{P'} Q\ \mathsf{wf}$, that the unification problem P (resp. equation Q) is well-formed modulo P':

$$\frac{\Delta' \sqsupseteq \Delta \qquad \Delta' \vdash_{P'} Q\ \mathsf{wf} \qquad \Delta \vdash_{P'} P\ \mathsf{wf}}{\Delta \vdash_{P'} Q \wedge P\ \mathsf{wf}}$$

$$\frac{}{\Delta \vdash_{P'} \top\ \mathsf{wf}}$$

$$\frac{\Delta; \Gamma \vdash_{P'} M_i' \Leftarrow A \qquad M_i' \sqsupseteq M_i \qquad (\forall i \in \{1,2\})}{\Delta \vdash_{P'} M_1 \doteq M_2\ \mathsf{wf}}$$

$$\frac{\Delta; \Gamma \vdash_{P'} R_i' \Rightarrow A_i \qquad R_i' \sqsupseteq R_i \qquad A_1 \equiv_{P'} A_2 \qquad (\forall i \in \{1,2\})}{\Delta \vdash_{P'} R_1 \doteq R_2\ \mathsf{wf}}$$

$$\frac{\Delta; \Gamma \vdash_{P'} S_i' : A > C_i \qquad S_i' \sqsupseteq S_i \qquad C_1 \equiv_{P'} C_2 \qquad (\forall i \in \{1,2\})}{\Delta \vdash_{P'} S_1 \doteq S_2\ \mathsf{wf}}$$

$$\frac{u :: (\Gamma \vdash a \cdot S) \in \Delta \qquad \Gamma \vdash_{P'} R' \Rightarrow A \qquad R' \sqsupseteq R \qquad A \equiv_{P'} a \cdot S}{\Delta \vdash_{P'} u \doteq R\ \mathsf{wf}}$$

$$\frac{u :: (\Gamma \vdash a \cdot S) \in \Delta \qquad \Gamma \vdash_{P'} R' \Rightarrow A \qquad R' \sqsupseteq R \qquad A \equiv_{P'} a \cdot S}{\Delta \vdash_{P'} u \leftarrow R\ \mathsf{wf}}$$

We say $\Delta \vdash P\ \mathsf{wf}$ if there exists an extension $\Delta' = \Delta, m_1 :: A_1, \ldots, m_n :: A_n$ of Δ by modal variables such that $\Delta' \vdash_P P\ \mathsf{wf}$.

First of all it is easy to show that taking a step in the set of equations in the 'modulo' preserves typing.

Lemma 6. *If* $P_0 \mapsto P_1$ *and* $\Delta \vdash_{P_0} P\ \mathsf{wf}$*, then* $\Delta \vdash_{P_1} P\ \mathsf{wf}$.

Proof. By induction on the derivation of $\Delta \vdash_{P_0} P\ \mathsf{wf}$, using Corollary 1 to transfer typing judgments and equivalences forward. ■

Although, as noted above, the definitions are arranged to make reasoning about the introduction of _ during the occurs-check and intersection transitions easy, it still remains to justify why inversion — the only other transition that creates underscores — preserves types. We need to construct a completion of ξ^{-1} that is free of underscores. Assuming $\Gamma \vdash \xi : \Gamma'$, this is defined as follows, similarly to inversion, as

$$\xi^*_{\cdot} = \cdot$$

$$\xi^*_{\Gamma,x:A} = \xi^*_{\Gamma}, \begin{cases} (y/x) & \text{if } (x/y) \in \xi \\ (u[\xi^*_{\Gamma}]/x) & \text{otherwise, for a fresh } u :: (\Gamma \vdash A). \end{cases}$$

This definition is in fact essentially identical to the standalone definition of inversion given by Dowek et al. [DHKP96] when their aim is to merely show that pattern substitutions have a one-sided inverse. The important idea is that for every new underscore we would have created by inversion, we insert a new metavariable of the correct type, so that that the resulting expression is still well-typed, and is a completion of inversion.

Since this definition is so close to inversion, it shares many of its properties, in particular being a one-sided inverse. Most importantly, however, the substitutions it outputs are well-typed for well-typed inputs.

Lemma 7. *If $\Gamma \vdash_P \xi : \Gamma'$, then $\Gamma' \vdash_P \xi^*_{\Gamma} : \Gamma$.*

Armed with this, we can show that unification preserves types.

