Extensions and Applications of Higher-order Unification

Conal M. Elliott

May, 1990
CMU-CS-90-134

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at Carnegie Mellon University.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404, and in part by NSF Grant CCR-8620191.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA or the U.S. government.

Keywords: unification, higher-order logic, logical frameworks, language encoding, program transformation

Abstract

This dissertation explores the problem of unification in various typed λ-calculi, developing and proving the correctness and completeness of unification algorithms for various calculi within a single general framework, and then demonstrating the practical importance of these algorithms by means of example applications.

We begin by presenting our general framework for unification, based on transformations of unification problems. Then, in this framework, we develop a new unification algorithm for a λ-calculus with dependent function (Π) types. This algorithm is especially useful as it provides for mechanization in the very expressive *Logical Framework (LF)*. The development involves significant complications not arising Huet's corresponding algorithm for the simply typed λ-calculus, primarily because it must deal with ill-typed terms. We then extend this algorithm first for dependent product (Σ) types, and second for implicit polymorphism. In the latter case, the algorithm is incomplete, though still quite useful in practice.

The last part of the dissertation provides examples of the usefulness of the algorithms. The general idea is to use a λ-calculus as a *meta-language* for representing various other languages (object-languages). The rich structure of a typed λ-calculus, as opposed to traditional, first-order abstract syntax trees, allows us to express rules, *e.g.*, program transformation and logical inference rules, that are more succinct, more powerful, and easier to reason about. We can then use unification in the meta-language to mechanize application of these rules.

Acknowledgements

I am grateful to many people for help and encouragement during my studies at Carnegie Mellon.

Frank Pfenning, my thesis advisor, has been a good friend and the best advisor one could hope for. Our collaborations have been stimulating and enjoyable. He helped me through periods of discouragement, and he set me on and kept me on the line of research and writing resulting in this thesis. His careful reading of and comments on drafts of the thesis have contributed greatly to its clarity and correctness.

The other members of my committee have also given generously of their time with reading and comments.

The ERGO project, founded by Bill Scherlis and Dana Scott, has been a terrific group to be part of. I greatly enjoyed the many software design and implementation projects, with the ERGO emphasis on conceptual elegance. Bill Scherlis, my first advisor, introduced me to many wonders, including ML, lazy evaluation, and program transformation.

Our close friends Ron and Mary Joy Collins and Debbie and Dave Kresh helped to provide happiness and spiritual growth during the Pittsburgh years. Because of them, it was hard to leave, in spite of our joy to return to family in Northern California. The loss of Debbie's physical presence this year, due to cancer, is mourned by those of us who were blessed to share in her life.

Our delightful children, Jacob, Becky, Charlotte, and Patrick, keep me real. My parents have patiently waited for us to return from Pittsburgh with their grandchildren. They have been supportive, emotionally and financially, when we most needed it.

Most of all, I am grateful to my wife Marianne, who has endured being taken out of her native Californian habitat for six years, and who loves me no matter what.

to Marianne

Contents

Chapter 1

Introduction

This dissertation addresses the problem of performing unification in various typed λ-calculi. It was motivated by and builds upon two areas of research whose importance is becoming increasingly recognized in computer science. The first of these is the mechanization of fragments of higher-order logic by means of higher-order unification, *i.e.*, unification in the typed λ-calculus (modulo $\beta\eta$-conversion). The second area is the use of rich type theories for formalizing theorem proving and program development.

The primary contribution of this thesis is to combine the advantages of mechanizability of the simply typed λ-calculus with the increased expressive power of these richer calculi, by developing new unification algorithms for calculi with dependent function types, products (again a dependent version, sometimes referred to as "strong sum" or simply "Σ" types), and implicit polymorphism. These new algorithms have important applications in the general area of formal language analysis and manipulation, for example, mechanically assisted theorem proving in a wide range of logics, automated or semi-automated type inference or type checking in various typed languages, and mechanically assisted program transformation. Of course, these various forms of applications have been implemented before for various languages (including logics). Our algorithms provide a general tool that can be more easily applied to a variety of languages than previously existing tools.

As a secondary contribution, we offer a new presentation of this kind of algorithm, which we hope serves to clarify the issues involved in these and other problems and algorithms.

Although many of the important ideas in our algorithms have their roots in Huet's algorithm for higher-order unification [36], there are serious technical difficulties that arise only in extensions to richer calculi. One is the necessity to deal with ill-typed terms during the unification process. Our technique for dealing with ill-typedness significantly complicates the proofs, but fortunately requires little additional complexity in the algorithms.

1.1 Higher-order Unification

The development of (first-order) unification, first studied by Herbrand [31], had a major impact on the field of automated theorem proving in first-order logic, because it was the key component of the new mechanization procedure *resolution*, due to Robinson [67, 66] (who also reintroduced unification). Because of the success of resolution, much work was focused on the efficient implementation of unification. (See [39] for a survey.) Another significant development was the observation that a subset of first-order logic, now called *Horn logic*, could serve as an elegant programming language, for which interpretation was performed by a restricted form of resolution known as *SLD resolution* (which is complete for Horn logic) [75]. Colmerauer and Roussel first implemented the programming language Prolog based on this idea [9].

In 1940, Church had formulated a *higher-order logic*, based on the incorporation of simple types in his λ-calculus. Formulas and proofs in higher-order logic can be much more succinct than their corresponding versions in first-order logic.

Given the success of first-order resolution in mechanizing first-order logic, it was natural to consider the possibility of higher-order resolution for mechanizing higher-order logic. This form of resolution depends on being able to enumerate complete sets of unifiers (CSUs) in the simply typed λ-calculus ($\lambda_{\rightarrow}$). Guard [27] pointed out that CSUs must sometimes be infinite, and the general problem of unifiability was shown to be undecidable by Lucchesi [40], Huet [35], and Goldfarb [26]. Goldfarb showed undecidability for even a restriction of the problem to second-order unification, with a single binary function constant. In contrast, Huet [34, 37] showed second-order matching to be decidable, and Farmer [21] showed decidability of *monadic* second-order unification. Decidability of higher-order matching is still an open problem. A complete algorithm for enumerating CSUs was presented by Jensen and Pietrzykowski [38], but it had the problem of being extremely undirected for certain kinds of problems. Huet showed that it is impossible in general to enumerate *minimal* (nonredundant) CSUs [34].

In [34], Huet presented a *pre-unification* algorithm that avoids some of the problems of undirectedness and redundancy. The key new idea was to postpone unification subproblems of a certain form, called "flexible-flexible". He noted that these subproblems were the source of unification's extreme undirectedness, and, importantly, proved that when all but flexible-flexible subproblems are eliminated, the remainder must always be unifiable. Thus a pre-unification algorithm suffices for unifiability. It also turns out to suffice for resolution, in which the remaining flexible-flexible subproblems are saved to be added to future unification problems. The addition of these new unification constraints often cause some of the flexible-flexible subproblems to be instantiated into forms in which they can be further reduced.

While the original purpose of higher-order unification was higher-order resolution, many diverse applications followed. Andrews developed the technique of *matings* for automated

theorem proving in higher-order logic [1]. Huet and Lang showed how to use a fragment of $\lambda_{\rightarrow}$ to encode program transformation rules and then use second-order matching and substitution to automatically apply them [37].

Just as first-order unification and first-order resolution led to Prolog, Nadathur applied higher-order unification and higher-order resolution in the design of a new programming language λProlog [53]. Although first based on a higher-order Horn logic, Miller *et al.* generalized the logic to *higher-order hereditary Harrop formulas*, which include use of explicit existential and universal quantification and implication [49, 47]. This extra expressiveness has proved extremely useful in many applications dealing with the manipulation of programs and formulas [50]. Some other applications of λProlog have been computational linguistics [46], specifying and implementing theorem provers for various logics [22, 23], program analysis [28], partial type inference in the ω-order polymorphic λ-calculus [59], and explanation based generalization (EBG) [16]. Donat and Wallen [17] also used higher-order unification for EBG. Paulson's Isabelle system for theorem proving in logics encoded in higher-order logic [56, 57] is similar in spirit to the λProlog work of Felty and Miller.

1.2 Richer Type Theories

Many of the applications of higher-order unification listed above involve the use of $\lambda_{\rightarrow}$ for representing (encoding) various formal languages, in particular logics and programming languages.[1] Another area of research has been the exploration of richer type theories and their application to formalizing theorem proving and program development. One very important paradigm is that of "formulas as types" introduced by Curry [13] and Howard [32], who observed a correspondence between types and terms of a given type on the one hand, and formulas and proofs of a given formula on the other. This correspondence has been further pursued in the work of deBruijn's group [15], Martin-Löf (*e.g.*, [42]), NuPrl [10], and the Calculus of Constructions (CoC) [11].

More recently, a related but different approach has been proposed for representing logics within a type theory. The *Logical Framework (LF)*, based on $\lambda_{\rightarrow}$ extended with *dependent function* (Π) *types*, makes a correspondence between types families (functions from terms to types) and the fundamental units of inference systems, called *judgements* (following Martin-Löf [43]). Logical formulas are encoded as terms rather than types, and the judgment type families are applied to these terms. The intent of LF is different from the previous formalisms, since it is intended for supporting not just one, but a wide range of logics, even nonconstructive ones. It is thus presented as a "first step towards a general theory of interactive proof checking and proof construction." A crucial property of LF, due to the rich representations allowed by dependent types, is that proof checking in appropriately encoded

[1] Variations on this general idea have also been suggested by Martin-Löf as a "system of arities" [43], and by Pfenning and Elliott as "higher-order abstract syntax" [60].

languages is reduced to type checking in the representing typed λ-calculus, and thus the decidability of type-checking is vital. As pointed out in [30], this is of great practical value, because it allows for the implementation of general tools, *e.g.*, an interactive proof editor, that work for a variety of logical systems. In comparison with CoC, the LF type theory is very weak, having the same computational power as the simply typed λ-calculus. The problem of unification in CoC's type theory seems to be significantly harder than unification in LF's type theory.

The enhanced representational ability of extending the $\lambda_{\rightarrow}$ with product types and implicit polymorphism (free type variables) has been demonstrated in [60]. The usefulness of these extensions comes from the fact that many languages contain constructs that are made up of a variable number of components (*e.g.*, a parallel "let" binding expression, as in ML or Lisp). The addition of dependent function types also allows the direct representation of the typing systems of various languages. This was observed in the LF encoding in [30] of Church's higher-order logic, and is explored in Chapter 7 of this thesis.

Pfenning has designed a programming language *Elf* [58] that combines the ideas of LF and λProlog. There is an implementation of it in Standard ML [19].

1.3 Overview of the Thesis

1.3.1 The Calculus λ_{Π}

We begin in Chapter 2 by presenting the calculus "λ_{Π}", which is an extension of the $\lambda_{\rightarrow}$ in two ways: First, in place of a simple function type $A{\rightarrow}B$, the type of the result of applying a function in λ_{Π} may depend on the term to which the function is applied. These types are written "$\Pi v{:}A.\ B$", where B may depend on (contain free occurrences of) v. Second, in order for such a B to depend on the variable v, the base types of $\lambda_{\rightarrow}$ are generalized to type families, as indexed by zero or more appropriately typed terms. This calculus is the one used by the Logical Framework (LF) [30] (which itself is derived from members of the AUTOMATH family of languages [15]) for the purpose of encoding the syntax, rules and proofs of a wide class of logics.

After presenting the syntax of the terms, types, and kinds of λ_{Π} and their associated typing rules, we go on to present our somewhat unconventional definition of substitutions and their composition operation. Our definition has the advantage of eliminating temporary variables, which arise frequently in unification. We demonstrate by example the difference between the conventional definition and ours, and show how to make our notion of substitution practical, by giving a compact representation and a method for composition of substitutions using this representation.

The conversion rules of λ_Π are the β and η rules at the level of terms and the level of types. These are extended in the usual way to the one-step and multi-step subterm reducing relations $\rightarrow_{\beta\eta}$ and $\rightarrow^*_{\beta\eta}$, and to the convertibility relation $\leftrightarrow^*_{\beta\eta}$. The Church-Rosser (CR) property for λ_Π with η as well as β has, for some time, been generally believed to be true. This conjecture has only recently been verified and the proof is quite complex [68]. We describe two alternatives to relying on CR. The importance of CR, together with the strong normalization property (SN), which is fairly easy to show, is that they reduce the question of convertibility (of well-typed terms and types) to equivalence (modulo α-conversion) of normal forms.

1.3.2 An Approach to Unification

Chapter 3 presents our framework for specifying unification in various calculi and developing and proving the correctness and completeness of algorithms for pre-unification in these calculi. Our approach is related to the transformation-based approaches of Martelli and Montanari for first-order unification [41] and of Snyder and Gallier for higher-order and equational unification [70, 69], which was itself inspired by the work of Martelli and Montanari [41]. However, as discussed below, unlike these works, our approach makes the important distinction between two kinds of "nondeterminism" present in the search for unifiers. This distinction is necessary to formulate an algorithm for enumerating *complete* and *minimal* sets of solutions (as defined in the chapter).

We begin by defining the notion of a *unification problem*, which encapsulates the information gained in making progress toward a subset of possible unifiers of an original pair of terms or types. Next, we define the *set of all solutions* of a unification problem, and then *minimal complete sets of pre-unifiers* (μCSPs). For us, a pre-unifier is not a substitution, but rather a special kind of unification problem (*solved form*), whose set of solutions is a subset of the set of solutions of a given unification problem.

We then present the notion of *transformations* on unification problems, central to our framework, which are relations between unification problems and sets of unification problems. Each particular pre-unification algorithm is given (in later chapters) as a collection of transformations. These transformations are required individually to have the property of *validity* and collectively to have the property of *completeness*. The purpose of such a collection of transformations is to be able to ultimately transform a unification problem into a μCSP.

We then define how a collection of transformations generates a set of pre-unifiers of a given unification problem by means of a nondeterministic search process. There are two kinds of choices are made in this process: first which unification problem to work on next, and second which transformation to apply and how to apply it. It turns out that the second kind of choice may be made completely arbitrarily, but, in order to have completeness, the

first kind must be done in a fair way. Finally, we prove that collections of transformations satisfying the validity and completeness properties do generate μCSPs.

1.3.3 A Pre-unification Algorithm

Chapter 4 presents the development of an algorithm for HOU_Π, *i.e.*, pre-unification in λ_Π, as a collection of transformations. The algorithm we construct is similar to Huet's algorithm for $\text{HOU}_\to$ (pre-unification in λ_Π). Under additional assumptions about the control structure, it behaves almost exactly the same on the subset of λ_Π corresponding to $\lambda_\to$. However, there are considerable technical difficulties in the justification of the algorithm that do not arise in $\text{HOU}_\to$.

Normal forms play a vital role in the development, and we begin by defining one that will be needed later in the chapter. This is the *β weak head normal form (WHNF)*, whose purpose is to reveal just the top level structure of the β normal form of a term or type. Interestingly, the role of η is quite different from β. In contrast, Huet used a long normal form (LNF), based on β-reduction and η-expansion.

Then we develop several useful properties of convertibility. Each of these properties may be interpreted as a decomposition method, in the following sense: Given any pair of well-typed terms or types of the same type or kind, one of the defined methods either shows that they are not convertible, or constructs a set of pairs of terms and/or types that (a) has the same set of (simultaneous) unifiers, (b) satisfies a certain *relative well-typedness* property, and (c) is, in a sense defined in the chapter, "smaller" than the given pair. One application of this set of decomposition methods is as an algorithm to test for convertibility. (In fact, it is a specialization of our HOU_Π algorithm when there are no unification variables.) However, the main purpose is to lay the groundwork for the formulation and justification of the transformation rules.

Next we examine some of the issues that arise in HOU_Π. Of particular importance is the need to deal with ill-typedness during pre-unification (unlike $\text{HOU}_\to$, where it is easily avoided). Managing ill-typedness is a major consideration in our transformations and their justification. Fortunately however, the extra complexity is more in the proofs than in the transformations themselves. We define an invariant on unification problems called "acceptability", on which the transformations depend and which they maintain. The main feature of an acceptable unification problem is that the ill-typedness present is "accounted for", as ensured by the existence of a certain kind of partial order on the pairs being unified.

With this background, we are ready to construct the transformations that make up our algorithm and prove their correctness. The decomposition methods defined previously yield with little additional work three transformations, which we call the "redex", "abstraction", and "rigid-rigid" transformations. The idea in each of these transformations is to either

show nonunifiablity or to replace a chosen disagreement pair (pair of terms or types being unified) with a finite collection of simpler disagreement pairs. These cases correspond roughly to Huet's SIMPL phase [36]. The final transformation deals with "flexible-rigid" disagreement pairs. In this case we deduce a useful constraint on the possible unifiers of the chosen disagreement pair. We then show how to use this constraint to instantiate the unification problem into a finite collection of alternate unification problems. This transformation corresponds to Huet's MATCH phase. Each of the transformations is proved valid, and the collection is proved complete. They thus define an algorithm for enumerating μCSPs, as described in Chapter 3.

The value of pre-unification in $\lambda_{\rightarrow}$ is that solved disagreement sets (ones containing only flexible-flexible pairs) are always unifiable, and so pre-unifiability implies unifiability [36]. By making vital use of our definition of *acceptability*, we can generalize Huet's constructive proof of this fact to acceptable solved-form unification problems in λ_{Π}. For the simply typed subset of λ_{Π}, the substitution that we construct specializes to Huet's.

Finally we demonstrate an application of our unification algorithm, to perform automatic "term inference". This problem has two important applications. One is making our unification algorithm more widely applicable. The other is to provide automatic type inference in encoded languages, as described in Chapter 7. As in many type inference algorithms, the basic idea is to combine type-checking and unification, in this case, HOU_{Π}. A similar problem is dealt with by Coquand and Huet [12, 33] and by Pollack [61] under the name of "argument synthesis".

We reported on a slightly different algorithm for HOU_{Π} in [18].

Pym has reported an independent development of an algorithm for HOU_{Π} as well [64].

1.3.4 Products

Chapter 5 extends the pre-unification algorithm developed in the previous chapter to the calculus "$\lambda_{\Pi\Sigma}$", which is λ_{Π} enriched with a dependent version of Cartesian product types, often called "strong sum types", or simply "Σ types". We begin by presenting the extensions to the language, notion of substitutions, conversion rules, and the new weak head normal form. We then develop decomposition methods analogous to those of the previous chapter. Interestingly, the rule of surjectivity for pairs turns out to play a role similar to that of η. With the exception of the flexible-rigid case, the transformations are quite similar to the ones for HOU_{Π}, using the new decomposition methods. The flexible-rigid case uses the same basic ideas as in HOU_{Π}, but there are some interesting differences. It is also simplified by the addition of a new transformation for eliminating *pair-producing* unification variables.

1.3.5 Polymorphism

Chapter 6 extends the pre-unification algorithm for $\lambda_{\Pi\Sigma}$ to a calculus "$\lambda_{\Pi\Sigma\delta}$" with implicit polymorphism, *i.e.*, type variables but no explicit type abstraction, and a very limited form of type application. The resulting algorithm is incomplete, but quite useful.

The transformations for handling weak head redices, abstraction and pairs, and pair-producing variables carry over unchanged from the previous chapter. The only change in the rigid-rigid transformation is the need to handle the type arguments of polymorphic constants. It is worth pointing out here that the analysis is greatly simplified by our use of weak head normal form instead of the long normal form. There is now a flexible-rigid case for *types* as well as terms, but it is simpler and does not cause branching in the search for pre-unifiers. The flexible-rigid case for terms contains the source of incompleteness of our algorithm. Considerable experience with applications programmed in λProlog, which uses an analogous treatment for unification in $\lambda_{\rightarrow}$ extended with implicit polymorphism, has shown it to be very useful in practice. For many unification problems of interest, the algorithm does indeed construct μCSPs.

1.3.6 Applications

Chapter 7 explores applications of our pre-unification algorithms. These applications all have in common that they use a typed λ-calculus as a *meta-language*, *i.e.*, a calculus in which to encode other languages, which we will call *object-languages*. The rich structure of a typed λ-calculus as opposed to traditional, first-order abstract syntax trees allows us to express rules, *e.g.*, program transformation and logical inference rules, that are more succinct, more powerful, and easier to reason about. We can then use unification in the meta-language to mechanize application of these rules.

We begin by giving some examples of the difficulties in using weak representation formalisms such as abstract syntax trees. These difficulties include consideration of variable capturing and shadowing, variable occurrence conditions, contexts, and object-language typing.

We then give an example of a representation of a simple typed expression language, and use it to illustrate our claim that our meta-language term inference algorithm directly yields object-language type inference (in simple enough object-languages). We then extend the language with new constructs, including programs defined by mutually recursive function definitions. These extensions show the value of products and implicit polymorphism.

Finally, we give several examples of transformation rules for our language, which are easily expressed because of the rich meta-language, and can be applied automatically by using $\text{HOU}_{\Pi\Sigma\delta}$.

Chapter 2

The Calculus λ_Π

In this chapter, we present the calculus λ_Π. After presenting the syntax of the terms, types, and kinds of λ_Π, we go on to present our somewhat unconventional definition of substitutions and their composition operation. The conversion rules of λ_Π are the β and η rules at the level of terms and the level of types. The Church-Rosser property for $\beta\eta$ reduction on well-typed terms and types in λ_Π, although generally believed to be true, has not been rigorously proved. For the purposes of this thesis, one can either accept it as a working hypothesis, or modify the definition of convertibility, as discussed in this chapter. Finally, we present the typing rules of the calculus and some of their important properties.

2.1 The Language

The calculus λ_Π is the one used by the Logical Framework (LF) [30], which itself is derived from members of the AUTOMATH family of languages [15]. The language of terms (called "objects" in [30]) in this calculus has the same structure as the simply typed λ-calculus ($\lambda_\rightarrow$) [8]. In place of simple function types "$A \rightarrow B$" we have *dependent function types*[1], "$\Pi v{:}A.\, B$", in which the type B of the result of a function may depend on the value v of the term to which it is applied. Along with types, there are *type families*, which are instantiated by applying them to terms. (They are thus different from, *e.g.*, the type constructors of ML [29], which are instantiated by applying them to types.) Types and type families are classified by their *kind*. Letting the meta-variables M and N range over terms, A and B

[1] These are also sometimes called "dependent product types". We prefer to use that term for the Σ types introduced in Chapter 5, which are sometimes called "strong sum types".

over types (and type families), and K over kinds, the language is as follows:

M	::=	c	constant
	|	v	variable
	|	$\lambda v{:}A.\ M$	abstraction
	|	$M\ N$	application
A	::=	c	constant
	|	$\Pi v{:}A.\ B$	dependent function type
	|	$\lambda v{:}A.\ B$	type family formation
	|	$A\ M$	type family instantiation
K	::=	Type	types of terms
	|	$\Pi v{:}A.\ K$	type family

We will often use the abbreviation "$A{\to}B$" for "$\Pi v{:}A.\ B$" when v is not free in B, and similarly for kinds. As usual, application is left associative, *i.e.*, $M\,M'\,M''$ is the same as $(M\,M')\,M''$. Also, in an abstraction, the scope of the dot reaches to the next closing parenthesis (or the end of the expression). We will sometimes use the meta-variable U to range over terms and types, and occasionally kinds.

