

Unification with Expansion Variables: Preliminary Results and Problems^{*}

Adam Bakewell and Assaf J. Kfoury

Boston University, <http://types.bu.edu/>

Abstract. Expansion generalises substitution. An *expansion* is a special term whose leaves can be substitutions. Substitutions map term variables to ordinary terms and *expansion variables* to expansions. Expansions (resp., ordinary terms) may contain expansion variables, each applied to an argument expansion (resp., ordinary term). Instances of the unification problem in this setting are constraint sets, where constraints are pairs of ordinary terms, and unifiers are expansions. This problem offers many interesting challenges.

The theory of unification with expansion variables was first considered in relation to the study of systems of intersection types for the λ -calculus. Solving constraint sets, under appropriate conditions, corresponds to type inference for lambda-terms in these systems.

We explain expansions and present a simple rewrite system for unification with expansion variables where ordinary terms uses the intersection type constructors. The simple rewrite system lacks some important properties. We indicate how it can be adapted to: simulate β -reduction, and intersection typing, of λ -terms; be a complete semi-decision procedure for unification; be confluent; produce most-general unifiers.

Every constraint set has a trivial unifier. However, finding a single most-general unifier is often impossible. We study the concept of most-general unifiers and introduce *principal* unifiers, which are easier to construct. Most-general unifiers exist for the unification problem formed by a certain restriction of substitutions, and we give an incomplete variant of simple unification to that finds them.

A second variant system addresses completeness and principality, producing *covering* substitution-unifier sets for constraints (every substitution-unifier is an instance of a set member, and all expansion-unifiers can be obtained from the set). For covering unifier sets we modify the problem to a form of E-unification where the constant ω is the unit of the intersection constructor.

1 Background and Motivation

The study of unification with expansion variables has both practical and theoretical ramifications. The motivation comes from type systems for the λ -calculus

^{*} Work partly funded by NSF grant CCR-0113193 *Implementing Modular Program Analysis via Intersection and Union Types*.

and functional programming languages. The theoretical framework gives rise to a host of problems whose solutions require appropriately adapted algebraic and combinatorial techniques.

The key novelty in our framework is the concept of an *expansion variable*. We start right off with a brief tutorial on expansion concepts in Section 1.1. The connection with type systems is briefly discussed in Section 1.2. The scope and organization of the paper are presented in Sections 1.3 and 1.4.

1.1 Expansion concepts

An *expansion* generalizes the notion of substitution. It is a term whose leaves can be substitutions. For example, an expansion called E_1 may be defined by¹

$$E_1 = (\{\mathbf{a}_1 \mapsto \mathbf{a}_1 \otimes \mathbf{a}_1\} \otimes \mathbf{a}_2) \otimes \{\mathbf{a}_1 \mapsto \mathbf{a}_3\}$$

where \otimes is a binary term constructor and each \mathbf{a}_i is a term variable (T-variable). Replacing each substitution leaf in E_1 by the identity substitution, we obtain the *term part* of E_1 , namely $(\{\} \otimes \mathbf{a}_2) \otimes \{\}$. Applying the expansion E_1 to a T-variable, say \mathbf{a}_1 , written $[E_1]\mathbf{a}_1$, results in the term part of E_1 with each substitution leaf S replaced by $S(\mathbf{a}_1)$. Thus,

$$[E_1]\mathbf{a}_1 = ((\mathbf{a}_1 \otimes \mathbf{a}_1) \otimes \mathbf{a}_2) \otimes \mathbf{a}_3.$$

Applying the expansion E_1 to a plain term τ_1 (without expansion variables) results in the term part of E_1 with each substitution leaf S replaced by $S(\tau_1)$. For example, if $\tau_1 = (\mathbf{a}_1 \otimes \mathbf{a}_1)$, then

$$[E_1]\tau_1 = [E_1](\mathbf{a}_1 \otimes \mathbf{a}_1) = (((\mathbf{a}_1 \otimes \mathbf{a}_1) \otimes (\mathbf{a}_1 \otimes \mathbf{a}_1)) \otimes \mathbf{a}_2) \otimes (\mathbf{a}_3 \otimes \mathbf{a}_3).$$

An *expansion variable* (E-variable) e , then, is a kind of function variable. Occurrences in terms are always applied to one argument. We say that E-variable e *wraps* its argument; the *namespace* of a subterm is the sequence of E-variables encountered on the path to it from the root of the term; and e occurs *outermost* if its namespace is empty. For example, in

$$\tau_2 = e_1(\mathbf{a}_1 \otimes e_2 \mathbf{a}_1),$$

E-variable e_1 occurs outermost; e_2 is in the namespace e_1 and T-variable \mathbf{a}_1 has occurrences in the namespaces e_1 and $e_1 \cdot e_2$. Substitutions are extended to map E-variables to expansions as well as T-variables to terms. Substitution application is defined such that only the outermost variables of the argument are affected. For a more involved example, consider the following expansion E_2 :

$$\begin{aligned} E_2 &= (\{\} \otimes S_1) \otimes S_2 && \text{where} \\ S_1 &= \{\mathbf{a}_1 \mapsto \mathbf{a}_4, e_2 \mapsto E_1\}, \\ S_2 &= \{\mathbf{a}_1 \mapsto \mathbf{a}_5, e_2 \mapsto e_3 \{\mathbf{a}_1 \mapsto \mathbf{a}_6\}\} \end{aligned}$$

¹ The notation $\{\mathbf{a}_1 \mapsto \mathbf{a}_1 \otimes \mathbf{a}_1\}$ defines the support of a total function f ; i.e. $f(x) = x$ for all x except \mathbf{a}_1 where $f(\mathbf{a}_1) = \mathbf{a}_1 \otimes \mathbf{a}_1$. The identity substitution is therefore the function whose support is $\{\}$, the empty set.

and E_1 was defined previously. The next expansion application shows how expansions can apply different substitutions to the same variable in different namespaces: outermost variable e_1 becomes E_2 ; at the leaves of E_2 , different substitutions S_1 and S_2 are applied; then under e_2 in the substitution S_1 , expansion E_1 is used.

$$\begin{aligned}
[\{e_1 \mapsto E_2\}] \tau_2 &= [E_2] (a_1 \otimes e_2 a_1) \\
&= ((a_1 \otimes e_2 a_1) \otimes [S_1] (a_1 \otimes e_2 a_1)) \otimes [S_2] (a_1 \otimes e_2 a_1) \\
&= ((a_1 \otimes e_2 a_1) \otimes (a_4 \otimes [E_1] a_1)) \otimes (a_5 \otimes [e_3 \{a_1 \mapsto a_6\}] a_1) \\
&= ((a_1 \otimes e_2 a_1) \otimes (a_4 \otimes ((a_1 \otimes a_1) \otimes a_2) \otimes a_3)) \otimes (a_5 \otimes e_3 a_6)
\end{aligned}$$

This layering created by E-variables is very useful because it makes the control of variable name disjointness or equality easier, which is often a difficulty in term rewriting that leads to cumbersome solutions like α -renaming and de Bruijn indices.

A graphical summary of the expansions and terms used in this section is in Figure 1. Edges in expansions, as well as edges in terms inherited from applying an expansion, are shown in boldface in Figure 1. Although the examples in this section do not show it, E-variables can occur anywhere in terms, not only at the root or at the leaves (as in term τ_2 above).

In the setting just described, a *unifier* of two terms τ and τ' containing E-variables is an expansion E such that $[E]\tau = [E]\tau'$. Note that standard *first-order unification* is a special case, when the terms τ and τ' contain no E-variables; in this case, a unifier of τ and τ' is a first-order substitution, and therefore trivially an expansion. There is an obvious resemblance between E-variables here and functional variables in *second-order unification*, but the two are different and potential relationships between them are yet to be worked out.

1.2 Developments that led to unification with E-variables

Unification of terms with E-variables in the very general sense introduced above is an interesting theoretical problem, but there is a lack of practical motivation for the general case in the absence of applications (at least so far). Moreover, desirable properties for unification such as the existence of unifiers, principal unifiers, complete or confluent unification systems, and so on, may or may not be achieved depending on the particular form of the general framework we choose.

We study a particular form of unification with E-variables which we previously called *β -unification* in [12, 16] and other more recent reports. Henceforth, ordinary terms are called *types*, as we build terms from the binary type constructors \rightarrow and \sqcap , and a special constant ω . Expansions include \sqcap but not \rightarrow .²

² In research papers on typed λ -calculi and programming languages, the binary constructors \wedge and \cap are commonly used instead of our \sqcap . We prefer to use \sqcap to avoid symbol overloading. We use \cap and \wedge exclusively to denote set intersection and logical “and”, respectively. Contrary to the standard uses of \cap and \wedge , our \sqcap is not always associative, commutative or idempotent.

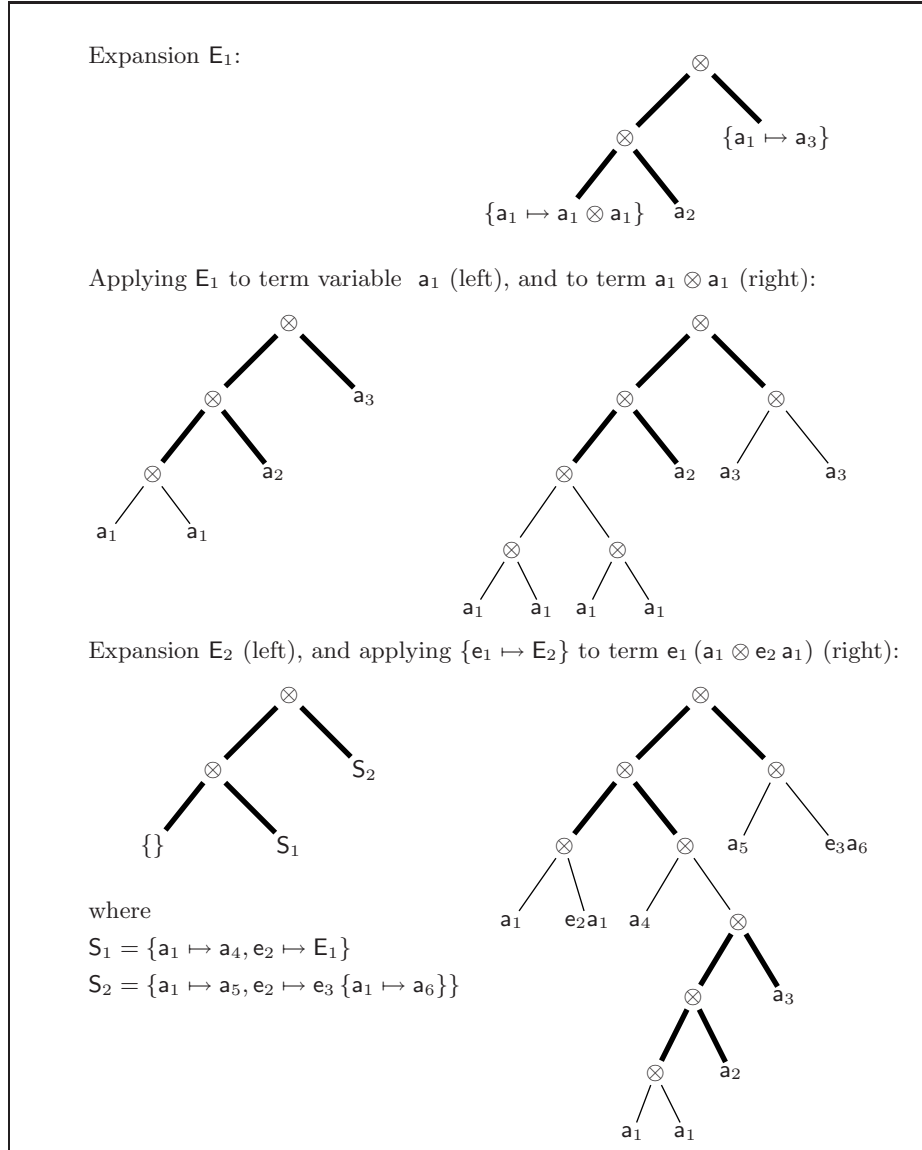


Fig. 1. Pictorial tutorial on expansion concepts (see Section 1.1).

The name β -unification originally referred to a precise connection. There is a constraint set not involving ω , i.e., an instance of ω -free unification with E-variables for every term of the pure λ -calculus. A term is β -strongly normalizing iff the corresponding constraint set has a unifier (not involving ω). This relationship was first expounded in [12], where a somewhat different form of E-variables was introduced. A theoretical presentation of another important connection was

described: Constraint-set normalization reveals an *intersection typing* for the corresponding λ -calculus term. System I, a system of intersection types with E-variables for the pure λ -calculus [16], gave the first procedure for unification with E-variables and applied it to infer *principal typings* (not just *principal types*). Principal typings are important for *compositional* program analysis.

Expansions and E-variables are now being applied and developed in the framework of System E, a more recent system of intersection types with E-variables [6]. This has a more sophisticated and intelligible formulation of expansions and E-variables than System I.³

1.3 What this paper is not

Apart from the connections just reviewed, this paper is not about type systems and the λ -calculus. Nor is it about the duplication operation known as “expansion” in the earlier work on intersection-type systems [23, 19, 20]. Nor is it about “E-variables” as we used them in [12], or even in [13, 16]. A cursory reading of these papers makes clear the differences with the framework of this paper, certainly in the form but also in the contents: The mechanisms we define and bring into play here, in order to formulate problems and resolve them, are not found in these papers.

But they are found in the more recent papers on System E [6, 7, 1], where they are also mixed with considerations related to type systems and the λ -calculus. Our goal here is to disconnect the concepts of *expansion* and *expansion variable* underlying the process of constraint solving from all other considerations in System E. This separate examination facilitates the explanation of the key concepts, which are quite natural, and offer new ways of tackling the still unresolved problems.

1.4 What is new in this paper

- Section 2: the unification problem is extricated from the rest of the System E framework, and formalized in its own right.
- Section 3: a simple rewrite system \xRightarrow{s} for unification with E-variables introduces our methodology for obtaining unifiers, including the key rewriting definitions, constraint-set simplification, and rules to generate partial unifiers. Some significant defects of the simple system are exposed.

³ We gloss over many of the differences in the use of unification with E-variables in System I [14, 13, 16] and System E [6, 7, 1]. A particularly important difference is in the built-in renaming mechanism which is required in System I whenever an expansion is substituted for an E-variable. The renaming mechanism can be omitted in System E, because E-variable application is now used to establish new namespaces. Another benefit is that arbitrary substitutions compose easily in System E, whereas substitution composition is extremely painful and fraught with difficulties in System I.

- Section 4: the notion of a most-general unifier is formalized in our setting; unlike first-order unification, a single most-general unifier is often impossible; the notion of a covering set of unifiers is introduced and we see that a covering set may be infinite; the notion of a principal unifier for a constraint set is defined in a weaker way for greater convenience; to simplify principal unification, we show how to construct expansion-unifiers from a set of substitution-unifiers.
- Section 5: a more refined rewrite system provides unique principal unifiers for many constraint-sets in a restriction of the general framework. It is also complete for certain constraint sets.
- Section 6: completeness and principality in the general case are addressed. The addition of a unit law for the constant ω makes it possible to devise a rewrite system that can generate finite principal unifiers.
- Section 7: the relationships to β -reduction, and intersection typing inference, are exposed; decidable and undecidable variations of the general framework are identified; the non-termination and non-confluence behaviour of the rewrite systems are exposed.
- Section 8: the rewrite systems for unification are summarised, and some important areas for further development are identified.

2 Problem Formulation

Section 2.1 defines the syntactic elements that comprise a β -unification problem. Section 2.2 defines the semantics of expansions. Section 2.3 defines unifiers.

2.1 Syntax

Definition 1 (Variables, E-Var, T-Var). *Expansion variables (E-variables) and type variables (T-variables) are disjoint sets whose elements are indexed by elements of two arbitrary countably infinite⁴ set of indices, \mathbb{I} and \mathbb{J} .*

$$\text{E-Var} = \{e_i \mid i \in \mathbb{I}\} \quad \text{T-Var} = \{a_i \mid i \in \mathbb{J}\}$$

Metavariable e ranges over E-variables, and metavariable a ranges over T-variables. Let metavariable v range over $\text{Var} = \text{E-Var} \cup \text{T-Var}$. \square

Definition 2 (Types, Typ, Typ $^{\rightarrow}$). *Restricted types and types are defined simultaneously as the least sets satisfying:*

$$\begin{aligned} \text{Typ}^{\rightarrow} &\supseteq \text{T-Var} \cup \{ \tau_1 \rightarrow \tau_2 \mid \tau_i \in \text{Typ} \} \\ \text{Typ} &\supseteq \{ \omega \} \cup \text{Typ}^{\rightarrow} \cup \{ \tau_1 \cap \tau_2 \mid \tau_i \in \text{Typ} \} \cup \{ e\tau \mid e \in \text{E-Var} \text{ and } \tau \in \text{Typ} \} \end{aligned}$$

⁴ An infinite set of variables is necessary to allow expansions to merge namespaces without sacrificing the existence of what we call most-general unifiers; what we call principal unifiers can actually be provided with a sufficiently large finite set of variables, the size depending on the constraint.

Metavariable $\bar{\tau}$ ranges over \mathbf{Typ}^\rightarrow and metavariable τ ranges over \mathbf{Typ} . The \mathbf{Typ}^\rightarrow types have constructors of the simply-typed λ -calculus outermost. The \mathbf{Typ} types add the binary intersection type constructor \cap , E-variables and the type constant ω .

The type syntax is ambiguous, e.g., $\alpha \cap \alpha \rightarrow \alpha$ can be understood as two different types. To disambiguate we use parentheses, as in $(\alpha \cap \alpha) \rightarrow \alpha$ or $\alpha \cap (\alpha \rightarrow \alpha)$. Alternatively, we assume that \rightarrow and \cap associate to the right and that \cap binds more tightly than \rightarrow , i.e., $\alpha \cap \alpha \rightarrow \alpha \cap \alpha$ means $(\alpha \cap \alpha) \rightarrow (\alpha \cap \alpha)$. \square

Remark 1. In this paper, the constructor \cap is not commutative, associative or idempotent and it does not absorb ω . These laws, and \cap -distributivity and ω -absorption laws for E-variables, are allowed in System E where the enlarged equality classes they induce are a benefit. It would be possible to add these laws, and develop the theory accordingly, without any significant changes to the results in this paper. But we think the reader will find the technical material more accessible without them (at the loss of a little flexibility): the size of a term M is always greater than the size of any other term N equal to a subterm of M , so inductive proofs work in the obvious way; and E-variables in M cannot move if we take a term equal to M , so there are no surprising equations here. \square

Example 1 (Types and equality). Let $\tau_3 = \omega \rightarrow \mathbf{a}_1 \rightarrow \mathbf{a}_1$. The five types τ_3 , $\tau_3 \cap \tau_3$, $\tau_3 \cap \omega$, $(\omega \rightarrow \mathbf{a}_1) \rightarrow \mathbf{a}_1$, and $\omega \rightarrow \mathbf{a}_2 \rightarrow \mathbf{a}_2$ are mutually unequal. \square

Definition 3 (Substitutions and expansions, Subst, Exp). *Substitutions and expansions are defined simultaneously as the least sets satisfying⁵*

$$\begin{aligned} \mathbf{Subst} &\supseteq \{\square\} \cup \{S : \mathbf{Var} \rightarrow (\mathbf{Typ} \cup \mathbf{Exp}) \mid S(\alpha) \in \mathbf{Typ} \text{ and } S(e) \in \mathbf{Exp}\} \\ \mathbf{Exp} &\supseteq \{\omega\} \cup \mathbf{Subst} \cup \{e E \mid E \in \mathbf{Exp}\} \cup \{E_1 \cap E_2 \mid E_i \in \mathbf{Exp}\} \end{aligned}$$

The nullary expansion ω is an annihilator, as in Section 1.1, and \square is a total function from \mathbf{Var} to $\mathbf{Typ} \cup \mathbf{Exp}$, recursively defined as follows.⁶

$$\square = \{a \mapsto a \mid a \in \mathbf{T-Var}\} \cup \{e \mapsto e \square \mid e \in \mathbf{E-Var}\}$$

The expansion semantics in Definition 5 lift \square to the identity function on \mathbf{Typ} ; \square is the same as $\{\}$ in Section 1.1, but we prefer the less ambiguous \square symbol. Thus substitutions are sort-preserving total functions from \mathbf{Var} to $\mathbf{Typ} \cup \mathbf{Exp}$; expansions are formal expressions built from binary constructor \cap , unary E-variables, and substitutions or ω at the leaves. \square

Remark 2. As substitutions are not formal expressions, strictly speaking, for every $S \in \mathbf{Subst}$ we should have a formal symbol, say $S^\#$, whose interpretation

⁵ If A and B are arbitrary sets, $f : A \rightarrow B$ denotes a total function f from A to B .

⁶ To understand \square informally, take the infinite unwinding of its definition. To simplify this, let $\mathbf{T-Var}$ and $\mathbf{E-Var}$ be the finite sets, $\{\mathbf{a}_1\}$ and $\{\mathbf{e}_1\}$. This also shows that substitutions are higher-order functions — their result for an E-variable arguments can include substitutions.

is the function S . The formal expressions in Exp should be built from $\{\omega, \sqsupset\} \cup \{S^\# \mid S \in \text{Subst}\}$ — not from $\{\omega, \sqsupset\} \cup \text{Subst}$ to be precise — with uses of \cap and applied E-variables. However, we do not need this kind of precision and do not mention the $S^\#$ again.