Theorem 3. *If $\Delta_0 \vdash P_0$ wf and $(\Delta_0 \vdash P_0) \mapsto (\Delta_1 \vdash P_1)$, then $\Delta_1 \vdash P_1$ wf.*

Proof. Deferred to appendix.

5 Conclusion

We have presented an algorithm for constraint simplification in a higher-order dependent type theory. It admits a proof that it terminates with either a well-typed most general unifier, failure, or a list of unsolvable constraints. An implementation is underway as a modification to the Twelf system [PS99], and there is a significant corpus of existing code on which we can evaluate its efficiency and effectiveness in type reconstruction and logic programming.

A future direction of work we would like to pursue is related to treatment of the case in which the algorithm terminates with remaining constraints. To the extent these constraints can be reified in a type theory as equational reasoning, discovering a set of irreducible constraints is nearly as useful as finding a most general unifier.

Furthermore, having a better theoretical foundation for unification in our setting makes it possible to explore unification in LF to extensions of it, for instance the hybrid logical framework HLF [Ree07].

References

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, April 1997.

[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.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, 1989.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Available as Technical Report CMU-CS-90-134.

[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.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MP92] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proceedings of the Workshop on the Prolog Programming Language*, pages 257–271, 1992.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.

[NPP05] Aleksander Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *ACM Transactions on Computational Logic*, 2005.

[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.

[Pym90] D.J. Pym. *Proofs, search and computation in general logic.* PhD thesis, University of Edinburgh, 1990.

[Ree07] Jason Reed. Hybridizing a logical framework. In P. Blackburn, T. Bolander, T. Braner, V. de Paiva, and J. Villadsen, editors, *Proceedings of the International Workshop on Hybrid Logic (HyLo 2006)*, 2007.

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, 2003.

A Appendix

A.1 Proof of Theorem 2

We require several standard lemmas.

Lemma 8. *Substitutions commute:*

- $[M/x][N/y]X = [[M/x]N/y][M/x]X$
- $[M/x][N \mid S] = [[M/x]N \mid [M/x]S]$

Lemma 9. *Modal and normal substitutions commute:* $\theta[M/x]N = [\theta M/x]\theta N$

Lemma 10. *If an expression X contains no bound variable not in* $\operatorname{rng}\xi$*, then* $[\xi][\xi^{-1}]X = X$.

Lemma 11. $[\xi^{-1}][\xi]X = X$.

Corollary 2 (Injectivity of pattern substitutions). *If* $[\xi]X = [\xi]X'$*, then* $X = X'$.

The proof of the theorem proceeds by case analysis on which transition rule was taken.

Decomposition: These are relatively easy. For example, $\theta(\lambda x.M) = \theta(\lambda x.M')$ iff $\theta M = \theta M'$, and similarly for homomorphic decomposition of the other constructs. If an underscore appears in a rigid position, then no substitution will get rid of it, and no $\equiv$ relation can hold with underscores in it.

Inversion: In this case $P_0 \mapsto P_1$ is $u[\xi] \doteq R \wedge P \mapsto u \doteq [\xi^{-1}]R \wedge P$. In one direction, suppose θ_1 is a $\vec{u}$-solution of P_1, so we have $\theta_1 u = \theta_1[\xi^{-1}]R$. Thus $\theta_1[\xi^{-1}]R = [\xi^{-1}]\theta_1 R$ has no underscores, so $\theta_1 R$ must only have variables from $\operatorname{rng}\xi$. Hit both sides with ξ, and we find $\theta_1 u = \theta_1[\xi]u = [\xi]\theta_1 u = [\xi][\xi^{-1}]\theta_1 R = \theta_1 R$. Thus θ_1 is also a $\vec{u}$-solution of P_0.

In the other direction, suppose θ_0 is a $\vec{u}$-solution of P_0, which means $\theta_0(u[\xi]) = [\xi](\theta_0 u) = \theta_0 R$.

Want to show: $\theta_0 u = \theta_0[\xi^{-1}]R$. It will suffice to show, by injectivity of pattern substitutions, that $\theta_0 R = [\xi]\theta_0[\xi^{-1}]R$. But by assumption we already know that $\theta_0 R$ is in the range of ξ, so $[\xi]\theta_0[\xi^{-1}]R = [\xi][\xi^{-1}]\theta_0 R = \theta_0 R$.