For the most part, we will ignore the issues of α-conversion (bound variable renaming) and variable capture. In implementations, we prefer de Bruijn's "nameless" representation [14], in which bound variable occurrences are integers denoting the number of λ's between the occurrence and its binding λ. The mechanics of substitution in this representation are described well in [33, Section 8.3]. For purposes of presentation, however, the conventional named representation is much easier to work with.

2.2 Substitution

Substitution is a fundamental notion in the study of unification. Our approach is somewhat unconventional, in that we associate every substitution with two sets of variables. Pragmatically, the most important difference is the way that composition of substitutions works. Our definition has the advantage of eliminating temporary variables, which arise frequently in higher-order (and equational) unification. It also eliminates the need for idempotence requirements.

First, we need two preliminary notions:

Definition 2.1 *The* set of free variables *of a term, type, or kind U, written "$\mathcal{F}(U)$", is given by the following. First for terms,*

$$\begin{aligned}
\mathcal{F}(\mathsf{c}) &= \{\ \} \\
\mathcal{F}(v) &= \{v\} \\
\mathcal{F}(M\ N) &= \mathcal{F}(M) \cup \mathcal{F}(N) \\
\mathcal{F}(\lambda v{:}A.\ M) &= \mathcal{F}(A) \cup (\mathcal{F}(M) - \{v\})
\end{aligned}$$

then for types,

$$\begin{aligned}
\mathcal{F}(\mathsf{c}) &= \{\ \} \\
\mathcal{F}(\Pi v{:}A.\ B) &= \mathcal{F}(A) \cup (\mathcal{F}(B) - \{v\}) \\
\mathcal{F}(A\ M) &= \mathcal{F}(A) \cup \mathcal{F}(M) \\
\mathcal{F}(\lambda v{:}A.\ B) &= \mathcal{F}(A) \cup (\mathcal{F}(B) - \{v\})
\end{aligned}$$

and finally for kinds

$$\begin{aligned}
\mathcal{F}(\mathsf{Type}) &= \{\ \} \\
\mathcal{F}(\Pi v{:}A.\ K) &= \mathcal{F}(A) \cup (\mathcal{F}(K) - \{v\})
\end{aligned}$$

Definition 2.2 *Given a variable set V, the set "λ_Π^V" contains just those terms, types, and kinds having free variables in V, i.e.,*

$$\{\, U \in \lambda_\Pi \mid \mathcal{F}(U) \subseteq V \,\}$$

Definition 2.3 *Given variables sets V and V', the* set of substitutions from V to V', *written "$\Theta_V^{V'}$", is the set of functions from V to $\lambda_\Pi^{V'}$.*

As a convention, we will use the meta-variable "θ", possibly subscripted and/or primed, to range over substitutions.

Later, we will introduce the more restrictive notion of a "well-typed" substitution.

Definition 2.4 *Given a variable set V, the* set of substitutions over V, *written Θ_V, is the union over all variable sets V' of $\Theta_V^{V'}$.*

Of particular interest are the identity substitutions:

Definition 2.5 *For any variable set V, the* identity substitution over V, *"θ_V^{id}", is defined by $\theta_V^{id} \in \Theta_V^V$ and $\theta_V^{id} v = v$ for all $v \in V$.*

As is customary, we now extend substitutions to functions over λ_Π^V. This extension is more complicated than in first-order languages because of variable binding. We first need one technical device:

Definition 2.6 *Given variable sets V and V', and a variable $u \notin V \cup V'$, let θ^{+u} be the unique member of $\Theta_{V\cup\{u\}}^{V'\cup\{u\}}$ such that*

$$\begin{array}{rcll} \theta^{+u}v & = & \theta v & \text{for } v \in V \\ \theta^{+u}u & = & u & \end{array}$$

Definition 2.7 *Given variable sets V and V', and a substitution $\theta \in \Theta_V^{V'}$, let $\overline{\theta}$ be the function from λ_Π^V to $\lambda_\Pi^{V'}$ satisfying the following properties. First for terms,*

$$\begin{array}{rcll} \overline{\theta}\mathsf{c} & = & \mathsf{c} & \\ \overline{\theta}v & = & \theta v & \text{if } v \in V \\ \overline{\theta}(M\,N) & = & (\overline{\theta}M)\,(\overline{\theta}N) & \\ \overline{\theta}(\lambda u{:}A.\,M) & = & \lambda u{:}\overline{\theta}A.\,\overline{\theta^{+u}}M & \text{if } u \notin V \cup V' \end{array}$$

The condition in the last case is for simplicity. We can always satisfy it by α-conversion. Then for types,

$$\begin{array}{rcll} \overline{\theta}\mathsf{c} & = & \mathsf{c} & \\ \overline{\theta}(\Pi u{:}A.\,B) & = & \Pi u{:}\overline{\theta}A.\,\overline{\theta^{+u}}B & \text{if } u \notin V \cup V' \\ \overline{\theta}(A\,M) & = & (\overline{\theta}A)\,(\overline{\theta}M) & \\ \overline{\theta}(\lambda u{:}A.\,B) & = & \lambda u{:}\overline{\theta}A.\,\overline{\theta^{+u}}B & \text{if } u \notin V \cup V' \end{array}$$

Finally, for kinds,

$$\begin{array}{rcll} \overline{\theta}\mathsf{Type} & = & \mathsf{Type} & \\ \overline{\theta}(\Pi u{:}A.\,K) & = & \Pi u{:}\overline{\theta}A.\,\overline{\theta^{+u}}K & \text{if } u \notin V \cup V' \end{array}$$

2.2.1 Composition

Composition of substitutions is an important operation, and is closely related to composition of functions.[2]

Definition 2.8 *Given variable sets V and V', and substitutions $\theta \in \Theta_V^{V'}$ and $\theta' \in \Theta_{V'}^{V''}$, define*

$$\theta' * \theta \quad = \quad \overline{\theta'} \circ \theta$$

Note then that $\theta' * \theta \in \Theta_V^{V''}$.

[2] We use funtional order composition, *i.e.*, $(f \circ g)\,x = f\,(g\,x)$.

The following theorem relates this notion of composition with standard notion of functional composition:

Theorem 2.9 *Let θ and θ' be as above. Then $\overline{\theta' * \theta} = \overline{\theta'} \circ \overline{\theta}$.*

To prove the theorem, we will first need some lemmas:

Lemma 2.10 *Let $\theta \in \Theta_V^{V'}$, and $u, u' \notin V \cup V'$, with $u \neq u'$. Then $(\theta^{+u})^{+u'} = (\theta^{+u'})^{+u}$.*

Proof: We will show that these substitutions yield the same result when applied to members of $V \cup \{ u, u' \}$.

- Case $v \in V$: $(\theta^{+u})^{+u'} v = \theta^{+u} v = \theta v$, by Definition 2.6. Similarly, $(\theta^{+u'})^{+u} v = \theta v$.
- Case u: $(\theta^{+u})^{+u'} u = \theta^{+u} u = u$. Also, $(\theta^{+u'})^{+u} u = u$.
- Case u': Dual to the previous case.

□

Lemma 2.11 *Let $\theta \in \Theta_V^{V'}$, $u \notin V \cup V'$, and $U \in \lambda_\Pi^V$. Then $\overline{\theta^{+u}} U = \overline{\theta} U$.*

Proof: We prove that for all U, and for V, V', such that $U \in \lambda_\Pi^V$, and for all $\theta \in \Theta_V^{V'}$, we have $\overline{\theta^{+u}} U = \overline{\theta} U$. The proof is by structural induction on U. We will only show the cases for terms, since types and kinds are handled analogously.

- Variable v: Let V,V', and θ be such that $v \in \lambda_\Pi^V$ and $\theta \in \Theta_V^{V'}$. Then $v \in V$, so $\overline{\theta^{+u}} v = \theta^{+u} v = \theta v = \overline{\theta} v$.
- Constant c: $\overline{\theta^{+u}} \mathsf{c} = \mathsf{c} = \overline{\theta} \mathsf{c}$.
- Application $(M\ N)$:

$$\begin{array}{rcll} \overline{\theta^{+u}}(M\ N) & = & (\overline{\theta^{+u}} M)\,(\overline{\theta^{+u}} N) & \\ & = & (\overline{\theta} M)\,(\overline{\theta} N) & \text{by the induction hypothesis (see below)} \\ & = & \overline{\theta}(M\ N) & \end{array}$$

  We can apply the induction hypothesis here, since $\mathcal{F}(M) \subseteq \mathcal{F}(M\ N)$ and $\mathcal{F}(N) \subseteq \mathcal{F}(M\ N)$.

- Abstraction $\lambda u'{:}A.\ M$: Let V, V', and θ be such that $(\lambda u'{:}A.\ M) \in \lambda_\Pi^V$ and $\theta \in \Theta_V^{V'}$, where we assume $u' \notin V \cup V' \cup \{u\}$. Then

$$\begin{array}{lcll} \overline{\theta^{+u}}(\lambda u'{:}A.\ M) & = & \lambda u'{:}\overline{\theta^{+u}}A.\ \overline{(\theta^{+u})^{+u'}}M & \\ & = & \lambda u'{:}\overline{\theta^{+u}}A.\ \overline{(\theta^{+u'})^{+u}}M & \text{by the previous lemma} \\ & = & \lambda u'{:}\overline{\theta}A.\ \overline{\theta^{+u'}}M & \text{by the induction hypothesis (see below)} \\ & = & \overline{\theta}(\lambda u'{:}A.\ M) & \end{array}$$

  We may apply the induction hypothesis here, since $M \in \lambda_\Pi^{V \cup \{u'\}}$ and $\theta^{+u'} \in \Theta_{V \cup \{u\}}^{V' \cup \{u'\}}$.

□

Lemma 2.12 *Let $\theta \in \Theta_V^{V'}$ and $\theta' \in \Theta_{V'}^{V''}$. Then $(\theta' * \theta)^{+u} = \theta'^{+u} * \theta^{+u}$. (Equivalently, by the definition of "$*$", we have $(\overline{\theta'} \circ \theta)^{+u} = \overline{\theta'^{+u}} \circ \theta^{+u}$.)*

Proof: We will show that these substitutions yield the same result when applied to members of $V \cup \{u\}$.

- Case $v \in V$: $(\theta' * \theta)^{+u}v = (\theta' * \theta)v = \overline{\theta'}(\theta v)$. Also, $\overline{\theta'^{+u}}(\theta^{+u}v) = \overline{\theta'^{+u}}(\theta v) = \overline{\theta'}(\theta v)$, because of the previous lemma (and since $u \notin \mathcal{F}(\theta v) \subseteq V'$).
- Case u: $(\theta' * \theta)^{+u}u = u$. Also $\overline{\theta'^{+u}}(\overline{\theta^{+u}}u) = \overline{\theta'^{+u}}u = u$.

□

Proof of Theorem 2.9: By definition, $\overline{\theta' * \theta} = \overline{\overline{\theta'} \circ \theta}$, so we will show that $(\overline{\overline{\theta'} \circ \theta})U = \overline{\theta'}(\overline{\theta}U)$, for all terms, types, and kinds U. Again, we will change the order of universal quantifiers in the claim, proving that for all U, and for V, V', V'', such that $U \in \lambda_\Pi^V$, and for all $\theta \in \Theta_V^{V'}$ and $\theta' \in \Theta_{V'}^{V''}$, we have $(\overline{\overline{\theta'} \circ \theta})U = \overline{\theta'}(\overline{\theta}U)$. The proof is by structural induction on U. Again, we will only show the cases for terms.

- Variable v: Let V be such that $v \in \lambda_\Pi^V$, and choose V', V'', $\theta \in \Theta_V^{V'}$ and $\theta' \in \Theta_{V'}^{V''}$. Then $v \in V$, so $(\overline{\overline{\theta'} \circ \theta})v = (\overline{\theta'} \circ \theta)v = \overline{\theta'}(\theta v) = \overline{\theta'}(\overline{\theta}v)$.
- Constant c: $(\overline{\overline{\theta'} \circ \theta})\mathsf{c} = \mathsf{c}$, by Definition 2.7. Also, $\theta'(\theta\mathsf{c}) = \theta'\mathsf{c} = \mathsf{c}$.

- Application $(M\ N)$: Let V be such that $(M\ N) \in \lambda_\Pi^V$, and choose V', V'', $\theta \in \Theta_V^{V'}$ and $\theta \in \Theta_{V'}^{V''}$. Then $M \in \lambda_\Pi^V$ and $N \in \lambda_\Pi^V$, so

$$\begin{array}{rcll}
(\overline{\overline{\theta'} \circ \theta})(M\ N) & = & (\overline{\overline{\theta'} \circ \theta})M)\,(\overline{\overline{\theta'} \circ \theta})N) & \\
& = & (\overline{\theta'}(\overline{\theta}M))\,(\overline{\theta'}(\overline{\theta}N)) & \text{by the induction hypothesis} \\
& = & \overline{\theta'}((\overline{\theta}M)\,(\overline{\theta}N)) & \\
& = & \overline{\theta'}(\overline{\theta}(M\ N)) &
\end{array}$$

- Abstraction $\lambda u{:}A.\,M$: Let V be such that $(\lambda u{:}A.\,M) \in \lambda_\Pi^V$, and choose V', V'', $\theta \in \Theta_V^{V'}$ and $\theta \in \Theta_{V'}^{V''}$. Then

$$\begin{array}{rcll}
(\overline{\overline{\theta'} \circ \theta})(\lambda u{:}A.\ M) & = & \lambda u{:}(\overline{\overline{\theta'} \circ \theta})A).\ (\overline{(\overline{\theta'} \circ \theta)^{+u}})M) & \\
& = & \lambda u{:}(\overline{\overline{\theta'} \circ \theta})A).\ \overline{(\overline{\theta'^{+u}} \circ \theta^{+u})}M & \text{by the Lemma above} \\
& = & \lambda u{:}\overline{\theta'}(\overline{\theta}A).\ \overline{\theta'^{+u}}(\overline{\theta^{+u}}M) & \text{by the induction hypothesis (see below)} \\
& = & \overline{\theta'}(\lambda u{:}\overline{\theta}A.\ \overline{\theta^{+u}}M) & \\
& = & \overline{\theta'}(\overline{\theta}(\lambda u{:}A.\ M)) &
\end{array}$$

  We may use the induction hypothesis since $M \in \lambda_\Pi^V V \cup \{u\}$, $\theta^{+u} \in \Theta_{V\cup\{u\}}^{V'\cup\{u\}}$, and $\theta'^{+u} \in \Theta_{V'\cup\{u\}}^{V''\cup\{u\}}$

□

Now that we have proved this close relationship between the two kinds of composition, we will no longer make explicit the difference between substitutions and their extensions to operate on λ_Π^V. Thus, we will use "θ" in place of "$\overline{\theta}$", "$\Theta_V^{V'}$" in place of "$\{\,\overline{\theta} \mid \theta \in \Theta_V^{V'}\,\}$", and "$\theta' \circ \theta$" in place of "$\theta' * \theta$". Moreover, we will generally omit the explicit addition of a variable to the domain of a substitution, choosing to write "θ", in place of θ^{+u}, where the u is clear from the context.

2.2.2 Notation

Although our notion of substitution is not the standard one, we will want to adopt something like the standard notation for writing down substitutions as collections of variable/term pairs. Since many of the substitutions used in practice are the identity on many of the variables in their domains, we want a notation that will allow these variables to be elided.

Definition 2.13 *Given variable sets V and V', variables $x_1, \ldots, x_m$, and terms $M_1, \ldots, M_m$ such that $\{\,x_1, \ldots, x_m\,\} \subseteq V$, $V - \{\,x_1, \ldots, x_m\,\} \subseteq V'$, and, for $1 \le i \le m$, $\mathcal{F}(M_i) \subseteq V'$, we will use "$[\,M_1/x_1, \ldots, M_m/x_m\,]_V^{V'}$" to mean the unique substitution $\theta \in \Theta_V^{V'}$ such that (a) for $1 \le i \le m$, $\theta x_i = M_i$, and (b) for each $v \in V - \{\,x_1, \ldots, x_m\,\}$, $\theta v = v$.*

2.2.3 Comparison to the Standard Notion of Substitutions

To appreciate the advantage of our notion of substitutions over the standard one, one must understand something of how substitutions and composition are used in higher-order (and equational) unification. Unifiers are constructed incrementally, using free variables as place holders to be filled in later via composition. For instance, a unifier $[\,\mathsf{g}\,\mathsf{a}/x\,]$ might be built in two steps: first we try $\mathsf{g}\,y$ for x where y is a new free variable, and then we try a for y. Expressed as a composition of substitutions this is

$$[\,\mathsf{a}/y\,]\circ[\,\mathsf{g}\,y/x\,]$$

However, the conventional meaning given to this composition is $[\,\mathsf{a}/y\ ,\ \mathsf{g}\,\mathsf{a}/x\,]$. After composition, one has to eliminate or explicitly ignore "temporary variables" like y.

By keeping track of the contexts involved, he temporary variables are eliminated as soon as they become unnecessary. For instance,

$$[\,\mathsf{a}/y\,]^{\{\,\}}_{\{y\}}\circ[\,\mathsf{g}\,y/x\,]^{\{y\}}_{\{x\}} \quad = \quad [\,\mathsf{g}\,\mathsf{a}/x\,]^{\{\,\}}_{\{x\}}$$

To make this approach to substitutions computationally useful, we have to show how in general to compute a representation $\theta'\circ\theta$ from a representation of θ and θ'.

Proposition 2.14 *For substitutions given by*

$$\theta = [\,M_1/x_1,\ldots,M_m/x_m\,]^{V'}_{V}$$

$$\theta' = [\,M'_1/x'_1,\ldots,M'_{m'}/x'_{m'}\,]^{V''}_{V'}$$

Let

$$\theta'' = [\,N_1/y_1,\ldots,N_n/y_n\,]^{V''}_{V}$$

where the set of pairs $\{\,\langle N_i,y_i\rangle \mid 1\le i\le n\,\}$ *is*

$$\{\,\langle\theta' M_i,x_i\rangle \mid 1\le i\le m\,\}\cup\{\,\langle M'_j,x'_j\rangle \mid x'_j\in V-\{\,x_1,\ldots,x_m\,\}\,\}$$

Then

$$\theta'\circ\theta=\theta''$$

Proof: We show $(\theta'\circ\theta)v=\theta''v$ for all $v\in V$. Consider the possible $v\in V$:

- If $v=x_i$ for $1\le i\le m$, then $\theta v=M_i$, so $(\theta'\circ\theta)v=\theta' M_i=\theta''v$.
- If $v=x_i\in V-\{\,x_1,\ldots,x_m\,\}$ then $\theta v=v$, so $(\theta'\circ\theta)v=\theta' v=M_i=\theta''v$

- If $v \in V - \{ x_1, \ldots, x_m, y_1, \ldots, y_n \}$ then $\theta v = v$ and $\theta' v = v$, so $(\theta' \circ \theta)v = v = \theta'' v$.