An alternative would have been to define substitutions as formal expressions too. This approach is used in [6]; it corresponds more directly to the implementation of expansions, but our concern here is to reduce the definitional overhead. \square

Remark 3. The symbols ω and \cap , and all E-variables, each denote two different things: a type constructor and an expansion constructor. The context always disambiguates. In Definition 5, applying an expansion to a type returns a *type*; applying an expansion to an expansion returns an *expansion*. In the presence of the equality rules allowed in System E, mentioned in Remark 1, the dual use of ω may be the source of some confusion, if used carelessly. For example, if the constructor \cap absorbs the type ω , i.e., $\tau \cap \omega = \tau$ as in System E but not here, then for *types* $\omega \cap \omega = \omega$ but for *expansions* $\omega \cap \omega \neq \omega$. It is possible to adjust the syntax in order to distinguish between the two different uses of ω , but at the price of a more complicated presentation. \square

Remark 4. T-variables and \rightarrow are not expansion constructors. So $S(a) = e \omega \cap a$ and $S(e) = e \omega \cap S_1$ are possible, but $S(e) = e \omega \rightarrow S_1$ is not. \square

Definition 4 (Substitution support, $\text{support} : \text{Subst} \rightarrow \text{Var}$). *The support of substitution S is:*
 $\text{support}(S) = \{a \in \text{T-Var} \mid S(a) \neq a\} \cup \{e \in \text{E-Var} \mid S(e) \neq e \sqsupset\}$ \square

Remark 5. For convenience, we may define a substitution by enumerating its restriction to its support, as in Section 1.1. So $\{a_1 \mapsto a_2\}$ is not partial; it is equal to \sqsupset on all variables apart from a_1 . \square

2.2 Expansion

Substitutions and expansions can be applied to any type or expansion. The definition of application is the same for the common constructors of these sorts.

Definition 5 (Expansion application, $[\cdot] \cdot$). *The application of an arbitrary expansion E to a type τ (respectively, an expansion E_1), written $[E]\tau$ (resp.,*

$[E] E_1$), returns a type (resp., an expansion). The inductive definition follows.

<p><i>Applying substitutions to types:</i></p> $\begin{aligned} [S] \alpha &= S(\alpha) \\ [S] (\tau_1 \rightarrow \tau_2) &= [S] \tau_1 \rightarrow [S] \tau_2 \\ [S] \omega &= \omega \\ [S] (e \tau) &= [S(e)] \tau \\ [S] (\tau_1 \cap \tau_2) &= [S] \tau_1 \cap [S] \tau_2 \end{aligned}$	<p><i>Applying substitutions to expansions:</i></p> $\begin{aligned} [S] S_1 &= \{v \mapsto [S](S_1(v)) \mid v \in \text{Var}\} \\ [S] \omega &= \omega \\ [S] (e E) &= [S(e)] E \\ [S] (E_1 \cap E_2) &= [S] E_1 \cap [S] E_2 \end{aligned}$
<p><i>Applying expansions to types:</i></p> $\begin{aligned} [\omega] \tau &= \omega \\ [e E] \tau &= e([E] \tau) \\ [E_1 \cap E_2] \tau &= [E_1] \tau \cap [E_2] \tau \end{aligned}$	<p><i>Applying expansions to expansions:</i></p> $\begin{aligned} [\omega] E &= \omega \\ [e E_1] E &= e([E_1] E) \\ [E_1 \cap E_2] E &= [E_1] E \cap [E_2] E \end{aligned}$

The formal definition matches the description in the introduction: substitutions distribute down to the outermost variables of their argument then get applied; expansions distribute their argument down into their leaves and then substitute or annihilate them. This interleaving effects the namespace layering. \square

Example 2 (Expansion application). $\{\{e_2 \mapsto \square \cap e_1 \square\} e_2 a$
 $= [\square \cap e_1 \square] a = [\square] a \cap [e_1 \square] a = a \cap e_1 [\square] a = a \cap e_1 a$. Another expansion and two applications of it:

$$\begin{aligned} E &= \{e_1 \mapsto \omega \cap e_3 \square \cap e_4 \square, e_3 \mapsto e_3 \{a_1 \mapsto e_2 a_1 \rightarrow a_2\}\} \\ [E] e_1 (e_2 a_1 \rightarrow a_2) &= \omega \cap e_3 (e_2 a_1 \rightarrow a_2) \cap e_4 (e_2 a_1 \rightarrow a_2) \\ [E] (\omega \cap e_3 a_1 \cap e_4 (a_4 \rightarrow a_5)) &= \omega \cap e_3 (e_2 a_1 \rightarrow a_2) \cap e_4 (a_4 \rightarrow a_5) \end{aligned}$$

\square

Proposition 1 (\square lifts to the identity function). For all types τ and expansions E :

1. $[\square] \tau = \tau$,
2. $[\square] E = E$, and
3. $[E] \square = E$.

\square

Proposition 2 (Expansion application is associative). For all types τ and expansions E_i : $[[E_1] E_2] \tau = [E_1] ([E_2] \tau)$, and $[[E_1] E_2] E_3 = [E_1] ([E_2] E_3)$. \square

Notation 1 For expansions E_1 and E_2 , let $(E_1; E_2)$ be shorthand for the composition $[E_2] E_1$ — that is E_1 then E_2 , not $[E_1] E_2$. By Proposition 2, “;” is an associative operation, so $E_1; E_2; E_3; E_4$ is unambiguous for example. That is, all five of its bracketings, shown below, produce the same result.

- $((E_1; E_2); E_3); E_4$
- $(E_1; E_2); (E_3; E_4)$
- $E_1; (E_2; (E_3; E_4))$

- $(E_1; (E_2; E_3)); E_4$
- $E_1; ((E_2; E_3); E_4)$ □

Remark 6. Expansion composition and expansion union are quite distinct notions: often, if $E_1 = E_2 \cup E_3$ there is no E_4 such that $E_1 = E_2; E_4$; often, if $E_1 = E_2; E_3$ there is no E_4 such that $E_1 = E_2 \cup E_4$. Composition can make an expansion more specific; union can make an expansion more general. □

2.3 Unifiers

In unification theory, a problem instance is a set of constraints, and a unifier is a solution. We define these standard concepts to fix our notation.

Definition 6 (Constraints and constraint sets, Constraint, CstrSet). A constraint $\tau_1 \doteq \tau_2$ is an unordered pair of types (i.e. \doteq is commutative.) The set **Constraint** comprises all constraints and **CstrSet** all constraint sets⁷ An instance of the (β -unification) problem is a finite set of such constraints. Metavariables δ and Δ , possibly decorated, range over constraints and constraint sets, respectively. □

Example 3 (Constraint set). Introducing a running example (recall that $\tau_3 = \omega \rightarrow a_3 \rightarrow a_3$): $\Delta_1 = \{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \cap e_1 a_2 \rightarrow a_4 \doteq e_2 \tau_3 \rightarrow a_4\}$ □

Definition 7 (Solved constraint set, solved : CstrSet \rightarrow Boolean). The predicate *solved* is defined on constraints and extended to sets as follows.

$$\begin{aligned} \text{solved}(\tau_1 \doteq \tau_2) &\text{ iff } \tau_1 = \tau_2; \\ \text{solved}(\Delta) &\text{ iff solved}(\delta) \text{ for every } \delta \in \Delta. \end{aligned} \quad \square$$

Definition 8 (Unifiers). If δ is the constraint $\tau_1 \doteq \tau_2$ and E an expansion, then $[E]\delta$ denotes the constraint $[E]\tau_1 \doteq [E]\tau_2$. If Δ is a constraint set, then $[E]\Delta$ denotes the constraint set $\{[E]\delta \mid \delta \in \Delta\}$. Expansion E is a unifier of constraint set Δ iff $\text{solved}([E]\Delta)$. □

Example 4 (Unifiers). Consider the following substitutions:

$$S_3 = \{e_2 \mapsto \square \cap e_1 \{a_2 \mapsto \tau_3\}, a_1 \mapsto \tau_3, e_1 \mapsto e_1 \{a_2 \mapsto \tau_3\}\}$$

$$S_4 = \{a_4 \mapsto a_1 \rightarrow a_1, e_1 \mapsto \omega\}$$

$$S_5 = \{e_2 \mapsto \square \cap \omega, a_1 \mapsto \tau_3, a_4 \mapsto a_1 \rightarrow a_1, e_1 \mapsto \omega\}$$

Although unequal in general, $(S_3; S_4)$ and S_5 both unify Δ_1 :

$$[S_3]\Delta_1 = \{\tau_3 \doteq e_1 \tau_3 \rightarrow a_4, \tau_3 \cap e_1 \tau_3 \rightarrow a_4 \doteq \tau_3 \cap e_1 \tau_3 \rightarrow a_4\}$$

$$[S_3; S_4]\Delta_1 = [S_5]\Delta_1 = \{\tau_3 \doteq \tau_3, \tau_3 \cap \omega \rightarrow a_1 \rightarrow a_1 \doteq \tau_3 \cap \omega \rightarrow a_1 \rightarrow a_1\} \quad \square$$

⁷ $\mathbb{P}(S)$ denotes the powerset of set S .

3 Simple Unification by Rewriting

Traditional unification systems aim to produce a *most general* unifier for all soluble constraint sets. For unification with expansion variables, achieving most-general, and complete, systems are rather complex subjects that we defer to Section 5, and Section 6. Here we lay the foundations by studying a very simple rewrite system for unification, which interleaves two phases. Section 3.1 defines simplification to factor common structure. Section 3.2 generates partial unifiers, and dissects some properties of the simple system through examples. But first, some helpful notation.

Notation 2

1. Let \vec{e} be a metavariable ranging over the set of all finite sequences of E -variables, including the empty sequence ε .
2. A constraint of the form $\vec{e}\tau_1 \doteq \vec{e}\tau_2$, where τ_1 and τ_2 do not have applications of the same E -variable outermost, may be written $\vec{e}(\tau_1 \doteq \tau_2)$. Call \vec{e} the prefix of such a constraint.
3. Let the notation e/S be shorthand for the substitution $\{e \mapsto eS\}$. Think of such a substitution as acting under e , or in namespace e .
4. We extend the “/” notation to arbitrary sequences of E -variables: $\vec{e} \cdot e/S$ means $\vec{e}/(e/S)$ and ε/S means S . \square

3.1 Simplification Rules

Definition 9 (Constraint simplify, simplify : CstrSet \rightarrow CstrSet). simplify factors out common structure and eliminates solved constraints:

$$\text{simplify}(\Delta) = \{ \delta \mid \delta \in \text{factor}(\Delta) \text{ and not solved}(\delta) \}$$

$$\text{factor}(\{\delta_1, \dots, \delta_n\}) = \text{factor}(\delta_1) \cup \dots \cup \text{factor}(\delta_n)$$

$$\text{factor}(\delta) = \begin{cases} \text{factor}(\{\vec{e}(\tau_1 \doteq \tau_2), \vec{e}(\tau'_1 \doteq \tau'_2)\}) & \text{if } \delta = \vec{e}(\tau_1 \cap \tau'_1 \doteq \tau_2 \cap \tau'_2) \\ \text{factor}(\{\vec{e}(\tau_1 \doteq \tau_2), \vec{e}(\tau'_1 \doteq \tau'_2)\}) & \text{if } \delta = \vec{e}(\tau_1 \rightarrow \tau'_1 \doteq \tau_2 \rightarrow \tau'_2) \\ \{\delta\} & \text{otherwise.} \end{cases}$$

For an arbitrary set Δ , $\text{solved}(\Delta)$ is true iff $\text{simplify}(\Delta) = \emptyset$. \square

Example 5 (Constraint simplify). For constraint set Δ_1 from Example 3 we have:

$$\text{simplify}(\Delta_1) = \{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \cap e_1 a_2 \doteq e_2 \tau_3\}.$$

$$\text{Another example: } \text{simplify}(a_1 \cap e_1 a_2 \doteq \tau_3 \cap e_1 \tau_3) = \{a_1 \doteq \tau_3, e_1(a_2 \doteq \tau_3)\}.$$

$$\text{simplify}(\{\omega \cap e_3(e_2 a_1 \rightarrow a_2) \cap e_4(e_2 a_1 \rightarrow a_2) \doteq \omega \cap e_3(e_2 a_1 \rightarrow a_2) \cap e_4(a_4 \rightarrow a_5)\}) \\ = \{e_4(a_4 \doteq e_2 a_1), e_4(a_2 \doteq a_5)\} \quad \square$$

Theorem 3 (Simplify sound). $\text{solved}([E] \Delta)$ iff $\text{solved}([E] (\text{simplify}(\Delta)))$. \square

3.2 Rewrite Rules

The rewrite relation $\xrightarrow[S]{S}$ defines direct reductions, which partly solve some constraint with substitution S . A few preliminary concepts are needed.

Definition 10 (Topmost expansion extraction, $\text{extract} : \text{Typ} \rightarrow \text{Exp}$). By induction on types:

$$\begin{aligned} \text{extract}(\bar{\tau}) &= \square \\ \text{extract}(\omega) &= \omega \\ \text{extract}(e \tau) &= e(\text{extract}(\tau)) \\ \text{extract}(\tau_1 \sqcap \tau_2) &= \text{extract}(\tau_1) \sqcap \text{extract}(\tau_2) \end{aligned}$$

In words, the topmost expansion of τ is the largest tree of expansion constructors from the root of τ down to where non-expansion constructors are encountered, with an identity substitution at each leaf. \square

Example 6 (extraction). $\text{extract}(\mathbf{a}_1 \sqcap \mathbf{e}_1 \mathbf{a}_2) = \square \sqcap \mathbf{e}_1 \square$
 $\text{extract}(\omega \sqcap \mathbf{e}_3 \mathbf{a}_1 \sqcap \mathbf{e}_4 (\mathbf{a}_4 \rightarrow \mathbf{a}_5)) = \omega \sqcap \mathbf{e}_3 \square \sqcap \mathbf{e}_4 \square$

\square

The expansion ω trivially unifies all constraints. We define the $\text{zip}\omega$ unifier to give a substitution-unifier for any constraint with a prefix length at least one. This substitutes is the most general trivial unifier; it substitutes ω for the deepest prefix variable (intuitively, it *zips* up the E-variables of the prefix and deploys ω at the first point of difference).

Definition 11 (Zip unify, $\text{zip}\omega : \text{Constraint} \rightarrow \text{Subst}$).

$$\text{zip}\omega(\bar{e}e(\tau_1 \doteq \tau_2)) = \bar{e}/\{e \mapsto \omega\}$$

\square

Example 7 (Zip unify).

$$\text{zip}\omega(\mathbf{e}_3(\mathbf{e}_2 \mathbf{a}_1 \rightarrow \mathbf{a}_2) \doteq \mathbf{e}_3 \mathbf{a}_1) = \mathbf{e}_3/\omega \text{ But } \text{zip}\omega(\mathbf{e}_2 \mathbf{a}_1 \rightarrow \mathbf{a}_2 \doteq \mathbf{a}_1) \text{ is undefined. } \square$$

Remark 7. Initially we are interested in finding *substitution*-unifiers, which we later extend to provide expansion unifiers, so it is not necessary to allow $\text{zip}\omega$ to produce ω when the constraint prefix is empty. Later definitions assume that $\text{zip}\omega$ produces a substitution not an expansion. Moreover, note the \bar{e}/S notation does not allow \bar{e}/ω . \square

The simple outer-part unifier improves on $\text{zip}\omega$ by creating substitutions for variables that occur outermost within the prefix — recall that by Notation 2, there is no E-variable e such that $\tau_1 = e \tau'_1$ and $\tau_2 = e \tau'_2$ in Definition 12. We generate *substitution*-unifiers: forming expansion-unifiers from a set of substitution-unifiers is easily done separately.

Definition 12 (Simple outer part unifier, $\text{unifier} \subseteq \text{Constraint} \times \text{Subst}$). Defined non-deterministically by:

$$\bar{e}(\tau_1 \doteq \tau_2) \xrightarrow{\text{unifier}} \begin{cases} \bar{e}/\{a \mapsto \tau\} & \text{if } \{\tau_1, \tau_2\} = \{a, \tau\} & \text{(T-unify)} \\ \bar{e}/\{e \mapsto \text{extract}(\tau)\} & \text{if } \{\tau_1, \tau_2\} = \{e \tau', \tau\} & \text{(E-unify)} \\ \text{zip}\omega(\bar{e}(\tau_1 \doteq \tau_2)) & \text{if } \bar{e} = \bar{e}_1 \cdot e & \text{(Z-unify)} \end{cases}$$

- (T-unify) attempts to solve constraints where one side is a T-variable.
- (E-unify) solves the topmost part of a constraint by substituting the topmost expansion of one side for the E-variable e on the other side.

- (Z-unify) trivially solves any constraint with a non-empty prefix. This unifier annihilates everything in the namespace \vec{e} . \square

Remark 8. The (T-unify) and (E-unify) rules do not include an *occur check*. This would be apt to prevent perpetual rewriting of constraints like $a \doteq a \cap a$ and $ea \doteq ea \cap ea$, and forcing a (Z-unify) solution where possible. It would be inapt for $ea_1 \doteq ea \rightarrow ea$, which can be solved by using (E-unify) and (T-unify) to generate $\{e \mapsto \square\}; \{a_1 \mapsto a \rightarrow a\}$.

The (Z-unify) rule is also essential for solving constraints of the form $\vec{e}e(\tau_1 \rightarrow \tau_2 \doteq \omega)$, $\vec{e}e(\tau_1 \rightarrow \tau_2 \doteq \tau_3 \cap \tau_3)$, or $\vec{e}e(\omega \doteq \tau_1 \cap \tau_2)$, which have no solution by substitution for any variable deeper than e . \square

Example 8 (Simple outer part unifiers). $(a_1 \cap e_1 a_2) \doteq e_2 \tau_3 \xrightarrow{\text{unifier}} \{e_2 \mapsto \square \cap e_1 \square\}$
 To illustrate the non-determinism in unifier: $e(a \doteq e_1 a) \xrightarrow{\text{unifier}} e/\{a \mapsto e_1 a\}$ and $e(a \doteq e_1 a) \xrightarrow{\text{unifier}} e/\{e_1 \mapsto \square\}$ and $e(a \doteq e_1 a) \xrightarrow{\text{unifier}} \{e \mapsto \omega\}$ And $\{e_1 e_3 a_1 \doteq e_2 \omega\} \xrightarrow{\text{unifier}} \{e_1 \mapsto e_2 \omega\}$ and $\{e_1 e_3 a_1 \doteq e_2 \omega\} \xrightarrow{\text{unifier}} \{e_2 \mapsto e_1 e_3 \square\}$. \square

The direct constraint set reduction relation $\xrightarrow[S]{S}$ simplifies, outer-part unifies then simplifies; a reduction is many direct reductions.

Definition 13 (Direct simple reduction, $\xRightarrow[S]{S} \subseteq \text{CstrSet} \times \text{CstrSet}$).

$$\Delta \xrightarrow[S]{S} \text{simplify}([S] \Delta) \quad \text{if there is } \delta \in \text{simplify}(\Delta) \text{ such that } \delta \xrightarrow{\text{unifier}} S.$$

We define $\Delta \xRightarrow[S]{S} \Delta_1$ if there is a substitution S such that $\Delta \xrightarrow[S]{S} \Delta_1$. \square

Example 9 (Direct simple reduction). Example 5 simplified Δ_1 . Now we reduce it to Δ_2 .

$$\Delta_1 \xrightarrow[S]{\{e_2 \mapsto \square \cap e_1 \square\}} \{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \doteq \tau_3, e_1(a_2 \doteq \tau_3)\} = \Delta_2$$

These two simple reductions solve another example:

$$\Delta_3 = \{\omega \doteq e_1 a_2, a_1 \rightarrow a_1 \doteq a_4, e_1(a_2 \doteq \tau_3)\} \xrightarrow[S]{\{e_1 \mapsto \omega\}} \{a_1 \rightarrow a_1 \doteq a_4\} \xrightarrow[S]{\{a_4 \mapsto a_1 \rightarrow a_1\}} \emptyset$$

This one shows one way to solve another: $\{e_1 e_3 a_1 \doteq e_2 \omega\} \xrightarrow[S]{\{e_1 \mapsto e_2 \omega\}} \emptyset$ \square

Definition 14 (Simple reduction, $\xRightarrow[S]{S} \subseteq \text{CstrSet} \times \text{CstrSet}$). Reduction $\xRightarrow[S]{S}$ is the reflexive transitive closure of $\xrightarrow[S]{S}$. Formally, we first define $\xrightarrow[S]{S}$ as follows. For all Δ, Δ_1 and Δ_2 :

$$\Delta \xrightarrow[S]{\square} \text{simplify}(\Delta) \quad \text{and} \quad \Delta \xrightarrow[S]{S_1; S_2} \Delta_2 \quad \text{if } \Delta \xrightarrow[S]{S_1} \Delta_1 \text{ and } \Delta_1 \xrightarrow[S]{S_2} \Delta_2.$$

We then define $\Delta \xRightarrow[S]{S} \Delta_1$ if there is a substitution S such that $\Delta \xrightarrow[S]{S} \Delta_1$. \square

Theorem 4 (Soundness of simple reduction). If $\Delta \xrightarrow[S]{S} \emptyset$ then $\text{solved}([S] \Delta)$. \square

Thus normalising simple reductions resulting in \emptyset produce a unifier. The length of a reduction is the number of direct reduction steps it comprises. Infinite reductions have an unbounded length; normalising reductions have a finite length.