Occurs-Check: There are two transitions. In both cases, observe that a subterm $u[\sigma]$ of R, being atomic, has one of two fates after the substitutions of a putative solution $\theta \models P_0$ are carried out: either it is completely projected away, or $[\sigma]R'$ occurs as a subterm of θR, where $(R'/u) \in \theta$. Thus the result of carrying out a substitution on, for example, a spine-with-hole to yield $(\theta\hat{S})$ is still an expression context, but it may not be linear even if the original $\hat{S}$ was.

Consider the first transition

$$(u \doteq H \cdot \hat{S}\{u[\xi]\}) \wedge P \mapsto (u \doteq H \cdot \hat{S}\{\_\}) \wedge P$$

Let a solution θ of P_0 be given, with $(R'/u) \in \theta$. We know then that $R' \equiv H \cdot (\theta\hat{S})\{[\xi]R'\}$. From this we can see that $(\theta\hat{S})$ must project out its argument, for otherwise R' is a larger term than itself, since $[\xi]R'$ is the same size as R', because ξ is a pattern. Therefore there is no difference between $\theta(\hat{S}\{u[\xi]\})$ and $\hat{S}\{\_\}$, and the latter state still has θ as a solution

Consider the other transition

$$(u \doteq c \cdot \hat{S}_{srig}\{u[\sigma]\}) \wedge P \mapsto \bot$$

Again let a solution θ of P_0 be given, with $(R'/u) \in \theta$. We know $R' = c \cdot (\theta\hat{S}_{rig})\{[\theta\sigma]R'\}$. We may use this equation to expand its own right side again to see

$$R' = c \cdot (\theta\hat{S}_{srig})\{c \cdot ([\theta\sigma]\theta\hat{S}_{rig})\{[\theta\sigma][\theta\sigma]R'\}\}$$

Since $\hat{S}_{srig}$ is strongly rigid, no substitution can project away its argument, and we can continue telescoping this expression to infer that R' has arbitrarily many occurrences of the constant c, a contradiction.

Intersection: Since pattern substitutions only do renaming, any solution of P_0 must refrain from using any variables that are not fixed by ξ. Thus any solution of P_0 is still a solution of P_1.

Pruning: Clearly any solution of the latter state is also a solution of the former. To show that no solutions are lost, consider a solution θ to P_0. It assigns some term R to u. If R has a free occurrence of x, then $\theta(u[\xi]) = [\xi]R$ will have an underscore in it, because $(\_/x) \in \xi$. Since $u[\xi]$ occurs rigidly, this cannot be projected away, and we have a contradiction. Therefore there is a term without occurrence of x, which can be substitued for v in P_1: the solution of P_1 consists of all of θ, plus this additional substitution for v.

Instantiation: By the commutativity of modal and ordinary substitutions.

■

A.2 Proof of Lemma 7

Lemma 12. *If an expression X contains no bound variable not in* $\operatorname{rng}\xi$*, then* $[\xi][\xi^*]X = X$.

Lemma 13. $[\xi^*][\xi]X = X$.

We want to show that ξ^* itself is well-typed. First we note a fact about the way that types of individual variables behave under pattern substitution:

Lemma 14. *Suppose $\Gamma \vdash \xi : \Gamma'$. If $(x/y) \in \xi$, and $x : A \in \Gamma$, then $y : B \in \Gamma'$ such that $[\xi]B = A$.*

Proof. By induction on the typing of ξ. If $\xi = (x/y).\xi_0$, then we read the conclusion directly off the typing

$$\frac{x : [\xi]B \in \Gamma \qquad \Gamma \vdash \xi_0 : \Gamma'}{\Gamma \vdash (x/y).\xi_0 : (\Gamma', y : B)}$$

Otherwise simply apply the induction hypothesis. ■

Having said that, we can now prove

Lemma 15. *If $\Gamma \vdash_P \xi : \Gamma'$, then $\Gamma' \vdash_P \xi^*_\Gamma : \Gamma$.*