□

Example 2.15 *Given the substitutions*

$$\theta = [\, \mathsf{g}\, y/x \,]^{\{y,u,v\}}_{\{x,u,v\}}$$

$$\theta' = [\, \mathsf{a}/y \;,\; \mathsf{b}\, w/u \,]^{\{v,w\}}_{\{y,u,v\}}$$

Then

$$\theta' \circ \theta = [\, \mathsf{g}\, \mathsf{a}/x \;,\; \mathsf{b}\, w/u \,]^{\{v,w\}}_{\{x,u,v\}}$$

2.3 Conversion

In this section, we define the convertibility relation as used in the definitions of typing and unification. We start out with the basic reduction relations:

Definition 2.16 *The β and η relations for terms and types are the smallest relations satisfying*

$$\begin{array}{rcll} (\lambda v{:}A.\, M)\, N & \beta & [\, N/v \,]M & \\ \lambda v{:}A.\, M\, v & \eta & M & \text{if } v \notin \mathcal{F}(M) \end{array}$$

$$\begin{array}{rcll} (\lambda v{:}A.\, B)\, N & \beta & [\, N/v \,]B & \\ \lambda v{:}A.\, B\, v & \eta & B & \text{if } v \notin \mathcal{F}(B) \end{array}$$

Convention 2.17 *Given basic reduction relations ρ_1, ρ_2, we will denote their union by "$\rho_1\rho_2$". For example, "$\beta\eta$" is $\beta \cup \eta$.*

These top level relations extend to one-step and multi-step subterm reducing relations, and convertibility relations:

Definition 2.18 *For a relation ρ on terms and types, the relation $\to_\rho$ is given by,*

$$\frac{U\ \rho\ U'}{U \to_\rho U'}$$

and then for kinds,

$$\frac{A \to_\rho A'}{\Pi v{:}A.\ K \to_\rho \Pi v{:}A'.\ K}$$

$$\frac{K \to_\rho K'}{\Pi v{:}A.\ K \to_\rho \Pi v{:}A.\ K'}$$

then for types,

$$\frac{A \to_\rho A'}{A\ M \to_\rho A'\ M}$$

$$\frac{M \to_\rho M'}{A\ M \to_\rho A\ M'}$$

$$\frac{A \to_\rho A'}{\Pi v{:}A.\ B \to_\rho \Pi v{:}A'.\ B}$$

$$\frac{B \to_\rho B'}{\Pi v{:}A.\ B \to_\rho \Pi v{:}A.\ B'}$$

and finally for terms,

$$\frac{A \to_\rho A'}{\lambda v{:}A.\ M \to_\rho \lambda v{:}A'.\ M}$$

$$\frac{M \to_\rho M'}{\lambda v{:}A.\ M \to_\rho \lambda v{:}A.\ M'}$$

$$\frac{M \to_\rho M'}{M\ N \to_\rho M'\ N}$$

$$\frac{N \to_\rho N'}{M\ N \to_\rho M\ N'}$$

Definition 2.19 *The relation $\to_\rho^*$ is the reflexive transitive closure of $\to_\rho$, and $\leftrightarrow_\rho^*$ is the equivalence closure of $\to_\rho$.*

This gives us our notion of convertibility:

Definition 2.20 *The convertibility relation $=_\lambda$ is $\leftrightarrow_{\beta\eta}^*$.*

The following properties will be important:

Definition 2.21 *A binary relation ρ on terms, types, and kinds, is said to be* substitutive *iff given any U, U' such that $U\ \rho\ U'$, it is the case that $(\theta U)\ \rho\ (\theta U')$ for any substitution θ.*

Proposition 2.22 *The reduction relations β and η are substitutive.*

The proof for the β rule will use the following fact:

Lemma 2.23 *We have $\theta([\, N/v \,]M) = [\, \theta N/v \,](\theta M)$.*

Proof: Simple induction on the structure of M. □

Proof of Proposition 2.22: We will treat only β and η at the level of terms, since the arguments for relations on the level of types is analogous. the Consider $\theta \in \Theta_V^{V'}$. For β, we reason as follows: If $\hat{M}\ \beta\ \hat{M}'$, then $\hat{M}$ and $\hat{M}'$ are of the form $(\lambda v{:}A.\ M)\, N$ and $[\, N/v \,]M$ respectively (where $v \notin V \cup V'$). Then

$$\begin{array}{rcll} \theta\hat{M} & = & \theta((\lambda v{:}A.\ M)\, N) & \\ & = & (\lambda v{:}\theta A.\ \theta M)\, (\theta N) & \text{by Definition 2.7} \\ & \beta & [\, (\theta N)/v \,](\theta M) & \\ & = & \theta([\, N/v \,]M) & \text{by the lemma} \\ & = & \theta\hat{M}' & \end{array}$$

Next, if $\hat{M}\ \eta\ \hat{M}'$, then $\hat{M}$ and $\hat{M}'$ are of the form $\lambda v{:}A.\ M\, v$ and M respectively, where $v \notin \mathcal{F}(M)$ (and $v \notin V \cup V'$). Then

$$\begin{array}{rcll} \theta\hat{M} & = & \theta(\lambda v{:}A.\ M\, v) & \\ & = & \lambda v{:}\theta A.\ (\theta M)\, (\theta v) & \\ & = & \lambda v{:}\theta A.\ (\theta M)\, v & \text{since } v \notin V \\ & \eta & \theta M & \text{since } v \notin \mathcal{F}(M) \text{ and } v \notin V' \\ & = & \theta(\hat{M}') & \end{array}$$

□

Proposition 2.24 *If ρ_1 and ρ_2 are substitutive, then $\rho_1\rho_2$ (i.e., $\rho_1 \cup \rho_2$) is substitutive.*

Proof: Let U, U' be such that $U \rho_1\rho_2 U'$ and let $\theta \in \Theta_V^{V'}$ for variables sets V and V'. Then either $U\ \rho_1\ U'$ or $U\ \rho_2\ U'$. Assume the former. By substitutivity of ρ_1, $(\theta U)\ \rho_1\ (\theta U')$, and so $(\theta U)\rho_1\rho_2(\theta U')$. Similarly, if $U\ \rho_2\ U'$. □

Proposition 2.25 *If ρ is substitutive, then $\rightarrow_\rho$, $\rightarrow_\rho^*$, and $\leftrightarrow_\rho^*$ are substitutive.*

Proof: Let U, U' be such that $U \to_\rho U'$ and let $\theta \in \Theta_V^{V'}$ for variable sets V and V'. We will prove that $(\theta U) \to_\rho (\theta U')$ by induction on the derivation $\mathcal{D}$ of $U \to_\rho U'$, following Definition 2.18:

- If $\mathcal{D}$ consists simply of an instance of the first rule in Definition 2.18, then $U \,\rho\, U'$, and the result is immediate from substitutivity of ρ.
- If $\mathcal{D}$ ends in an instance of the second rule, then $U = \Pi v{:}A.\ K$, $U' = \Pi v{:}A'.\ K$, and there is a subderivation of $\mathcal{D}$ ending in $A \to_\rho A'$. By induction, we may assume $\theta A \to_\rho \theta A'$. Then

$$\begin{array}{rcl} \theta U & = & \theta(\Pi v{:}A.\ K) \\ & = & \Pi v{:}\theta A.\ \theta K \\ & \to_\rho & \Pi v{:}\theta A'.\ \theta K \\ & = & \theta(\Pi v{:}A'.\ K) \\ & = & \theta U' \end{array}$$

- The other cases are all quite similar.

Given this, substitutivity of $\to_\rho^*$ and $\leftrightarrow_\rho^*$ follows by a simple induction. □

Corollary 2.26 *The convertibility relation $=_\lambda$ is substitutive.*

2.4 Typing

To define the typing rules, we will need devices to declare the types and kinds of constants and the types of variables.

Definition 2.27 *A* signature *is a sequence of pairs associating constants with types or kinds. We write a signature, typically denoted by Σ, as*

$$\langle\, \mathsf{c}_1{:}A_1, \ldots, \mathsf{c}_n{:}A_n \,\rangle$$

(where here the U_i stand for types and kinds). We write "$\mathsf{dom}(\Sigma)$" for the domain $\{\, \mathsf{c}_1, \ldots, \mathsf{c}_n \,\}$ of Σ, and "$\Sigma \oplus \mathsf{c}{:}U$" to mean the result of extending the signature Σ by adding $\mathsf{c}{:}U$ to the end.

Definition 2.28 *Similarly, a* context *is a sequence of pairs associating variables with types (but not kinds). We write a context, typically denoted by* Γ*, as*

$$[\, v_1{:}\, A_1, \ldots, v_n{:}\, A_n \,]$$

and write "$\mathsf{dom}(\Gamma)$*" for its domain, and "*$\Gamma \oplus v{:}\, A$*" for its extension. Occasionally, we will also use "*$\mathsf{ran}(\Gamma)$*" for the range* $A_1, \ldots, A_n$ *of* Γ*.*

Following [30], we then define five judgments:

Definition 2.29 *The five basic typing judgments are defined below. We read (a) "*$\vdash \Sigma$ *sig" as* Σ *is a* valid signature*, (b) "*$\vdash_\Sigma \Gamma$ *context" as* Γ *is a* valid context *given* Σ*, (c) "*$\Gamma \vdash_\Sigma K$ *kind" as* K *is a* valid kind *given* Σ *and* Γ*, (d) "*$\Gamma \vdash_\Sigma A \in K$*" as* A has kind K *given* Σ *and* Γ*, and (e) "*$\Gamma \vdash_\Sigma M \in A$*" as* M has type A *given* Σ *and* Γ*. The typing rules below are taken from [30], and depend on the notions of substitution and of convertibility.*[3]

Signatures

$$\vdash \langle\ \rangle\ \mathit{sig}$$

$$\frac{\vdash \Sigma\ \mathit{sig} \qquad [\,] \vdash_\Sigma K\ \mathit{kind} \qquad \mathsf{c} \notin \mathsf{dom}(\Sigma)}{\vdash \Sigma \oplus \mathsf{c}{:}\, K\ \mathit{sig}}$$

$$\frac{\vdash \Sigma\ \mathit{sig} \qquad [\,] \vdash_\Sigma A \in \mathsf{Type} \qquad \mathsf{c} \notin \mathsf{dom}(\Sigma)}{\vdash \Sigma \oplus \mathsf{c}{:}\, A\ \mathit{sig}}$$

Thus valid signatures assign types and kinds to distinct constants, and these types and kinds must be valid in the part of the signature preceding their use.

Contexts

$$\frac{\vdash \Sigma\ \mathit{sig}}{\vdash_\Sigma [\ ]\ \mathit{context}}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathit{context} \qquad \Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad v \notin \mathsf{dom}(\Gamma)}{\vdash_\Sigma \Gamma \oplus v{:}\, A\ \mathit{context}}$$

Thus valid contexts assign types (but not kinds) to distinct variables, and these types must be valid in the current signature and the part of the context preceding their use. Also, note that no context is valid in an invalid signature.

[3] This dependence is why we presented substitution and conversion before typing.

Kinds

$$\frac{\vdash_\Sigma \Gamma \ \mathit{context}}{\Gamma \vdash_\Sigma \mathsf{Type}\ \mathit{kind}}$$

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma \oplus v{:}A \vdash_\Sigma K\ \mathit{kind}}{\Gamma \vdash_\Sigma \Pi v{:}A.\ K\ \mathit{kind}}$$

Note that no kind is valid in an invalid context.

Types

$$\frac{\vdash_\Sigma \Gamma \ \mathit{context} \qquad \mathsf{c}{:}K \in \Sigma}{\Gamma \vdash_\Sigma \mathsf{c} \in K}$$

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma \oplus v{:}A \vdash_\Sigma B \in \mathsf{Type}}{\Gamma \vdash_\Sigma \Pi v{:}A.\ B \in \mathsf{Type}}$$

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma \oplus v{:}A \vdash_\Sigma B \in K}{\Gamma \vdash_\Sigma \lambda v{:}A.\ B \in \Pi v{:}A.\ K}$$

$$\frac{\Gamma \vdash_\Sigma A \in \Pi v{:}B.\ K \qquad \Gamma \vdash_\Sigma M \in B}{\Gamma \vdash_\Sigma A\,M \in [\,M/v\,]K}$$

$$\frac{\Gamma \vdash_\Sigma A \in K \qquad \Gamma \vdash_\Sigma K'\ \mathit{kind} \qquad K =_\lambda K'}{\Gamma \vdash_\Sigma A \in K'}$$

Note that there is no ambiguity in the first rule. If Σ contained two different kind assignments for c, then Σ would not be a valid signature, so Γ would not be a valid context with respect to Σ. In the last rule, the condition "$\Gamma \vdash_\Sigma K'$ kind" is necessary because a valid kind can be convertible to an invalid one.

Terms

$$\frac{\vdash_\Sigma \Gamma \ \mathit{context} \qquad \mathsf{c}{:}A \in \Sigma}{\Gamma \vdash_\Sigma \mathsf{c} \in A}$$

$$\frac{\vdash_\Sigma \Gamma \ \mathit{context} \qquad v{:}A \in \Gamma}{\Gamma \vdash_\Sigma v \in A}$$

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma \oplus v{:}A \vdash_\Sigma M \in B}{\Gamma \vdash_\Sigma \lambda v{:}A.\ M \in \Pi x{:}A.\ B}$$

$$\frac{\Gamma \vdash_\Sigma M \in \Pi v{:}A.\ B \qquad \Gamma \vdash_\Sigma N \in A}{\Gamma \vdash_\Sigma M\,N \in [\,N/v\,]B}$$

$$\frac{\Gamma \vdash_\Sigma M \in A \qquad \Gamma \vdash_\Sigma B \in \mathsf{Type} \qquad A =_\lambda B}{\Gamma \vdash_\Sigma M \in B}$$

Again, there is no ambiguity in the rules for typing constants or variables.

Proposition 2.30 *In any context, every term has at most one type, modulo convertibility. More precisely, given a context* Γ*, term* M *and types* A, A'*, if* $\Gamma \vdash_\Sigma M \in A$ *and* $\Gamma \vdash_\Sigma M \in A'$*, then* $A =_\lambda A'$*. Similarly for types and their kinds.*

Proof: The reason is that to any term, only two rules might apply, one of which is the conversion rule. Moreover, the other rule, specific to the given form of term assigns at most one type. For instance, consider an application $M\,N$ in a context Γ. A typing derivation for $M\,N$ must end in an instance of the application typing rule, followed by zero or more instances of the conversion typing rule. Let the following be the instance of the application typing rule:

$$\frac{\Gamma \vdash_\Sigma M \in \Pi v{:}A.\,B \qquad \Gamma \vdash_\Sigma N \in A}{\Gamma \vdash_\Sigma M\,N \in [\,N/v\,]B}$$

By induction on the structure of the terms being typed, we may assume that every type of M in Γ is convertible to $\Pi v{:}A.\,B$, and every type of B in Γ is convertible to A. Therefore, every typing of $M\,N$ must end in this same instance, modulo convertibility, of the application typing rule, followed by some number of type conversions, and hence concludes in a type for $M\,N$ that is convertible with $[\,N/v\,]B$. □

We will have use of some simple properties of this calculus. Many of these properties are proved in [30] for a very similar calculus based on β convertibility.

Proposition 2.31 (Strengthening) *If (a)* $\Gamma_1 \oplus \Gamma_2 \vdash_\Sigma M \in A$*, (b)* $v \notin \mathsf{dom}(\Gamma_1 \oplus \Gamma_2)$*, and (c)* $\Gamma_1 \vdash_\Sigma A \in \mathsf{Type}$*, then* $\Gamma_1 \oplus v{:}A \oplus \Gamma_2 \vdash_\Sigma M \in A$*. Similarly for types and their kinds.*

Proof: Simple induction on the derivation of $\Gamma_1 \oplus \Gamma_2 \vdash_\Sigma M \in A$. □

Proposition 2.32 (Weakening) *If (a)* $\Gamma_1 \oplus v{:}A \oplus \Gamma_2 \vdash_\Sigma M \in B$*, and (b)* $\Gamma_1 \vdash_\Sigma N \in A$*, then*

$$\Gamma_1 \oplus [\,N/v\,]\Gamma_2 \vdash_\Sigma [\,N/v\,]M \in [\,N/v\,]B$$

Proof: The proof is by induction on the derivation $\mathcal{D}$ of $\Gamma_1 \oplus v{:}A \oplus \Gamma_2 \vdash_\Sigma M \in B$, making use of the derivation $\mathcal{D}_N$ of $\Gamma_1 \vdash_\Sigma N \in A$. The idea is to replace every use of the typing $v{:}A$ in $\mathcal{D}$ by $\mathcal{D}_N$. As usual, we only treat the case of terms.

- If $\mathcal{D}$ consists of the rule for typing variables, then either $M = v$, $M \in \mathsf{dom}(\Gamma_1)$, or $M \in \mathsf{dom}(\Gamma_2)$. If $M = v$, then $[\,N/v\,]M = N$, and $B = A$. Also $v \notin \mathsf{dom}(\Gamma_1)$ and $\mathcal{F}(A) \subseteq \mathsf{dom}(\Gamma_1)$, so $[\,N/v\,]A = A$. But we already know that $\Gamma_1 \vdash_\Sigma N \in A$. If $v \in \mathsf{dom}(\Gamma_1)$, then $[\,N/v\,]M = M$ and $[\,N/v\,]B = B$ (since again, $\mathcal{F}(B) \subseteq \mathsf{dom}(\Gamma_1)$). Otherwise, for some variable u and type C, $M = u$ and $u{:}C \in \Gamma_2$. Then, $[\,N/v\,]M = u$, and $u{:}[\,N/v\,]C \in [\,N/v\,]\Gamma_2$, so the result follows.
- If $\mathcal{D}$ ends in a constant typing then the treatment is similar to the above.
- If $\mathcal{D}$ ends in an instance of the application typing rule, then M is an application $\hat{M}\,\hat{N}$ and $B = [\,\hat{N}/v\,]\hat{B}$, where $\mathcal{D}$ contains subderivations of (a) $\Gamma_1 \oplus v{:}\,A \oplus \Gamma_2 \vdash_\Sigma \hat{M} \in \Pi v'{:}\hat{A}.\,\hat{B}$ and $\Gamma_1 \oplus v{:}\,A \oplus \Gamma_2 \vdash_\Sigma \hat{N} \in \hat{A}$. By induction, we may assume that (a) $\Gamma_1 \oplus [\,N/v\,]\Gamma_2 \vdash_\Sigma [\,N/v\,]\hat{M} \in [\,N/v\,](\Pi v'{:}\hat{A}.\,\hat{B})$, and (b) $\Gamma_1 \oplus \Gamma_2 \vdash_\Sigma [\,N/v\,]\hat{N} \in [\,N/v\,]\hat{A}$. However, $[\,N/v\,](\Pi v'{:}\hat{A}.\,\hat{B}) = \Pi v'{:}[\,N/v\,]\hat{A}.\,[\,N/v\,]\hat{B}$, so by the application typing rule, $\Gamma_1 \oplus [\,N/v\,]\Gamma_2 \vdash_\Sigma ([\,N/v\,]\hat{M})\,([\,N/v\,]\hat{N}) \in [\,([\,N/v\,]\hat{N})/v'\,]\hat{B}$. Then, since $([\,N/v\,]\hat{M})\,([\,N/v\,]\hat{N}) = [\,N/v\,]M$, and $[\,([\,N/v\,]\hat{N})/v'\,]\hat{B} = ([\,N/v\,] \circ [\,\hat{N}/v'\,])\hat{B} = [\,N/v\,]B$, the result follows.
- The abstraction typing case is similar.
- Finally, assume that $\mathcal{D}$ ends in an instance of the conversion typing rule. Then $\mathcal{D}$ has a subderivation of $\Gamma_1 \oplus v{:}\,A \oplus \Gamma_2 \vdash_\Sigma M \in B'$, where $\Gamma_1 \oplus v{:}\,A \oplus \Gamma_2 \vdash_\Sigma B' \in \mathsf{Type}$, and $B' =_\lambda B$. By the induction hypothesis then, we may assume that (a) $\Gamma_1 \oplus [\,N/v\,]\Gamma_2 \vdash_\Sigma [\,N/v\,]M \in [\,N/v\,]B'$, and (b) $\Gamma_1 \oplus [\,N/v\,]\Gamma_2 \vdash_\Sigma [\,N/v\,]B' \in \mathsf{Type}$. (The latter holds because $[\,N/v\,]\mathsf{Type} = \mathsf{Type}$.) However, since $B' =_\lambda B$, we have $[\,N/v\,]B' =_\lambda [\,N/v\,]B$ by Corollary 2.26, so the result follows by the conversion typing rule.

□

We will also need a variant of weakening

Proposition 2.33 *If (a)* $\Gamma_1 \oplus v{:}\,A \oplus \Gamma_2 \vdash_\Sigma M \in B$, *and* $v \notin \mathcal{F}(\Gamma_2) \cup \mathcal{F}(M) \cup \mathcal{F}(B)$, *then* $\Gamma_1 \oplus \Gamma_2 \vdash_\Sigma M \in B$.

Proof: Similar to the previous proposition. We cannot simply appeal to that proposition, however, since there might not exist a term N such that $\Gamma_1 \vdash_\Sigma N \in A$. □

Proposition 2.34 *If* $\Gamma \vdash_\Sigma M \in A$ *then* $\Gamma \vdash_\Sigma A \in \mathsf{Type}$, *and if* $\Gamma \vdash_\Sigma A \in K$ *then* $\Gamma \vdash_\Sigma K$ *kind.*

Proof: Simple induction on the derivation of $\Gamma \vdash_\Sigma M \in A$ or $\Gamma \vdash_\Sigma A \in K$, using weakening in cases the derivation ends in an instance of the term or type application rule. □

2.4.1 Typing and Conversion

For $\lambda_\rightarrow$, it is well known that $\rightarrow_{\beta\eta}$ has the strong normalization (SN) and Church-Rosser (CR) properties for well-typed terms, guaranteeing the existence and uniqueness of normal forms[63]. For well-typed terms in λ_Π, SN is fairly simple to show [30, Theorem A.7]. The CR property for λ_Π with η as well as β has, for some time, been generally believed to be true. This conjecture has only recently been verified and the proof is quite complex [68]. The importance of CR, together with the strong normalization property (SN) is that they reduce the question of convertibility (of well-typed terms and types) to equivalence (modulo α-conversion) of normal forms. An alternative to relying on CR for $\rightarrow_{\beta\eta}$ would be to use just $\rightarrow_\beta$, for which CR has been shown (even for ill-typed terms) [30]. The algorithm we develop in Chapter 4 could be modified to perform unification for this calculus by following the reasoning behind Huet's β unification algorithm for $\lambda_\rightarrow$. However, this does not seem worthwhile, because the η rule is necessary for language representation, on which most applications of interest depend.

As another alternative to relying on CR, we could redefine convertibility in terms of a particular deterministic process of comparing terms. This process is defined and discussed in Section 4.2. Then our unification procedure is guaranteed correct with respect to this notion of convertibility. The CR conjecture (together with SN) implies that this notion is equivalent to the conventional notion of convertibility.

Another important property is the following:

Definition 2.35 *A relation ρ on λ_Π is said to* preserve typing *iff for any Γ, M, M', and A, if $\Gamma \vdash_\Sigma M \in A$ and $M \rho M'$, then $\Gamma \vdash_\Sigma M' \in A$, and similarly for types and their kinds.*

Proposition 2.36 *("Subject reduction") The reduction relations β and η preserve typing.*

We will first need a lemma:

Lemma 2.37 *If $\Gamma \vdash_\Sigma \lambda v{:}A.\ M \in C$ and $C =_\lambda \Pi v{:}A'.\ B'$, then (a) $A =_\lambda A'$, and (b) $\Gamma \oplus v{:}A \vdash_\Sigma M \in B'$. Similarly for types and kinds.*

Proof: Consider a derivation $\mathcal{D}$ of $\Gamma \vdash_\Sigma \lambda v{:}A.\, M \in C$, where $C =_\lambda \Pi v{:}A'.\, B'$. The only two possible final rules are the abstraction and type conversion ones.

- If $\mathcal{D}$ ends in an instance of the conversion rule, then it contains a subderivation of $\Gamma \vdash_\Sigma \lambda v{:}A.\, M \in C'$ for some C' such that $\Gamma \vdash_\Sigma C' \in \mathsf{Type}$, and $C' =_\lambda C$, so that $C' =_\lambda \Pi v{:}A'.\, B'$. By derivation induction, we have (a) $A =_\lambda A'$, and (b) $\Gamma \oplus v{:}A \vdash_\Sigma M \in B'$, which is what we are proving.
- If $\mathcal{D}$ ends in an instance of the abstraction rule, then C itself is of the form $\Pi v{:}A.\, B$, so part (a) is immediate. Part (b) is also immediate because $\mathcal{D}$ contains a subderivation of $\Gamma \oplus v{:}A \vdash_\Sigma M \in B$.

□

Now we can prove that β and η preserve typing.