Example 10 (Simple reduction). We normalise Δ_2 from Example 9 in four steps:

$$\begin{aligned} \Delta_2 &\xrightarrow[\mathfrak{s}]{\{a_1 \mapsto \tau_3\}} \{\omega \doteq e_1 a_2, a_1 \rightarrow a_1 \doteq a_4, e_1 (a_2 \doteq \tau_3)\} & (\Delta_3) \\ &\xrightarrow[\mathfrak{s}]{e_1 / \{a_2 \mapsto \tau_3\}} \{\omega \doteq e_1 \tau_3, a_1 \rightarrow a_1 \doteq a_4\} & (\Delta_4) \\ &\xrightarrow[\mathfrak{s}]{\{a_4 \mapsto a_1 \rightarrow a_1, e_1 \mapsto \omega\}} \emptyset \end{aligned}$$

□

Remark 9 (Incompleteness). There are constraints, such as $\omega \doteq a \rightarrow a$, which have expansion-unifiers, but $\xrightarrow[\mathfrak{s}]{} \Rightarrow$ cannot solve. There are also constraints with substitution-unifiers that $\xrightarrow[\mathfrak{s}]{} \Rightarrow$ cannot solve; for an idea of the issues involved, witness:

$$\Delta_5 = \{e (\omega \doteq a \rightarrow a), e a \doteq a \cap a\}$$

This is soluble with $S_6 = \{e \mapsto \omega \cap \omega, a \mapsto \omega\}$ but the simple system has various inadequacies, preventing the discovery of such unifiers: it always assigns outermost E-variables the *topmost* extraction, rather than allowing lesser extractions; it does not rename variables under an expanded E-variable; and, most relevant for the above case, it does not introduce fresh E-variables so that such solutions can be built up gradually, e.g. $[S_6] \Delta_5 = [\{e \mapsto e_1 \cap e_1\}; \{e_1 \mapsto \omega\}; \{a \mapsto \omega\}] \Delta_5$.

For an example of why less than the topmost extraction can be needed, consider:

$$\Delta_9 = \{e ((a \rightarrow a) \cap (a \rightarrow a)) \doteq (a \rightarrow a) \cap (a \rightarrow a) \cap (a \rightarrow a) \cap (a \rightarrow a)\}$$

is solved by $\{e \mapsto \square \cap \square\}$ but (E-unify) uses maximal extraction to obtain the substitution $\{e \mapsto \square \cap \square \cap \square \cap \square\}$, and Δ_5 reduces to the stuck constraint set $\{(a \rightarrow a) \cap (a \rightarrow a) \doteq (a \rightarrow a)\}$.

For an example of why renamings can be needed, consider:

$$\Delta_{10} = \{e a \doteq a \rightarrow a\}$$

This is solved by $\{e \mapsto \{a \mapsto a \rightarrow a\}\}$, but (E-unify) destroys the possibility of such a solution by merging the outermost and e namespaces.

We address completeness in Section 6 where a more complex version of extract is used to cover all possibilities. □

Remark 10 (Normalisation). Some constraints have infinite reductions. For example, the following reduction forms a kind of repeatable cycle.

$$\begin{array}{l}
\Delta_6 = \quad \{e_1 ((e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)) \rightarrow e_4 a_2) \dot{=} e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1), \\
\quad \quad \quad e_4 a_2 \dot{=} a_2\} \\
\begin{array}{l} \xrightarrow[\mathcal{S}]{\{e_4 \mapsto \square\}} \\ \xrightarrow[\mathcal{S}]{\{e_1 \mapsto e_2 (\square \cap e_3 \square)\}} \end{array} \quad \{e_1 ((e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)) \rightarrow e_4 a_2) \dot{=} e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)\} \\
\begin{array}{l} \xrightarrow[\mathcal{S}]{S} \\ \xrightarrow[\mathcal{S}]{S} \end{array} \quad \{e_2 (e_3 a_1 \dot{=} e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)), e_2 (e_4 a_2 \dot{=} a_2), \\
\quad \quad \quad e_2 e_3 (((e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)) \rightarrow e_4 a_2) \dot{=} a_1)\} \\
\begin{array}{l} \xrightarrow[\mathcal{S}]{S} \\ \xrightarrow[\mathcal{S}]{S} \end{array} \quad [e_2] \Delta_6 \text{ where } S = e_2 e_3 / \{a_1 \mapsto (e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)) \rightarrow e_4 a_2\}
\end{array}$$

So normalisation depends on the reduction strategy because Δ_6 is solved by another strategy that generates the substitutions $\{e_2 \mapsto e_1 \square\}$; $\{e_1 \mapsto \omega\}$; $\{e_4 \mapsto \square\}$. Infinite reduction is an essential ingredient for the correspondence to β -reduction (Section 7.1) and it is not a problem for this form of unification per se because we can usually force a solution — use a strategy that tries (T-unify) and (E-unify) to search for precise unifiers, then try (Z-unify) when some complexity limit is exceeded. \square

4 Most General and Principal Unifiers

We want *most general*, unifiers. In first-order unification it is possible to provide a single most-general unifier for every soluble constraint. In our setting this is not true. Section 4.2 defines *most-general unifier sets*. Section 4.3 motivates what we call a *principal* unifier (set), which is easier to construct. Section 4.4 shows how a set of substitution-unifiers may be extended to form all expansion unifiers. But first, Section 4.1 introduces some useful classes of substitutions including a definition of isomorphisms.

4.1 Renamings and Isomorphisms

Definition 20 adapts the standard definition of an isomorphism to our setting. Isomorphisms are a special case of a larger class of substitutions explained first, which we call “renamings”. These are basically substitutions that only map variables to variables.⁸

Definition 15 (Renamings, Renaming). *Let \mathcal{V} be a set of E-variables and \mathcal{S} a set of substitutions. We define $\mathcal{V} \cdot \mathcal{S}$ as follows: $\mathcal{V} \cdot \mathcal{S} = \{e S \mid e \in \mathcal{V} \text{ and } S \in \mathcal{S}\}$. The set $\mathcal{V} \cdot \mathcal{S}$ is a subset of Exp. We define the set of renamings as the least such that:*

$$\begin{array}{l}
\text{Renaming} \supseteq \{\square\} \cup \{R : \text{Var} \rightarrow (\text{T-Var} \cup \text{E-Var} \cdot \text{Renaming}) \mid \\
\quad \quad \quad R(\alpha) \in \text{T-Var} \text{ and } R(e) \in \text{E-Var} \cdot \text{Renaming}\}
\end{array}$$

⁸ Renaming a term can result in a new term that is not isomorphic (or “ α -equivalent”) to the original because variable merging is possible. Definition 18 distinguishes possibly merging (surjective) and non-merging (injective) renamings (in some related literature, the term renaming denotes what we call an injective renaming).

The Renaming set is therefore a particular subset of Subst . Informally, applying a renaming R to a type τ produces a type $[R]\tau$ obtained from τ by renaming all variables (not only those occurring outermost). \square

The domain of a renaming is Var .earlier. In the range of a renaming R we only include the T-variables that result from applying R to T-variables the applied E-variables that result from applying R to E-variables.

Definition 16 (Renaming range, $\text{range} : \text{Subst} \rightarrow \mathbb{P}(\text{Var})$).

$$\text{range}(R) = \{ R(a) \mid a \in \text{Var} \} \cup \{ e' \mid e \in \text{E-Var and } R(e) = e' R' \}. \quad \square$$

Renamings that act non-trivially only in the outer namespace are often useful. These *outer renamings* are specified by the following non-recursive definition.

Definition 17 (Outer renamings, Outer-Renaming).

$$\text{Outer-Renaming} \supseteq \{ \square \} \cup \{ R \in \text{Renaming} \mid R(e) \in \text{E-Var} \cdot \{ \square \} \} \quad \square$$

Proposition 3 (Characterization of renamings).

1. Substitution S is a renaming iff: for all \vec{e} and α , there are \vec{e}' and α' such that $[S](\vec{e}\alpha) = \vec{e}'\alpha'$ and the lengths of \vec{e} and \vec{e}' are the same.
2. Substitution S is an outer renaming iff: for all \vec{e} , e and α , there is an e' such that $[S](e\vec{e}\alpha) = e'\vec{e}\alpha$. \square

Definition 18 (Injective and surjective renamings). A renaming R is injective if

- $a \neq a'$ implies $R(a) \neq R(a')$;
- $e \neq e'$ and $R(e) = e_1 R_1$ and $R(e') = e'_1 R'_1$ implies $e_1 \neq e'_1$ and R_1 is injective and R'_1 is injective.

A renaming R is surjective if

- for all a there is a a' such that $R(a') = a$;
- for all e there are e' and R' such that $R(e') = e R'$ and R' is surjective. \square

Definition 19 (Inverse of renaming, $(R)^{-1}$). The inverse of injective renaming R , denoted $(R)^{-1}$, is recursively defined by:

$$(R)^{-1} = \{ \alpha' \mapsto \alpha \mid R(\alpha) = \alpha' \} \cup \{ e' \mapsto e (R')^{-1} \mid R(e) = e' R' \} \quad \square$$

Remark 11. A surjective renaming like $\{ a_1 \mapsto a_2, a_2 \mapsto a_2 \}$, or just $\{ a_1 \mapsto a_2 \}$, has no inverse because of the variable merging.

It may be that $(R)^{-1}$ exists and $R; (R)^{-1} = \square$ but $(R)^{-1}; R \neq \square$ because of variable merging in $(R)^{-1}$. For example, if $R = \{ a_i \mapsto a_{i \times 2} \mid i \in \mathbb{I} \}$ (where we assume the index set \mathbb{I} is the natural numbers), then $[R; (R)^{-1}] a_1 = a_1$ but $[(R)^{-1}; R] a_1 = a_2$. If R has an inverse which inverts to R , we call it bijective, or an isomorphism. \square

Definition 20 (Isomorphism). A renaming I is an isomorphism if there is a renaming I' such that $I; I' = \square$ and $I'; I = \square$.

Isomorphisms are lifted to mappings from Typ to Typ , and from Exp to Exp , according to Definition 5. If $\tau = [I]\tau'$ (respectively, $E = [I]E'$) for some isomorphism, we say that τ and τ' are isomorphic types (resp., E and E' are isomorphic expansions). \square

Proposition 4 (Isomorphisms).

1. *Isomorphisms are injective and surjective renamings.*
2. \square *is an isomorphism* \square

4.2 Most General Unifiers

Remark 12. We define a *most-general* unifier of Δ as any such that no other unifier is more general. That is, every other unifier is either an instance of the most general, or else the most general is not an instance of the other unifier. (note E_3 is always a substitution in Definition 21). \square

Definition 21 (Most general unifier).

- S_1 is a most general substitution-unifier of Δ if $\text{solved}([S_1] \Delta)$; and for all S_2, S_3 , if $\text{solved}([S_2] \Delta)$ and $S_2 = S_1; S_3$, then S_3 is an isomorphism.
- E_1 is a most general expansion-unifier of Δ if $\text{solved}([E_1] \Delta)$; and for all E_2, E_3 , if $\text{solved}([E_2] \Delta)$ and $E_2 = E_1; E_3$, then E_3 is an isomorphism. \square

Example 11 (Several most-general unifiers). There can be several *non-isomorphic* most-general unifiers. There are basically two ways to unify the following constraint set Δ , the S_1 way and the S_2 way.

$$\Delta = \{e_1 \ a_1 \doteq \omega\} \quad S_1 = \{e_1 \mapsto \omega\} \quad S_2 = \{e_1 \mapsto \{a_1 \mapsto \omega\}\}$$

The second, S_2 , is not most-general for reasons we consider later. But for now the important observation is that neither S_1 nor S_2 is a substitution (or expansion) instance of the other: intuitively, S_2 eliminates e_1 so it is impossible to devise a substitution that will insert expansion ω wherever e_1 used to occur; S_1 annihilates every e_1 so it is impossible to devise a substitution to restore the annihilated information. There are at least two non-isomorphic substitution unifiers of Δ . \square

Remark 13. We define a *covering* unifier set of Δ to be any set of unifiers such that every unifier of Δ is an instance of some set member. \square

Notation 5 *Let the metavariable \mathcal{S} range over sets of substitutions and the metavariable \mathcal{E} range over sets of expansions.* \square

Definition 22 (Covering unifier set).

- *Substitution set \mathcal{S} is a covering substitution-unifier set of Δ if $\text{solved}([S_1] \Delta)$ implies there are $S_2 \in \mathcal{S}$ and S_3 such that $S_1 = S_2; S_3$.*
- *Expansion set \mathcal{E} is a covering expansion-unifier set of Δ if $\text{solved}([E_1] \Delta)$ implies there are $E_2 \in \mathcal{E}$ and E_3 such that $E_1 = E_2; E_3$.* \square

Remark 14. If a covering expansion-unifier set of Δ is empty then Δ is insoluble. If a covering unifier set of Δ is a singleton then Δ has a most-general unifier, in the usual sense of of first-order unification. If a singleton covering unifier set is impossible we seek a finite one; ideally a *minimal* finite set. \square

Definition 23 (Minimal unifier set). *Substitution unifier set \mathcal{S} (resp., expansion unifier set \mathcal{E}) is minimal if $\{S_1, S_2\} \subseteq \mathcal{S}$ and $S_1 \neq S_2$ implies there is no S_3 such that $S_1 = S_2; S_3$ (resp., $\{E_1, E_2\} \subseteq \mathcal{E}$ and $E_1 \neq E_2$ implies there is no E_3 such that $E_1 = E_2; E_3$).* \square

Remark 15. Section 6 constructs covering unifier sets. They are not minimal in general. It may be possible to minimise (perhaps non-uniquely) such sets by matching — removing members that are instances of other member — but this subject is left to further work. \square

4.3 Principal Unifiers

Remark 16. Constructing most-general unifiers can be cumbersome so we introduce the weaker idea of a *principal* unifier. The following example illustrates the problem. \square

Example 12 (Most general unifiers). Consider constraint set Δ .

$$\Delta = \{e_1 \ a_1 \doteq a_1 \rightarrow a_1\}$$

It has the following most general substitution-unifier.

$$S_1 = \{e_1 \mapsto (\{a_1 \mapsto (a_1 \rightarrow a_1)\} \cup S_2)\} \cup S_7$$

where S_2 and S_7 are injective outer renamings such that:
the support of S_2 is $\text{Var} \setminus \{a_1\}$;
the support of S_7 is $\text{Var} \setminus \{e_1\}$;
 $\text{range}(S_2)$ and $\text{range}(S_7)$ are disjoint.

This difficult construction is needed because any substitution-unifier S_8 of Δ must eliminate e_1 . So S_8 is obtained from specialising S_1 according to $S_8 = S_1; ((S_2)^{-1}; S_8(e_1)); ((S_7)^{-1}; S_8)$. So it is important not to merge any variables in the e_1 namespace with their namesakes in the empty namespace. In particular, $S_3 = \{e_1 \mapsto \{a_1 \mapsto (a_1 \rightarrow a_1)\}\}$ is not a most general unifier of Δ : another unifier is $S_4 = \{e_1 \mapsto (\{a_1 \mapsto (a_1 \rightarrow a_1)\}), a_2 \mapsto a_1\}$. Now $[S_3] e_1 \ a_2 \cap a_2 = a_2 \cap a_2$ and $[S_4] e_1 \ a_2 \cap a_2 = a_2 \cap a_1$ so there cannot be a substitution S_5 such that $S_4 = S_3; S_5$.

If substitutions were restricted to have finite supports while Var remains infinite, as in System E, most general unifiers are impossible as we can extend the different treatment of the same variable in different namespaces, by an arbitrary unifier, to any number of variables. \square

Remark 17. Basically we call an expansion-unifier E_1 of Δ principal if $\text{solved}([E_2] \Delta)$ implies there is an expansion E_3 such that $[E_2] \Delta = [E_1; E_3] \Delta$ (and similarly for substitution-unifiers). That is, principal unifiers ignore the effect of unifiers on variables not in Δ . \square

Remark 18. To clarify the relationship between principal and most general, and to simplify later proofs, our formal definition uses an associative, commutative and idempotent expansion *join* operator. \square

Definition 24 (Expansion join, \sqcup). A partial operator defined inductively:

$$\begin{aligned}
\omega & \sqcup \omega & = \omega \\
(E_1 \cap E_2) \sqcup (E_3 \cap E_4) & = (E_1 \sqcup E_3) \cap (E_2 \sqcup E_4) \\
e E_1 & \sqcup e E_2 & = e (E_1 \sqcup E_2) \\
S_1 & \sqcup S_2 & = S, \text{ if for all } v, S(v) \neq \perp \text{ where:} \\
S(v) & = \begin{cases} S_1(v) & \text{if } v \notin \text{support}(S_2) \text{ or } S_1(a) = S_2(a) \\ S_2(v) & \text{if } v \notin \text{support}(S_1) \\ S_1(e) \sqcup S_2(e) & \text{if } e \in \text{support}(S_1) \cap \text{support}(S_2) \cap \text{E-Var} \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

□

Remark 19. It is also useful to define the partial order \sqsubseteq (i.e. a reflexive transitive relation) of which \sqcup is the semi-lattice join operator. Strictly, \perp should be included in **Exp** and then \perp is the supremum of the \sqsubseteq semi-lattice. There is no unique lattice infimum because expansions with differing shapes outermost have no meet (and we choose not to invent one), e.g., there is no E such that $E \sqsupseteq \omega$ and $E \sqsupseteq \square \cap \square$. Thus, the topmost elements of the lattice are all the expansions whose substitution elements are all \square . □

Definition 25 (Sub-expansion partial order, \sqsubseteq). We define $E_1 \sqsubseteq E_2$ (equivalently, $E_2 \sqsupseteq E_1$) iff there is an E_3 such that $E_2 = E_1 \sqcup E_3$. □

Example 13 (Join and sub-expansions). Substitutions S_3 and S_4 map e_1 to an intersection of two substitutions: $S_3 = \{e_1 \mapsto \{a_1 \mapsto a_3\} \cap \{a_1 \mapsto a_4\}\}$; $S_4 = \{e_1 \mapsto \{a_2 \mapsto a_5\} \cap \{a_2 \mapsto a_6\}\}$. Therefore $S_3 \sqcup \square = S_3$, and $\square \sqsubseteq S_3$, and $S_3 \sqcup S_4 = \{e_1 \mapsto \{a_1 \mapsto a_3, a_2 \mapsto a_5\} \cap \{a_1 \mapsto a_4, a_2 \mapsto a_6\}\}$. □

Remark 20. The domain restriction of an expansion E for a constraint or type Y is the “smallest” expansion in the \sqsubseteq partial order that acts as E on Y . The restriction $E|_Y$ is unique by Theorem 6. □

Definition 26 (Expansion domain restriction, $|$). Let E be an expansion and Y a type or constraint. The restriction of E for Y , denoted $E|_Y$, is the expansion E_1 such that:

1. $[E_1]Y = [E]Y$, and
2. for every expansion E_2 such that $[E_2]Y = [E]Y$, $E_1 \subseteq E_2$. □

Example 14 (Expansion restriction). Referring back to the constraint and unifiers of Example 12, $S_1|_\Delta = S_3$ and $S_4|_\delta = S_3$. □

Theorem 6 (Existence of unique restriction). For every expansion E and type or constraint Y , there is an expansion $E|_Y$. □

Remark 21. An expansion unifier of Δ can only be principal if it has no proper sub-expansion which is also a unifier of Δ . In addition, it must not be a proper expansion instance of any other unifier (for substitution-unifiers the statement is analogous). □

Definition 27 (Principal unifier). *Expansion E_1 is a principal unifier of Δ iff:*

1. $\text{solved}([E_1] \Delta)$, i.e., E_1 is a unifier of Δ ,
2. for all E_2 such that $\text{solved}([E_2] \Delta)$, there is an expansion E_3 such that $E_1; E_3 \sqsubseteq E_2$. \square

Remark 22. For all most general unifiers there is a principal sub-expansion and for all principal unifiers there is a most-general super-unifier. The constructive proof is similar to the infinite-support renaming technique of Example 12. \square

Theorem 7 (Every most general unifier restricts to be principal). *If E_1 is a most general unifier of Δ then $E_1|_{\Delta}$ is a principal unifier of Δ . \square*

Theorem 8 (Every principal unifier extends to be most general). *If E_1 is a principal unifier of Δ then there is an $E_1 \sqsubseteq E_2$ such that E_2 is a most general unifier of Δ . \square*

4.4 Covering Substitution Unifier Sets

Remark 23. A minimal covering unifier set for a constraint set can be infinite. Typically when there is a covering substitution-unifier set with more than one member all covering expansion-unifier sets are infinite. However, all expansion-unifiers can be formed by taking all combinations of instances of a covering substitution-unifier set. This justifies only considering substitution-unifiers: expansion-unifiers do not really reveal any more information about how to unify a constraint set not revealed by its substitution-unifiers. \square