Proof. By induction on Γ. The base case is easy. Of the two non-base cases, one is when the variable, say x, does not occur in $\operatorname{rng}\xi$, and $\xi^*_{\Gamma,x:A} = (u[\xi^*_\Gamma]/x).\xi^*_\Gamma$. We can use the derivation arising from the appeal to the induction hypothesis twice to get the derivation

$$\dfrac{\dfrac{\dfrac{u :: \Gamma \vdash A \in \Delta \qquad \Gamma' \vdash_P \xi^*_\Gamma : \Gamma}{\Gamma' \vdash_P u[\xi^*_\Gamma] \Rightarrow [\xi^*_\Gamma]A} \qquad [\xi^*_\Gamma]A \equiv_P [\xi^*_\Gamma]A}{\Gamma' \vdash_P u[\xi^*_\Gamma] \Leftarrow [\xi^*_\Gamma]A} \qquad \Gamma' \vdash_P \xi^*_\Gamma : \Gamma}{\Gamma' \vdash_P (u[\xi^*_\Gamma]/x).\xi^*_\Gamma : (\Gamma, x : A)}$$

In the other case x does actually occur in the range of ξ as $(x/y) \in \xi$, and so $\xi^*_{\Gamma,x:A} = (y/x).\xi^*_\Gamma$. By the previous lemma, pick a B such that $A = [\xi]B$ and $y : B \in \Gamma'$. Then $y : [\xi^*_\Gamma]A (= [\xi^*_\Gamma][\xi]B = B) \in \Gamma'$, which together with a use of the induction hypothesis allows the derivation

$$\dfrac{y : [\xi^*_\Gamma]A \in \Gamma' \qquad \Gamma' \vdash \xi^*_\Gamma : \Gamma}{\Gamma' \vdash (y/x).\xi^*_\Gamma : (\Gamma, x : A)}$$

■

A.3 Proof of Theorem 3

Lemma 16. *If $(M_1; S_1) \doteq (M_2; S_2) \in P_0$, and $M_i' \sqsupseteq M_i$, then $M_1' \equiv_{P_0} M_2'$.*

Proof. Let θ be given such that $\theta \models P_0$. In particular, $(\theta M_1; \theta S_1) \equiv (\theta M_2; \theta S_2)$. But then $\theta M_1 \equiv \theta M_2$, and this equation must be underscore-free, so $\theta M_1' \equiv \theta M_2'$. ■

Lemma 17. *Assume $M_1 \equiv_P M_2$ and $X_1 \equiv_P X_2$ and $S_1 \equiv_P S_2$. Then $[M_1/x]X_1 \equiv_P [M_2/x]X_2$ and $[M_1 \mid S_1] \equiv_P [M_2 \mid S_2]$.*

Lemma 18. *Suppose $A \equiv_P A'$ and $\Gamma \equiv_P \Gamma'$.*

- *If $\Gamma \vdash_P M \Leftarrow A$, then $\Gamma' \vdash_P M \Leftarrow A'$.*
- *If $\Gamma \vdash_P R \Leftarrow C$, then there exists C' such that $\Gamma' \vdash_P R \Leftarrow C'$ and $C \equiv_P C'$.*
- *If $\Gamma \vdash_P S \Leftarrow A > C$, then there exists C' such that $\Gamma' \vdash_P S \Leftarrow A' > C'$ and $C \equiv_P C'$.*

Proof. ■

The main theorem proceeds again by case analysis on the possible steps. If the step is not instantiation, we get the preservation of well-formedness of all the equations we didn't touch from the above corollary, and what needs to be checked is just that the new equations are still well-typed.

Case:

$$(\lambda x.M_1 \doteq \lambda x.M_2) \wedge P \mapsto (M_1 \doteq M_2) \wedge P$$

By assumption there exist $\Gamma, \Pi x{:}A.B, M_1' \sqsupseteq M_1, M_2' \sqsupseteq M_2$ such that

$$\Delta; \Gamma \vdash_{P_0} \lambda x.M_1' \Leftarrow \Pi x{:}A.B$$

$$\Delta; \Gamma \vdash_{P_0} \lambda x.M_2' \Leftarrow \Pi x{:}A.B$$

By inversion, we have

$$\Delta; \Gamma, x : A \vdash_{P_0} M_1' \Leftarrow B$$

$$\Delta; \Gamma, x : A \vdash_{P_0} M_2' \Leftarrow B$$

so after pushing forward with Lemma 1, we have

$$\Delta \vdash_{P_1} M_1 \doteq M_2 \ \mathsf{wf}$$

Case:

$$(M_1; S_1) \doteq (M_2; S_2) \wedge P \mapsto (M_1 \doteq M_2) \wedge (S_1 \doteq S_2) \wedge P$$