Proof of Proposition 2.36: We will begin with β and will only treat the term cases, since the type cases are analogous. Assume that there is a derivation $\mathcal{D}$ of $\Gamma \vdash_\Sigma (\lambda v{:}A.\, M)\, N \in C$ for some type C. There are two cases to consider:

- Assume $\mathcal{D}$ ends in an instance of the conversion typing rule. Then $\mathcal{D}$ has subderivations of (a) $\Gamma \vdash_\Sigma (\lambda v{:}A.\, M)\, N \in C'$ and (b) $\Gamma \vdash_\Sigma C' \in \mathsf{Type}$, where $C' =_\lambda C$. By derivation induction, we may assume that $\Gamma \vdash_\Sigma [\, N/v \,]M \in C'$. However, we can then use the conversion rule again to conclude $\Gamma \vdash_\Sigma [\, N/v \,]M \in C$.
- Assume $\mathcal{D}$ ends in an instance of the application typing rule. Then for some A' and B, C has the form $[\, N/v \,]B$, and $\mathcal{D}$ contains subderivations of (a) $\Gamma \vdash_\Sigma \lambda v{:}A.\, M \in \Pi v{:}A'.\, B$, and (b) $\Gamma \vdash_\Sigma N \in A'$. By the lemma above, we know that $A =_\lambda A'$, so $\Gamma \vdash_\Sigma N \in A$, and $\Gamma \oplus v{:}A \vdash_\Sigma M \in B$, so that, by weakening (Proposition 2.32), $\Gamma \vdash_\Sigma [\, N/v \,]M \in [\, N/v \,]B$, *i.e.*, $\Gamma \vdash_\Sigma [\, N/v \,]M \in C$.

For η, there are again two cases, this time depending on whether the derivation $\mathcal{D}$ of $\Gamma \vdash_\Sigma \lambda v{:}A.\, M\, v \in C$ ends in an instance of the conversion or abstraction typing rule. The reasoning in the conversion rule case is exactly as above. Otherwise, C has the form $\Pi v{:}A.\, B$ and $\mathcal{D}$ contains a subderivation $\mathcal{D}_{M\, v}$ of $\Gamma \oplus v{:}A \vdash_\Sigma M\, v \in B$. Recall that we want to show $\Gamma \oplus v{:}A \vdash_\Sigma M \in \Pi v{:}A.\, B$, from which we will conclude by Proposition 2.33 that $\Gamma \vdash_\Sigma M \in \Pi v{:}A.\, B$ since $v \notin \mathcal{F}(M)$ and $v \notin \mathcal{F}(\Pi v{:}A.\, B)$. Again there are two cases, depending on whether $\mathcal{D}_{M\, v}$ ends with an instance of the conversion or application typing rule. The conversion case is handled as usual. Otherwise, there are subderivations of $\mathcal{D}_{M\, v}$ of the form (a) $\Gamma \oplus v{:}A \vdash_\Sigma M \in \Pi v'{:}A'.\, B'$ and (b) $\Gamma \oplus v{:}A \vdash_\Sigma v \in A'$, and we have

$B = [\, v/v' \,]B'$. Therefore, $A =_\lambda A'$, and so $\Gamma \oplus v{:}A \vdash_\Sigma M \in \Pi v'{:}A.\ B'$. However, $\Pi v'{:}A.\ B'$ is α-equivalent to $\Pi v{:}A.\ B$, since $B = [\, v/v' \,]B'$, so the result follows by weakening. □

Proposition 2.38 *If the relations ρ_1 and ρ_2 preserve typing, then $\rho_1\rho_2$ (i.e., $\rho_1 \cup \rho_2$) does. If ρ preserves typing, then $\rightarrow_\rho$ and $\rightarrow_\rho^*$ do.*

Proof: The first claim is immediate, since $M \rho_1\rho_2 M'$ iff $M\ \rho_1\ M'$ or $M\ \rho_2\ M'$. The second follows by induction on the derivation of $M \rightarrow_\rho M'$ and the third follows by induction on the number of $\rightarrow_\rho$ steps. □

However, it is not necessarily the case that $\leftrightarrow_\rho^*$ preserves typing.

Example 2.39 *Consider a signature Σ containing type constants i, o, and a term constant $\mathsf{c}{:}\mathsf{i}$. Then $(\lambda x{:}\mathsf{o}.\ x)\,\mathsf{c} \rightarrow_\beta \mathsf{c}$, and so $\mathsf{c} \leftrightarrow_\beta^* (\lambda x{:}\mathsf{o}.\ x)\,\mathsf{c}$, but $(\lambda x{:}\mathsf{o}.\ x)\,\mathsf{c}$ is ill-typed.*

On the other hand, we do have the following:

Proposition 2.40 *Assume that ρ preserves typing, and that $\rightarrow_\rho$ is CR on well-typed terms, and let U and U' be well-typed terms or types such that $U \leftrightarrow_\rho^* U'$. Then U and U' have the same type or kind.*[4]

Proof: By CR, there is some $\hat{U}$ such that $U \rightarrow_\rho^* \hat{U}$ and $U' \rightarrow_\rho^* \hat{U}$. By Proposition 2.38 however, $\hat{U}$ has the same type or kind as both U and U' and therefore U and U' have the same type or kind. □

2.4.2 Well-typed Substitutions

In unification over a typed calculus, we are given not just the set of variables to be instantiated during unification, but also their types, *i.e.*, the types of the terms that must be assigned to these variables.

[4] More precisely, any type or kind that U has, U' also has. This distinction becomes important in calculi like that of Chapter 5, which do not have unique typing.

Definition 2.41 *For contexts* Γ *and* Γ'*, the* set of well-typed substitutions from Γ to Γ'*, written "*$\Theta_\Gamma^{\Gamma'}$*", is the set of* $\theta \in \Theta_{\mathsf{dom}(\Gamma)}^{\mathsf{dom}(\Gamma')}$ *such that for any variable* v *and type* B*,*

$$v : B \in \Gamma \quad \Rightarrow \quad \Gamma' \vdash_\Sigma \theta v \in \theta B$$

(Note that we do not assume Γ to be valid, but that the validity of Γ' is implied by this condition if Γ is nonempty, since, by the typing rules, no term has a type in an invalid context.)

Definition 2.42 *For a context* Γ*, the* set of well-typed substitutions over Γ*, written "*Θ_Γ*", is the union over all contexts* Γ' *of* $\Theta_\Gamma^{\Gamma'}$

Chapter 3

An Approach to Unification

In this chapter, we present our approach to unification. Although several details are calculus-dependent, the properties used are quite weak and apply to the calculi in later chapters as well as other problems, *e.g.*, equational unification. A related approach, by Snyder and Gallier, for $\text{HOU}_{\to}$, (unification in the simply typed λ-calculus, $\lambda_{\to}$), and equational unification, is presented in [70, 69], which was itself inspired by the work of Martelli and Montanari [41]. However, as discussed below, unlike these works, our approach makes the important distinction between two kinds of "nondeterminism" present in the search for unifiers. This distinction is necessary to formulate an algorithm for enumerating *complete* and *minimal* sets of solutions (as defined in this chapter).

3.1 The Specification

Our formulation of higher-order unification is a generalization of the usual formulation, designed for exposition of the algorithm.

Definition 3.1 *For a context* Γ*, a* disagreement pair over Γ *is a triple* $\langle \Psi , U, U' \rangle$*, where* U *and* U' *are both terms or both types,* Ψ *is a "universal context", typing those variables not to be substituted for, such that* $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Psi) = \{\ \}$*, and all of the variables occurring in* U *and* U' *are in* $\mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Psi)$*.*[1] *We will use the meta-variable* P *to range over disagreement pairs.*

[1] There is no conceptual difficulty in allowing *kind* disagreement pairs as well, but it does not appear to be useful.

The *unification context* Γ and the *universal context* Ψ serve quite different purposes here: Γ serves to enumerate and provide types for the variables that may be instantiated during unification, while Ψ enumerates and provides types for the variables that are not subject to instantiation. These latter, universal, variables arise because at a certain point in each of our algorithms, binding constructs are removed, *e.g.*, an abstraction term $\lambda v{:}A.\ M$ is replaced by M. The Ψ's record the types of these variables, which would otherwise be lost.[2] In contrast, Huet's presentation [36] maintains explicit λ abstractions, which accumulate during execution of his algorithm. A potential disadvantage of our approach is that with a traditional representation for the calculus that uses variable *names*, we would have to perform α-conversion, before removing binding constructs. This disadvantage is removed by adopting de Bruijn's index based representation [14].

Definition 3.2 *A* disagreement set over Γ *is a finite multiset of disagreement pairs over* Γ.[3] *We will use the meta-variable* D *to range over disagreement sets.*

Definition 3.3 *A* unification problem *is a triple* $\langle \Gamma, \theta_0, D\rangle$ *consisting of a context* Γ*, a substitution* $\theta_0 \in \Theta^{\mathsf{dom}(\Gamma)}_{\mathsf{dom}(\Gamma_0)}$ *for some context* Γ_0*, and a disagreement set* D *over* Γ*. We will use* Q *to range over unification problems.*

Usually one presents unification as taking simply a pair of terms having some free variables. The extra complexity here is motivated by the transformation-based framework developed in this chapter. In practice, one begins with an initial unification context, Γ, the identity substitution, θ^{id}_{Γ}, over Γ, and a single disagreement pair, $\langle [\,], M, M'\rangle$ whose free variables are all contained in $\mathsf{dom}(\Gamma)$. Then, in a search process described in general terms later in this chapter, progress is made incrementally towards unifiers. In this process, substitutions are performed (as in Huet's MATCH phase), and disagreement pairs are decomposed into sets of disagreement pairs (as in Huet's SIMPL phase). We compose the individual substitutions leading toward a unifier and keep them as the θ_0 component of a unification problem. By comparison, in [36], the individual substitutions are kept in the edges of a "matching tree". We do not wish to emphasize this difference, as it seems to be mostly one of convenience of presentation.

An important point here, which we will discuss in detail later, is that we do not assume that θ_0 is well-typed, and thus we say $\theta_0 \in \Theta^{\mathsf{dom}(\Gamma)}_{\mathsf{dom}(\Gamma_0)}$ rather than $\theta_0 \in \Theta^{\Gamma}_{\Gamma_0}$. Similarly, we do not say in this definition that the disagreement pairs in D relate well-typed terms or types of the same type or kind. There is however a somewhat weaker condition which we will

[2] In fact, the Ψ's are used only in the proofs and are never examined by the algorithm itself. They could thus be removed as an optimization.

[3] We will use conventional set-like notation for these multisets, with the exception of using $\uplus$ instead of $\cup$.

call "acceptability" of a unification problem Q, written "$\mathcal{A}(Q)$". This condition is required and maintained as an invariant by the unification process described in general terms in this chapter and specifically in later chapters. There we will also show how to initially establish the invariant. The meaning of acceptability is not made precise until Definition 4.38, since it involves technical details developed in Chapter 4. As an approximation, the reader may think of acceptability as saying that all of the disagreement pairs relate well-typed terms or types of the same type or kind.

Definition 3.4 *Given a context* Γ*, a substitution* $\theta \in \Theta_\Gamma$*, and a disagreement pair* $P = \langle \Psi\ ,\ U,\ U' \rangle$ *over* Γ*, we say that* θ unifies P *iff* $\theta U =_\lambda \theta U'$*. For a disagreement set* D *over* Γ*, we say that* θ unifies D *iff* θ *unifies every* $P \in D$*.*

Given a unification problem $\langle \Gamma, \theta_0, D \rangle$, we are interested in the result of composing θ_0 (representing what has already been learned about the original unification problem) with well-typed unifiers of D.

Definition 3.5 *The* set of solutions *of a unification problem* $\langle \Gamma, \theta_0, D \rangle$ *is*

$$\mathcal{U}(\langle \Gamma, \theta_0, D \rangle) \quad = \quad \{\, \hat{\theta} \mid \exists \theta .\, \hat{\theta} =_\lambda \theta \circ \theta_0 \wedge \theta \in \Theta_\Gamma \wedge \theta \textit{ unifies } D \,\}$$

In general, to avoid confusion, we will use $\hat{\theta}$ *to refer to potential solutions of unification problems, and* θ *for potential unifying substitutions, where for a given* θ_0*,* $\hat{\theta} =_\lambda \theta \circ \theta_0$*.*

Note that when θ_0 is an identity substitution and D is of the form $\{\langle [\,]\ ,\ M,\ M' \rangle\}$, we have the usual problem of simply unifying two terms (except for the additional well-typedness condition on θ). An important use of $\mathcal{U}$ is

Definition 3.6 *A unification problem* Q has a solution *iff* $\mathcal{U}(Q) \neq \{\ \}$*.*

In the problem of first-order unification, it is well known that when two terms have a unifier, they have a most general unifier (MGU), of which all other unifiers are instances. (In particular, every MGU is an instance of every other MGU, so in that sense they are unique.) There are many efficient algorithms to decide whether a first-order unification problem has a unifier, and if so, to produce an MGU. (See [39] for a survey.)

With higher-order unification (and equational unification), MGUs no longer exist. However, we can still look for a *complete set of unifiers (CSU)*, whose instances forms the set of all unifiers [20]. One would also like to have the property of *minimality* (non-redundancy), saying that the enumerated unifiers have no instances in common. However, as shown in [34], even for $\lambda_\rightarrow$, it is not generally possible to enumerate minimal CSUs.

Huet's idea of *pre-unification* [34] (implicit in [36]) solved this difficulty. For this, we need the notion of a unification problem being in *solved form.* (We have borrowed this term from [70].) The precise meaning of this property varies from one calculus to another and is motivated by considerations in developing the pre-unification algorithms. The precise meaning of this property will be given as Definition 4.46. For now we need only the following:

Assumption 3.7 *The solved form property satisfies*

1. *It is decidable whether a unification problem is in solved form.*
2. *A unification problem whose disagreement set is empty is in solved form.*
3. *Every acceptable unification problem in solved form has a solution.*

For us, a "pre-unifier" is not a substitution, but rather a special kind of unification problem:

Definition 3.8 *A unification problem Q' is a* pre-unifier *of a unification problem Q iff $\mathcal{U}(Q') \subseteq \mathcal{U}(Q)$ and Q' is in solved form.*

An important property of pre-unification is

Proposition 3.9 *An acceptable unification problem has a solution iff it has a pre-unifier.*

Proof: Let $Q = \langle \Gamma, \theta_0, D \rangle$ be a unification problem, and assume that Q has a solution $\hat{\theta} \in \Theta^{\Gamma'}_{\Gamma_0}$. Then, by the substitutivity property of convertibility (Proposition 2.25), and part 2 of Assumption 3.7, $\langle \Gamma', \hat{\theta}, \{\ \} \rangle$ is a pre-unifier of Q. Next, assume that Q has a pre-unifier Q'. Since Q' is in solved form, it has a solution $\hat{\theta}$ by part 3 of Assumption 3.7, but $\mathcal{U}(Q') \subseteq \mathcal{U}(Q)$, so $\hat{\theta}$ is also a solution of Q. □

Given a unification problem, we will want to compute a representative subset of its unifiers, in the following sense:

Definition 3.10 *Let Q be a unification problem and $\mathcal{Q}$ be a set of unification problems. Then $\mathcal{Q}$ is a* minimal complete set of pre-unifiers (μCSP) *of Q iff*

1. *Every $Q' \in \mathcal{Q}$ is acceptable.*
2. *Every $Q' \in \mathcal{Q}$ is a pre-unifier of Q (i.e., it is in solved form and $\mathcal{U}(Q') \subseteq \mathcal{U}(Q)$).*
3. *$\mathcal{Q}$ is complete with respect to Q, i.e., for any solution $\hat{\theta}$ of Q, there is a $Q' \in \mathcal{Q}$ such that $\hat{\theta} \in \mathcal{U}(Q')$.*
4. *$\mathcal{Q}$ is minimal, i.e., for any two distinct members Q', Q'' of $\mathcal{Q}$, $\mathcal{U}(Q') \cap \mathcal{U}(Q'') = \{\ \}$.*

3.2 Transformations on Unification Problems

The unification algorithms in the following chapters are presented as collections of (nondeterministic) transformations on unification problems. This allows us, for the most part, to ignore issues of control structure and focus on the main ideas. The transformations map unification problems to sets of unification problems, preserving sets of unifiers. The goal of the transformations is to eventually construct only pre-unifiers.

Definition 3.11 *A* transformation *is a relation between unification problems and finite sets of unification problems.*

Definition 3.12 *Let Q be an acceptable unification problem and $\mathcal{Q}$ be a set of unification problems. We say that the transition "$Q \mapsto \mathcal{Q}$" is*

- acceptable *iff every $Q' \in \mathcal{Q}$ is acceptable,*
- correct *iff $\mathcal{U}(Q) = \bigcup_{Q' \in \mathcal{Q}} \mathcal{U}(Q')$,*
- minimal *iff for any two distinct $Q', Q'' \in \mathcal{Q}$, $\mathcal{U}(Q') \cap \mathcal{U}(Q'') = \{\ \}$, and*
- valid *iff it is acceptable, correct, and minimal.*

Definition 3.13 *A transformation ρ is* acceptable, correct, minimal, *or* valid, *if for any acceptable unification problem Q, and set of unification problems $\mathcal{Q}$ such that $Q\ \rho\ \mathcal{Q}$, the transition $Q \mapsto \mathcal{Q}$ is respectively acceptable, correct, minimal, or valid.*

A simple but useful fact is

Proposition 3.14 *The union of acceptable, correct, minimal, or valid transformations is respectively an acceptable, correct, minimal, or valid transformation.*

Proof: Trivial from Definition 3.13. □

3.3 Algorithms from Transformations

Given a collection of transformations, we will now describe a search process that operates on a set of unification problems and enumerates a set of pre-unifiers. Informally, the process goes as follows: If there are no unification problems left, stop. Otherwise, choose a unification problem to work on next. If it is in solved form, output it as a pre-unifier. Otherwise, apply one of the transformations, in some way, to replace the unification problem by a finite set of new unification problems. Then continue.

Note that two kinds of choices are made in this process. First, there is the choice of which unification problem to work on next, and second, there is the choice of which transformation to apply and how to apply it. It turns out that the second kind of choice may be made completely arbitrarily, but, in order to have completeness, the first kind must be done in a fair way.[4] As pointed out in [36], this allows for various strategies. Huet formulated this difference by constructing "matching trees", in which the nodes are disagreement sets and the edges are substitutions, and then showed that all matching trees are complete. His pre-unifiers are constructed by composing substitutions along edges that form a path from the original disagreement set to one in solved form. In our formulation, these composed substitutions are part of the unification problem.

Definition 3.15 *For a transformation ρ and a unification problem Q, a* ρ search tree from *Q is a (possibly infinite) tree T of unification problems such that*

- *The root of T is Q.*
- *For every node Q' in T, the set of children of Q' in T is either empty if Q' is in solved form, or is some $\mathcal{Q}$ satisfying $Q' \rho \mathcal{Q}$ if Q' is not in solved form.*

Definition 3.16 *For a transformation ρ, we define the relation ρ^{**} as follows: $Q \rho^{**} \mathcal{Q}$ iff there is some ρ search tree from Q whose set of solved form nodes is $\mathcal{Q}$.*

We will want our combined transformations to be sufficient to find any unifier, in the following sense:

Definition 3.17 *A transformation relation ρ is* complete *iff for any acceptable unification problem Q and any $\mathcal{Q}$ such that $Q \rho^{**} \mathcal{Q}$, and any $\hat{\theta} \in \mathcal{U}(Q)$, there is some $Q' \in \mathcal{Q}$ such that $\hat{\theta} \in \mathcal{U}(Q')$.*

[4] In implementation terms, this means that we can use *e.g.*, breadth-first search or depth-first search with iterative deepening, but not simple depth-first search.

We can then show the following

Proposition 3.18 *Let ρ be any valid complete transformation relation, Q be an acceptable unification problem, and $\mathcal{Q}$ be any set of unification problems such that $Q\ \rho^{**}\ \mathcal{Q}$. Then $\mathcal{Q}$ is a μCSP of Q.*

Proof: The completeness requirement (part 3 of Definition 3.10) is immediate from Definition 3.17.[5] To see why the other three requirements are satisfied, let T be a ρ search tree, the set of whose solved form nodes is $\mathcal{Q}$, and let $Q' \in \mathcal{Q}$, so that Q' lies on some path in T (of finite length) from Q. For each parent and child, Q_p and Q_c, in T, if $\mathcal{A}(Q_p)$, then (a) $\mathcal{A}(Q_c)$ (by the acceptability of ρ), and (b) we have $\mathcal{U}(Q_c) \subseteq \mathcal{U}(Q_p)$ (by correctness of ρ). Thus, by induction on path length, (a) $\mathcal{U}(Q') \subseteq \mathcal{U}(Q)$, and (b) $\mathcal{A}(Q')$. But also, Q' is in solved form by definition of ρ^{**}, so Q' is a pre-unifier. The remaining condition, minimality of $\mathcal{Q}$, follows from minimality of ρ: Let Q', Q'' be distinct members of $\mathcal{Q}$. Then Q' and Q'' have a common ancestor Q_a in T that is a descendant of all of the other common ancestors of Q', Q''. Then there are distinct children Q'_c, Q''_c of Q_a, which are ancestors of Q' and Q'' respectively. By minimality of ρ, $\mathcal{U}(Q'_c) \cap \mathcal{U}(Q''_c) = \{\ \}$. But $\mathcal{U}(Q') \subseteq \mathcal{U}(Q'_c)$ and $\mathcal{U}(Q'') \subseteq \mathcal{U}(Q''_c)$, so $\mathcal{U}(Q') \cap \mathcal{U}(Q'') = \{\ \}$. □

The following will be useful in proving completeness.

Definition 3.19 *Let ρ be a transformation relation, and assume that for any substitution $\hat{\theta}$ there is a well founded ordering "$\succ_{\hat{\theta}}$" such that for any acceptable unification problem Q with $\hat{\theta} \in \mathcal{U}(Q)$, and set of unification problems $\mathcal{Q}'$, with $Q\ \rho\ \mathcal{Q}'$, and any $Q' \in \mathcal{Q}'$, we have $Q \succ_{\hat{\theta}} Q'$. Then we say that ρ is* decreasing.

Proposition 3.20 *Every correct, acceptable, decreasing transformation relation is complete.*

Proof: Let ρ be a correct, acceptable, decreasing transformation relation, and let Q be an acceptable unification problem with $\hat{\theta} \in \mathcal{U}(Q)$, and $\mathcal{Q}$ be such that $Q\ \rho^{**}\ \mathcal{Q}$. We will show that there is a $Q_{\hat{\theta}} \in \mathcal{Q}$ such that $\hat{\theta} \in \mathcal{U}(Q_{\hat{\theta}})$. Using the definition of ρ^{**}, let T be a ρ search tree from Q whose set of solved form nodes is $\mathcal{Q}$. Consider a sequence, finite or infinite, of unification problems Q_i defined as follows. Let $Q_0 = Q$. Then for a given Q_i, if Q_i is not in solved form, let Q_{i+1} be a child of Q_i in T such that $\hat{\theta} \in \mathcal{U}(Q_{i+1})$. (It exists and is acceptable, by induction, given the correctness and acceptability of ρ, and is unique if ρ is minimal.) Then $Q_0 \succ_{\hat{\theta}} Q_1 \succ_{\hat{\theta}} \cdots$, and since $\succ_{\hat{\theta}}$ is a well founded ordering, the sequence is finite, ending with some Q_n. But then $Q_n \in \mathcal{Q}$ and $\hat{\theta} \in \mathcal{U}(Q_n)$. □

[5] If ρ were not assumed to be complete, we could take it to be the trivially valid transformation that relates every unification problem Q to the set $\{Q\}$.