Definition 28 (Expansion-unifiers from substitution-unifiers, $\text{sustoeus} : \mathbb{P}(\text{Subst}) \rightarrow \mathbb{P}(\text{Exp})$). *$\text{sustoeus}(\mathcal{S})$ is the expansion set formed by taking all combinations of a substitution set \mathcal{S} ; it is the least set such that:*

$$\text{sustoeus}(\mathcal{S}) \supseteq \{ \omega \} \cup \mathcal{S} \cup \{ e E \mid E \in \text{sustoeus}(\mathcal{S}) \} \cup \{ E_1 \cap E_2 \mid \{ E_1, E_2 \} \subseteq \text{sustoeus}(\mathcal{S}) \}$$

Proposition 5 (Substitution-unifier sets suffice). *If \mathcal{S} is a covering substitution-unifier set of Δ then $\text{sustoeus}(\mathcal{S})$ is a covering expansion-unifier set of Δ . \square*

Remark 24. Using sustoeus to describe all expansion-unifiers is beneficial when there is a finite covering substitution-unifier set but only infinite covering expansion-unifier sets. For example, if $\{S_1, S_2\}$ is a complete covering substitution-unifier set for Δ then we cannot obtain all expansion-unifiers from any finite set of expansions combining S_1 and S_2 . In particular, the expansion $E = e_1 S_1 \cap e_2 S_2$ cannot be expanded to give unifiers like $S_1 \cap S_1$ (The closest we can get is $E; \{e_1 \mapsto \square \cap \square, e_2 \mapsto \omega\}$, giving $S_1 \cap S_1 \cap \omega$). This motivates the addition of a law to make ω the unit of \cap , and the subsequent developments in Section 6. As additional motivation, the following example shows that covering substitution-unifier sets can also be infinite. \square

Example 15 (Infinite covering substitution-unifier set). Consider the following constraint and some of its substitution-unifiers, S_7 to S_5 .

$$\begin{aligned}
& e_1 e_3 a_1 \doteq e_2 \omega \\
S_1 &= \{e_3 \mapsto \omega\} \\
S_2 &= \{e_3 \mapsto \{a_1 \mapsto \omega\}\} \\
S_3 &= \{e_1 \mapsto e_2 S_1\} \\
S_4 &= \{e_1 \mapsto e_2 S_2\} \\
S_5 &= \{e_1 \mapsto e_2 (S_1 \cap S_2), e_2 \mapsto e_2 (\square \cap \square)\} \\
S_6 &= \{e_1 \mapsto e_2 (S_1 \cap S_2 \cap S_1 \cap S_2), e_2 \mapsto e_2 (\square \cap \square \cap \square \cap \square)\} \\
S_7 &= \{e_1 \mapsto e_2 (S_1 \cap S_2 \cap S_1 \cap S_2), e_2 \mapsto e_2 (\square \cap \square \cap \square \cap \square)\}
\end{aligned}$$

Unifiers S_3 , S_4 and S_5 are mutually unobtainable from each other. Worse, S_1 and S_2 may be combined in any number of ways such that any covering substitution-unifier set must be infinite: some, like $S_6 = S_5; \{e_2 \mapsto e_2 (\square \cap \square)\}$ can be obtained from smaller unifiers; others, like S_7 cannot. \square

5 Restricted Principal Unifiers

This section derives a variant of simple reduction that incorporates renamings to produce single principal unifiers (that is, singleton covering principal substitution-unifier sets) for a large class of constraints, by imposing a restriction on substitutions and expansions.

Remark 25. This overcomes the difficulty raised in Example 11 as follows. The only permissible restricted-substitution-unifier of $e_3 a_1 \doteq \omega$ is $S_1 = \{e_3 \mapsto \omega\}$; all unifiers formed from S_2 are disallowed.

For the full example, $e_1 e_3 a_1 \doteq e_2 \omega$, the system we present gives no result (it is incomplete). A single principal unifier is still impossible. the following unifiers cannot be obtained from each other $S_1 = \{e_1 \mapsto \{e_3 \mapsto e_2 \omega\}\}$ $S_2 = \{e_1 \mapsto e_2 S_1\}$ (including renamings to make these more general does not solve the problem as the deletion of e_1 in both cases makes it impossible to introduce e_2 underneath correctly after applying S_1 , and impossible to eliminate e_2 correctly after applying S_2). \square

Definition 29 (Restricted substitutions and expansions, RSub, RExp). *These sets restrict Subst and Exp, they are defined simultaneously as the least sets satisfying:*

$$\begin{aligned}
\text{RSub} &\supseteq \{\square\} \cup \{\bar{S} : \text{Var} \rightarrow (\text{Typ}^{\rightarrow} \cup \text{RExp}) \mid \bar{S}(\alpha) \in \text{Typ}^{\rightarrow}, \bar{S}(e) \in \text{RExp}\} \\
\text{RExp} &\supseteq \{\omega\} \cup \text{RSub} \cup \{e \bar{E} \mid \bar{E} \in \text{RExp}\} \cup \{\bar{E}_1 \cap \bar{E}_2 \mid \{\bar{E}_1, \bar{E}_2\} \subseteq \text{RExp}\} \quad \square
\end{aligned}$$

Remark 26. We define the reextract relation to replace the extract function. It takes the topmost extraction but puts a different renaming for the variables in set V at each leaf. \square

Definition 30 (Topmost renaming expansion extraction, $\xrightarrow{\text{reextract}}$). *By induction on types, for an arbitrary finite set V of variables:*

$$\begin{aligned}
(\bar{\tau}, V) & \xrightarrow{\text{retract}} (S, \text{range}(S)) \text{ for injective renaming } S \text{ if } \text{range}(S) \cap V = \emptyset \\
(\omega, V) & \xrightarrow{\text{retract}} (\omega, \emptyset) \\
(\tau_1 \cap \tau_2, V) & \xrightarrow{\text{retract}} (E_1 \cap E_2, V_1 \cup V_2) \text{ if } (\tau_i, V) \xrightarrow{\text{retract}} (E_i, V_i) \text{ and } V_1 \cap V_2 = \emptyset \\
(e \tau, V) & \xrightarrow{\text{retract}} (e E, V_1) \text{ if } (\tau, V) \xrightarrow{\text{retract}} (E, V_1) \quad \square
\end{aligned}$$

Remark 27. The runifier relation modifies the unifier relation as needed. It uses the `ovars` function to find all variables in the empty namespace of types. \square

Definition 31 (Outer variables, `ovars` : $\text{Typ} \rightarrow \mathbb{P}(\text{Var})$). *By induction on types:*

$$\begin{aligned}
\text{ovars}(a) & = \{a\} \\
\text{ovars}(\tau_1 \rightarrow \tau_2) & = \text{ovars}(\tau_1) \cup \text{ovars}(\tau_2) \\
\text{ovars}(\omega) & = \emptyset \\
\text{ovars}(e \tau) & = \{e\} \\
\text{ovars}(\tau_1 \cap \tau_2) & = \text{ovars}(\tau_1) \cup \text{ovars}(\tau_2)
\end{aligned}$$

\square

Example 16 (Outer variables). $\text{ovars}(e_1 (e_2 a_1 \rightarrow a_2)) = \{e_1\}$ and $\text{ovars}(\omega \cap e_3 a_1 \cap e_4 (a_4 \rightarrow a_5)) = \{e_3, e_4\}$. \square

Definition 32 (Topmost outer part renaming unifier, `runifier` $\subseteq \text{Constraint} \times \mathbb{P}(\text{Var}) \times \text{Subst}$). *By case analysis:*

$$\begin{aligned}
(\vec{e}(\tau_1 \doteq \tau_2), V) & \xrightarrow{\text{runifier}} \\
\left\{ \begin{array}{ll} \vec{e}/\{a \mapsto \bar{\tau}\} & \text{if } \{\tau_1, \tau_2\} = \{a, \bar{\tau}\} \text{ and } a \notin \text{ovars}(\bar{\tau}) \\ \vec{e}/\{e \mapsto E\} & \text{if } \{\tau_1, \tau_2\} = \{e \bar{\tau}, \tau\} \text{ and } (\tau, V) \xrightarrow{\text{retract}} (E, V') \\ \square & \text{if } \{\tau_1, \tau_2\} = \{e \tau_3, \tau_4\} \text{ and } \tau_3 \notin \text{Typ}^\rightarrow \text{ and } \tau_4 \neq e_1 \bar{\tau} \\ \text{zip}\omega(\vec{e}(\tau_1 \doteq \tau_2)) & \text{otherwise, if } \vec{e} = \vec{e}_1 \cdot e \end{array} \right. \quad \begin{array}{l} \text{(R T-unify)} \\ \text{(R E-unify)} \\ \text{(R N-unify)} \\ \text{(R Z-unify)} \end{array}
\end{aligned}$$

\square

Remark 28. `runifier` incorporates a number of important changes.

- (R T-unify) is only applicable when the non-T-variable type is in Typ^\rightarrow . As a nicety, it includes an occur check so that (R Z-unify) will solve constraints that would otherwise proliferate under an endless series of (R T-unify) steps.
- (R E-unify) uses the `retract` relation and it is only applicable when E-variable e wraps a type in Typ^\rightarrow . The constraint forms that could benefit from an occur check are not identified to keep the rule simple.
- (R N-unify) is a new rule to catch problematic constraints forms that were handled by (E-unify) but are excluded by (R E-unify). The result \square should be read as an indicator that this constraint is stuck, so ignore it (this is intended to be easier to understand than the alternative of putting complex side conditions on other rules); a solution to the whole constraint set may still be possible after uses of other rules on non-stuck constraints. The excluded forms must not be given to (R Z-unify) as principality could easily be lost (even on a constraint set that just contains (R N-unify) form constraints). Moreover, single principal solutions for most of the excluded forms are not possible. Of course, there are many special-case exceptions

like the set $\{e_1 \omega\}e_2 \omega$, so the single principal unification system described in this section cannot succeed for every constraint set that has a single principal unifier.

- (R Z-unify) is applicable where there is an occur check problem or differing outer constructors as before, it also applies to constraints of the form $a \doteq \tau$ where $\tau \notin \text{Typ}^{\rightarrow}$. \square

Remark 29. The “renaming”, or “restricted”, constraint reduction relation modifies \Rightarrow to use `runifier`. Note that (R N-unify) does not induce a reduction. It uses `factor` instead of `simplify` to keep the solved parts of the constraint set. Solved parts are kept to prevent accidental loss of a principal solution by reintroduction of variable names. For example, we must not use `simplify` to eliminate the solved constraint in $\{a_1 \doteq a_1, e a_2 \doteq a_3\}$ then rewrite with $\{e \mapsto \{a_2 \mapsto a_1\}\}$, which reintroduces a_1 and leads to a non-principal unifier. Therefore solved constraints are not passed to `runifier`. In practice, a more sophisticated renaming strategy to track which variables actually occur in the affected namespaces and replaces them with new names from a fresh supply is more apt. \square

Definition 33 (Restricted renaming reduction, $\Rightarrow_{\text{r}} \subseteq \text{CstrSet} \times \text{CstrSet}$).

We define the relation $\xrightarrow{\text{r}}^{\text{S}}$ as follows, where `Vars` gives the set of variables in its argument: $\Delta \xrightarrow{\text{r}}^{\text{S}} \text{factor}([\bar{S}] \Delta)$ if $\delta \in \text{factor}(\Delta)$ and not `solved`(δ) and $(\delta, \text{Vars}(\Delta)) \xrightarrow{\text{runifier}} \bar{S}$ and $\bar{S} \neq \square$. We define the reflexive transitive closure $\xrightarrow{\text{r}}^{\text{S}}$ by analogy with $\xrightarrow{\text{S}}$. \square

Example 17 (Restricted renaming reduction). In this example the substitutions generated are all the same as for simple reduction, except at (3) where two renamings are introduced by the renaming extraction (let $S_1(a_3) = a_6$ and $S_2(a_3) = a_7$). We show the constraints in fully simplified form but it is important in practice to only `factor` them to avoid loss of principal solutions through variable merging.

$$\begin{array}{l}
\{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \cap e_1 a_2 \doteq e_2 \tau_3, a_4 \doteq a_5\} \\
\begin{array}{l} \xrightarrow{\text{r}}^{\{a_5 \mapsto a_4\}} \\ \xrightarrow{\text{r}}^{\{e_2 \mapsto S_1 \cap e_1 S_2\}} \end{array} \{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \cap e_1 a_2 \doteq e_2 \tau_3\} \\
\begin{array}{l} \xrightarrow{\text{r}}^{\{a_1 \mapsto \omega \rightarrow a_6 \rightarrow a_6\}} \\ \xrightarrow{\text{r}}^{\{e_1 / \{a_2 \mapsto \omega \rightarrow a_7 \rightarrow a_7\}\}} \end{array} \{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \doteq (\omega \rightarrow (a_6 \rightarrow a_6)), e_1 (a_2 \doteq (\omega \rightarrow (a_7 \rightarrow a_7)))\} \\
\begin{array}{l} \xrightarrow{\text{r}}^{\{a_1 \mapsto \omega \rightarrow a_6 \rightarrow a_6\}} \\ \xrightarrow{\text{r}}^{\{e_1 \mapsto \omega\}} \end{array} \{\omega \doteq e_1 a_2, a_6 \rightarrow a_6 \doteq a_4, e_1 (a_2 \doteq (\omega \rightarrow (a_7 \rightarrow a_7)))\} \\
\begin{array}{l} \xrightarrow{\text{r}}^{\{a_4 \mapsto a_6 \rightarrow a_6\}} \\ \xrightarrow{\text{r}}^{\{e_1 \mapsto \omega\}} \end{array} \{\omega \doteq e_1 (\omega \rightarrow a_7 \rightarrow a_7), a_6 \rightarrow a_6 \doteq a_4\} \\
\begin{array}{l} \xrightarrow{\text{r}}^{\{a_4 \mapsto a_6 \rightarrow a_6\}} \\ \xrightarrow{\text{r}}^{\{e_1 \mapsto \omega\}} \end{array} \{\omega \doteq e_1 (\omega \rightarrow (a_7 \rightarrow a_7))\} \\
\begin{array}{l} \xrightarrow{\text{r}}^{\{e_1 \mapsto \omega\}} \\ \xrightarrow{\text{r}}^{\{e_1 \mapsto \omega\}} \end{array} \emptyset
\end{array}$$

\square

Remark 30. Restricted reduction produces single principal *restricted* unifiers, as shown by Theorem 9, using the following lemma. \square

Lemma 1 (Outer part unifiers expand to any restricted unifier). *If $\Delta \xrightarrow{\bar{E}} \Delta_1$ and $\text{solved}([\bar{E}'] \Delta)$ then there is an \bar{E}'' such that $\bar{E}'|_{\Delta} = \bar{E}; \bar{E}''$. \square*

Theorem 9 (\Rightarrow gives principal restricted unifiers). *If $\Delta \xrightarrow{\bar{E}} \Delta_1$ and $\text{solved}(\Delta_1)$ and $\text{solved}([\bar{E}'] \Delta)$ then there is an expansion \bar{E}'' such that $\bar{E}'|_{\Delta} = \bar{E}; \bar{E}''$. \square*

6 Complete Unification and Matching

This section addresses completeness. A naive complete system is easily obtained by enumerating expansions and testing whether they are unifiers. Section 6.2 shows how to solve constraints to produce covering substitution-unifier sets by rewriting in a style similar to the simple and renaming systems. To do this simply we have to enrich the theory, as well as using many rules with a high degree of non-determinism. As preparation, Section 6.1 gives rules to solve the simpler type matching problem.

6.1 Matching

Remark 31. Matching — i.e. finding an expansion E such that $[E] \tau_1 = \tau_2$ when there is one — is decided by the rules in Definition 34. using the join operator. This contrasts with the tricky higher-order matching problem, known decidable only to fourth order [18]. \square

Definition 34 (Type matching, $\text{match} \subseteq \text{Typ} \times \text{Typ} \times \text{Exp}$). *Defined non-deterministically by:*

$$\begin{array}{l}
(\tau_1 \quad , \omega \quad) \xrightarrow{\text{match}} \omega \\
(\tau_1 \quad , \tau_2 \cap \tau_3) \xrightarrow{\text{match}} E_1 \cap E_2 \quad \text{if } (\tau_1, \tau_2) \xrightarrow{\text{match}} E_1 \text{ and } (\tau_1, \tau_3) \xrightarrow{\text{match}} E_2 \\
(\tau_1 \quad , e \tau_2 \quad) \xrightarrow{\text{match}} e E \quad \text{if } (\tau_1, \tau_2) \xrightarrow{\text{match}} E \\
(a \quad , \tau_1 \quad) \xrightarrow{\text{match}} \{a \mapsto \tau_1\} \\
(e \tau_1 \quad , \tau_2 \quad) \xrightarrow{\text{match}} \{e \mapsto E\} \quad \text{if } (\tau_1, \tau_2) \xrightarrow{\text{match}} E \\
(\omega \quad , \omega \quad) \xrightarrow{\text{match}} \square \\
(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2) \xrightarrow{\text{match}} S_1 \sqcup S_2 \quad \text{if } (\tau_i, \tau'_i) \xrightarrow{\text{match}} S_i \text{ and } [S_1 \sqcup S_2] \tau_i = \tau'_i \\
(\tau_1 \cap \tau_2, \tau'_1 \cap \tau'_2) \xrightarrow{\text{match}} S_1 \sqcup S_2 \quad \text{if } (\tau_i, \tau'_i) \xrightarrow{\text{match}} S_i \text{ and } [S_1 \sqcup S_2] \tau_i = \tau'_i \quad \square
\end{array}$$

Example 18 (Type matching). Substitutions S_7 and S_8 each map e_1 to an intersection: $S_7 = \{e_1 \mapsto \{a_1 \mapsto a_3\} \cap \{a_1 \mapsto a_4\}\}$; $S_8 = \{e_1 \mapsto \{a_2 \mapsto a_5\} \cap \{a_2 \mapsto a_6\}\}$. Thus $S_7 \sqcup S_8 = \{e_1 \mapsto \{a_1 \mapsto a_3, a_2 \mapsto a_5\} \cap \{a_1 \mapsto a_4, a_2 \mapsto a_6\}\}$. Letting $\tau_1 = e_1 a_1 \mapsto e_1 a_2$, and $\tau_2 = a_1 \cap a_4 \mapsto a_5 \cap a_6$, we have $(\tau_1, \tau_2) \xrightarrow{\text{match}} S_7 \sqcup S_8$, and $(\tau_1, \tau_2) \xrightarrow{\text{match}} \{e_1 \mapsto \{a_1 \mapsto a_1 \cap a_4, a_2 \mapsto a_5 \cap a_6\}\}$. \square

Theorem 10 (Type matching is sound and complete).