By assumption there exist $\Gamma, \Pi x{:}A.B, C_i, M_i' \sqsupseteq M_i, S_i' \sqsupseteq S_i$ such that

$$\Delta; \Gamma \vdash_{P_0} (M_i'; S_i') \Leftarrow \Pi x{:}A.B > C_i$$

and $C_1 \equiv_{P_0} C_2$. By inversion, we have

$$\Delta; \Gamma \vdash_{P_0} M_1' \Leftarrow A \qquad \Delta; \Gamma \vdash_{P_0} S_1' \Leftarrow [M_1'/x]B > C_1$$

$$\Delta; \Gamma \vdash_{P_0} M_2' \Leftarrow A \qquad \Delta; \Gamma \vdash_{P_0} S_2' \Leftarrow [M_2'/x]B > C_2$$

At this stage we observe that the spine tails S_1' and S_2' are at different types because they got different arguments substituted in. Take advantage of P_0 to bring them back together. By Lemma 16, since $(M_1; S_1) \doteq (M_2; S_2) \in P_0$, we know $M_1' \equiv_{P_0} M_2'$. Then $[M_1'/x]B \equiv_{P_0} [M_2'/x]B$ by Lemma 17. Finally, using Lemma 18 and $\Delta; \Gamma \vdash_{P_0} S_1' \Leftarrow [M_1'/x]B > C_1$, we get the existence of $C_3 \equiv_{P_0} C_1$ such that

$$\Delta; \Gamma \vdash_{P_0} S_1' \Leftarrow [M_2'/x]B > C_3$$

Using Lemma 1, push forward all the judgments from $\vdash_{P_0}$ to $\vdash_{P_1}$ to see

$$\Delta \vdash_{P_1} S_1 \doteq S_2 \ \mathsf{wf}$$

Case:

$$u[\xi] \doteq R \wedge P \mapsto u \doteq [\xi^{-1}]R \wedge P$$

Say $u :: (\Gamma' \vdash a \cdot S) \in \Delta$. By assumption there exist $\Gamma, A, R' \sqsupseteq R$ such that

$$\Gamma \vdash_{P_0} R' \Rightarrow A$$

$$\Gamma \vdash_{P_0} \xi : \Gamma'$$

$$[\xi](a \cdot S) \equiv_{P_0} A$$

Note that we didn't have to consider $\xi' \sqsupseteq \xi$, because since ξ is a strong pattern substitution, it has no underscores to fill in.

By Lemma 7, we get

$$\Gamma' \vdash_{P_0} [\xi^*]R' \Rightarrow [\xi^*]A$$

$$[\xi^*][\xi]a \cdot S \equiv_{P_0} [\xi^*]A$$

It is easy to see that $[\xi^*]R' \sqsupseteq [\xi^{-1}]R$. But $[\xi^*][\xi]a \cdot S \equiv_{P_0} a \cdot S$ by Lemma 13.

Case: $(\Delta \vdash (u \doteq R) \wedge P) \mapsto ([R/u]\Delta \vdash (u \leftarrow R) \wedge [R/u]P)$, with the side-condition that $(u \notin FV(R))$.

We have $\Delta' \sqsupseteq \Delta$, $u :: (\Gamma \vdash a \cdot S) \in \Delta'$, $P' \sqsupseteq P$, $R' \sqsupseteq R$ such that $\Delta'; \Gamma \vdash_{P_0} R' \Rightarrow A$, $a \cdot S \equiv_{P_0} A$. So by the modal substitution theorem for $\vdash_{P_0}$, we can see

$$[R/u]\Delta \vdash_{P_0} (u \leftarrow R) \wedge [R/u]P \ \mathsf{wf}$$

which we can push forward with Lemma 1 to get

$$[R/u]\Delta \vdash_{P_1} (u \leftarrow R) \wedge [R/u]P \ \mathsf{wf}$$

■