Chapter 4

A Pre-unification Algorithm

In this chapter we develop an algorithm for HOU_Π in the framework established in Chapter 3, *i.e.*, as a collection of transformations on unification problems. The algorithm we end up with is quite similar to Huet's algorithm for $\text{HOU}_\to$, and in fact behaves almost exactly the same on the subset of λ_Π corresponding to $\lambda_\to$. However, there are considerable technical difficulties in the justification of the algorithm that do not arise in $\text{HOU}_\to$.

We begin by presenting a useful normal form, the *β weak head normal form*, and then use it to reduce convertibility of terms or types to convertibility of simpler terms or types. Next we point out some of the complications arising in HOU_Π that are not present in $\text{HOU}_\to$. In particular, we are forced to deal with ill-typed terms. These considerations serve to motivate the development of our algorithm and, in particular, the definition of "acceptability", an invariant maintained by the transformations, constraining the ill-typedness present in unification problems. We then go on to develop a collection of transformations that suffice for pre-unification. The next two sections prove completeness of the set of transformations, and unifiability of solved form unification problems. Finally, we show how our algorithm allows for a simple type checking/"term inference" algorithm. This problem is not only useful in its own right, but also allows for performing unification on terms that may become well-typed only after substitution.

4.1 Weak Head Normal Forms

Convertibility, as defined in Section 2.3 is an undirected notion. However, the CR and SN properties allow us to replace convertibility by reducibility to common normal form. In higher-order unification, we are considering convertibility after substitution, and as we will see, it is sufficient to normalize just enough to reveal the ultimate "top level structure" of the fully normalized term.

The reduction rules β and η will turn out to play quite different roles. Our use of η is more in the spirit of extensionality than reduction.

Definition 4.1 *The (β)* weak head reduction *relation "*wh_β*" is a restriction of* $\to_\beta$*, applying* β *only "at the head":*

$$\frac{U\ \beta\ U'}{U\ \mathrm{wh}_\beta\ U'}$$

where U *and* U' *range over terms and types. Then for types,*

$$\frac{A\ \mathrm{wh}_\beta\ A'}{A\,M\ \mathrm{wh}_\beta\ A'\,M}$$

and then for terms,

$$\frac{M\ \mathrm{wh}_\beta\ M'}{M\,N\ \mathrm{wh}_\beta\ M'\,N}$$

Additionally, we define a weak head redex *to be any term or type* U *such that there is some* U' *for which* $U\ \mathrm{wh}_\beta\ U'$*.*

Example 4.2 *The following weak head reductions hold (using the first and third rules respectively):*

$$\begin{array}{rcl}(\lambda x{:}\mathsf{i}.\,\mathsf{c}\,(\mathsf{c}\,x))\,\mathsf{b} & \mathrm{wh}_\beta & \mathsf{c}\,(\mathsf{c}\,\mathsf{b}) \\ ((\lambda x{:}\mathsf{i}.\,\mathsf{c}\,(\mathsf{c}\,x))\,\mathsf{b})\,\mathsf{b} & \mathrm{wh}_\beta & \mathsf{c}\,(\mathsf{c}\,\mathsf{b})\,\mathsf{b}\end{array}$$

Note that another way of expressing wh_β would be to say

$$\begin{array}{rcl}(\lambda v{:}A.\,\hat{M})\,N\,N_1\cdots N_n & \mathrm{wh}_\beta & ([\,N/v\,]\hat{M})\,N_1\cdots N_n \\ (\lambda v{:}A.\,\hat{B})\,N\,N_1\cdots N_n & \mathrm{wh}_\beta & ([\,N/v\,]\hat{B})\,N_1\cdots N_n\end{array}$$

for $n \geq 0$. The form we have chosen extends better to the calculus of the next chapter.

Proposition 4.3 *If* $U\ \mathrm{wh}_\beta\ U'$ *then* $U \to_\beta U'$*.*

Proof: A simple induction of the derivation of $U\ \mathrm{wh}_\beta\ U'$. □

Corollary 4.4 wh_β *preserves typing.*

Proposition 4.5 wh_β *is substitutive.*

Proof: Again, a simple induction on the derivation of $U \, \mathrm{wh}_\beta \, U'$, this time using the substitutivity of β, and the distributive properties of substitution with respect to application. □

Definition 4.6 *A term or type U is in (β)* weak head normal form (WHNF) *iff there is no U' such that $U \, \mathrm{wh}_\beta \, U'$.*

We now focus on a subset of the WHNF terms and types:

Definition 4.7 *A* body *is a WHNF term or type (not necessarily well-typed) that is not an abstraction.*

Proposition 4.8 *A well-typed term body (i.e., a body at the level of terms) is either*

- *a variable,*
- *a constant, or*
- *$M \, N$ for a well-typed body M,*[1]

and a well-typed type body is either

- *a type constant,*
- *$\Pi v{:}A.\, B$, or*
- *$A \, M$ for a well-typed body A that is not a Π type.*

Proof: Follows easily from Definitions 4.6 and 4.1. The restriction of well-typedness insures that, in $A \, M$, A is not a Π type. □

The following will be convenient:

Definition 4.9 *An* atom *is a term constant or variable.*

Definition 4.10 *An* atomic type *is one of the form $\mathsf{c} \, N_1 \cdots N_n$ for some c, n, and $N_1, \ldots, N_n$.*

Then the previous proposition can be restated to say that (a) a well-typed term body has the form $a \, M_1 \cdots M_m$ for some atom a, and (b) a type body is either a Π type or an atomic type.

[1] Of course, N is also well-typed, but we want to emphasize that this proposition applies recursively to M.

4.2 Some Useful Properties of Convertibility

The development of our algorithm is divided into two main parts. In this section, we present the first part, which consists of revealing some useful properties of convertibility (as opposed to unifiability). The properties will play a key role in the second part of the development, which forms and justifies of the transformations that make up the pre-unification algorithm..

We will construct various methods by which to decompose questions of convertibility of pairs of terms or types to convertibility of "simpler" pairs. It will be convenient even at this stage to use disagreement pairs, even though we are dealing with conversion and not unification. First, the following definitions will be useful.

Definition 4.11 *Given a substitution $\theta \in \Theta_V^{V'}$ and a universal context $\Psi = [\, v_1{:}A_1, \ldots, v_n{:}A_n \,]$, by "$\theta\Psi$" we mean the context $[\, v_1{:}\theta A_1, \ldots, v_n{:}\theta A_n \,]$[2]. For simplicity, we assume that $\{\, v_1, \ldots, v_n \,\}$ is disjoint from $V \cup V'$. Similarly for unification contexts.*

Definition 4.12 *For a context Γ and a disagreement pair $P = \langle \Psi\ ,\ U,\ U' \rangle$ over Γ, the* set of free variables of *P, written "$\mathcal{F}(P)$", is*

$$(\mathcal{F}(U) \cup \mathcal{F}(U')) - \mathsf{dom}(\Psi)$$

(This will be a subset of $\mathsf{dom}(\Gamma)$.) Similarly, for a disagreement set D, "$\mathcal{F}(D)$" is $\bigcup_{P \in D} \mathcal{F}(P)$.

Definition 4.13 *For a disagreement pair $P = \langle \Psi\ ,\ U,\ U' \rangle$ and a substitution θ, by "θP", we mean the disagreement pair $\langle \theta\Psi\ ,\ \theta U,\ \theta U' \rangle$. Note that we may have to α-convert P (i.e. rename some of the variables in Ψ and perform the corresponding renaming in U and U'). For a disagreement set D, by "θD", we mean the multiset $\{\, \theta P \mid P \in D \,\}$.*

Definition 4.14 *Given a unification context Γ, we will say that a disagreement pair $\langle \Psi, U, U' \rangle$ is* well-typed over Γ *iff U and U' have the same type or kind (and are therefore well-typed) in the context $\Gamma \oplus \Psi$. A disagreement set D is* well-typed *iff every disagreement pair $P \in D$ is well-typed.*

Definition 4.15 *Given a disagreement pair P, we will write "$\mathrm{eq}_\lambda(P)$", to mean that P relates convertible terms or types, i.e., $P = \langle \Psi\ ,\ U,\ U' \rangle$, where $U =_\lambda U'$.*

[2] To be more precise, $\theta\Psi = [\, v_1{:}A_1, v_2{:}\theta^{+v_1}A_2, \ldots, v_n{:}\theta^{+v_1\cdots+v_{n-1}}A_n \,]$

In our eventual use of these properties of convertibility, the pairs of terms or types whose convertibility is in question is the result of applying a substitution θ to a disagreement set P being unified, since, by definition, θ unifies P iff $\mathrm{eq}_\lambda(\theta P)$. Although our disagreement sets may contain ill-typed terms, it will turn out that we will only need to consider substitutions θ that instantiate a given disagreement pair to be well-typed. Thus it will suffice in this section to consider only well-typed disagreement pairs (in the sense of Definition 4.14).

To show that our decomposition methods make progress, we will need a notion of *size*:

Definition 4.16 *Given a term or type U in β normal form, we define the* size *of U, written "$\mathsf{size}(U)$", as follows. First for terms,*

$$\begin{aligned} \mathsf{size}(\mathsf{c}) &= 1 \\ \mathsf{size}(v) &= 1 \\ \mathsf{size}(\lambda v{:}\, A.\ M) &= \mathsf{size}(A) + \mathsf{size}(M) + 1 \\ \mathsf{size}(M\ N) &= \mathsf{size}(M) + \mathsf{size}(N) \end{aligned}$$

Then for types,

$$\begin{aligned} \mathsf{size}(\mathsf{c}) &= 1 \\ \mathsf{size}(\Pi v{:}\, A.\ B) &= \mathsf{size}(A) + \mathsf{size}(B) + 1 \\ \mathsf{size}(\lambda v{:}\, A.\ B) &= \mathsf{size}(A) + \mathsf{size}(B) + 1 \\ \mathsf{size}(A\ M) &= \mathsf{size}(A) + \mathsf{size}(M) \end{aligned}$$

Then we extend this notion to all terms and types that are well-typed in some context by saying that for an arbitrary well-typed term or type U, $\mathsf{size}(U) = \mathsf{size}(\hat{U})$, where $\hat{U}$ is the β normal form of U. (Note that CR and SN of β for well-typed terms and types makes this well-defined.)

We also extend this notion to well-typed disagreement pairs by

$$\mathsf{size}(\langle \Psi\ ,\ U,\ U' \rangle) \quad = \quad \mathsf{size}(U) + \mathsf{size}(U')$$

but not to disagreement sets.

We will also need a measure of how far from being head-normalized a term or type is:

Definition 4.17 *Given a term or type U, well-typed in some context, we write "$\mathsf{dn}(U)$" to mean the number of weak head β reductions required to convert U to weak head normal form. More precisely, it is the number n such that there is a sequence $U_0, \ldots, U_n$ for which (a) $U_0 = U$, (b) U_n is in β weak head normal form, and (c) $U_i\ \mathrm{wh}_\beta\ U_{i+1}$ for $0 \leq i < n$. (Note that SN and the determinacy of wh_β ensure that $\mathsf{dn}(U)$ is well-defined.) We extend* dn *to disagreement pairs as with* size.

Our various decomposition methods will be concrete ways of satisfying the following abstract requirement (explanation follows):

Definition 4.18 *Given a disagreement pair P, and a "disagreement sequence" $\hat{D} = \langle\!\langle\ P'_1, \ldots, P'_k\ \rangle\!\rangle$, both over a unification context Γ, P is* decomposable to $\hat{D}$, *written "$P \lhd \hat{D}$", iff*

1. $\mathrm{eq}_\lambda(P)$ *iff for* $1 \leq i \leq k$, *we have* $\mathrm{eq}_\lambda(P'_i)$, *and*
2. *Each P'_i is well-typed (in the sense of Definition 4.14)* relative to *the convertibility of the preceding sequence of P'_j, i.e., for $1 \leq i \leq k$, P'_i is well-typed if for $1 \leq j < i$,* $\mathrm{eq}_\lambda(P'_j)$.
3. *For $1 \leq i \leq k$, either (a) $\mathsf{size}(P'_i) < \mathsf{size}(P)$, or (b) $\mathsf{size}(P'_i) = \mathsf{size}(P)$ and $\mathsf{dn}(P'_i) < \mathsf{dn}(P)$.*
4. *No new free variables are introduced, i.e., $\mathcal{F}(P'_i) \subseteq \mathcal{F}(P)$, for $1 \leq i \leq k$.*

The first part is the fundamental property, stating that the question of convertibility before decomposition is equivalent to the question of convertibility after. The second says that, while we may construct ill-typed disagreement pairs (ill-typed terms or types or even well-typed terms or types of different types or kinds), this only happens if one or more of the preceding disagreement pairs is not convertible. Note that, in particular, if $k \geq 1$, it requires P'_1 to be well-typed. The third part says that the decomposition is making a kind of progress. Given any tree of disagreement pairs, where each node is *decomposable* to a sequence of its children, this part says that the tree can only have finite paths. Finally, the fourth part is a technical condition needed for later proofs. Note that since P and the P'_i are over Γ, the free variables referred to are in $\mathsf{dom}(\Gamma)$.

It is interesting to note that the decomposition methods defined below form a complete recursive algorithm for deciding whether two well-typed terms or types are convertible. After decomposing, we would (recursively) first test earlier P'_i for convertibility and, only if they succeed, then test later ones. Then part 1 guarantees correctness, part 2 guarantees that each disagreement pair that is tested is well-typed, and part 3 guarantees termination. One might think that we could do something similar for unification, and thus avoid dealing with ill-typed terms or types, but in fact, there will be other, conflicting, requirements on the order in which disagreement pairs are processed.

The following sections develop decomposition methods that together apply to all possible forms of well-typed terms and types. The cases come from the fact that for a well-typed disagreement pair $P = \langle \Psi\ ,\ U,\ U' \rangle$, either

- U or U' is a weak head redex,
- U or U' is an abstraction, or
- Both U and U' are (well-typed) bodies.

4.2.1 Weak head redices

Convertibility of terms or types subject to weak head reduction is handled by the following method.

Definition 4.19 *The decomposition method (relation)* $\rightsquigarrow_{\mathrm{wh}}$ *is given by*

$$\frac{U \ \mathrm{wh}_\beta \ V}{\langle \Psi \ , \ U, \ U' \rangle \rightsquigarrow_{\mathrm{wh}} \langle\!\langle \ \langle \Psi \ , \ V, \ U' \rangle \ \rangle\!\rangle}$$

$$\frac{U' \ \mathrm{wh}_\beta \ V'}{\langle \Psi \ , \ U, \ U' \rangle \rightsquigarrow_{\mathrm{wh}} \langle\!\langle \ \langle \Psi \ , \ U, \ V' \rangle \ \rangle\!\rangle}$$

Then we have

Proposition 4.20 *Let P be a well-typed disagreement pair and $\hat{D}$ a disagreement sequence such that $P \rightsquigarrow_{\mathrm{wh}} \hat{D}$. Then $P \lhd \hat{D}$.*

Proof: Letting $\hat{D} = \langle\!\langle \ P' \ \rangle\!\rangle$, note that P and P' are of the form $\langle \Psi \ , \ U, U' \rangle$ and $\langle \Psi \ , \ V, V' \rangle$ respectively where either (a) $U \ \mathrm{wh}_\beta \ V$ and $U' = V'$, or (b) $U = V$ and $U' \ \mathrm{wh}_\beta \ V'$. In either case, by Proposition 4.3, $U =_\lambda V$ and $U' =_\lambda V'$. Therefore, if $U =_\lambda U'$ then by transitivity and reflexivity of $=_\lambda$, $V =_\lambda V'$. By the same argument, if $V =_\lambda V'$ then $U =_\lambda U'$. The second requirement, well-typedness of P', follows from Corollary 4.4. For the third requirement, we have $\mathsf{size}(P') = \mathsf{size}(P)$ and $\mathsf{dn}(P') < \mathsf{dn}(P)$, because P' results from a β weak head reduction of U or U'. Finally, the fourth condition is satisfied because reduction does not introduce new free variables. □

4.2.2 Abstractions

Convertibility involving abstractions is reduced via the following method. (Note here the role of Ψ.)

Definition 4.21 *The decomposition method* $\rightsquigarrow_\Pi$ *is given by*

$$\langle \Psi \ , \ \lambda v{:}A.\ \hat{M}, \ \lambda v'{:}A'.\ \hat{M}' \rangle \rightsquigarrow_\Pi \langle\!\langle \ \langle \Psi \oplus v : A \ , \ \hat{M}, \ [\, v/v' \,]\hat{M}' \rangle \ \rangle\!\rangle$$

$$\langle \Psi \ , \ \lambda v{:}A.\ \hat{B}, \ \lambda v'{:}A'.\ \hat{B}' \rangle \rightsquigarrow_\Pi \langle\!\langle \ \langle \Psi \oplus v : A \ , \ \hat{B}, \ [\, v/v' \,]\hat{B}' \rangle \ \rangle\!\rangle$$

$$\frac{M' \textit{ is a body}}{\langle \Psi\ ,\ \lambda v{:}A.\ \hat{M},\ M' \rangle \rightsquigarrow_\Pi \langle\!\langle\ \langle \Psi \oplus v{:}A\ ,\ \hat{M},\ M'\, v \rangle\ \rangle\!\rangle}$$

$$\frac{M \textit{ is a body}}{\langle \Psi\ ,\ M,\ \lambda v{:}A.\ \hat{M}' \rangle \rightsquigarrow_\Pi \langle\!\langle\ \langle \Psi \oplus v{:}A\ ,\ M\, v,\ \hat{M}' \rangle\ \rangle\!\rangle}$$

$$\frac{B' \textit{ is a body}}{\langle \Psi\ ,\ \lambda v{:}A.\ \hat{B},\ B' \rangle \rightsquigarrow_\Pi \langle\!\langle\ \langle \Psi \oplus v{:}A\ ,\ \hat{B},\ B'\, v \rangle\ \rangle\!\rangle}$$

$$\frac{B \textit{ is a body}}{\langle \Psi\ ,\ B,\ \lambda v{:}A.\ \hat{B}' \rangle \rightsquigarrow_\Pi \langle\!\langle\ \langle \Psi \oplus v{:}A\ ,\ B\, v,\ \hat{B}' \rangle\ \rangle\!\rangle}$$

Proposition 4.22 *Let P be a well-typed disagreement pair and $\hat{D}$ a disagreement sequence such that $P \rightsquigarrow_\Pi \hat{D}$. Then $P \lhd \hat{D}$.*

(Proof below.)

Example 4.23 *Suppose $P = \langle [\ ]\ ,\ \lambda x{:}\mathsf{i}.\ x,\ \lambda y{:}\mathsf{i}.\ y \rangle$. Then*

$$P \rightsquigarrow_\Pi \langle\!\langle\ \langle [\, x{:}\mathsf{i}\,]\ ,\ x,\ x \rangle\ \rangle\!\rangle$$

which can be handled by the next decomposition method. It is important to note that in this rule we do not have to compare the types of the abstracted variables. This is precisely because well-typedness of P guarantees them to be convertible.

Example 4.24 *Suppose $P = \langle [\, f{:}\mathsf{i}{\to}\mathsf{i}\,]\ ,\ \lambda x{:}\mathsf{i}.\ f\, x,\ f \rangle$. Then*

$$P \rightsquigarrow_\Pi \langle\!\langle\ \langle [\, f{:}\mathsf{i}{\to}\mathsf{i}\ ,\ x{:}\mathsf{i}\,]\ ,\ f\, x,\ f\, x \rangle\ \rangle\!\rangle$$

This decomposition method is strongly reminiscent of the rule of extensionality. As we will see in the proof below, the justification for this decomposition relies on the η rule. In fact the unification algorithm (as well as the convertibility algorithm implicitly defined by this collection of decomposition methods) performs no other form of η conversion. In contrast, Huet's algorithm, in the case of η, performs η-expansion to put terms into long normal form (LNF). Our development shows that these expansions are sometimes unnecessary. Another, more fundamental difficulty with LNF, or even long *head* normal form, is that its definition involves considerations of whether certain subterms are of functional type. In HOU_Π, however, as we have remarked, we will be forced to deal with terms that are ill-typed, and so

the concept of long normal form would need careful reconsideration. This was the approach we took in [18], using the notion of "approximate well-typedness".

We can also make a comparison to the use of η in the standard method for testing convertibility of well-typed terms, which is to completely β and η normalize them, and then test the result for α-equivalence. Instead, we perform β reductions and some η-expansions.

We can now explain why we use *weak* head normal forms, as opposed to the more common "head normal form", which requires some beta-reductions even inside an abstraction [3, page 41]. The reason is simply that we know how to decompose a disagreement pair involving an abstraction, regardless of the reductions that apply inside the abstraction.

Proof of Proposition 4.22: We will treat only two of the six cases, since the others are analogous.

Let $P \rightsquigarrow_\Pi \hat{D}$ by the first rule. Then $P = \langle \Psi\,,\, \lambda v{:}A.\hat{M},\, \lambda v'{:}A'.\hat{M}' \rangle$ and $\hat{D} = \langle\!\langle\ P'\ \rangle\!\rangle$, where $P' = \langle \Psi \oplus v{:}A, \hat{M}, [\,v/v'\,]\hat{M}' \rangle$. First assume that $\mathrm{eq}_\lambda(P)$, *i.e.* $\lambda v{:}A.\hat{M} =_\lambda \lambda v'{:}A'.\hat{M}'$. Then by α-conversion, $\lambda v{:}A.\,\hat{M} =_\lambda \lambda v{:}A'.\,[\,v/v'\,]\hat{M}'$. Therefore, $(\lambda v{:}A.\,\hat{M})\,v =_\lambda (\lambda v{:}A'.\,[\,v/v'\,]\hat{M}')\,v$, so, by β-reduction, $\hat{M} =_\lambda [\,v/v'\,]\hat{M}'$, *i.e.*, $\mathrm{eq}_\lambda(P')$. The converse, that $\mathrm{eq}_\lambda(P')$ implies $\mathrm{eq}_\lambda(P)$, follows from the abstraction rule of Definition 2.18, and α-conversion.