1. *If $(\tau_1, \tau_2) \xrightarrow{\text{match}} E$ then $[E] \tau_1 = \tau_2$.*
2. *If $[E] \tau_1 = \tau_2$ then $(\tau_1, \tau_2) \xrightarrow{\text{match}} E_1$ for some E_1 .* \square

6.2 Covering Unifier Sets

Remark 32. Example 15 shows that a finite covering set of substitution-unifiers is impossible for many constraint sets. In this section we present a constraint solving system broadly in the style of the simple and renaming systems. But this is designed to allow every unifier for a constraint set to be an instance of that inferred by some reduction path. We could follow the principle of the matching rules and just enumerate all possibilities, but instead we consider a more interesting and efficient alternative. We do not study the efficiency issues here, but concentrate on the completeness property and one simple change to the theory that enables this new approach. We define a new equivalence relation $\stackrel{\text{unit}}{=}$ which incorporates algebraic laws to make ω the unit of \cap . \square

Definition 35 (Unit law and unit equivalence, $\stackrel{\text{unit}}{=}$). *The symbol $\stackrel{\text{unit}}{=}$ denotes the equivalence relation formed by taking the compatible closure of: $\tau \cap \omega \stackrel{\text{unit}}{=} \tau$ and $\omega \cap \tau \stackrel{\text{unit}}{=} \tau$.* \square

Remark 33. Many types match the pattern $\tau_1 \cap \tau_2$ under $\stackrel{\text{unit}}{=}$. Such as: ω , \mathbf{a}_1 , $\omega \rightarrow \omega$, $\mathbf{e}_1 (\mathbf{a}_1 \cap \mathbf{a}_2)$, $\mathbf{e}_1 \mathbf{a}_1 \cap \mathbf{e}_1 \mathbf{a}_2$, and so on. We want to exclude then from matching certain rules for \cap decomposition. So we define a meta-constructor (i.e. a type *pattern* constructor) \sqcap which only matches types that must be constructed with an \cap . \square

Definition 36 (True intersection meta-constructor, $\sqcap : \text{Typ} \times \text{Typ} \rightarrow \text{Typ}$). $\tau_1 \sqcap \tau_2$ if $\tau_1 \not\stackrel{\text{unit}}{=} \omega$ and $\tau_2 \not\stackrel{\text{unit}}{=} \omega$. \square

Remark 34. Constraint factorisation is redefined as a relation to allow for the non-determinism introduced by the unit law. Now $\text{solved}(\Delta)$ and $\Delta \xrightarrow{\text{simplify}} \Delta_1$ and not $\text{solved}(\Delta_1)$ is possible. \square

Definition 37 (Factor relation, $\text{rfactor} \subseteq \text{CstrSet} \times \text{CstrSet}$). *Factor out common structure. Defined non-deterministically:*

$$\begin{aligned} & \{\delta_i\}_{i=1}^n \xrightarrow{\text{rfactor}} \Delta_1 \cup \dots \cup \Delta_n \text{ if } \delta_i \xrightarrow{\text{rfactor}} \Delta_i \\ & \delta \xrightarrow{\text{rfactor}} \begin{cases} \Delta & \text{if } \delta \stackrel{\text{unit}}{=} \vec{e}(\tau'_1 \sqcap \tau''_1 \doteq \tau'_2 \cap \tau''_2) \text{ and } \{\vec{e}(\tau'_1 \doteq \tau'_2), \vec{e}(\tau''_1 \doteq \tau''_2)\} \xrightarrow{\text{rfactor}} \Delta \\ \Delta & \text{if } \delta \stackrel{\text{unit}}{=} \vec{e}(\tau_1 \rightarrow \tau'_1 \doteq \tau_2 \rightarrow \tau'_2) \text{ and } \{\vec{e}(\tau_1 \doteq \tau_2), \vec{e}(\tau'_1 \doteq \tau'_2)\} \xrightarrow{\text{rfactor}} \Delta \\ \{\delta\} & \text{if } \delta \neq \tau_1 \rightarrow \tau'_1 \doteq \tau_2 \rightarrow \tau'_2 \text{ and } \delta \neq \tau_1 \sqcap \tau'_1 \doteq \tau_2 \sqcap \tau'_2 \end{cases} \end{aligned}$$

\square

Example 19 (Finite covering unifier sets possible). The constraint $\mathbf{e}_1 \mathbf{e}_3 \mathbf{a}_1 \doteq \mathbf{e}_2 \omega$ considered in Example 15 has a single covering unifier with the unit law: \square

Remark 35. The following *opus* relation takes the part of *extract* and *unifier* in the rewrite system introduced below. It maps every constraint δ , and some variable set V (assumed to contain all variables in Δ where $\delta \in \Delta$) to a set of substitutions. Lemma 2 shows the covering property that if $\delta \xrightarrow{\text{opus}} \mathcal{S}$ then all substitution-unifiers are instances of some member of \mathcal{S} . As with *extract*, members of \mathcal{S} only solve an outer part of δ . \square

Definition 38 (Outer-part covering unifier sets, $\text{opus} \subseteq \text{Constraint} \times$

$\mathbb{P}(\text{Var}) \times \mathbb{P}(\text{Subst})$). *A set of outer-part substitution-unifiers for every constraint:*

$$\begin{aligned}
(e(\tau_1 \doteq \tau_2), V) &\xrightarrow{\text{opus}} \left\{ \begin{array}{l} \{e \mapsto e e_1 S_1 \cap \dots \cap e_n S_n\} \\ | \\ \{e_1, \dots, e_n\} \not\subseteq \text{Vars}(\Delta) \\ \text{and } (\tau_1 \doteq \tau_2, V) \xrightarrow{\text{opus}} (\mathcal{S} \cup \{S_i\}_{i=1}^n) \end{array} \right\} & \text{(C E-unify)} \\
(a \doteq \tau, V) &\xrightarrow{\text{opus}} \left\{ \begin{array}{l} \{a \mapsto \tau\} \\ | \\ a \notin \text{ovars}(\tau) \end{array} \right\} & \text{(C T-unify)} \\
(e\tau \doteq \tau_1 \rightarrow \tau_2, V) &\xrightarrow{\text{opus}} \left\{ \begin{array}{l} \{e \mapsto R\} \\ | \\ R \text{ is an injective renaming} \\ \text{and } \text{range}(R) \cap V = \emptyset \end{array} \right\} & \text{(C EO-unify)} \\
(e\tau \doteq \omega, V) &\xrightarrow{\text{opus}} \left\{ \begin{array}{l} \{e \mapsto R\}, \{e \mapsto \omega\} \\ | \\ R \text{ is an injective renaming} \\ \text{and } \text{range}(R) \cap V = \emptyset \end{array} \right\} & \text{(C EO-unify)} \\
(e\tau \doteq \tau_1 \cap \tau_2, V) &\xrightarrow{\text{opus}} \left\{ \begin{array}{l} \{e \mapsto R\}, \{e \mapsto e_1 \sqcap \cap e_2 \sqcap\} \\ | \\ R \text{ is an injective renaming} \\ \text{and } \text{range}(R) \cap V = \emptyset \\ \text{and } e_1, e_2 \notin V \end{array} \right\} & \text{(C EI-unify)} \\
(e_1 \tau_1 \doteq e_2 \tau_1, V) &\xrightarrow{\text{opus}} \left\{ \begin{array}{l} \{e_1 \mapsto e_5 ([S_L] e_3 \sqcap \cap S_R |_{\tau_3}), \\ e_2 \mapsto e_5 (S_L |_{\tau_4} \cap [S_R] e_4 \sqcap)\} \\ | \\ \{e_3, e_4, e_5, e_1^L, \dots, e_p^L, e_1^R, \dots, e_q^R\} \cap V = \emptyset \\ \text{and } e_1 \neq e_2 \\ \text{and } (e_3 \tau_1 \doteq \tau_2, V) \xrightarrow{\text{opus}} (\mathcal{S} \cup \{S_i^L\}_{i=1}^p) \\ \text{and } (\tau_1 \doteq e_4 \tau_2, V) \xrightarrow{\text{opus}} (\mathcal{S}' \cup \{S_i^R\}_{i=1}^q) \\ \text{and } S_L = e_1^L S_1^L \cap \dots \cap e_p^L S_p^L \\ \text{and } S_R = e_1^R S_1^R \cap \dots \cap e_q^R S_q^R \end{array} \right\} & \text{(C EE-unify)}
\end{aligned}$$

□

Remark 36. The opus outer part is not delineated by expansion constructors, as in *extract*; and it is really necessary to incorporate the actions of what were (T-unify) and (Z-unify) in previous systems within opus to get a substitution set with the covering property. Consider each case defining the result. Unlike in *extract*, the variables \vec{e} are not necessarily the full common prefix of the two sides of the constraint.

- (C E-unify) replaces the notions of working inside a prefix and the (Z-unify) rules in previous systems. An arbitrary unifier could substitute for e_1 an intersection of many different unifiers for the inner constraint. This rule creates unifiers that can cover all such unifiers.
- (C T-unify) is like (T-unify), so opus extractions can extend down to include the type constructors immediately below the topmost tree of expansion constructors.
- (C EA-unify) eliminates E-variable e safely by replacing it with a suitable renaming substitution. This is similar to the base case of *extract*, but it can be needed whatever constructor is outermost in τ_3 (the same unifier is generated in the (C EO-unify) and (C EI-unify) rules).
- (C EO-unify) is analogous to the ω case of *extract*; it also gives the elimination possibility.

- (C EI-unify) is analogous to the \sqcap case of *extract*. Rather than forcing a maximal extraction by inductively extracting, to cover all solutions, it is simpler to make subsequent uses of *opus* do the recursive extractions. Again, the elimination possibility is provided.
- (C EE-unify) with E-variables e_1, e_2 outermost we provide a solution of which all E_1, E_2 such that $[E_1] \tau_1 = [E_2] \tau_2$ are instances. This is considerably more complex than the E-variable case of *extract* in previous systems, a better intuition is to think of it as a generalisation of (C E-unify). Lemma 2 shows it is sufficient to find all E and S such that $\text{solved}([E] \tau_1 \doteq [S] \tau_2)$ or $\text{solved}([S] \tau_1 \doteq [E] \tau_2)$. These are all covered by inductively taking *opus* of the constraints $e_3 \tau_1 \doteq \tau_2$ and $\tau_1 \doteq e_4 \tau_2$. The result is the intersection of all these $p + q$ outer-part unifiers, each wrapped in a distinct fresh E-variable so that any instance can be formed by substituting for e_5 an expansion to select the desired combination of the unifiers. This is the innovative part of this system, enabled by the unit laws. Without them we would be forced to achieve completeness by enumerating all possible combinations at such points. \square

Example 20 (Outer-part unifier sets). Continuing with the constraint of Example 15, which concentrates our attention on the interesting (C EE-unify) rule.

$$\begin{array}{l}
\left[\begin{array}{l}
e_4 e_3 a_1 \doteq \omega \xrightarrow{\text{opus}} \{\{e_4 \mapsto \omega\}, \{e_4 \mapsto \{e_3 \mapsto e_6\}\}\} \\
[e_7 a_1 \doteq \omega \xrightarrow{\text{opus}} \{\{e_7 \mapsto \omega\}, \{e_7 \mapsto \{a_1 \mapsto a_2\}\}\} \\
[a_1 \doteq e_8 \omega \xrightarrow{\text{opus}} \{\{a_1 \mapsto e_8 \omega\}\} \\
e_3 a_1 \doteq e_5 \omega \xrightarrow{\text{opus}} \{\{e_3 \mapsto e_9 (e_1^L \omega \sqcap e_2^L \{a_1 \mapsto a_2\}) \sqcap e_1^R \{a_1 \mapsto e_8 \omega\}\}, \\
\quad e_5 \mapsto e_9 (e_1^L \sqcap \sqcap e_2^L \sqcap \sqcap e_1^R \sqcap)\} \} = S \\
e_1 e_3 a_1 \doteq e_2 \omega \xrightarrow{\text{opus}} \{\{e_1 \mapsto e_{10} (e_3^L \omega \sqcap e_4^L \{\{e_3 \mapsto e_6\}\}) \sqcap e_3^R S|_{e_3}\}, \\
\quad e_2 \mapsto e_{10} (e_3^L \sqcap \sqcap e_4^L \sqcap \sqcap e_1^R S(e_5))\}
\end{array} \right.
\end{array}$$

The bottommost use of *opus* uses (C EE-unify) which requires the two recursive uses of *opus* above it (in the “tree”). The first of these simply uses (C EO-unify) to match the ω by substitution and renaming respectively. The second *opus* is another (C EE-unify) case requiring another two recursive *opus* uses: one is just like the first before; the other is a (C T-unify) case. Thus the inner (C EE-unify) result is an intersection of the three possibilities generated by its offspring; the outer (C EE-unify) gives an intersection of three outer-part unifiers, the third components taken from the inner (C EE-unify) case.

Applying the substitution to the constraint $e_1 e_3 a_1 \doteq e_2 \omega$ gives:
 $e_{10} e_3^L \omega \sqcap e_4^L e_6 a_1 \sqcap e_3^R e_9 (e_1^L \omega \sqcap e_2^L a_2 \sqcap e_1^R e_8 \omega) \doteq$
 $e_{10} e_3^L \omega \sqcap e_4^L \omega \sqcap e_3^R e_9 (e_1^L \omega \sqcap e_2^L \omega \sqcap e_1^R \omega)$ \square

Remark 37. We make no attempt to generate minimal unifier sets in the sense of Definition 23. Doing so could be very tricky at best. \square

Remark 38. Lemma 2 shows how the *opus* function produces finite covering substitution sets in the sense of Definition 22. \square

Applying an *opus* substitution to a constraint set Δ that is solved by the application of expansion E decreases a certain size measure. This measure (Definition 40) is an ordered pair of numbers. Both numbers are measures of the unsolved parts of Δ . The least significant number is the number of distinct variables in the unsolved part — the set of *vpaths*. The most significant number is the size of the parts of $[E]\Delta$ that result from expansion of the unsolved parts of Δ .

Definition 39 (Variable paths, *vpaths*). *By induction on types:*

$$\begin{aligned}
\text{vpaths}(a) &= \{a \cdot \epsilon\} \\
\text{vpaths}(e\tau) &= \{e \cdot \epsilon\} \cup \{e \cdot \vec{v} \mid \vec{v} \in \text{vpaths}(\tau)\} \\
\text{vpaths}(\omega) &= \emptyset \\
\text{vpaths}(\tau_1 \cap \tau_2) &= \text{vpaths}(\tau_1) \cup \text{vpaths}(\tau_2) \\
\text{vpaths}(\tau_1 \rightarrow \tau_2) &= \text{vpaths}(\tau_1) \cup \text{vpaths}(\tau_2)
\end{aligned}$$

Definition 40 (Solving size and variables measure, *evsize*). *Let $\langle m, X \rangle + \langle n, Y \rangle = \langle m+n, X \cup Y \rangle$ and $\langle m, X \rangle < \langle n, Y \rangle$ iff $m < n$ or $m = n$ and $X \subset Y$. The size measure on is defined by cases on E and Δ by:*

$$\begin{aligned}
\text{evsize}(E, \Delta) &= \Sigma_{\delta \in \Delta} \text{evsize}(E, \delta) \\
\text{evsize}(\omega, \delta) &= \langle 0, \emptyset \rangle \\
\text{evsize}(e E, \delta) &= \text{evsize}(E, \delta) \\
\text{evsize}(E_1 \cap E_2, \delta) &= \text{evsize}(E_1, \delta) + \text{evsize}(E_2, \delta) \\
\text{evsize}(S, \tau \doteq \tau) &= \langle 0, \emptyset \rangle \\
\text{evsize}(S, e\tau_1 \doteq e\tau_2) &= \text{evsize}(S(e), \tau_1 \doteq \tau_2) \\
\text{evsize}(S, \tau_1 \cap \tau_2 \doteq \tau_3 \cap \tau_4) &= \text{evsize}(S, \tau_1 \doteq \tau_3) + \text{evsize}(S, \tau_2 \doteq \tau_4) \\
\text{evsize}(S, \tau_1 \rightarrow \tau_2 \doteq \tau_3 \rightarrow \tau_4) &= \text{evsize}(S, \tau_1 \doteq \tau_3) + \text{evsize}(S, \tau_2 \doteq \tau_4) \\
\text{evsize}(S, \delta) &= \langle \text{size}([S]\delta), \text{vpaths}(\delta) \rangle, \text{ otherwise}
\end{aligned}$$

Lemma 2 (opus sets are coverings). *If $\text{solved}([S_1]\Delta)$ and $\Delta \xrightarrow{\text{rfactor}} \Delta$ and $\delta \in \Delta$ and not $\text{solved}(\delta)$ then $(\delta, \text{Vars}(\Delta)) \xrightarrow{\text{opus}} S$ and there are substitutions $S_2 \in S$ and S_3 such that $S_1|_{\Delta} = S_2; S_3$ and $\text{eemeasure}(S_1, \Delta) > \text{eemeasure}(S_3, [S_2]\Delta)$.*

Remark 39. The “complete”, or “covering”, constraint set reduction relation \xRightarrow{c} uses *opus* where \xRightarrow{s} and \xRightarrow{r} used unifier or runifier. Again, for simplicity, the definition passes the set of all variables in Δ as the rename set V . The soundness proof for \xRightarrow{c} is analogous to Theorem 4, but the statement is slightly different because only some simplifications with the factor relation preserve solvedness under expansion.

Definition 41 (Covering direct reduction, $\xRightarrow{c} \subseteq \text{CstrSet} \times \text{CstrSet}$). *The relation \xRightarrow{c}^S is defined by: $\Delta \xRightarrow{c}^S \Delta_2$ if $\Delta \xrightarrow{\text{rfactor}} \Delta_1$ and $\delta \in \Delta_1$ and not $\text{solved}(\delta)$ and $(\delta, \text{Vars}(\Delta)) \xrightarrow{\text{opus}} S$ and $S \in S$ and $[S]\Delta_1 \xrightarrow{\text{rfactor}} \Delta_2$.*

Define $\Delta \xRightarrow{c} \Delta'$ if there is a substitution S such that $\Delta \xRightarrow{c}^S \Delta'$, and define \xRightarrow{c} as the reflexive transitive closure by analogy with \xRightarrow{s} .

Example 21 (Covering reductions). The example constraint set has the following direct reduction.

$$\{e_1 e_3 a_1 \doteq e_2 \omega\} \xRightarrow{c} \{e_{10} e_3^L (\omega \doteq \omega), e_{10} e_4^L (e_6 a_1 \doteq \omega), e_{10} e_3^R e_9 e_1^L (\omega \doteq \omega), \\ e_{10} e_3^R e_9 e_2^L (a_2 \doteq \omega), e_{10} e_3^R e_9 e_1^R (e_8 \omega \doteq \omega)\}$$

There are many ways to normalise this constraint set. Here is one substitution generated by a normalisation:

$$e_{10} e_4^L / \{e_6 \mapsto \omega\}; e_{10} e_3^R e_9 e_2^L / \{a_2 \mapsto \omega\}; e_{10} e_3^R e_9 e_1^R / \{e_8 \mapsto \omega\}$$

The whole unifier (restricted to the example constraint) is:

$$\{e_1 \mapsto e_{10} (e_3^L \omega \cap e_4^L \{e_3 \mapsto \omega\} \cap e_3^R \{e_3 \mapsto e_9 (e_1^L \omega \cap e_2^L \{a_1 \mapsto \omega\} \cap e_1^R \{a_1 \mapsto \omega\})\}), \\ e_2 \mapsto e_{10} (e_3^L \square \cap e_4^L \square \cap e_1^R e_9 (e_1^L \square \cap e_2^L \square \cap e_1^R \square))\}$$

This is not minimal and we can eliminate some parts, giving:

$$S_8 = \{e_1 \mapsto e_{10} (e_3^L \omega \cap e_4^L \{e_3 \mapsto \omega\} \cap e_3^R \{e_3 \mapsto e_9 (e_1^L \omega \cap e_1^R \{a_1 \mapsto \omega\})\}), \\ e_2 \mapsto e_{10} (e_3^L \square \cap e_4^L \square \cap e_1^R e_9 (e_1^L \square \cap e_1^R \square))\}$$

Returning to the unifiers given in Example 15, all are instances of the one we infer now with \xRightarrow{c} :

$$S_3 = \{e_1 \mapsto e_2 S_1\} \\ = S_8; \{e_{10} \mapsto e_2 \{e_3^L \mapsto \omega, e_4^L \mapsto \square, e_3^R \mapsto \omega\}\} \\ S_4 = \{e_1 \mapsto e_2 S_2\} \\ = S_8; \{e_{10} \mapsto e_2 \{e_3^L \mapsto \omega, e_4^L \mapsto \omega, e_3^R \mapsto \{e_9 \mapsto \{e_1^L \mapsto \omega, e_1^R \mapsto \square\}\}\}\} \\ S_5 = \{e_1 \mapsto e_2 (S_1 \cap S_2), e_2 \mapsto e_2 (\square \cap \square)\} \\ = S_8; \{e_{10} \mapsto \{e_3^L \mapsto \omega, e_4^L \mapsto \square, e_3^R \mapsto \omega\} \cap \\ \{e_3^L \mapsto \omega, e_4^L \mapsto \omega, e_3^R \mapsto \{e_9 \mapsto \{e_1^L \mapsto \omega, e_1^R \mapsto \square\}\}\}\}$$

□

Theorem 11 (\xRightarrow{c} gives covering substitution-unifier sets). *If solved($[S_1] \Delta$) then there are S_2, S_3, Δ_1 such that $\Delta \xRightarrow{S_2, c} \Delta_1$ and solved(Δ_1) and $S_1|_{\Delta} = S_2; S_3$.* □

7 Properties of Constraint Reduction

This section considers, at varying levels of detail, how the simple rewrite system can be adapted to incorporate some important properties. Section 7.1 presents a key property: constraint set reduction corresponds to β -reduction. Section 7.2 shows that there are decidable, and undecidable, forms of unification with expansion variables. Section 7.3 looks at termination and confluence; the results are mostly negative, so finding better behaved restrictions of the general framework is important future work.

7.1 Relationship to β -reduction

This section shows that β -unification relates to β -unification. And how it can be used for *intersection typing* inference. A relation *pretyping* maps every λ -term to its *pretypings*. A pretyping is an intersection *typing* (a type environment and a result type), paired with a constraint set. For β -normal forms, reduction of the constraint set is strongly normalising; for a β -reducible term, the constraint set reduces to a constraint set in a pretyping of the reduced term (Theorem 12). The \Rightarrow_r rewrite system is used to demonstrate this correspondence. First we recall relevant λ -calculus definitions.

Definition 42 (λ -calculus definitions). *Term variables and terms are the least sets satisfying:*

$\text{TermVar} = \{x_i \mid i \in \mathbb{K}\}$ where \mathbb{K} is an arbitrary countable set of indices
 $\text{Term} \supseteq \text{TermVar} \cup \{\lambda x. M \mid x \in \text{TermVar} \text{ and } M \in \text{Term}\} \cup \{M_1 @ M_2 \mid M_i \in \text{Term}\}$

The λI terms restrict Term to those M such that x occurs free at least once in M_1 for every abstraction $\lambda x. M_1$ in M . The capture-free replacement of all free occurrences of variable x in term M_1 with the term M_2 is written $M_1[x := M_2]$. We take term equality to be syntactic equality. The β -reduction relation $\xrightarrow{\beta}$ on terms is defined by the following rules.

1. $(\lambda x. M_1) @ M_2 \xrightarrow{\beta} M_1[x := M_2]$
2. $\lambda x. M_1 \xrightarrow{\beta} \lambda x. M_2$ if $M_1 \xrightarrow{\beta} M_2$
3. $M_1 @ M_2 \xrightarrow{\beta} M_3 @ M_2$ if $M_1 \xrightarrow{\beta} M_3$
4. $M_1 @ M_2 \xrightarrow{\beta} M_1 @ M_3$ if $M_2 \xrightarrow{\beta} M_3$ □

Example 22 (β -reduction).

$(\lambda x. x @ x) @ (\lambda x. \text{id}) \xrightarrow{\beta} (\lambda x. \text{id}) @ (\lambda x. \text{id})$ if $\text{id} = \lambda x. x$. □

Next we define pretypings and their operations.

Definition 43 (Type environments, typings and pretypings).

1. A type environment is a total function $A : \text{TermVar} \rightarrow \text{Typ}$ which maps finitely many term variables to non- ω types.
2. The empty environment $()$ is defined $\{x \mapsto \omega \mid x \in \text{TermVar}\}$ and $(x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n)$ abbreviates $() [x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n]$.⁹
3. Environment intersection: $A \cap A' = \{x \mapsto \tau \mid \{A(x), A'(x)\} = \{\omega, \tau\}\} \cup \{x \mapsto \tau \cap \tau' \mid \{A(x), A'(x)\} \neq \{\omega, \tau\}\}$.
4. Environment expansion: $[E] A = \{x \mapsto [E] A(x) \mid A(x) \neq \omega\} \cup \{x \mapsto \omega \mid A(x) = \omega\}$.
5. E-variable application: $e A = [e \square] A$.
6. A typing is a type environment A paired with a type τ , written $\langle A \vdash \tau \rangle$.
7. A pretyping P is a typing paired with a constraint set Δ , written $\langle A \vdash \tau \rangle / \Delta$.

⁹ The function modification operator $A[x \mapsto \tau]$ gives a function which is the same as A for every variable except x which it maps to τ .

8. Pretyping reduction is defined: $\langle A \vdash \tau \rangle / \Delta \xrightarrow[r]{S} \langle [S] A \vdash [S] \tau \rangle / \Delta'$
 if $\Delta \xrightarrow[r]{S} \Delta'$ where r is a constraint-set reduction relation. \square

Example 23 (Pretyping). $P_1 = \langle () \vdash a_5 \rangle / \Delta_7$ where
 $\Delta_7 = \{a_1 \doteq e_1 a_2 \rightarrow a_4, a_1 \cap e_1 a_2 \doteq e_2 \tau_3, a_4 \doteq a_5\}$. \square

Every application in term M corresponds to one constraint (some are split into two by simplification) in pretypings of M . The constraint expresses that the applied subterm has a function type from the type of the argument subterm to the type of the application itself.

Definition 44 (Pretypings of a term, $\xrightarrow{\text{pretyping}}$). Defined inductively on terms by:

$x \xrightarrow{\text{pretyping}} \langle (x : a) \vdash a \rangle / \emptyset$
 $\lambda x. M \xrightarrow{\text{pretyping}} \langle A[x : \omega] \vdash A(x) \rightarrow \tau \rangle / \Delta$
 if $M \xrightarrow{\text{pretyping}} \langle A \vdash \tau \rangle / \Delta$
 $M_1 @ M_2 \xrightarrow{\text{pretyping}} \langle A_1 \cap e A_2 \vdash a \rangle / \Delta_1 \uplus e \Delta_2 \uplus \text{simplify}(\{\tau_1 \doteq e \tau_2 \rightarrow a\})$
 if $M_i \xrightarrow{\text{pretyping}} \langle A_i \vdash \tau_i \rangle / \Delta_i$ and e, a do not occur outermost in Δ_1 \square

Remark 40. Whereas β -reduction rewrites an application $\lambda x. M @ N$ by substituting a copy of N for each of the n free occurrence of x in M ; reduction of the corresponding pretyping creates n copies of the constraint of N , specialises the application type to the type of M , and specialises the type of x to an intersection of n copies of the type of N .

To show this relationship, Theorem 12 uses the restricted renaming reduction relation, \xRightarrow{r} . This allows, for example, the correspondence to succeed for the example term $\text{id}@id \xrightarrow{\text{pretyping}} P_2$ where $P_2 = \langle () \vdash a_2 \rangle / \{a_1 \rightarrow a_1 \doteq e_1 (a_1 \rightarrow a_1) \rightarrow a_2\}$. The (E-unify) of \xRightarrow{s} breaks the correspondence by reducing P_2 to $\langle () \vdash a_1 \rangle / \{a_1 \doteq a_1 \rightarrow a_1\}$ and then to $\langle () \vdash \omega \rangle / \emptyset$. With \xRightarrow{r} , P_2 reduces to $\langle () \vdash a_1 \rangle / \{a_1 \doteq a_1 \rightarrow a_1\}$ and then $\langle () \vdash a_1 \rightarrow a_1 \rangle / \emptyset$, which is a pretyping of id as required. \square

For the pretypings of terms in β -normal form, all reduction paths lead to a normal form because the opportunity to expand an E-variable never arises.

Lemma 3 (β -normal form pretypings strongly normalise). *If M is a β -normal form and $M \xrightarrow{\text{pretyping}} P$ then P has no infinite \xRightarrow{r} reductions.* \square

Theorem 12 (Constraint reduction simulates β -reduction). *If $M \xrightarrow{\text{pretyping}} P$ and M is β -normalising then P is \xRightarrow{r} -normalising. If $M \xrightarrow{\text{pretyping}} P$ and M is not β -strongly normalising then P has an infinite \xRightarrow{r} reduction.* \square

Example 24 (Constraint reduction simulating β -reduction). The simulation for our example is shown below (P_1 was defined in Example 23; the pretyping reduction follows the steps shown in Example 17).

$$\begin{array}{ccccc}
(\lambda x. x @ x) @ \lambda x. id & \xrightarrow{\beta} & (\lambda x. id) @ (\lambda x. id) & \xrightarrow{\beta} & id \\
\downarrow \text{pretyping} & & \downarrow \text{pretyping} & & \downarrow \text{pretyping} \\
P_1 \xrightarrow{r} \langle () \vdash a_4 \rangle / \{ \omega \doteq e_1 (\omega \rightarrow a_7 \rightarrow a_7), a_6 \rightarrow a_6 \doteq a_4 \} & \xrightarrow{r} & \langle () \vdash a_8 \rightarrow a_8 \rangle / \emptyset & &
\end{array}$$

The first β -reduction copies the argument term twice into the body of $\lambda x. x @ x$; the corresponding constraint reduction puts two distinctly named copies of the argument term type into the constraint of the body of body of $\lambda x. x @ x$. The second β -reduction eliminates the argument term and returns the body, id ; the corresponding constraint reduction solves the constraint by eliminating the argument type, and adjusts the overall type to the type of id . \square

Remark 41. Normalising the pretyping of term M results in an intersection typing (possibly containing some E-variables) for M . Such typings are derivable in System E, or a conventional intersection type system if we drop the E-variables.

Instead of directly simulating β -reduction, the same result could be derived by showing how constraint reduction simulates a System E type inference algorithm, which itself simulates β -reduction [7].

The correspondence to β -reduction does not prove that our unification system has undecidable (strong) normalisability, although this is true for other, more restricted β -unification systems [16, 3]. \square

Lemma 4 (β -normal form constraint reductions are finite). *If M is a β -normal form and $M \xrightarrow{\text{pretyping}} \langle A \vdash \tau \rangle / \Delta$ then the length of every reduction $\Delta \xrightarrow{s} \Delta'$ and every reduction $\Delta \xrightarrow{c} \Delta'$ is finite.* \square

Example 25 (Normal form constraint reduction). A normalisation of a pretyping of the β -normal form $x @ x$: $\langle (x \mapsto e_1 a_1 \cap e_2 a_2) \vdash a_4 \rangle / \{ e_1 a_1 \doteq e_2 a_2 \rightarrow a_4 \} \xrightarrow{s} \langle (x \mapsto (e_2 a_2 \rightarrow a_4) \cap e_2 a_2) \vdash a_4 \rangle / \emptyset$ \square

Remark 42. With such a strong link to β -normalisation, an undecidability result for constraint, or pretyping, normalisation under renaming reduction is anticipated. A reduction of β -normalisation (shown undecidable in [5]), or β -strong-normalisation (shown undecidable in [12]) would be, in principle, a good way to show undecidability. However, β -normalisation, or β -strong-normalisation, does not reduce to show the undecidability of pretyping normalisation. There are β -unnormalisable terms whose pretypings normalise. For example, for the following unnormalising term,

$$(\lambda x. x @ x) @ (\lambda x. x @ x) \xrightarrow{\text{pretyping}} \langle () \vdash a_7 \rangle / \{ a_1 \doteq e_1 a_2 \rightarrow a_3, e_2 (a_4 \doteq e_3 a_5 \rightarrow a_6), e_2 (a_4 \cap e_3 a_5 \rightarrow a_6) \doteq (a_1 \cap e_1 a_2), a_3 \doteq a_7 \}$$

We can easily normalise the constraint to a stuck but non- \emptyset form by using a (Z-unify), or (R Z-unify), reduction $\omega \doteq (a_1 \cap e_1 a_2)$.

It may be possible to reduce β -strong-normalisability to pretyping-strong-normalisability; investigating this route is left to future work. By restricting to the constraints that arise during the simulation of β -reduction, and eliminating (R Z-unify), β -normalisation reduces to pretyping normalisation [1], showing the undecidability of \Rightarrow_{τ} normalisation for an interesting class of constraints. \square

7.2 Decidability

Decidable forms. The unification with expansion variables problem with an ω expansion-constructor is trivially decidable. (In general, any nullary expansion constructor unifies all constraints.) This does not make the problem useless or trivial: ω is only a most-general unifier when there is no substitution-unifier, it should be seen as a catchall solution.

Another decidable form arises when there are no expansion constructors — so expansion variables and substitutions are the only kinds of expansion, and expansion variables only serve as namespace separators. Solving these forms is equivalent to first-order unification because a constraint is soluble iff the constraint that results from replacing its expansion variables with distinct fresh variable renaming substitutions is soluble.

Another decidable form is when there is only one unary constructor (which may be a case of expansion). Then unification can never fail through a clash of constructors, so expansions are only useful as namespace separators, and the problem reduces to first-order unification.

Undecidable forms. Undecidable forms arise when there are expansion constructors, but no nullary ones. The simplest case we know of is when \cap is the only expansion constructor and \rightarrow is a type constructor. This form we call ω -free unification with E-variables. In general, the undecidability result extends to unification with E-variables over any constructor signature where:

1. there is at least one expansion constructor;
2. there is at least one term constructor;
3. the arity of both of these constructors is at least two.

To demonstrate undecidability we consider the connection between to a minimal system of intersection types with expansion variables. Judgements of System $E^{\cap, \rightarrow}$ assign *pretypings* to λI terms.

Definition 45 (Pretyping and typing judgements). *A pretyping judgement pairs term M with pretyping $P = \langle A \vdash \tau \rangle / \Delta$, written as $M : P$. If $\text{solved}(\Delta)$ then $M : P$ is a typing judgement.* \square

The pretyping judgements of System $E^{\cap, \rightarrow}$ are defined by the following natural deduction rules. If a typing judgement $M : P$ is derivable in System $E^{\cap, \rightarrow}$ we say that M is System $E^{\cap, \rightarrow}$ -typable.

$$\begin{array}{c}
\text{(tv)} \frac{}{x : \langle (x : \tau) \vdash \tau \rangle / \emptyset} \quad \text{(ab)} \frac{M : \langle A \vdash \tau \rangle / \Delta}{\lambda x. M : \langle A[x \mapsto \omega] \vdash A(x) \rightarrow \tau \rangle / \Delta} \\
\text{(ap)} \frac{M : \langle A \vdash \tau \rangle / \Delta \quad M_1 : \langle A_1 \vdash \tau_1 \rangle / \Delta_1}{M @ M_1 : \langle A \cap A_1 \vdash \tau_2 \rangle / \{\tau \doteq \tau_1 \rightarrow \tau_2\} \cup \Delta \cup \Delta_1} \\
\text{(in)} \frac{M : \langle A \vdash \tau \rangle / \Delta \quad M : \langle A_1 \vdash \tau_1 \rangle / \Delta_1}{M : \langle A \cap A_1 \vdash \tau \cap \tau_1 \rangle / \Delta \cup \Delta_1} \quad \text{(ev)} \frac{M : \langle A \vdash \tau \rangle / \Delta}{M : \langle e A \vdash e \tau \rangle / \{e \delta \mid \delta \in \Delta\}}
\end{array}$$

Proposition 6 (System $E^{\cap, \rightarrow}$ -typable terms are normalising). *A term M of the λI -calculus is System $E^{\cap, \rightarrow}$ -typable iff it is β -normalizing.* \square

Lemma 5 (All System $E^{\cap, \rightarrow}$ pretypings expand initial pretyping). *If $M : P$ is derivable in System $E^{\cap, \rightarrow}$ and $M \xrightarrow{\text{pretyping}} P_1$ then there exists an expansion E such that $P = [E] P_1$.* \square

Theorem 13 (Undecidability ω -free unification with E-variables). *In the ω -free restriction of unification with E-variables, for any constraint set Δ the existence of an expansion E such that $\text{solved}([E] \Delta)$ is undecidable.* \square

Remark 43. The undecidability and simulation of β -reduction results give us a completeness result for constraint solving with $\xRightarrow{\text{r}}$ because when we restrict terms to the λI terms, and unifiers to be ω -free, we know that $\xRightarrow{\text{r}}$ solves the constraints of all β -normalising terms, and the constraints of all other λI terms are insoluble. \square

Theorem 14 (Completeness of $\xRightarrow{\text{r}}$ for ω -free unification). *If M is a λI term and $M \xrightarrow{\text{pretyping}} \langle A \vdash \tau \rangle / \Delta$ then there is an ω -free expansion E such that $\text{solved}([E] \Delta)$ iff $\Delta \xRightarrow{\text{r}} \emptyset$.* \square

7.3 Rewriting Properties

Termination A rewrite system without infinite reductions would be the ideal starting point for designing a decision procedure. Remark 9 shows the simple system has infinite reductions. The correspondence to β -reduction shows the renaming system has infinite reductions, even for where reduction only uses (T-unify) and (E-unify) .

For the simple and renaming systems there are soluble constraints that they cannot solve and will reduce infinitely; with the covering system they are soluble but there are still infinite reductions.

In the simple system reductions using only (Z-unify) and (T-unify) are finite; (E-unify) alone can increase constraint set size and produce infinite reductions such as repetition of the following:

Example 26 (Simple (E-unify) is non-terminating). The following reduction results in the same constraint nested inside E-variable e_2 , this behaviour can repeat indefinitely.

$$\begin{aligned} & \{e_1 a \doteq e_2 (a \cap e_1 a), e_1 (e_1 a \doteq e_2 (a \cap e_1 a))\} \xrightarrow[\substack{e_1 \mapsto e_2 (\Box \cap e_1 \Box)}]{s} \\ & \{e_2 (e_1 a \doteq e_2 (e_1 a \cap a)), e_2 e_1 (e_1 a \doteq e_2 (e_1 a \cap a))\} \end{aligned}$$

□

With renaming reduction we conjecture that (R E-unify) has no infinite reductions — at least for the constraints involved in simulating β -reduction. The covering system reintroduces rather trivial ways to form infinite reductions by using (C EI-unify). For example: $\{e_1 a \doteq e_2 a \cap e_2 a, e_1 a \doteq e_2 a\}$ can expand limitlessly as descendants of e_1 and e_2 expand alternately. The other covering system rules in isolation are conjectured to be terminating.

Confluence Non-deterministic rewrite systems with some confluence property can be implemented without backtracking, improving their complexity enormously, confluence is often essential for rewriting on large inputs and a great aid for compositional solving. Reduction of unrestricted constraints as presented in this report is not confluent. We describe some varieties of confluence that might be apt and highlight some of the problems to be overcome in this area.

Unifier confluence. As the result of constraint reduction is really the unifier it generates as well as the reduced constraint, one might hope for the following very strong form of confluence. $\Delta_1 \xleftarrow{E_1} \Delta \xrightarrow{E_2} \Delta_2$ implies there are Δ_3, E_3, E_4 such that $\Delta_1 \xrightarrow{E_3} \Delta_3 \xleftarrow{E_4} \Delta_2$ and $E_1; E_3 = E_2; E_4$. However, this would be excessive if we use a rewrite system that always gives a principal unifier: we do not really care which principal unifier we get. Unifier confluence is only really be useful because it implies all the following weaker forms.

Constraint confluence. The standard statement for rewrite systems. $\Delta_1 \Leftarrow \Delta \Rightarrow \Delta_2$ implies there is a Δ_3 such that $\Delta_1 \Rightarrow \Delta_3 \Leftarrow \Delta_2$. So any normal form — stuck or solved — of Δ can always be reached by reducing a reduct of Δ .

Local confluence. For terminating rewrite systems one is usually interested in the following. $\Delta_1 \Leftarrow \Delta \Rightarrow \Delta_2$ implies there is a Δ_3 such that $\Delta_1 \Rightarrow \Delta_3 \Leftarrow \Delta_2$. In combination with termination (which we do not have), this implies constraint confluence. For systems like \xrightarrow{s} , where there is only one normal form \emptyset , and rewriting has the *progress* property (any non-empty constraint set is reducible), termination would imply confluence.

None of the above forms of confluence hold for a rather trivial reason. Constraints sets that include a constraint like $a_1 \doteq a_2$ can reduce in two different ways to give constraints that are identical up to variables renaming! When the rest of the constraint cannot be normalised a confluence is impossible. Therefore we should seek confluence modulo isomorphism.

Soluble constraint confluence, or closedness. The real benefit of confluence is that for constraints soluble by reduction, a unifier can always be obtained by

reduction of *any* reduct. In short, we do not care if reduction is non-confluent for constraints we cannot solve. This property is $\Delta_1 \Leftarrow \Delta \Rightarrow \Delta_2$ and $\text{solved}(\Delta_2)$ implies $\Delta_1 \Rightarrow \Delta_2$.¹⁰ Soluble-constraint confluence does not have to be taken modulo isomorphism. It is implied by, and does not imply, confluence.

Simple, renaming and covering reduction do not enjoy any of these properties. The following example demonstrates a case where reduction is not even closed modulo isomorphism.

Example 27 (Constraint reduction not closed). We can use a simple variation on Example 26 to demonstrate non-closedness. In the first reduction below, the additional constraint $e_2 \omega \doteq e_3 \omega$ is solved and the remainder can reduce infinitely as before.

$$\begin{array}{l} \{e_2 \omega \doteq e_3 \omega, e_1 a \doteq e_2 (a \cap e_1 a), e_1 (e_1 a \doteq e_2 (a \cap e_1 a))\} \\ \{e_1 a \doteq e_2 (a \cap e_1 a), e_1 (e_1 a \doteq e_2 (a \cap e_1 a))\} \end{array} \xrightarrow[\text{s}]{e_3 \mapsto e_2 \omega}$$

Choosing to reduce the new constraint differently quickly leads to a normal form:

$$\begin{array}{l} \{e_2 \omega \doteq e_3 \omega, e_1 a \doteq e_2 (a \cap e_1 a), e_1 (e_1 a \doteq e_2 (a \cap e_1 a))\} \\ \{e_1 a \doteq e_3 \omega\} \\ \omega \end{array} \xrightarrow[\text{s}]{e_2 \mapsto e_3 \omega} \xrightarrow[\text{s}]{e_1 \mapsto e_3 \omega}$$

With renamings, this example constraint has no infinite reductions. However, the same general situation adapts to the constraint used to illustrate an infinite reduction in Remark 9. Modifying it as shown below, we can then choose to substitute for e_2 or e_3 and, as above, the result can be then either be normalised or reduced infinitely.

$$\begin{array}{l} \{e_2 \omega \doteq e_3 \omega, \\ e_1 ((e_2 ((e_3 a_1 \rightarrow a_2) \cap e_3 a_1)) \rightarrow e_4 a_2) \doteq e_2 (e_3 a_1 \rightarrow a_2 \cap e_3 a_1), e_4 a_2 \doteq a_2\} \end{array}$$

Similar situations exist with covering reduction. □

8 Prospects

The overall impression is that the rewrite systems for β -unification are powerful with much potential, but sometimes badly behaved.

¹⁰ This would be called *closedness* in formal language theory. Constraint reduction can be viewed as a way of defining a language of constraints, $\mathcal{L} = \{\Delta \mid \Delta \Rightarrow \Delta_1 \text{ and } \text{solved}(\Delta_1)\}$. If \Rightarrow is closed then reducing a member of \mathcal{L} cannot produce a non-member of \mathcal{L} . The converse, that non-members cannot be made members by reduction, is immediate.

8.1 Unification system summary

Simple rewriting with \Rightarrow_s This is our system of choice for pedagogical purposes.¹¹ It demonstrates well how the mechanism of expansion works and even comes up with satisfactory unifiers for many constraints, including a large subclass of those we need to simulate β -reduction. However, its namespace merging means it will not produce valid typings from the constraint set of every λ -term. Moreover, it lacks so many important rewriting properties that it really provides little practical potential or deeper theoretical interest.