For the second requirement, we must show that P' is well-typed. Since P is well-typed, there must be some B such that $\lambda v{:}A.\,\hat{M}$ and $\lambda v'{:}A'.\,\hat{M}'$ both have type B in $\Gamma \oplus \Psi$. But then B must be convertible to $\Pi v{:}A.\,\hat{B}$ for some $\hat{B}$ such that $\Gamma \oplus \Psi \oplus v{:}A \vdash_\Sigma \hat{M} \in \hat{B}$,[3] and B must also be convertible to $\Pi v'{:}A'.\,\hat{B}'$ for some $\hat{B}'$ such that $\Gamma \oplus \Psi \oplus v'{:}A' \vdash_\Sigma \hat{M}' \in \hat{B}'$. By consideration of normal forms, we have $\hat{A} =_\lambda \hat{A}'$ and $\hat{B} =_\lambda [\,v/v'\,]\hat{B}'$. Therefore, $\Gamma \oplus \Psi \oplus v{:}A \vdash_\Sigma [\,v/v'\,]\hat{M}' \in \hat{B}$, *i.e.*, P' is well-typed.

For the third requirement, we have

$$\begin{array}{rcll} \mathsf{size}(P') & = & \mathsf{size}(\hat{M}) + \mathsf{size}([\,v/v'\,]\hat{M}') & \\ & = & \mathsf{size}(\hat{M}) + \mathsf{size}(\hat{M}') & \text{by a simple induction} \\ & < & \mathsf{size}(\lambda v{:}A.\,\hat{M}) + \mathsf{size}(\lambda v'{:}A'.\,\hat{M}') & \\ & = & \mathsf{size}(P) & \end{array}$$

The fourth condition, that there are no new free variables, is immediate.

Next, let $P \rightsquigarrow_\Pi \hat{D}$ by the third rule. Then $P = \langle \Psi\ ,\ \lambda v{:}A.\,\hat{M},\ M' \rangle$ where M' is a body, and $\hat{D} = \langle\!\langle\ P'\ \rangle\!\rangle$, where $P' = \langle \Psi \oplus v{:}A\,,\, \hat{M},\, M'\,v \rangle$. First assume that $\lambda v{:}A.\,\hat{M} =_\lambda M'$. Since $(\lambda v{:}A.\,\hat{M})\,v \to_\beta \hat{M}$, we then have $\hat{M} =_\lambda M'\,v$, *i.e.*, $\mathrm{eq}_\lambda(P')$. Next, assume $\hat{M} =_\lambda M'\,v$. Then $\lambda v{:}A.\,\hat{M} =_\lambda \lambda v{:}A.\,M'\,v$. Since $\lambda v{:}A.\,\hat{M}$ is well-typed in $\Gamma \oplus \Psi$, we know that $v \notin \mathsf{dom}(\Gamma \oplus \Psi)$,

[3] This is because any derivation of $\Gamma \vdash_\Sigma \lambda v{:}A.\,M \in B$ ends in an instance of the abstraction typing rule followed by zero or more instances of the conversion typing rule.

and then because M' is well-typed in $\Gamma \oplus \Psi$, $v \notin \mathcal{F}(M')$. Thus $\lambda v{:}A.\, M'\, v \to_\eta M'$, so $\lambda v{:}A.\, \hat{M} =_\lambda M'$. This shows that the first requirement for $\lhd$ is satisfied.

For the second requirement, that P' is well-typed, the reasoning is similar to the first case.

For the third requirement, we will again show that $\mathsf{size}(P') < \mathsf{size}(P)$.

$$\begin{array}{rcll} \mathsf{size}(P') & = & \mathsf{size}(\hat{M}) + \mathsf{size}(M'\, v) & \\ & = & \mathsf{size}(\hat{M}) + \mathsf{size}(M') + 1 & \text{since } M' \text{ is a body} \\ & < & \mathsf{size}(\lambda v{:}A.\, \hat{M}) + \mathsf{size}(M') & \text{since } \mathsf{size}(\lambda v{:}A.\, \hat{M}) > \mathsf{size}(\hat{M}) + 1 \\ & = & \mathsf{size}(P) & \end{array}$$

The fourth requirement is immediate. □

4.2.3 Bodies

The only remaining case to consider is a disagreement pair relating two bodies. We will use the following facts:

Proposition 4.25 *If U, U' are bodies well-typed in a context Γ (though not necessarily of the same type or kind), then $U =_\lambda U'$ iff one of the following holds. First for terms:*

- *U and U' are the same atom, or*
- *$U = M\, N$ and $U' = M'\, N'$, for bodies M, M' well-typed in Γ, such that $M =_\lambda M'$ and $N =_\lambda N'$.*

and for types

- *U and U' are the same type constant,*
- *$U = A\, M$ and $U' = A'\, M'$, for bodies A, A' well-typed in Γ, such that $A =_\lambda A'$ and $M =_\lambda M'$, or*
- *$U = \Pi v{:}A.\, B$ and $U' = \Pi v{:}A'.\, B'$, such that $A =_\lambda A'$ and $B =_\lambda B'$.*

To prove this proposition, we will need the following fact.[4]

Lemma 4.26 *The $\beta\eta$ normal form of a body is a body. In particular, let U be a body. Then for terms,*

- *$U = a$, for some atom a, iff the $\beta\eta$ normal form of U is a.*
- *$U = M\,N$ iff the $\beta\eta$ normal form of U is $\hat{M}\,\hat{N}$, where $\hat{M}$ and $\hat{N}$ are the $\beta\eta$ normal forms of M and N respectively.*[5]

and for types,

- *If $U = \mathsf{c}$, the $\beta\eta$ normal form of U is c.*
- *If $U = A\,M$, the $\beta\eta$ normal form of U is $\hat{A}\,\hat{M}$, where $\hat{A}$ and $\hat{M}$ are the $\beta\eta$ normal forms of A and M respectively.*
- *If $U = \Pi v{:}A.\ B$, the $\beta\eta$ normal form of U is $\Pi v{:}\hat{A}.\ \hat{B}$, where $\hat{A}$ and $\hat{B}$ are the $\beta\eta$ normal forms of A and B respectively.*

Proof: A simple induction on the structure of U. □

Proof of Proposition 4.25: The "if" part is immediate. The "only if" part follows from considering the possible forms of well-typed bodies, listed in Proposition 4.8, and their $\beta\eta$ normal forms, described in the lemma. For example, if U is $M\,N$, then the $\beta\eta$ normal form of U is $\hat{M}\,\hat{N}$, where $\hat{M}$ and $\hat{N}$ are the $\beta\eta$ normal forms respectively of M and N. Let $\hat{U'}$ be the $\beta\eta$ normal form of U'. Since $U =_\lambda U'$ though, $\hat{U'} = \hat{M}\,\hat{N}$. But then by the lemma, $U' = M'\,N'$ for some M' and N' whose $\beta\eta$ normal forms are $\hat{M}$ and $\hat{N}$ respectively, which says that $M =_\lambda M'$ and $N =_\lambda N'$. □

This proposition gives rise to a decomposition method, almost giving a sufficient condition for $\lhd$. The only problem is the relative well-typedness condition on the new disagreement pairs, as is demonstrated in the following.

Example 4.27 *Let our signature be a small fragment of the one given in [30] for encoding first-order logic:*

$$\langle\ \mathsf{o}{:}\mathsf{Type}\ ,\ \mathsf{i}{:}\mathsf{Type}\ ,\ \forall{:}(\mathsf{i}{\to}\mathsf{o}){\to}\mathsf{o}\ ,\ \neg{:}\mathsf{o}{\to}\mathsf{o}\ ,\ \top{:}\mathsf{o}\ \rangle$$

[4]The usefulness of the lemma is based the CR property. Alternatively, we would take Proposition 4.25 as part of the definition of convertibility.

[5]More precisely, the "if" direction of this statement should be that if the $\beta\eta$ normal form of (the body) U is $\hat{M}\ \hat{N}$, then there are terms M, N with $\beta\eta$ normal forms $\hat{M}$ and $\hat{N}$ respectively such that $U = M\,N$.

and consider the two terms $\forall\,(\lambda x{:}\mathsf{i}.\top)$ *and* $\neg\,\top$*, both well-typed bodies (even of the same type). From the preceding proposition, we know that these terms are convertible iff (a)* $\forall =_\lambda \neg$*, and (b)* $\lambda x{:}\mathsf{i}.\,\top =_\lambda \top$*. However, it is* not *the case that*

$$\langle [\,]\,,\ \forall\,(\lambda x{:}\mathsf{i}.\,\top),\ \neg\,\top\rangle \lhd \langle\!\langle\ \langle [\,]\,,\ \forall,\ \neg\rangle\ ,\ \langle [\,]\,,\ \lambda x{:}\mathsf{i}.\,\top,\ \top\rangle\ \rangle\!\rangle$$

since both of these new disagreement pairs are ill-typed (in that they relate terms of different types.)

To avoid this problem, we instead use the following more complicated decomposition method:

Definition 4.28 *The decomposition method* $\leadsto_{\mathrm{rr}}$ *is given as follows. First for term bodies,*[6]

$$\langle \Psi\ ,\ v,\ v\rangle \leadsto_{\mathrm{rr}} \langle\!\langle\ \ \rangle\!\rangle$$

$$\langle \Psi\ ,\ \mathsf{c},\ \mathsf{c}\rangle \leadsto_{\mathrm{rr}} \langle\!\langle\ \ \rangle\!\rangle$$

$$\frac{\langle \Psi\ ,\ M,\ M'\rangle \leadsto_{\mathrm{rr}} \hat{D}}{\langle \Psi\ ,\ M\,N,\ M'\,N'\rangle \leadsto_{\mathrm{rr}} \hat{D} \oplus \langle \Psi\ ,\ N,\ N'\rangle}$$

and then for type bodies,

$$\langle \Psi\ ,\ \mathsf{c},\ \mathsf{c}\rangle \leadsto_{\mathrm{rr}} \langle\!\langle\ \ \rangle\!\rangle$$

$$\frac{\langle \Psi\ ,\ A,\ A'\rangle \leadsto_{\mathrm{rr}} \hat{D}}{\langle \Psi\ ,\ A\,M,\ A'\,M'\rangle \leadsto_{\mathrm{rr}} \hat{D} \oplus \langle \Psi\ ,\ M,\ M'\rangle}$$

$$\langle \Psi\ ,\ \Pi v{:}A.\,B,\ \Pi v'{:}A'.\,B'\rangle \leadsto_{\mathrm{rr}} \langle\!\langle\ \langle \Psi\ ,\ A,\ A'\rangle, \langle \Psi \oplus v : A\ ,\ B,\ [\,v/v'\,]B'\rangle\ \rangle\!\rangle$$

Definition 4.29 *Given a disagreement pair* $P = \langle \Psi\ ,\ U,\ U'\rangle$*, we say that "*topeq$(P)$*" iff there is a* $\hat{D}$ *such that* $P \leadsto_{\mathrm{rr}} \hat{D}$*. From the definition of* $\leadsto_{\mathrm{rr}}$*, we can see that the choice of* Ψ *is irrelevant. We will thus say that two terms or types* U *and* U' *have the* same top level structure*, written "*$U \approx U'$*", iff* topeq$(\langle \Psi\ ,\ U,\ U'\rangle)$*, where* Ψ *is an arbitrary context.*

Proposition 4.30 *Given bodies* U *and* U'*,* $U \approx U'$ *iff one of the following holds: First for terms,*

- U *and* U' *are the same atoms, or*
- $U = M\,N$ *and* $U' = M'\,N'$ *for bodies* M, M' *and terms* N, N' *such that* $M \approx M'$*;*

[6] We use the notation "$\hat{D} \oplus P$", for a disagreement sequence $\hat{D}$ and a disagreement pair P, to mean the disagreement sequence that results from adding P onto the end of $\hat{D}$.

and for types

- U *and* U' *are the same constant, or*
- $U = A\,M$ *and* $U' = A'\,M'$ *for bodies* A, A' *and terms* M, M' *such that* $A \approx A'$*, or*
- U *and* U' *are both* Π *types.*

Proof: The "if" part is a direct consequence of the definition of $\leadsto_{\mathrm{rr}}$. The "only if" part follows by induction on the derivation of $\langle \Psi\ ,\ U,\ U' \rangle \leadsto_{\mathrm{rr}} \hat{D}$. □

Two simple consequences are as follows:

Proposition 4.31 *"*$\approx$*" is an equivalence relation.*

Proof: Each of reflexivity, transitivity and symmetry, follows by induction on the structure of the bodies involved, given Proposition 4.30. □

Proposition 4.32 *Let* P *be a well-typed disagreement pair. If* $\mathrm{eq}_\lambda(P)$ *then* $\mathrm{topeq}(P)$*. (Hence, if* $\neg\mathrm{topeq}(P)$ *then* $\neg\mathrm{EQ}_\lambda(P)$*.)*

Proof: By induction on U, using Propositions 4.25 and 4.30. □

Example 4.33 *Return to the previous example, where* $P = \langle [\ ]\ ,\ \forall\,(\lambda x{:}\mathsf{i}.\ \top),\ \neg\,\top \rangle$*. Then* $\neg\mathrm{topeq}(P)$*, and indeed* $\forall\,(\lambda x{:}\mathsf{i}.\ \top) \not=_\lambda \neg\,\top$*.*

Proposition 4.34 *Let* P *be a disagreement pair relating well-typed terms or types (though not necessarily of the same type or kind), and let* $\hat{D}$ *be such that* $P \leadsto_{\mathrm{rr}} \hat{D}$*. Then* $P \lhd \hat{D}$*.*

(Proof below.)

Example 4.35 *Let* $P = \langle [\,], \mathsf{q}\,M\,N, \mathsf{q}\,M'\,N' \rangle$ *be a well-typed disagreement pair in a signature including*

$$\langle\, \mathsf{a}{:}\mathsf{Type}\ ,\ \mathsf{b}{:}\mathsf{a}{\to}\mathsf{Type}\ ,\ \mathsf{c}{:}\mathsf{Type}\ ,\ \mathsf{q}{:}\Pi x{:}\mathsf{a}.\,(\mathsf{b}\,x){\to}\mathsf{c}\,\rangle$$

Note that both terms have type c*. Then*

$$P \rightsquigarrow_{\mathrm{rr}} \langle\!\langle\ \langle [\,]\ ,\ M,\ M' \rangle, \langle [\,]\ ,\ N,\ N' \rangle\ \rangle\!\rangle$$

By our assumption that P *is well-typed,* M *and* M' *both have type* a*, while* N *and* N' *have types* $(\mathsf{b}\,M)$ *and* $(\mathsf{b}\,M')$ *respectively. If however,* $M =_\lambda M'$*, then these types are convertible.*

Proof of Proposition 4.34: We will use induction on the derivation $\mathcal{D}$ of $P \rightsquigarrow_{\mathrm{rr}} \hat{D}$.

- If $\mathcal{D}$ consists of an instance of one of the first two rules, then P relates the same variable or constant, and $\hat{D} = \langle\!\langle\ \ \rangle\!\rangle$. Then $\mathrm{eq}_\lambda(P)$ and so the first part of Definition 4.18 is trivially satisfied. The second and third parts are vacuously true. For the fourth part, note that $\mathcal{F}(\hat{D}) = \{\ \}$.

- Next, assume $\mathcal{D}$ ends in an instance of the third rule. Then $P = \langle \Psi\ ,\ M\,N,\ M'\,N' \rangle$, and for some $\hat{D}'$ such that $\hat{D} = \hat{D}' \oplus \langle \Psi\ ,\ N,\ N' \rangle$, there is a subderivation of $\mathcal{D}$ showing that $\langle \Psi\ ,\ M,\ M' \rangle \rightsquigarrow_{\mathrm{rr}} \hat{D}'$. Since $M\,N$ and $M'\,N'$ are well-typed in $\Gamma \oplus \Psi$, M and M' are well-typed. Therefore, by the induction hypothesis, we have $\langle \Psi\ ,\ M,\ M' \rangle \lhd \hat{D}'$. Letting $\hat{D}' = \langle\!\langle\ P'_1, \ldots, P'_{k'}\ \rangle\!\rangle$, we have $\hat{D} = \langle\!\langle\ P'_1, \ldots, P'_{k'+1}\ \rangle\!\rangle$, where $P'_{k'+1} = \langle \Psi\ ,\ N,\ N' \rangle$. For part 1 of Definition 4.18, we reason

$$\begin{array}{lcll}
\mathrm{eq}_\lambda(P) & \Leftrightarrow & M\,N =_\lambda M'\,N' & \\
& \Leftrightarrow & M =_\lambda M' \wedge N =_\lambda N' & \text{by Proposition 4.25} \\
& \Leftrightarrow & \mathrm{eq}_\lambda(P'_1) \wedge \cdots \wedge \mathrm{eq}_\lambda(P'_{k'}) \wedge N =_\lambda N' & \text{by the induction hypothesis} \\
& \Leftrightarrow & \mathrm{eq}_\lambda(P'_1) \wedge \cdots \wedge \mathrm{eq}_\lambda(P'_{k'+1}) & \text{by definition of } P'_{k'+1}
\end{array}$$

  For part 2, given the induction hypothesis, all that remains to show is that if $\mathrm{eq}_\lambda(P'_j)$ for $1 \le j \le k'$, then $P'_{k'+1}$, *i.e.*, $\langle \Psi\ ,\ N,\ N' \rangle$, is well-typed. By the above, this is the same as saying that if $M =_\lambda M'$ then $\langle \Psi\ ,\ N,\ N' \rangle$ is well-typed. Assume $M =_\lambda M'$. Since M and M' are well-typed, and $\to_{\beta\eta}$ is CR for well-typed terms, M and M' have the same type, by Proposition 2.40. Also, since $M\,N$ and $M'\,N'$ are well-typed, M and M' have some type $\Pi v{:}A.\,B$, and N and N' have the same type A. Thus $\langle \Psi\ ,\ N,\ N' \rangle$ is well-typed.

  For the third requirement of $\lhd$, we show $\mathsf{size}(P'_i) < \mathsf{size}(P)$, for $1 \le i \le k'$, reasoning as follows. For $1 \le i \le k'$,

$$\begin{array}{lcll}
\mathsf{size}(P'_j) & < & \mathsf{size}(\langle \Psi\ ,\ M,\ M' \rangle) & \text{by the induction hypothesis} \\
& < & \mathsf{size}(P) &
\end{array}$$

  and, finally, $\mathsf{size}(P'_{k'+1}) = \mathsf{size}(\langle \Psi\ ,\ N,\ N' \rangle) < \mathsf{size}(P)$.

  The fourth requirement follows by induction and since $\mathcal{F}(\langle \Psi\ ,\ M,\ M' \rangle) \subseteq \mathcal{F}(P)$ and $\mathcal{F}(\langle \Psi\ ,\ N,\ N' \rangle) \subseteq \mathcal{F}(P)$.

- If $\mathcal{D}$ consists of an instance of the fourth rule (involving a type constant), the argument is analogous to the first case.

- If $\mathcal{D}$ ends in an instance of the fifth rule (involving a type application), the argument is analogous to the second case.

- Finally, suppose that $\mathcal{D}$ consists of an instance of the last rule (involving Π types). Then $P = \langle \Psi \,,\, \Pi v{:}A.\, B,\, \Pi v'{:}A'.\, B' \rangle$ and $\hat{D} = \langle\!\langle\, \langle \Psi \,,\, A,\, A' \rangle, \langle \Psi \oplus v{:}A \,,\, B,\, [\, v/v' \,] B' \rangle \,\rangle\!\rangle$. The first part of Definition 4.18 follows immediately from Proposition 4.25. For the second, since $\Pi v{:}A.\, B$ and $\Pi v'{:}A'.\, B'$ are well-typed in $\Gamma \oplus \Psi$, A and A' are both of kind Type in $\Gamma \oplus \Psi$ and if $A =_\lambda A'$ then B and B' have kind Type in $\Gamma \oplus \Psi \oplus v{:}A$. The third and fourth parts are trivial.

□

4.3 From Conversion to Unification

The previous section developed a collection of methods for decomposing questions of convertibility of a given pair of terms or types into the question of convertibility of a finite set of pairs of terms or types. In this section, we prepare for the second main part of our development, which is the construction and justification of the transformations that form our pre-unification algorithm.

Huet's algorithm for $\text{HOU}_{\to}$ relies on and maintains an important invariant on the unification problems under consideration, namely that their disagreement sets are well-typed (in the sense of Definition 4.14, being made up of only disagreement pairs relating well-typed terms of the same type). As we will demonstrate, we cannot maintain this invariant for HOU_{Π}, and so we use a more complicated one. Fortunately, the additional complexity is not as much in the final algorithm as in its justification.

One of the ways in which ill-typedness can enter our unification problems is illustrated in the following:

Example 4.36 *Consider again the disagreement pair in Example 4.35. As we shall justify in Section 4.4.4, this disagreement pair is unified by any unifier of* $\{\, \langle [\,] \,,\, M,\, M' \rangle, \langle [\,] \,,\, N,\, N' \rangle \,\}$. *However, unless M and M' are convertible (not just unifiable), the disagreement pair* $\langle [\,], N, N' \rangle$ *is ill-typed in the sense that it relates terms of different types* $(\mathsf{b}\, M)$ *and* $(\mathsf{b}\, M')$.

Recall that the same kind of ill-typedness arose in the discussion of convertibility. There we were able to avoid comparing $(\mathsf{b}\,M)$ and $(\mathsf{b}\,M')$ until insuring that they have the same type (by first finding that $M =_\lambda M'$). This suggests a similar treatment when performing unification: First unify M and M'. If this succeeds with some unifier θ, *i.e.*, $\theta M =_\lambda \theta M'$, then continue by unifying θN and $\theta N'$. These terms have types $\theta(\mathsf{b}\,M) = \mathsf{b}\,(\theta M)$ and $\theta(\mathsf{b}\,M') = \mathsf{b}\,(\theta M')$ respectively, but since $\theta M =_\lambda \theta M'$, these types are convertible. Thus we might think we could avoid ever dealing with ill-typed disagreement pairs. The fatal flaw in this approach is that it relies on doing full *unification* instead of *pre-unification*. Trying the analogous approach with pre-unification does not always eliminate all of the differences between the types $(\mathsf{b}\,M)$ and $(\mathsf{b}\,M')$.