Restricted renaming rewriting with \Rightarrow_r Adding renamings is sufficient to demonstrate the full relationship to β -reduction and exact intersection typing inference. So this system is practically useful and an appropriate starting point for investigation of the production of “approximate” typings and typings that provide exact information about terms under evaluation strategies other than call-by-name. We are investigating these subjects in the context of System E in [1]. This requires some small modifications to allow for the richer algebraic theory, but the underlying principles of the unification system we use are the same. Future work will show that this system also offers better rewriting properties for a restricted class of constraints that are sufficient to simulate β -reduction: in addition to principal solutions, confluence, and termination of (R E-unify) rewriting, are possible.

Covering rewriting with \Rightarrow_c This system provides a fairly simple starting point for investigating how to produce finite representations of all unifiers for an arbitrary constraint set. It is somewhat more complex than the other systems and developing efficient, confluent, or approximate, variants seems an interesting area for theoretical research, but there is a lack of clear motivation for this strand (compared to the restricted approach) until a wider range of practical instances of β -unification problems are identified.

8.2 Areas for development

Decidable and confluent restrictions Termination is always possible by rewriting to some limit, then forcing a solution with ω . This suggests *ranked* unification, by analogy with ranked type inference [16], controlled by some structural measure. Regarding confluence, restrictions on constraints akin to the *polar types* discipline [10] will be necessary.

Applications Exact typing inference, as outlined in Section 7.1 and [7], is one important application of β -unification. The typings produced are suitable for exact analysis for call-by-name languages. Extensions for analysis of call-by-need or call-by-value languages, and for richer languages than the λ -calculus are desirable, and likely to lead to interesting variants on β -unification.

¹¹ A gentle slide presentation introducing the key concepts with the simple system without ω and (Z-unify) is available from <http://types.bu.edu/modular/compositional/>.

Relationship to other forms of unification We presented unification with expansion variables as a generalisation of first-order unification. Certain features are needed before it can be dubbed β -unification. Less powerful variants — unification with substitution variables, for example — have practical potential. Interesting richer variants can be formed by altering the term language, or the meaning of expansion.

Investigating connections to other unification problems, especially those bearing on type inference, will be enlightening, and reveal new uses for expansion variables. The undecidable *higher-order unification* problem [9] is one target; in [21], a restricted, undecidable, form of second-order unification is reduced to type inference in Church-style System F. *Semi-unification* is another undecidable problem, equivalent to type inference in Curry-style System F [25]; The relationship to linear logic, expounded in [17], is also promising: expansion variables play the role of proofnet *boxes*.

References

1. A. Bakewell, S. Carlier, A. J. Kfoury, and J. B. Wells. Exact intersection typing inference and call-by-name evaluation. Technical report, Department of Computer Science, Boston University, Dec. 2004. Superseded by [2].
2. A. Bakewell, S. Carlier, A. J. Kfoury, and J. B. Wells. Inferring intersection typings that are equivalent to call-by-name and call-by-value evaluations. Technical report, Church Project, Boston University, Apr. 2005.
3. A. Bakewell and A. J. Kfoury. Unification with expansion variables. Technical report, Department of Computer Science, Boston University, Dec. 2004.
4. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, 1983.
5. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
6. S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*, pp. 294–309. Springer-Verlag, 2004.
7. S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β -reduction. In *Proc. 6th Int’l Conf. Principles & Practice Declarative Programming*, 2004. Completely supersedes [8].
8. S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β -reduction. Technical Report HW-MACS-TR-0012, Heriot-Watt Univ., School of Math. & Comput. Sci., Jan. 2004. Completely superseded by [7].
9. G. Huet. A unification algorithm for typed λ -calculus. *Theoret. Comput. Sci.*, 1(1):27–58, 1975.
10. T. Jim. A polar type system. In J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, and J. B. Wells, eds., *ICALP Workshops 2000: Proceedings of the Satellite Workshops of the 27th International Colloquium on Automata, Languages, and Programming*, vol. 8 of *Proceedings in Informatics*, pp. 323–338, Geneva, Switzerland, July 2000. Carleton Scientific.

11. A. J. Kfoury. Beta-reduction as unification. A refereed extensively edited version is [12]. This preliminary version was presented at the Helena Rasiowa Memorial Conference, July 1996.
12. A. J. Kfoury. Beta-reduction as unification. In D. Niwinski, ed., *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996)*, Banach Center Publication, Volume 46, pp. 137–158. Springer-Verlag, 1999. Supersedes [11] but omits a few proofs included in the latter.
13. A. J. Kfoury, G. Washburn, and J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In *Proceedings of the 2nd Workshop on Intersection Types and Related Systems*, 2002. The ITRS '02 proceedings appears as vol. 70, issue 1 of *Elec. Notes in Theoret. Comp. Sci.*
14. A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999. Superseded by [16].
15. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [14], Aug. 2003.
16. A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004. Supersedes [14]. For omitted proofs, see the longer report [15].
17. P. Møller Neergaard and H. G. Mairson. Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In *Proc. 9th Int'l Conf. Functional Programming*. ACM Press, Sept. 2004.
18. V. Padovani. Decidability of fourth-order matching. *Math. Structures Comput. Sci.*, 3(10):361–372, 2000.
19. S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2):181–209, Mar. 1988.
20. S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2):151–169, Jan. 1984.
21. A. Schubert. Second-order unification and type inference for Church-style polymorphism. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.
22. S. J. van Bakel. Complete restrictions of the intersection type discipline. *Theoret. Comput. Sci.*, 102(1):135–163, 1992.
23. S. J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. Ph.D. thesis, Catholic University of Nijmegen, 1993.
24. J. B. Wells. Typability and type checking in the second-order λ -calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. Logic in Comput. Sci.*, pp. 176–185, 1994. Superseded by [25].
25. J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999. Supersedes [24].

A Proofs

Proposition 1 (\sqsupset lifts to the identity function). *For all types τ and expansions E :*

1. $\sqsupset \tau = \tau$;
2. $\sqsupset E = E$;
3. $[E] \sqsupset = E$.

Proof. By structural induction on τ or E as appropriate. \square

Proposition 2 (Expansion application is associative). *For all types τ and expansions E_i : $[[E_1] E_2] \tau = [E_1] ([E_2] \tau)$, and $[[E_1] E_2] E_3 = [E_1] ([E_2] E_3)$.*

Proof. By structural induction on E_1 , following the definition of expansion. \square

Theorem 3 (Simplify Sound). *solved($[E] \Delta$) iff solved($[E]$ (simplify(Δ))).*

Proof. As $\Delta = \{\delta_i\}_{i=1}^n$, we show that solved($[E] \delta_i$) iff solved($[E]$ simplify(δ_i)). Proceed by case analysis of an arbitrary $\delta = \tau_1 \doteq \tau_2$, following Definition 9.

1. Case $\tau_i = \bar{e}(\tau'_i \cap \tau''_i)$; or $\tau_i = \bar{e}(\tau'_i \rightarrow \tau''_i)$. We have solved($[E] \delta$) iff solved($[E]$ ($\bar{e}(\tau'_1 \doteq \tau''_2)$)) and solved($[E]$ ($\bar{e}(\tau''_1 \doteq \tau'_2)$)) by structural induction on δ then the theorem follows by induction on the two sub-constraints.
2. Case $\tau_1 = \tau_2$. Then solved(δ) implies solved($[E] \delta$) and solved($[E] \emptyset$).
3. Case ‘otherwise’ is immediate. \square

Theorem 4 (Soundness of simple unification). *If $\Delta \xrightarrow[S]{S} \emptyset$ then solved($[S] \Delta$).*

Proof. By induction on the reduction.

1. Case $\Delta \xrightarrow[S]{\emptyset} \emptyset$. Then simplify(Δ) = \emptyset and the result follows by Theorem 3.
2. Case $\Delta \xrightarrow[S]{S_1} \Delta_1 \xrightarrow[S]{S_2} \emptyset$. By I.H., solved($[S_2] \Delta_1$). As $\Delta_1 = \text{simplify}([S_1] \Delta)$ it follows that solved($[S_1; S_2] \Delta$) by Theorem 3. \square

Theorem 6 (Existence of unique restriction). *For every expansion E and type or constraint Y , there is an expansion $E|_Y$.*

Proof. By structural induction on E and for the case where E is a substitution, by structural induction on Y . \square

Theorem 7 (A principal unifier for every most general). *If E_1 is a most general unifier of Δ then $E_1|_\Delta$ is a principal unifier of Δ .*

Proof. As E_1 is most-general, there is an E_3 such that $E_2 = E_1; E_3$ for every unifier E_2 of Δ . By structural induction on E_1 , $E_1; E_3|_\Delta = E_1|_\Delta; E_3|_{[E_1] \Delta}$. Therefore E_2 is a principal unifier of Δ . \square

Remark 44. The following definitions are used in the proof of Theorem 8.

The `superexp` function extends all the substitutions in a principal unifier to make it most general.

To find out which variables occur in each namespace it uses the `outspace` function to get the set of variables in the outer namespace of a type τ (that is, `ovars`(τ)). But for every outer E-variable e , `outspace` gives $e \tau_e$ where τ_e is the intersection of the subterms in τ that are wrapped inside an outermost occurrence of e . So all variables in the namespace e in τ are in τ_e . \square

Definition 46 (Namespace destruction, $\text{outspace} : \text{Typ} \rightarrow \mathbb{P}(\text{Typ})$).

$$\text{outspace}(\tau) = \begin{cases} \{\tau\} & \text{if } \tau = a \text{ or } \tau = e \tau_1 \\ \emptyset & \text{if } \tau = \omega \\ \{a \mid a \in V\} \cup \{e \tau_1 \cap \tau_2 \mid e \tau_i \in \text{outspace}(\tau_i)\} & \text{if } \tau = \tau_1 \rightarrow \tau_2 \text{ or } \tau = \tau_1 \cap \tau_2 \\ \cup \left\{ \begin{array}{l} e \tau_1 \in \text{outspace}(\tau_i) \mid \\ \{i, j\} = \{1, 2\} \text{ and } \forall \tau_2. e \tau_2 \notin \text{outspace}(\tau_j) \end{array} \right\} & \end{cases}$$

□

Definition 47 (Principal unifier extension, $\text{superexp} : \text{Exp} \times \mathbb{P}(\text{Constraint}) \rightarrow \text{Exp}$). *Let*

$$\text{superexp}(E, \{\tau_i \doteq \tau'_i\}_{i=1}^n) = \text{superexp}(E, \tau_1 \cap \tau'_1 \cap \dots \cap \tau_n \cap \tau'_n, \mathcal{R})$$

where the ternary superexp is defined below and \mathcal{R} is a countably infinite set of injective outer renamings. such that distinct renamings in \mathcal{R} have disjoint ranges.

Let $\text{split}(\mathcal{R}, n)$ form a set of n mutually disjoint subsets of infinite outer renaming sets \mathcal{R}_i , where each subset is infinite.

$$\begin{aligned} \text{superexp}(\omega, \tau, \mathcal{R}) &= \omega \\ \text{superexp}(e E, \tau, \mathcal{R}) &= e(\text{superexp}(E, \tau, \mathcal{R})) \\ \text{superexp}(E \cap E', \tau, \mathcal{R}) &= (\text{superexp} E, \tau, \mathcal{R}_1) \cap (\text{superexp} E', \tau, \mathcal{R}_2) \\ &\quad \text{where } \text{split}(\mathcal{R}, 2) = \{\mathcal{R}_1, \mathcal{R}_2\} \\ \text{superexp}(S, \tau, \mathcal{R}) &= S_0; \{ S_0(a) \mapsto S(a) \mid a \in \text{outspace}(\tau) \}; \\ &\quad \{ e'_i \mapsto \text{superexp}(S(e_i), \tau', \mathcal{R}_i) \mid S_0(e_i) = e'_i \tau' \}_{i=1}^n \\ &\quad \text{where } \{e_i \tau_i\}_{i=1}^n = \text{outspace}(\tau) \setminus \text{T-Var} \\ &\quad \text{and } \{RS_i\}_{i=1}^n = \text{split}(\mathcal{R} \setminus \{S_0\}, n) \end{aligned}$$

The function superexp maps a unifier E_1 which acts on the variables in τ and an \mathcal{R} . It gives an E_2 with the same shape as E_1 but every substitution in E_2 acts the same as the corresponding substitution in E_1 on variables in τ and every other variable is renamed to a new and distinct variable. □

Theorem 8 (A most general unifier for every principal). *If E_1 is a principal unifier of Δ then there is an $E_1 \sqsubseteq E_2$ such that E_2 is a most general unifier of Δ .*

Proof. Let $E_2 = \text{superexp}(E_1, \Delta)$. That $E_1 \sqsubseteq E_2$ follows from the definition of superexp by structural induction on E_1 .

For E_2 to be most general: if $\text{solved}([E_3] \Delta)$ and $E_3 \sqsupseteq E_1; E_4$ then there is an E_5 such that $E_3 = E_2; E_5$. Proceed by induction on E_4 .

- $E_4 = \omega$. Then $E_5 = \omega$.
- $E_4 = e E'_4$. Then $E_3 = e E'_3$ and $E'_3|_{\Delta} = E_1; E'_4$ implies an E'_5 s.t. $E'_3 = E_2; E'_5$ by I.H., so $E_5 = e E'_5$.
- $E_4 = E'_4 \cap E''_4$. Similar to previous case.

- If $E_4 = S_4$. Then $E_2 = \text{superexp}(E_1, \text{tau}, RS)$ as above. Proceed by induction on E_1 .
 1. $E_1 = \omega$. Then $E_2 = \omega$ and $E_3 = \omega$ and $E_4 = E_5$.
 2. $E_1 = e E'_1$. Then $E_2 = e E'_2$ and $E'_2 = \text{superexp}(E'_1, \tau, RS)$ and $E_3 = e E'_3$ and $E'_3|_\tau = E'_1; [E_5]e$. By I.H., there is an E'_5 s.t. $E'_3 = E'_2; E'_5$. So $E_3 = E_2; \{e \mapsto E'_5\}$.
 3. $E_1 = E_1^1 \cap E_1^2$. Then $E_2 = \text{extract}(E_1^1, \tau, RS^1) \cap \text{extract}(E_1^2, \tau, RS^2)$ and $E_3 = E_3^1 \cap E_3^2$ and $E_3^i|_\tau = E_1^i; S_4$. By I.H., there are S_5^i s.t. $E_3^i = E_2^i; S_5^i$. For $\{i, j\} = \{1, 2\}$, $S_5^i(v) \neq \square(v)$ implies that either $S^j(v) = \square(v)$ or else $S^i(v) = S_4(v)$ and $S^j(v) = S_4(v)$ because substitutions in RS^1 and RS^2 have mutually disjoint ranges. So $S_5 = S_5^1 \cup S_5^2$ is well formed and $E_3 = E_2; S_5$.
 4. $E_1 = S_1$. Then $E_2 = S_2$ as defined by `extract` above and $E_3 = S_3$ and $S_3|_\tau = S_1; S_4$.
 Let $V = (\text{outspace}(\tau) \cap \text{T-Var}) \cup \{e_i \mid e_i \tau_i \in \text{outspace}(\tau)\}$. Then $S_1(e_i) = E_i$ and $S_2(e_i) = E'_i$ and $S_3|_{\tau_i} = E_i; S_4$. So by I.H., there is an S_i^5 s.t. $S_3(e_i) = E'_i; S_i^5$.
 The substitution $S_5^i = S_4 \cup S_1^5 \cup \dots \cup S_n^5$, where there are n E-variables in V , is well-formed because for distinct $\{i, j\} \subseteq \{1..n\}$, and for all v , either $v \in V$ and $S_i^5(v) = S_j^5(v)$ and $S_4(v) = S_i^5(v)$ or else $S_4(v) = \square(v)$ and $S_j^5(v) \neq \square(v)$ implies $S_j^5(v) = \square(v)$.
 Similarly, for variables $v \notin V$, $S_0(v)$ is distinct from $S_5^i(v)$. So $S_5 = S_5^i; (S_0)^{-1}; S_3|_{(\text{Var} \setminus V)}$.

□

Lemma 1 (Outer part unifiers expand to any restricted unifier). *If $\Delta \xrightarrow{\bar{E}} \Delta_1$ and $\text{solved}([\bar{E}'] \Delta)$ then there is an \bar{E}'' such that $\bar{E}'|_\Delta = \bar{E}; \bar{E}''$.*

Proof. By Definition 33 there is a $\delta \in \text{factor}(\Delta)$ and $(\delta, \text{Vars}(\Delta)) \xrightarrow{\text{runifier}} \bar{E}$. The proof is by structural induction on \bar{E}' . Proceed by case analysis, letting $V = \text{Vars}(\Delta)$ throughout.

1. $\bar{E}' = \omega$. Then $\bar{E}'' = \omega$.
2. $\bar{E}' = e \bar{E}'_1$. Then $\text{solved}([\bar{E}'_1] \Delta)$ and by I.H. there is a \bar{E}''_1 s.t. $\bar{S}'_1 = \bar{E}; \bar{E}''_1$. So $\bar{E}'|_\Delta = \bar{E}; e \bar{E}''$.
3. $\bar{E}' = \bar{E}'_1 \cap \bar{E}'_2$. Similar to the previous case.
4. $\bar{E}' \in \text{RSub}$. Proceed by induction on the constraint prefix \vec{e} where $\delta = \vec{e}(\tau_1 \doteq \tau_2)$.
 - (a) $\vec{e} = e_1 \cdot \vec{e}_1$ and $\bar{E} = e_1 / \bar{E}_1$. Then let $\Delta' = \{\delta_1 \mid e_1 \delta_1 \in \Delta\}$ and $\bar{E}'_1 = [\bar{E}'] e_1$, so $\text{solved}([\bar{E}'_1] \Delta_1)$. By I.H., there is an \bar{E}''_1 such that $\bar{E}'_1|_{\Delta'} = \bar{E}_1; \bar{E}''_1$, so $\bar{E}'|_\Delta = \bar{E}; \bar{E}''[e_1 \mapsto \bar{E}''_1]$.
 - (b) $\vec{e} = \epsilon$. Proceed by case analysis of the `runifier` rules.
 - i. (R T-unify). $\{\tau_1, \tau_2\} = \{a, \bar{\tau}\}$. and $\bar{E} = \{a \mapsto \bar{\tau}\}$. Then $[\bar{E}'] a = [\bar{E}'] \bar{\tau}$ and $\bar{E}' = \bar{E}; \bar{E}'$.
 - ii. (R E-unify). $\{\tau_1, \tau_2\} = \{e \bar{\tau}, \text{tau}\}$, and $(\tau, V) \xrightarrow{\text{reextract}} (E, V')$ and $\bar{E} = \{e \mapsto E\}$. Proceed by case analysis of τ .

- A. $\tau = \omega$. Then $E = \omega$ and $\bar{E}'(e) = \omega$ and $\bar{E}'' = \bar{E}'|_{[\bar{E}]} \Delta$.
 - B. $\tau = e_1 \tau'$. Then $E = e_1 E'$ and $(\tau', V) \xrightarrow{\text{rextract}} (E', V')$ and $\bar{E} = \{e \mapsto e_1 E'\}$ and $\bar{E}'(e) = e_1 E_1$.
Let $\bar{E}'_1 = \bar{E}'[e \mapsto E_1]$ and $\Delta' = e \bar{\tau} \doteq \tau'$. Then $\text{solved}([\bar{E}'_1] \Delta')$ and $\Delta' \xrightarrow{\text{runifier}} \bar{E}_1$ by I.H. there is a \bar{E}''_1 such that $\bar{E}'_1|_{\Delta'} = \bar{E}_1; \bar{E}''_1$.
So $\bar{E}'|_{\Delta} = \bar{E}; \bar{E}'[e_1 \mapsto \bar{E}''_1]|_{[\bar{E}]} \Delta$.
 - C. $\tau = \tau_1 \cap \tau_2$. Similar to the previous case.
 - D. $\tau \in \text{Typ}^{\rightarrow}$. Then E is a renaming and $\bar{E}'(e) \in \text{RSub}$. So $bE'|_{\Delta} = \bar{E}; \bar{E}'; (((\)^{-1} E); \bar{E}'(e))|_{[\bar{E}]} \Delta$.
- iii. (R N-unify) . No reduction.
- iv. (R Z-unify) . A restricted substitution unifier \bar{S}' for these forms is impossible. Because:
- A. there is a (R T-unify) occur check problem. Then for all $\bar{S}, [\bar{S}] \tau_1 \neq [\bar{S}] \tau_2$, by simultaneous induction on $[\bar{S}] a$ and $\bar{\tau}$, down to the position where a occurs in $\bar{\tau}$; similarly for (E-unifier), by induction down to the position where e occurs in τ .
 - B. $\tau_1 \doteq \tau_2$ has one of the forms $\bar{\tau} \doteq \omega$, $\bar{\tau} \doteq \tau_1 \cap \tau_2$, or $\omega \doteq \tau_1 \cap \tau_2$ where differing outer constructors preclude a substitution solution.
- So any solution must be an expansion instance of ω .

□

Theorem 9 (\Rightarrow gives principal restricted unifiers). *If $\Delta \xrightarrow{\bar{E}} \Delta_1$ and $\text{solved}(\Delta_1)$ and $\text{solved}([\bar{E}'] \Delta)$ then there is an expansion \bar{E}'' such that $\bar{E}'|_{\Delta} = \bar{E}; \bar{E}''$.*

Proof. By induction on the reduction $\Delta \xrightarrow{E} \Delta_1$.

1. Case $\text{solved}(\Delta)$ and $\bar{E} = \square$. Then $\bar{E}'' = \bar{E}'$.
2. Case $\Delta \xrightarrow{E_0} \Delta_2 \xrightarrow{\bar{E}_1} \Delta_1$. Then $\delta \in \text{factor}(\Delta)$ and $(\delta, \text{Vars}(\Delta)) \xrightarrow{\text{runifier}} \bar{E}_0$ and $\Delta_2 = \text{factor}([\bar{E}_0] \Delta)$. By Lemma 1, there is an \bar{E}'_1 such that $[\bar{E}'] \Delta = [\bar{E}_0; \bar{E}'_1] \Delta$. By Theorem 3, $\text{solved}([\bar{E}_0; \bar{E}'_1] \Delta) \Leftrightarrow \text{solved}([\bar{E}'_1] \Delta_2)$. By I.H., $\text{solved}([\bar{E}'_1] \Delta_2)$ implies there is an \bar{E}''_1 such that $\bar{E}'_1|_{\text{Vars}(\Delta_2)} = \bar{E}_1; \bar{E}''_1$. So $\bar{E} = \bar{E}_0; \bar{E}_1$ and $\bar{E}'' = \bar{E}''_1$ and $\bar{E}'|_{\text{Vars}(\Delta)} = \bar{E}; \bar{E}''$.

□

Theorem 10 (Type matching is sound and complete).

1. If $(\tau_1, \tau_2) \xrightarrow{\text{match}} E$ then $[E] \tau_1 = \tau_2$.
2. If $[E] \tau_1 = \tau_2$ then $(\tau_1, \tau_2) \xrightarrow{\text{match}} E_1$ for some E_1 . □

Proof.

1. By structural induction, following the definition of match.
2. By simultaneous structural induction on τ_1, τ_2, E (many cases are combined in the following proof). In this direction we also show that $E \sqsubseteq E_1$, i.e., there is an E_2 such that $E = E_1 \sqcup E_2$.

- (a) $\tau_1 = a$, $E = \{a \mapsto \tau_2\} \sqcup S$. Then $E_1 = \{a \mapsto \tau_2\}$ and $E \sqsupseteq E_1$.
- (b) $\tau_1 = \tau_3 \rightarrow \tau_4$, $\tau_2 = \tau_5 \rightarrow \tau_6$, $E \in \mathbf{Subst}$. Then $[E] \tau_3 = \tau_5$ and $[E] \tau_4 = \tau_6$.
By I.H., $(\tau_3, \tau_5) \xrightarrow{\text{match}} S_1$ and $(\tau_4, \tau_6) \xrightarrow{\text{match}} S_2$ and $E \sqsupseteq S_1$ and $E \sqsupseteq S_2$
thus $E_1 = S_1 \sqcup S_2$ is defined and $E \sqsupseteq E_1$.
- (c) $\tau_1 = \omega$, $\tau_2 = \omega$, $E \in \mathbf{Subst}$. Then $E_1 = \sqsupset$ and $E \sqsupseteq E_1$.
- (d) $\tau_2 = \omega$, $E = \omega$. Then $E_1 = \omega$.
- (e) $\tau_1 = \tau_3 \cap \tau_4$, $\tau_2 = \tau_5 \cap \tau_6$, $E \in \mathbf{Subst}$. Then $[E] \tau_3 = \tau_5$ and $[E] \tau_4 = \tau_6$.
By I.H., $(\tau_3, \tau_5) \xrightarrow{\text{match}} S_1$ and $(\tau_4, \tau_6) \xrightarrow{\text{match}} S_2$ and $E \sqsupseteq S_1$ and $E \sqsupseteq S_2$
thus $E_1 = S_1 \sqcup S_2$ is defined and $E \sqsupseteq E_1$.
- (f) $\tau_2 = \tau_3 \cap \tau_4$, $E = E_2 \cap E_3$. Then $[E_2] \tau_1 = \tau_3$ and $[E_3] \tau_1 = \tau_4$. By I.H.,
 $(\tau_1, \tau_3) \xrightarrow{\text{match}} E_4$ and $(\tau_1, \tau_4) \xrightarrow{\text{match}} E_5$ and $E_2 \sqsupseteq E_4$ and $E_3 \sqsupseteq E_5$ thus
 $E_1 = E_4 \cap E_5$ is defined and $E \sqsupseteq E_1$.
- (g) $\tau_1 = e_1 \tau_3$, $\tau_2 = e_1 \tau_4$, $E \in \mathbf{Subst}$. Then $[E(e_1)] \tau_3 = \tau_4$. By I.H., $(\tau_3, \tau_4) \xrightarrow{\text{match}} E_2$ and $E(e_1) \sqsupseteq E_2$ thus $E_1 = \{e_1 \mapsto E_2\}$ is defined and $E \sqsupseteq E_1$.
- (h) $\tau_2 = e \tau_3$ and $E = e E_2$. Then $[E_2] \tau_1 = \tau_3$. By I.H., $(\tau_1, \tau_3) \xrightarrow{\text{match}} E_3$
and $E_2 \sqsupseteq E_3$ thus $E_1 = e E_3$ is defined and $E \sqsupseteq E_1$. \square

Lemma 2 (opus sets are coverings). *If solved($[S_1] \Delta$) and $\Delta \xrightarrow{\text{rfactor}} \Delta$ and $\delta \in \Delta$ and not solved(δ) then $(\delta, \text{Vars}(\Delta)) \xrightarrow{\text{opus}} S$ and there are substitutions $S_2 \in S$ and S_3 such that $S_1|_{\Delta} = S_2; S_3$ and $\text{eemeasure}(S_1, \Delta) > \text{eemeasure}(S_3, [S_2] \Delta)$.*

Proof. By case analysis of unsolved simplified δ following the structure of the definition of opus, which is a complete case analysis of all such forms.

1. $e \delta$.

Let $\{S_i\}_{i=1}^n$ be the set of substitution-unifiers of δ at the n leaves of $S_1(e)$.
By I.H., there are $\{S'_i\}_{i=1}^n, \{S''_i\}_{i=1}^n$ such that $\delta \xrightarrow{\text{opus}} (\{S'_i\}_{i=1}^n \cup S')$. Let E
be an expansion of the same shape as $S_1(e)$, with the projection-expansion
 $\{e_i \mapsto S''_i\} \cup \{e_j \mapsto \omega \mid j \in \{1, \dots, n\} \setminus \{i\}\}$ replacing each S_i . Thus $S_3 =$
 $\{e \mapsto E\}$.

2. $a \doteq \tau$ where $a \notin \text{ovars}(\tau)$.

Then $S_2 = \{a \mapsto \tau\}$ and $S_3 = S_1[a \mapsto a]$.

3. $e \tau \doteq \tau_1$, and $S_1(e) = S_4$.

Then $S_2 = \{e \mapsto R\}$ and $S_3 = S_1 \cup (((\)^{-1} R); S_4)$.

4. $e \tau \doteq \omega$, and $S_1(e) = \omega$. Then $S_2 = \{e \mapsto \omega\}$ and $S_3 = S_1[e \mapsto e \sqsupset]$.

5. $e \tau \doteq \tau_1 \cap \tau_2$, and $S_1(e) = E_1 \cap E_2$.

Then $S_2 = \{e \mapsto e_1 \sqsupset \cap e_2 \sqsupset\}$ and $S_3 = S_1[e \mapsto e \sqsupset, e_1 \mapsto E_1, e_2 \mapsto E_2]$.

6. $e_1 \tau_1 \doteq e_2 \tau_1$ and $e_1 \neq e_2$.

Let $\{E_i\}_{i=1}^n$ and $\{E'_i\}_{i=1}^n$ be the expansions at the leaves of the common
part of $S_1(e_1)$ and $S_1(e_2)$. For each $1 \leq i \leq n$, either E_i or E'_i must be
a substitution (if both were something else they could not be a leaf of the
common part).

Thus for each $1 \leq i \leq n$: E_i is a substitution and $S'_i = E_i[e_4 \mapsto E'_i]$ and $\delta_i =$
 $\tau_1 \doteq e_4 \tau_2$ and solved($[S'_i] \delta_i$); or E'_i is a substitution and $S'_i = E'_i[e_3 \mapsto E_i]$
and $\delta_i = e_3 \tau_1 \doteq \tau_2$ and solved($[S'_i] \delta_i$).

By I.H., there are $\{S''_i\}_{i=1}^n, \{S'''_i\}_{i=1}^n$ such that $\delta_i \xrightarrow{\text{opus}} (\{S''_i\}_{i=1}^n \cup S')$. Let
 E be an expansion of the same shape as the common part of $S_1(e_1)$ and

$S_1(e_2)$, with the following projection substitution at each leaf i : $\{e_k^L \mapsto S_i'''\} \cup \{e_j^R \mapsto \omega \mid j \in \{1, \dots, p\}\} \cup \{e_j^L \mapsto \omega \mid j \in \{1, \dots, q\} \setminus \{k\}\}$ when E_i' is a substitution and S_i'' is S_k^L ; swap the L and R labels when E_i is a substitution.

Theorem 11 ($\Rightarrow_{\mathcal{C}}$ gives covering substitution-unifier sets). *If solved($[S_1] \Delta$) then there are S_2, S_3, Δ_1 such that $\Delta \xrightarrow{\mathcal{C}}_{S_2} \Delta_1$ and solved(Δ_1) and $S_1|_{\Delta} = S_2; S_3$.*

Proof. To see that there is some finite solving $\Rightarrow_{\mathcal{C}}$ reduction of Δ : every direct reduction corresponds to one or more steps in the application of S_1 . We do not give a more detailed proof in this paper. The existence of S_2 and S_3 is a simple induction using Lemma 2. \square

Lemma 3 (β -normal form pretypings strongly normalise). *If M is a β -normal form and $M \xrightarrow{\text{pretyping}} P$ then P has no infinite $\Rightarrow_{\mathcal{P}}$ reductions.*

Proof. Every applied term in M is a variable or an application. So P has the form $\langle A \vdash \tau \rangle / \{\vec{e}_i (a_i \doteq \tau_i \rightarrow \tau'_i)\}_{i=1}^n$ where $i \neq j$ implies $\vec{e}_i \neq \vec{e}_j$ or $a_i \neq a_j$. The only possible reductions of such constraint sets use the outer-part $\xrightarrow{\text{unifier}}$ rules:

1. (T-unify) gives a smaller pretyping of the same general form;
2. (Z-unify) gives a smaller pretyping, should an occur clash happen. This situation does not arise, but it is not necessary to show this for the lemma. \square

Theorem 12 (Constraint reduction simulates β -reduction). *If $M \xrightarrow{\text{pretyping}} P$ and M is β -normalising then $P \xRightarrow{\mathcal{P}} \emptyset$. If $M \xrightarrow{\text{pretyping}} P$ and M is not β -strongly normalising then P has an infinite $\Rightarrow_{\mathcal{P}}$ reduction.*

Proof. We use the function eExtract , a variant on extract giving the set of E-variable paths in the topmost extraction together with the types they lead to.

$$\begin{aligned} \text{eExtract}(\bar{\tau}) &= \{\bar{\tau}\} \\ \text{eExtract}(\omega) &= \emptyset \\ \text{eExtract}(\tau_1 \cap \tau_2) &= \text{eExtract}(\tau_1) \cup \text{eExtract}(\tau_2) \\ \text{eExtract}(e \tau_1) &= \{e \tau \mid \tau \in \text{eExtract}(\tau_1)\} \end{aligned}$$

If M is a β -normal form use Lemma 3 otherwise the result follows by a chase of the following diagram.

$$\begin{array}{ccc} M_1 & \xrightarrow{\beta} & M_2 \\ \text{sim} \downarrow & & \downarrow \text{sim} \\ P_1 & \dashrightarrow_{\mathcal{P}} & P_2 \end{array}$$

By simultaneous structural induction on M_1, M_2, P_1 , following the definition of β -reduction.

1. Case $M_1 = (\lambda x. M_3) @ M_4$, $M_2 = M_3[x := M_4]$,
 $P_1 = \langle A_3[x : \omega] \uplus e A_4 \vdash a_0 \rangle / \Delta$ where $\Delta = \Delta_3 \cup e \Delta_4 \cup \{\tau \doteq e \tau_4, \tau_3 \doteq a_0\}$
and $\text{eExtract}(\tau) = \{\vec{e}_i a_i\}_{i=1}^n$. The n \vec{e}_i and a_i of τ are all distinct because each is a path to a distinct free occurrence of x in M_3 . The only occurrences

of a_0 in P_1 are those shown; e does not occur in Δ_3 ; a_i do not occur in Δ_4 .
Reduction to P_2 proceeds as follows where $S = \{\vec{e}_i / \{a_i \mapsto [S_i] \tau_4\}\}_{i=1}^n$.

$$\begin{aligned} & \xrightarrow[\text{r}]{\{a_0 \mapsto \tau_3\}} \langle A_3[x : \omega] \uplus e A_4 \vdash \tau_3 \rangle / \Delta_3 \cup e \Delta_4 \cup \{\tau \doteq e \tau_4\} \\ & \xrightarrow[\text{r}]{\{e \mapsto E\}} \langle A_3[x : \omega] \uplus \{\vec{e}_i [S_i] A_4\}_{i=1}^n \vdash \tau_3 \rangle / \Delta_3 \cup \{\vec{e}_i [S_i] \Delta_4\}_{i=1}^n \cup \{\vec{e}_i (a_i \doteq [S_i] \tau_4)\}_{i=1}^n \\ & \xrightarrow[\text{r}]{S} \langle [S] A_3[x : \omega] \uplus \{\vec{e}_i [S_i] A_4\}_{i=1}^n \vdash [S] \tau_3 \rangle / [S] \Delta_3 \cup \{\vec{e}_i [S_i] \Delta_4\}_{i=1}^n = P_2 \end{aligned}$$

By the following structural induction on M_3 , $M_3[x := M_4] \xrightarrow{\text{pretyping}} P_2$.

- (a) Case M_3 where $M_3[x := M_4] = M_3$. Then $n = 0$ and $P_2 = [S] P_3$ and $M_3 \xrightarrow{\text{pretyping}} P_2$ by choosing different variables.
 - (b) Case $M_3 = x$. Then $\Delta_3 = \emptyset$, $n = 1$, $\vec{e}_1 = \epsilon$ and $P_2 = [S_1] P_4$ and $M_4 \xrightarrow{\text{pretyping}} P_4$ implies $M_3[x := M_4] \xrightarrow{\text{pretyping}} P_2$ by choosing different variables.
 - (c) Case $M_3 = \lambda y. M_5$. Then $M_3 \xrightarrow{\text{pretyping}} \langle A_5[y : \omega] \vdash A_5(y) \rightarrow \tau_5 \rangle / \Delta_5$ and (1) $M_3[x := M_4] = M_3$ and $y = x$ and $P_2 = [S] P_3$. or (2) $M_3[x := M_4] = \lambda y. M_5[x := M_4]$, and $y \neq x$ and y is not a free variable in M_5 by our assumption of capture-free substitution, thus by I.H. on M_5 , $M_5[x := M_4] \xrightarrow{\text{pretyping}} \langle [S] A_5 \uplus \{\vec{e}_i [S_i] A_4\}_{i=1}^n \vdash [S] \tau_5 \rangle / [S] \Delta_5 \cup \{\vec{e}_i [S_i] \Delta_4\}_{i=1}^n$ and $M_3[x := M_4] \xrightarrow{\text{pretyping}} P_2$.
 - (d) Case $M_3 = M_5 @ M_6$. Similar to the previous case.
2. Case $M_3 @ M_4, M_5 @ M_4, \langle A_3 \cap e A_4 \vdash \mathbf{a}_0 \rangle / \Delta_3 \cup e \Delta_4 \cup \{\tau_3 \doteq e \tau_4 \rightarrow \mathbf{a}_0\}$ where $M_3 \xrightarrow{\beta} M_5$. Then $M_3 \xrightarrow{\text{pretyping}} \langle A_3 \vdash \tau_3 \rangle / \Delta_3$ implies there is a $\langle A_5 \vdash \tau_5 \rangle / \Delta_5$ s.t. $M_5 \xrightarrow{\text{pretyping}} P_5$ and $P_3 \xrightarrow[\text{r}]{E} P_5$ by I.H. So $P_1 \xrightarrow[\text{r}]{E} P_2$ and $M_2 \xrightarrow{\text{pretyping}} P_2$ where $P_2 = \langle A_5 \cap e A_4 \vdash \mathbf{a}_0 \rangle / \Delta_5 \cup e \Delta_4 \cup \{\tau_5 \doteq e \tau_4 \rightarrow \mathbf{a}_0\}$.
 3. Cases $M_1 = M_4 @ M_3$ and $M_1 = \lambda y. M_3$ where $M_3 \xrightarrow{\beta} M_5$ are similar to the previous case. \square

proposition 6 (System $E^{\cap, \rightarrow}$ -typable terms are normalising). *A term M of the λI -calculus is System $E^{\cap, \rightarrow}$ -typable iff it is β -normalizing.*

Proof. The system of intersection types without ω considered in [4] is shown to type precisely the β -normalizing terms. System $E^{\cap, \rightarrow}$ adds the (ev) typing rule, which adds no typing power. It omits several features of the [4] system, but it extends the system of *strict* intersection types [22] (without ω), which is still sufficient to type the normalizing terms. \square

Lemma 5 (All System $E^{\cap, \rightarrow}$ pretypings expand initial pretyping). *If $M : P$ is derivable in System $E^{\cap, \rightarrow}$ and $M \xrightarrow{\text{pretyping}} P_1$ then there exists an expansion E such that $P = [E] P_1$.*

Proof. By cases on the judgement $M : P$.

- Case $M : \langle e A' \vdash e \tau' \rangle / \Delta' \cup \Delta''$. By I.H. $\exists E^i. \langle A^i \vdash \tau^i \rangle / \Delta^i = [E^i] P_1$ so $E = E' \cap E''$.

- Case $M : \langle e A' \vdash e \tau' \rangle / \{ e \delta \mid \delta \in \Delta' \}$ By I.H. $\exists E'. \langle A' \vdash \tau' \rangle / \Delta' = [E'] P_1$ so $E = e E'$.
- Case $x : \langle (x : \tau) \vdash \tau \rangle / \emptyset$. so $P_1 = \langle (x \mapsto \tau) \vdash \tau \rangle / \emptyset$ and $E = \{ a \mapsto \tau \}$.
- Case $\lambda x. M' : \langle A' [x \mapsto \omega] \vdash A'(x) \rightarrow \tau' \rangle / \Delta'$. Then $P_1 = \langle A'' [x \mapsto \omega] \vdash A''(x) \rightarrow \tau'' \rangle / \Delta''$. By I.H. $\exists E'. \langle A' \vdash \tau' \rangle / \Delta' = [E'] (\langle A'' \vdash \tau'' \rangle / \Delta'')$ so $E = E'$.
- Case $M' @ M'' : \langle A' \cap A'' \vdash \tau \rangle / \tau' \doteq \tau'' \rightarrow \tau \cap \Delta' \cap \Delta''$ and $\tau' = \tau'' \rightarrow \tau$ as P is a typing. Then $P_1 = \langle A'_1 \cap e A''_1 \vdash a \rangle / \Delta'_1 \uplus e \Delta''_1 \uplus \text{simplify}(\{ \tau'_1 \doteq e \tau''_1 \rightarrow a \})$ and e, a do not occur outermost in Δ_1 .
By I.H. $\exists E^i. \langle A^i \vdash \tau^i \rangle / \Delta^i = [E^i] (\langle A^i_1 \vdash \tau^i_1 \rangle / \Delta^i_1)$. As P is a typing, $[E^1] \tau'_1 = [E^2] \tau''_1 \rightarrow \tau$. So E^1 is a substitution and $E = E^1 [e \mapsto E^2] [a \mapsto \tau]$. \square

Theorem 13 (Undecidability of ω -free unification with E-variables).
In the ω -free restriction of unification with E-variables, for any constraint set Δ the existence of an expansion E such that $\text{solved}([E] \Delta)$ is undecidable.

Proof. By reduction of the undecidable β -normalisation problem for λI terms. By Lemma 6, M is β -normalising iff it is typable in System $E^{\cap, \rightarrow}$; by Lemma 5, the constraint set Δ where $M \xrightarrow{\text{pretyping}} \langle A \vdash \tau \rangle / \Delta$ is soluble iff M is typable in System $E^{\cap, \rightarrow}$; hence unifiability is undecidable for the constraints generated by $\xrightarrow{\text{pretyping}}$. \square

Theorem 14 (Completeness of \Rightarrow_r for ω -free unification). *If M is a λI term and $M \xrightarrow{\text{pretyping}} \langle A \vdash \tau \rangle / \Delta$ then there is an ω -free expansion E such that $\text{solved}([E] \Delta)$ iff $\Delta \Rightarrow_r \emptyset$.*

Proof. If M is β -normalising then Δ is insoluble by the proof of Theorem 13, and $\Delta \Rightarrow_r \emptyset$ by Theorem 12. If M is not β -normalising then Δ is insoluble by the proof of Theorem 13, and \Rightarrow_r is sound so it will not reduce Δ to \emptyset . \square