Another source of ill-typedness comes from the fact that, as mentioned following Definition 3.3, our algorithm builds up unifiers incrementally by composing certain substitutions and applying them to appropriate disagreement sets. These substitutions are the natural extensions of the *imitations* and *projections* used in the MATCH phase of Huet's algorithm. In that algorithm, certain potential projections are ruled out immediately, because they would be of the wrong type. The test for allowable projections consists simply of a comparison of type constants. In HOU_Π, however, the situation is much more difficult, because the types we would have to compare contain terms. Since they are therefore subject to instantiation, determining possible type correctness of the substitution requires unifying these types, which is as difficult as unifying terms. As mentioned above, in pre-unification, we cannot simply perform a full unification before continuing.

Our solution to this problem is simply to carry out the possibly ill-typed substitution, and to add a disagreement pair relating the types that would have to be unified to make the substitution well-typed. However, this raises an issue that requires careful treatment: Applying an ill-typed substitution can result in an ill-typed term, which may therefore fail to be strongly normalizing, or even weakly head normalizing. We will explain following Definition 4.38 of *acceptability* why this possibility does not jeopordize completeness.

Another potential problem with allowing ill-typedness in our unification problems is that, if unrestricted, it would destroy the crucial property that solved form unification problems, as we shall define them, have solutions (as expressed in Assumption 3.7).

The way we avoid these potential problems is to carefully restrict the structure of ill-typedness involved in acceptable unification problems. For this we need

Definition 4.37 *Given a unification problem $Q = \langle \Gamma, \theta_0, D\rangle$, an* accounting for *$Q$ is a strict partial order (i.e., a transitive, antisymmetric,* non*reflexive relation) "$\sqsubset$", between D and $D \cup \mathsf{ran}(\Gamma)$[7], such that (a) for any $X \in D \cup \mathsf{ran}(\Gamma)$, and unifier $\theta \in \Theta_\Gamma$ of $\{\, P \in D \mid P \sqsubset X \,\}$, θX is well-typed, and (b) for any Γ_1, v, A, Γ_2 such that $\Gamma = \Gamma_1 \oplus v{:}A \oplus \Gamma_2$, and any $P \in D$,*

[7]To be more precise: "$\sqsubset$" does not relate disagreement pairs with disagreement pairs and types, but rather occurrences of these.

if $P \sqsubset A$, *we have* $\mathcal{F}(P) \subseteq \mathsf{dom}(\Gamma_1)$. *For* $X \in D \cup \mathsf{ran}(\Gamma)$, *define "*$D^{\sqsubset}_X$*" to be the set* $\{ P \in D \mid P \sqsubset X \}$. *We will often refer to* $D^{\sqsubset}_X$ *as the set of pairs that* account for *(the ill-typedness of)* X.

Definition 4.38 *A unification problem* $Q = \langle \Gamma, \theta_0, D \rangle$ *is* acceptable *iff the following conditions hold:*

1. *There is an accounting for* Q.
2. θ_0 *is permanent. See Definition 4.39 below.*
3. Γ *is weakly valid. See Definition 4.43 below.*

It is important to note that our algorithm only maintains the *existence* of accountings, but never actually constructs them.

Now we can informally explain why we do not need to avoid non-normalizing terms. More rigorous justification is contained in the validity and completeness proofs of the transformations in Section 4.4. Suppose a disagreement set contains some disagreement pair P, containing a term M that is not strongly normalizing. Then for any substitution θ, θM is also not strongly normalizing, since by substitutivity we can parallel reduction sequences from M by reduction sequences from θM. By the SN property for well-typed terms, we can then conclude that θM is ill-typed for all θ. However, we know by acceptability that there is a subset $D^{\sqsubset}_P$ of D all of whose unifiers instantiate P, and therefore M, to be well-typed. Therefore $D^{\sqsubset}_P$, and hence D has no unifier. In summary, if an acceptable unification problem contains a non-SN term then it is nonunifiable. Completeness, however, makes no claims about unification problems with no solutions. This is one reason we choose to treat one-step weak head reduction as a transformation, rather than taking normalization for granted as is usually done in $\text{HOU}_{\to}$ (*e.g.*, [36] and [70]). In [18], we described an optimization based on the idea of *approximate well-typedness* that allows us to avoid ever constructing terms that are not strongly normalizing.

Given a pair of terms M and M' to unify, we can satisfy the invariant initially in either of two ways. The first is to simply check that M and M' are well-typed and have the same type. The second method, defined in Section 4.7, is much more flexible, allowing for disagreement pairs that will become well-typed after substitution.

The property of a substitution being *permanent*, mentioned in Definition 4.38 will be used to show minimality of our transformations (in the sense of Definition 3.13) and hence of the sets of pre-unifiers enumerated by the algorithm (in the sense of Definition 3.10). We adopted the word from the notion of a "permanent occurrence" in [48].

Definition 4.39 *For contexts Γ_0 and Γ, a substitution $\theta_0 \in \Theta^{\mathsf{dom}(\Gamma)}_{\mathsf{dom}(\Gamma_0)}$ is* permanent *iff for any substitutions $\theta, \theta' \in \Theta_\Gamma$, we have*

$$\theta \circ \theta_0 =_\lambda \theta' \circ \theta_0 \quad \Rightarrow \quad \theta =_\lambda \theta'$$

One trivial example of a permanent substitution is the identity substitution over any given variable set.

Example 4.40 *Given the signature $\Sigma = \langle\, \mathsf{i}{:}\mathsf{Type}\ ,\ \mathsf{c}{:}\mathsf{i}{\to}\mathsf{i}{\to}\mathsf{i} \,\rangle$, and the contexts $\Gamma_0 = [\, z{:}\mathsf{i} \,]$ and $\Gamma = [\, x{:}\mathsf{i}\ ,\ y{:}\mathsf{i} \,]$, the substitution $\theta_0 = [\, \mathsf{c}\,x\,y/z \,]^\Gamma_{\Gamma_0}$ is permanent. To see this, consider any two substitutions $\theta, \theta' \in \Theta_\Gamma$.*

$$\begin{array}{lll}
 & \theta \circ \theta_0 =_\lambda \theta' \circ \theta_0 & \\
\Leftrightarrow & (\theta \circ \theta_0)z =_\lambda (\theta' \circ \theta_0)z & \text{since } \mathsf{dom}(\theta \circ \theta_0) = \{z\} \\
\Leftrightarrow & \theta(\theta_0 z) =_\lambda \theta(\theta_0 z) & \text{by definition of composition} \\
\Leftrightarrow & \theta(\mathsf{c}\,x\,y) =_\lambda \theta'(\mathsf{c}\,x\,y) & \\
\Leftrightarrow & \mathsf{c}\,(\theta x)\,(\theta y) =_\lambda \mathsf{c}\,(\theta' x)\,(\theta' y) & \text{by Definition 2.7} \\
\Leftrightarrow & \theta x =_\lambda \theta' x \wedge \theta y =_\lambda \theta' y & \text{by Proposition 4.34} \\
\Leftrightarrow & \theta =_\lambda \theta' & \text{since } \mathsf{dom}(\theta) = \mathsf{dom}(\theta') = \{\, x, y \,\}
\end{array}$$

Example 4.41 *Given the signature $\Sigma = \langle\, \mathsf{i}{:}\mathsf{Type}\ ,\ \mathsf{a}{:}\mathsf{i}\ ,\ \mathsf{b}{:}\mathsf{i} \,\rangle$, and the contexts $\Gamma_0 = [\, z{:}\mathsf{i} \,]$ and $\Gamma = [\, f{:}\mathsf{i}{\to}\mathsf{i} \,]$, the substitution $\theta_0 = [\, f\,x/z \,]^\Gamma_{\Gamma_0}$ is not permanent. Let $\theta = [\, (\lambda v{:}\mathsf{i}.\,\mathsf{c})/f\ ,\ \mathsf{a}/x \,]^{[\,]}_\Gamma$ and $\theta' = [\, (\lambda v{:}\mathsf{i}.\,\mathsf{c})/f\ ,\ \mathsf{b}/x \,]^{[\,]}_\Gamma$. Then $\theta \circ \theta_0 = [\, \mathsf{c}/z \,]^{[\,]}_\Gamma = \theta' \circ \theta_0$, but $\theta \neq_\lambda \theta'$.*

Since our θ_0 are built up by composition, we will need

Proposition 4.42 *The composition of permanent substitutions is permanent*

Proof: For contexts $\Gamma_0, \Gamma_1, \Gamma$, let $\theta_0 \in \Theta^{\mathsf{dom}(\Gamma_1)}_{\mathsf{dom}(\Gamma_0)}$ and $\theta_1 \in \Theta^{\mathsf{dom}(\Gamma)}_{\mathsf{dom}(\Gamma_1)}$ be permanent substitutions and $\theta, \theta' \in \Theta_\Gamma$ be such that $\theta \circ (\theta_1 \circ \theta_0) =_\lambda \theta' \circ (\theta_1 \circ \theta_0)$. Then by associativity of composition, $(\theta \circ \theta_1) \circ \theta_0 =_\lambda (\theta' \circ \theta_1) \circ \theta_0$. But then by permanence of θ_0, $\theta \circ \theta_1 =_\lambda \theta' \circ \theta_1$, and then by permanence of θ_1, $\theta =_\lambda \theta'$. □

Although we will not rely on maintaining valid unification contexts (in the sense of Definition 2.29), we will need a weaker property in the proofs of our transformations:

Definition 4.43 *A context $\Gamma = [\, x_1{:}A_1, \ldots, x_m{:}A_m \,]$ is* weakly valid *iff for each i with $1 \leq i \leq n$, we have (a) $\mathcal{F}(A_i) \subseteq \{\, x_1, \ldots, x_{i-1} \,\}$, and (b) A_i is reducible to the form $\Pi y_1{:}B_1.\,\cdots\,\Pi y_k{:}B_k.\,B_0$, where B_0 is an atomic type. (For example, B_0 cannot be an abstraction.)*

4.4 The Transformations

For unification, we will need to distinguish between two forms of bodies.

Definition 4.44 *The* head *of a body, which is an atom or the symbol* Π*, is given by, first for terms,*

- $head(v) = v$,
- $head(\mathsf{c}) = \mathsf{c}$,
- $head(M\,N) = head(M)$,

and then for types

- $head(\mathsf{c}) = \mathsf{c}$,
- $head(A\,M) = head(A)$,
- $head(\Pi v{:}A.\,B) = \Pi$,

Then we have

Definition 4.45 *Given a unification context* Γ *and a body* U*, we say that* U *is* flexible *if* $head(U) \in \mathsf{dom}(\Gamma)$*. Otherwise,* U *is* rigid*.*

To know the goal of the transformations, we need to define the solved form property. It turns out that we can use exactly the same criterion as in Huet's algorithm for $\text{HOU}_{\to}$:

Definition 4.46 *A unification problem is in* solved form *iff its disagreement set contains only flexible-flexible (term) disagreement pairs.*

Clearly, this definition satisfies parts 1 and 2 of Assumption 3.7. We will prove part 3, unifiability of acceptable solved form unification problems, in Section 4.6.

The transformations making up our algorithm come from considering all possible forms of disagreement pairs other than flexible-flexible ones, and is guided by the properties of convertibility developed in Section 4.2. If no transformation applies to a given unification problem, it is in solved form.

4.4.1 Preliminaries

Some of the transformations have the form of replacing a given disagreement pair P by the members of a sequence $\hat{D}$ of disagreement pairs. Clearly such a transformation would be correct (in the sense of Definition 3.13) if the set of unifiers of P is equal to the set of unifiers of $\hat{D}$. However, this will turn out to be too strong a requirement. We must make use of the accounting for acceptable unification problems, as the proof of the following proposition shows.[8]

Proposition 4.47 *Let $Q = \langle \Gamma, \theta_0, D \uplus \{ P \} \rangle$ be acceptable, $\hat{D}$ be such that $\theta P \lhd \theta \hat{D}$, for any $\theta \in \Theta_\Gamma$ for which θP is well-typed, and let Q' be $\langle \Gamma, \theta_0, D \uplus \hat{D} \rangle$.*[9] *Then the transition $Q \mapsto \{Q'\}$ is valid (i.e., correct, acceptable, and minimal).*

Proof: Let $\sqsubset$ be an accounting for Q, and let $D_a = (D \uplus \{ P \})^{\sqsubset}_P$. Consider an arbitrary $\theta \in \Theta_\Gamma$ such that θP is well-typed. To show *correctness* (in the sense of Definition 3.13), consider two cases:

- θ does not unify D_a: Then, since $D_a \subseteq D$ (by nonreflexivity of $\sqsubset$), θ does not unify $D \uplus \{ P \}$. However, for the same reason, such a θ does not unify $D \uplus \hat{D}$.
- θ unifies D_a: Since $\sqsubset$ is an accounting, θP is well-typed. But then $\theta P \lhd \theta \hat{D}$, so by part 1 of Definition 4.18 of $\lhd$, θ unifies P iff θ unifies $\hat{D}$. Then expanding the definition of $\mathcal{U}$, we see

$$\begin{aligned} \mathcal{U}(Q) &= \{ \hat{\theta} \mid \exists \theta . \hat{\theta} =_\lambda \theta \circ \theta_0 \wedge \theta \in \Theta_\Gamma \wedge \theta \text{ unifies } D \uplus \{ P \} \} \\ &= \{ \hat{\theta} \mid \exists \theta . \hat{\theta} =_\lambda \theta \circ \theta_0 \wedge \theta \in \Theta_\Gamma \wedge \theta \text{ unifies } D \uplus \hat{D} \} \\ &= \mathcal{U}(Q') \end{aligned}$$

Thus, in either case, the transition $Q \mapsto \{Q'\}$ is correct.

To see the first condition, we construct an accounting for Q': Let $\hat{D} = \langle\!\langle\ P'_1, \ldots, P'_k\ \rangle\!\rangle$. In the new ordering, the role of P will be shared by the P'_i, and each P'_i will be below later ones. More precisely, define $\sqsubset'$ by

- $\hat{P} \sqsubset' X$ if $\hat{P} \neq P$, $X \neq P$, and $\hat{P} \sqsubset X$;
- $\hat{P} \sqsubset' P'_i$ for $1 \leq i \leq k$, if $\hat{P} \sqsubset P$

[8] For a disagreement sequence $\hat{D} = \langle\!\langle\ P'_1, \ldots, P'_k\ \rangle\!\rangle$, we write "$\theta\hat{D}$" to mean the disagreement sequence $\langle\!\langle\ \theta P'_1, \ldots, \theta P'_k\ \rangle\!\rangle$.

[9] For a disagreement multiset D and a disagreement sequence $\hat{D}$, we write "$D \uplus \hat{D}$" to mean the multiset union of D and the (multiset of) members of $\hat{D}$.

- $P'_i \sqsubset' X$ for $1 \leq i \leq k$, if $P \sqsubset X$
- $P'_i \sqsubset' P'_j$ if $1 \leq i < j \leq k$

It is straightforward, though rather tedious to see that $\sqsubset'$ is a strict partial order.[10] To show that $\sqsubset'$ is an accounting for Q', consider first an arbitrary $X \in D \cup \mathsf{ran}(\Gamma)$. There are two cases:

1. $P \not\sqsubset X$. Then $D_X^{\sqsubset'} = D_X^{\sqsubset}$ and therefore since $\sqsubset$ is an accounting for Q, any unifier of $D_X^{\sqsubset'}$ instantiates X be well-typed.
2. $P \sqsubset X$. Then $D_X^{\sqsubset'} = (D_X^{\sqsubset} - \{P\}) \uplus \hat{D}$. Let θ be a unifier of $D_X^{\sqsubset'}$. Since $P \sqsubset X$, $D_P^{\sqsubset} \subseteq D_X^{\sqsubset} - \{P\}$, so because $\sqsubset$ is an accounting for Q, θP is well-typed. Therefore, by Definition 4.18, of $\lhd$, since θ unifies $\hat{D}$, it unifies P. But we already knew that θ unifies $D_X^{\sqsubset'} \subseteq D_X^{\sqsubset} - \{P\}$, so it follows that θ unifies $D_X^{\sqsubset}$, so θX is well-typed.

Next, consider P_i for $1 \leq i \leq k$. We can see that $D_{P_i}^{\sqsubset'} = D_P^{\sqsubset} \uplus \{ P'_1, \ldots, P'_{i-1} \}$. Let θ be a unifier of $D_{P_i}^{\sqsubset'}$. Then θ unifies $D_P^{\sqsubset}$, and hence θP is well-typed. Then the definition of $\lhd$, since θ also unifies $P'_1, \ldots, P'_{i-1}$, it follows that θP_i is well-typed.

To show the second requirement of $\sqsubset'$ to be an accounting, let $\Gamma = \Gamma_1 \oplus v{:}A \oplus \Gamma_2$, and assume for some $\hat{P} \in D \uplus \hat{D}$ that $\hat{P} \sqsubset' A$. Then either (a) $\hat{P} \in D$, in which case $\hat{P} \sqsubset A$, so $\mathcal{F}(\hat{P}) \subseteq \mathsf{dom}(\Gamma_1)$, because $\sqsubset$ is an accounting for Q, or (b) $\hat{P} = P'_i$ for some i, in which case also $P \sqsubset A$, so $\mathcal{F}(P'_i) \subseteq \mathcal{F}(P) \subseteq \mathsf{dom}(\Gamma_1)$.

The second and third conditions for acceptability (Definition 4.38) are trivially satisfied, since neither the unification context Γ nor the substitution θ_0 changes.

Finally, minimality is vacuously true. □

4.4.2 Redices

The idea here is very simple — if a member of a disagreement pair is a weak head redex, perform a reduction.

Proposition 4.48 *Let Γ be a context, P be a disagreement pair over Γ, and $\hat{D}$ a disagreement set such that $P \rightsquigarrow_{\mathrm{wh}} \hat{D}$. Then for any $\theta \in \Theta_\Gamma$, $\theta P \rightsquigarrow_{\mathrm{wh}} \theta \hat{D}$, and in particular if θP is well-typed then $\theta P \lhd \theta \hat{D}$.*

[10] Given we really mean relating *occurrences* of disagreement pairs.

Proof: From Definition 4.19 of $\leadsto_{\text{wh}}$, we have $P = \langle \Psi\ ,\ U,\ U' \rangle$ and $\hat{D} = \langle\!\langle\ \langle \Psi\ ,\ V,\ V' \rangle\ \rangle\!\rangle$, where either (a) $U\ \text{wh}_\beta\ V$ and $U' = V'$, or (b) $U = V$ and $U'\ \text{wh}_\beta\ V'$. We will consider the first case only, as the second is analogous:

$$\begin{array}{rcll} \theta P & = & \langle \theta\Psi\ ,\ \theta U,\ \theta U' \rangle & \\ & \leadsto_{\text{wh}} & \langle\!\langle\ \langle \theta\Psi\ ,\ \theta V,\ \theta V' \rangle\ \rangle\!\rangle & \text{by Proposition 4.5 and } V = V' \\ & = & \theta\langle\!\langle\ \langle \Psi\ ,\ V,\ V' \rangle\ \rangle\!\rangle & \\ & = & \theta\hat{D} & \end{array}$$

The final conclusion follows from Proposition 4.20. □

Transformation 4.1 *A Let $Q = \langle \Gamma, \theta_0, D \uplus \{\ P\ \} \rangle$ and let $\hat{D}$ be such that $P \leadsto_{\text{wh}} \hat{D}$. Then make the transition*[11]

$$Q \quad \mapsto \quad \{\langle \Gamma, \theta_0, D \uplus \hat{D} \rangle\}$$

Proposition 4.49 *Transformation 4.1 is valid.*

Proof: Immediate from Propositions 4.48 and 4.47. □

4.4.3 Abstractions

The key to handling a unification problem that contains a disagreement pair including at least one top-level abstraction is the following:

Proposition 4.50 *Let Γ be a context, P be a disagreement pair over Γ, and $\hat{D}$ a disagreement set such that $P \leadsto_{\Pi} \hat{D}$. Then for any $\theta \in \Theta_\Gamma$, $\theta P \leadsto_{\Pi} \theta\hat{D}$, and in particular if θP is well-typed then $\theta P \lhd \theta\hat{D}$.*

Proof: From Definition 4.21 of $\leadsto_{\Pi}$, we can see that there are four cases to consider. We will treat just the first and third cases, since the others are similar. Let $\theta \in \Theta_\Gamma^{\Gamma'}$, and as always, assume that $v, v' \notin \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Gamma')$.

[11] The intended interpretation of this is to define Transformation 4.1 to be the transformation relation that relates Q to $\{\langle \Gamma, \theta_0, D \uplus \hat{D} \rangle\}$ for all $Q, \Gamma, \theta_0, D, P$, and $\hat{D}$ such that $Q = \langle \Gamma, \theta_0, D \uplus \{\ P\ \} \rangle$ and $P \leadsto_{\text{wh}} \hat{D}$. Recall from Definition 3.13 that validity refers only to *acceptable* Q.

Let $P \leadsto_\Pi \hat{D}$ by the first rule of Definition 4.21. Then $P = \langle \Psi\ ,\ \lambda v{:}A.\ \hat{M},\ \lambda v'{:}A'.\ \hat{M}' \rangle$ and $\hat{D} = \langle\!\langle\ P'\ \rangle\!\rangle$, where $P' = \langle \Psi \oplus v{:}A\ ,\ \hat{M},\ [\,v/v'\,]\hat{M}' \rangle$. Thus

$$\begin{array}{rcll}
\theta P & = & \theta\langle \Psi\ ,\ \lambda v{:}A.\ \hat{M},\ \lambda v'{:}A'.\ \hat{M}' \rangle & \\
& = & \langle \theta\Psi\ ,\ \lambda v{:}\theta A.\ \theta\hat{M},\ \lambda v'{:}\theta A'.\ \theta\hat{M}' & \\
& \leadsto_\Pi & \langle\!\langle\ \langle \theta\Psi \oplus v{:}\theta A\ ,\ \theta M,\ [\,v/v'\,](\theta\hat{M}') \rangle\ \rangle\!\rangle & \\
& = & \langle\!\langle\ \langle \theta(\Psi \oplus v{:}A)\ ,\ \theta M,\ \theta([\,v/v'\,]\hat{M}') \rangle\ \rangle\!\rangle & \text{since } v, v' \notin \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Gamma') \\
& = & \theta\hat{D} &
\end{array}$$

Next, let $P \leadsto_\Pi \hat{D}$ by the third rule of Definition 4.21. Then $P = \langle \Psi\ ,\ \lambda v{:}A.\ \hat{M},\ M' \rangle$, and $\hat{D} = \langle\!\langle\ \langle \Psi \oplus v{:}A\ ,\ \hat{M},\ M'\,v \rangle\ \rangle\!\rangle$. Thus

$$\begin{array}{rcll}
\theta P & = & \theta\langle \Psi\ ,\ \lambda v{:}A.\ \hat{M},\ M' \rangle & \\
& = & \langle \theta\Psi\ ,\ \lambda v{:}\theta A.\ \theta\hat{M},\ \theta M' \rangle & \\
& \leadsto_\Pi & \langle\!\langle\ \langle (\theta\Psi) \oplus v{:}\theta A\ ,\ \theta\hat{M},\ (\theta M')\,v \rangle\ \rangle\!\rangle & \text{by Definition 4.21} \\
& = & \langle\!\langle\ \langle \theta(\Psi \oplus v{:}A)\ ,\ \theta\hat{M},\ \theta(M'\,v) \rangle\ \rangle\!\rangle & \text{since } v \notin \mathsf{dom}(\Psi) \\
& = & \theta\langle\!\langle\ \langle \Psi \oplus v{:}A\ ,\ \hat{M},\ M'\,v \rangle\ \rangle\!\rangle & \\
& = & \theta\hat{D} &
\end{array}$$

The final conclusion follows from Proposition 4.22. □

Transformation 4.2 *Let $Q = \langle \Gamma, \theta_0, D \uplus \{\, P \,\} \rangle$, and let $\hat{D}$ be such that $P \leadsto_\Pi \hat{D}$. Then make the transition*

$$Q \quad \mapsto \quad \{\langle \Gamma, \theta_0, D \uplus \hat{D} \rangle\}$$

Proposition 4.51 *Transformation 4.2 is valid.*

Proof: Immediate from Propositions 4.47 and 4.50. □

This case does not correspond directly to anything in Huet's algorithm. There, λ's simply accumulate in the normal form.

4.4.4 Rigid-rigid

The treatment of this case will be much like the previous ones. The key is

Proposition 4.52 *Let U be a rigid body with respect to Γ, and let $\theta \in \Theta_\Gamma$. Then θU is a body and $\theta U \approx U$. (See Definition 4.29.)*

Proof: Simple induction on U, using Proposition 4.8 and Definition 2.7. □

Proposition 4.53 *Let Γ be a context and P be a rigid-rigid disagreement pair over Γ. If $\neg\text{topeq}(P)$, then P is nonunifiable. If $P \rightsquigarrow_{\text{rr}} \hat{D}$, then for all $\theta \in \Theta_\Gamma$, $\theta P \rightsquigarrow_{\text{rr}} \theta\hat{D}$, and in particular when θP is well-typed, we have $\theta P \lhd \theta\hat{D}$.*

Proof: Suppose that $\neg\text{topeq}(P)$, where $P = \langle \Psi\ ,\ U,\ U' \rangle$. If P is unifiable then for some $\theta \in \Theta_\Gamma$, $\theta U =_\lambda \theta U'$, and hence by Proposition 4.32, $\theta U \approx \theta U'$. However, from Proposition 4.52, $\theta U \approx U$, and $\theta U' \approx U'$, so by transitivity and reflexivity of $\approx$ (Proposition 4.31), $U \approx U'$, which is a contradiction. Next assume that $P \rightsquigarrow_{\text{rr}} \hat{D}$. We will prove by induction on the derivation $\mathcal{D}$ of $P \rightsquigarrow_{\text{rr}} \hat{D}$ that $\theta P \rightsquigarrow_{\text{rr}} \theta\hat{D}$.

- If $\mathcal{D}$ consists solely of an instance of the first rule in Definition 4.28 (involving variables), then $P = \langle \Psi\ ,\ v,\ v \rangle$ and $\hat{D} = \langle\!\langle\ \ \rangle\!\rangle$. Since P is rigid-rigid, $v \notin \mathsf{dom}(\Gamma)$, so $\theta P = \langle \theta\Psi\ ,\ v,\ v \rangle \rightsquigarrow_{\text{rr}} \langle\!\langle\ \ \rangle\!\rangle = \theta\hat{D}$.
- If $\mathcal{D}$ consists solely of an instance of the second rule (involving constants), the argument is analogous.
- If $\mathcal{D}$ ends in an instance of the third rule (involving applications) then $P = \langle \Psi, M\ N, M'\ N' \rangle$ and $\hat{D} = \hat{D}' \oplus \langle \Psi, N, N' \rangle$, where there is a subderivation of $\mathcal{D}$ ending in $\langle \Psi\ ,\ M,\ M' \rangle \rightsquigarrow_{\text{rr}} \hat{D}'$. By induction, we may assume that $\theta\langle \Psi\ ,\ M,\ M' \rangle = \langle \theta\Psi\ ,\ \theta M,\ \theta M' \rangle \rightsquigarrow_{\text{rr}} \theta\hat{D}'$. But then

$$\begin{array}{rcll} \theta P & = & \langle \theta\Psi\ ,\ \theta(M\ N),\ \theta(M'\ N') \rangle & \\ & = & \langle \theta\Psi\ ,\ (\theta M)\,(\theta N),\ (\theta M')\,(\theta N') \rangle & \\ & \rightsquigarrow_{\text{rr}} & \theta\hat{D}' \oplus \langle \theta\Psi\ ,\ \theta N,\ \theta N' \rangle & \text{by the induction hypothesis} \\ & = & \theta\hat{D} & \end{array}$$

- If $\mathcal{D}$ consists of or ends in an instance of the rules for a type constants or application, the argument is analogous to the first two cases.

- Finally, if $\mathcal{D}$ consists of an instance of the last rule (involving Π types), then $P = \langle \Psi\ ,\ \Pi v{:}A.\ B,\ \Pi v{:}A'.\ B' \rangle$ and $\hat{D} = \langle\!\langle\ \langle \Psi\ ,\ A,\ A' \rangle, \langle \Psi \oplus v{:}A\ ,\ B,\ B' \rangle\ \rangle\!\rangle$. Thus

$$\begin{array}{rcl} \theta P & = & \langle \theta\Psi\ ,\ \theta(\Pi v{:}A.\ B),\ \theta(\Pi v{:}A'.\ B') \rangle \\ & = & \langle \theta\Psi\ ,\ \Pi v{:}\theta A.\ \theta B,\ \Pi v{:}\theta A'.\ \theta B' \rangle \\ & \leadsto_{\mathrm{rr}} & \langle\!\langle\ \langle \theta\Psi\ ,\ \theta A,\ \theta A' \rangle, \langle (\theta\Psi) \oplus v{:}\theta A\ ,\ \theta B,\ \theta B' \rangle\ \rangle\!\rangle \\ & = & \theta \langle\!\langle\ \langle \Psi\ ,\ A,\ A' \rangle, \langle \Psi \oplus v{:}A\ ,\ B,\ B' \rangle\ \rangle\!\rangle \\ & = & \theta \hat{D} \end{array}$$

The final conclusion follows from Proposition 4.34. □

Transformation 4.3 *Let $Q = \langle \Gamma, \theta_0, D \uplus \{\ P\ \} \rangle$ where P is rigid-rigid. If $\neg\mathrm{topeq}(P)$ then make the transition*

$$Q \quad \mapsto \quad \{\ \}$$

Otherwise, let $P \leadsto_{\mathrm{rr}} \hat{D}$, and make the transition

$$Q \quad \mapsto \quad \langle \Gamma, \theta_0, D \uplus \hat{D} \rangle$$

Proposition 4.54 *Transformation 4.3 is valid.*

Proof: Immediate from Propositions 4.47 and 4.53. □

This case corresponds to one step of Huet's SIMPL phase.

4.4.5 Flexible-rigid

In the preceding cases we either showed nonunifiablity, or replaced the chosen disagreement pair by a finite collection of other disagreement pairs. In this case, the strategy is different. Here we deduce a useful constraint on the possible unifiers of the chosen disagreement pair, and hence the whole disagreement set. We then show how to use this constraint to instantiate the unification problem into a finite collection of alternate unification problems. This corresponds to Huet's MATCH phase. Although we refer to this case as "flexible-rigid", it also handles the symmetric rigid-flexible case.

For the analysis of this case, assume that our acceptable unification problem Q is $\langle \Gamma, \theta_0, D \rangle$, where D contains a flexible-rigid disagreement pair $P = \langle \Psi\ ,\ M,\ M' \rangle$ or a rigid-flexible disagreement pair $P = \langle \Psi\ ,\ M',\ M \rangle$.[12] Let the unification variable v be the head of M, and a' be the head of M'. Since M' is rigid, a' is either a constant, or a variable in $\mathsf{dom}(\Psi)$. Let $\Gamma = \Gamma_1 \oplus v{:}A \oplus \Gamma_2$, where by part three of Definition 4.38 for acceptability, A is reducible to the form $\Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ A_0$, for an atomic A_0. To determine the possible structure (modulo $=_\lambda$) of θv, we will need the following fact:

Proposition 4.55 *For some Γ, $\hat{M}$, $x_1, \ldots, x_m$, $A_1, \ldots, A_m$, suppose that*

$$\Gamma \vdash_\Sigma \hat{M} \in \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ A_0$$

for an atomic A_0. Then for some atom b, and terms $N_1, \ldots, N_n$, we have

$$\hat{M} =_\lambda \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ b\, N_1 \cdots N_n$$

where the right hand side is well-typed in Γ.

Proof: By induction on m:

- If $m = 0$, let M' be the wh_β normal form of $\hat{M}$. Then $\Gamma \vdash_\Sigma M' \in A_0$ (by Proposition 4.4). M' cannot be an abstraction, since otherwise A_0 would be convertible to a Π type, which contradicts Proposition 4.25. Thus, since M' is in wh_β normal form, it has the form $b\, N_1 \cdots N_n$.
- Assume $m \geq 1$. Without loss of generality, assume $x_1 \notin \mathsf{dom}(\Gamma)$, and so $x_1 \notin \mathcal{F}(\hat{M})$. By strengthening (Proposition 2.31), $\Gamma \oplus x_1{:}A_1 \vdash_\Sigma \hat{M} \in \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ A_0$. Also, $\Gamma \oplus x_1{:}A_1 \vdash_\Sigma x_1 \in A_1$. Thus, by the application typing rule,

  $$\Gamma \oplus x_1 : A_1 \vdash_\Sigma \hat{M}\, x_1 \in \Pi x_2{:}A_2.\ \cdots \Pi x_m{:}A_m.\ A_0$$

  and then by the abstraction typing rule,

  $$\Gamma \vdash_\Sigma \lambda x_1{:}A_1.\ \hat{M}\, x_1 \in \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ A_0$$

  and $\lambda x_1{:}A_1.\ \hat{M}\, x_1$ is well-typed in Γ. Also, since

  $$\Gamma \oplus x_1 : A_1 \vdash_\Sigma \hat{M}\, x_1 \in \Pi x_2{:}A_2.\ \cdots \Pi x_m{:}A_m.\ A_0$$

  we may assume, by the induction hypothesis, that

  $$\hat{M}\, x_1 =_\lambda \lambda x_2{:}A_2.\ \cdots \lambda x_m{:}A_m.\ b\, N_1 \cdots N_n$$

  where the right hand side is well-typed. Thus,

  $$\hat{M} =_\lambda \lambda x_1{:}A_1.\ \hat{M}\, x_1 =_\lambda \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ b\, N_1 \cdots N_n$$

  which is well-typed.

[12] There are no flexible types, so P must relate terms.

□

Now we return to the problem of determining the possible top level structure of θv. Recall from the definition of $\Theta_\Gamma^{\Gamma'}$ that for $\theta \in \Theta_\Gamma^{\Gamma'}$, we have $\Gamma' \vdash_\Sigma \theta v \in \theta(\Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ A_0)$, so that for some types $A'_1, \ldots, A'_m$, atom b, and terms $N_1, \ldots, N_n$, θv is convertible to a term of the form

$$\lambda x_1{:}A'_1.\ \cdots \lambda x_m{:}A'_m.\ b\, N_1 \cdots N_n$$

for some atom b, by the Proposition we just proved. We are interested in the possibilities for b when θ is a unifier.

Proposition 4.56 *Let θ be as above. If θ unifies P then b is either*

- *a', if a' is a constant (rather than a variable in $\mathsf{dom}(\Psi)$), or*
- *some x_i, for $1 \leq i \leq m$.*

Proof: If b is not one of the x_i, then the head of any β normal form of θM is b. However, the head of $\theta M'$ is a' for any θ, so if θ unifies P, then, by Proposition 4.32, b must be a'. The reason that b cannot be a variable in $\mathsf{dom}(\Psi)$ is that if b is a variable, then $b \in \mathcal{F}(\theta v)$, *i.e.*, $b \in \mathsf{dom}(\Gamma')$, where $\theta \in \Theta_\Gamma^{\Gamma'}$, but we assume that (possibly after α-conversion), $\mathsf{dom}(\Psi) \cap (\mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Gamma')) = \{\ \}$. □

Definition 4.57 *Let H be this set of possible values of b.*

Next we see how possible values of b translate into "approximating substitutions".

Definition 4.58 *Let Γ, v, A, Γ_1, etc. be as above. Let $b \in H$ and let B be the type of b according to Σ or $[\, x_1{:}A_1, \ldots, x_m{:}A_m\,]$, where $B = \Pi y_1{:}B_1.\ \cdots \Pi y_n{:}B_n.\ B_0$ for an atomic B_0. Note that either $b{:}B \in \Sigma$, or for some k, $b = x_k$, and so $B = A_k$. Let*

$$N_b \quad = \quad \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ b\,(v_1\, x_1 \cdots x_m) \cdots (v_n\, x_1 \cdots x_m)$$

where the v_j are new variables, i.e., $\{\, v_1, \ldots, v_n\,\} \cap (\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \cup \mathsf{dom}(\Psi)) = \{\ \}$. Then define the approximating substitution

$$\theta_b \quad = \quad [\, N_b/v\,]_{\mathsf{dom}(\Gamma)}^{\mathsf{dom}(\Gamma_1) \cup \{\, v_1, \ldots, v_n\,\} \cup \mathsf{dom}(\Gamma_2)}$$

For the types of the new variables $v_1, \ldots, v_n$, let

$$C_j \quad = \quad \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\, [\, (v_1\, x_1 \cdots x_m)/y_1, \ldots, (v_{j-1}\, x_1 \cdots x_m)/y_{j-1}\,]B_j$$

for $1 \leq j \leq n$. Then we define the new context

$$\Gamma_b \quad = \quad \Gamma_1 \oplus [\, v_1{:}C_1, \ldots, v_n{:}C_n \,] \oplus (\theta_b \Gamma_2)$$

(The reason we have to apply θ_b to Γ_2 is that the types assigned in Γ_2 might contain the variable v.)

Finally, for the type of N_b, let

$$C_b \quad = \quad \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\, [\, (v_1 x_1 \cdots x_m)/y_1, \ldots, (v_n x_1 \cdots x_m)/y_n \,] B_0$$

Proposition 4.59 *For any $b \in H$ and $\theta \in \Theta_\Gamma$, θv is convertible to a term of the form*

$$\lambda x_1{:}A'_1.\ \cdots \lambda x_m{:}A'_m.\, b\, N_1 \cdots N_n$$

iff there is a θ' such that $\theta =_\lambda \theta' \circ \theta_b$.

Proof: Assume θv is as stated. We need to construct a θ' such that $\theta =_\lambda \theta' \circ \theta_b$. Let $\{\, \hat{v}_1, \ldots, \hat{v}_l \,\} = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ and $\theta \hat{v}_i = \hat{M}_i$, for $1 \leq i \leq l$. It is easy to check that the following suffices for θ':

$$[\, (\lambda x_1{:}A'_1.\ \cdots \lambda x_m{:}A'_m.\, N_1)/v_1, \ldots, (\lambda x_1{:}A'_1.\ \cdots \lambda x_m{:}A'_m.\, N_n)/v_n \ ,\ \hat{M}_1/\hat{v}_1, \ldots, \hat{M}_l/\hat{v}_l \,]$$

The reverse implication follows from the fact that the body of N_b (*i.e.*, the result of stripping off the λ's) is rigid. □

The following fact will also be useful:

Lemma 4.60 *For any Γ' and $\theta' \in \Theta^{\Gamma'}_{\Gamma_b}$, if $\Gamma' \vdash_\Sigma \theta' A \in \mathsf{Type}$, then (a) for $1 \leq j \leq n$, $\Gamma' \vdash_\Sigma \theta' C_j \in \mathsf{Type}$, and (b) $\Gamma' \vdash_\Sigma \theta' C_b \in \mathsf{Type}$.*

Proof: Assume $\Gamma' \vdash_\Sigma \theta' A \in \mathsf{Type}$, and let B be as in Definition 4.58. Since B is either some A_i or the type of a constant, we know that

$$\Gamma' \oplus [\, x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m \,] \vdash_\Sigma \theta' B \in \mathsf{Type}$$

and, since $B = \Pi y_1{:}B_1.\ \cdots \Pi y_n{:}B_n.\, B_0$, we have

$$\Gamma' \oplus [\, x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m \,] \oplus [\, y_1{:}\theta' B_1, \ldots, y_{j-1}{:}\theta' B_{j-1} \,] \vdash_\Sigma \theta' B_j \in \mathsf{Type}$$

for $1 \leq j \leq n$, and also

$$\Gamma' \oplus [\, x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m \,] \oplus [\, y_1{:}\theta' B_1, \ldots, y_n{:}\theta' B_n \,] \vdash_\Sigma \theta' B_0 \in \mathsf{Type}$$

The result follows, using weakening (Proposition 2.32) n times and the Π typing rule m times. □

In order to show that the flexible-rigid transformation makes progress, we will need a notion of *height* of a term:[13]

Definition 4.61 *The* height *of a well-typed term M with respect to a context Γ, written "$\mathsf{height}_\Gamma(M)$", is given by*

- *If M wh_β M' for some M', then $\mathsf{height}_\Gamma(M) = \mathsf{height}_\Gamma(M')$.*
- *If $M = \lambda v{:}A.\, M'$ for some v, A, M', then $\mathsf{height}_\Gamma(M) = \mathsf{height}_{\Gamma \oplus v:A}(M')$.*
- *If M is a body and $\Gamma \vdash_\Sigma M \in \Pi v{:}A.\, B$, then $\mathsf{height}_\Gamma(M) = \mathsf{height}_\Gamma(\lambda v{:}A.\, M\, v)$.*
- *If M is an atom not of Π type in Γ, then $\mathsf{height}_\Gamma(M) = 1$.*
- *If M is an application $M'\,N'$ not of Π type, then $\mathsf{height}_\Gamma(M) = \max(\mathsf{height}_\Gamma(M'), 1 + \mathsf{height}_\Gamma(N'))$.*

(Note that this is well-defined for well-typed terms because of the SN and determinacy properties of wh_β and unicity of types.[14])

It will be important to note that height is invariant under conversion.

Proposition 4.62 *Given two terms M and M', well-typed in a context Γ, such that $M =_\lambda M'$, we have $\mathsf{height}_\Gamma(M) = \mathsf{height}_\Gamma(M')$.*

Proof: By induction on the structure of M and M', using the properties proved in Section 4.2, recalling that either (a) M or M' is a weak head redex, (b) M or M' is an abstraction, and if the other is not then it has Π type, or (c) both M and M' are bodies. □

Then to compare substitutions, we will use a multiset ordering based on height:

[13] The reason we cannot simply use size is that we must not take into account the λ's and abstracted variable types.

[14] It will be important to note that we only use here unicity of types for bodies.

Definition 4.63 *Given contexts Γ, Γ' and two substitutions $\theta, \theta' \in \Theta_\Gamma^{\Gamma'}$, define $\theta \gg \theta'$ iff there is an integer h such that*

$$|\{\, v \in V \mid \mathsf{height}_{\Gamma'}(\theta v) = h \,\}| > |\{\, v \in V \mid \mathsf{height}_{\Gamma'}(\theta' v) = h \,\}|$$

but for each $k > h$,

$$|\{\, v \in V \mid \mathsf{height}_{\Gamma'}(\theta v) = k \,\}| = |\{\, v \in V \mid \mathsf{height}_{\Gamma'}(\theta' v) = k \,\}|$$

Proposition 4.64 *In Proposition 4.59, θ' can be chosen such that $\theta \gg \theta'$.*

Proof: The θ' in that proof suffices, since the difference between θ and θ' is the replacement of one substitution term by a collection of substitution terms with strictly smaller heights. (Note that $\mathsf{height}_{\Gamma'}(\lambda x_1{:}A'_1.\cdots\lambda x_m{:}A'_m.b\,N_1\cdots N_n) = \max_{1\le i\le n}(1+\mathsf{height}_{\Gamma'}(N_i)) > \mathsf{height}_{\Gamma'}(N_j)$, for $1 \le j \le n$.) □

For minimality of the transformation defined below, we will need

Proposition 4.65 *For each $b \in H$, θ_b is permanent.*

Proof: For some Γ', let θ_1, θ_2 be arbitrary substitutions in $\Theta_{\Gamma_b}^{\Gamma'}$ such that $\theta_1 \circ \theta_b =_\lambda \theta_2 \circ \theta_b$. We will show that, therefore, $\theta_1 =_\lambda \theta_2$, *i.e.*, for every $u \in \mathsf{dom}(\Gamma_b)$, $\theta_1 u =_\lambda \theta_2 u$.

- Assume $u \in \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Then $(\theta_1 \circ \theta_b)u = \theta_1 u$, and $(\theta_2 \circ \theta_b)u = \theta_2 u$, so $\theta_1 u = \theta_2 u$.
- Assume $u = v_i$ where $1 \le i \le n$. Note that for some $A'_1, \ldots, A'_m$ and $A''_1, \ldots, A''_m$, we have

$$\begin{aligned}(\theta_1 \circ \theta_b)v &= \theta_1 N_b \\ &= \lambda x_1{:}A'_1.\cdots\lambda x_m{:}A'_m.\, b\,((\theta_1 v_1)x_1\cdots x_m)\cdots((\theta_1 v_n)x_1\cdots x_m)\end{aligned}$$

  and

$$\begin{aligned}(\theta_2 \circ \theta_b)v &= \theta_2 N_b \\ &= \lambda x_1{:}A''_1.\cdots\lambda x_m{:}A''_m.\, b\,((\theta_2 v_1)x_1\cdots x_m)\cdots((\theta_2 v_n)x_1\cdots x_m)\end{aligned}$$

Applying the abstraction and body decomposition methods (Definitions 4.21 and 4.28), we find that for $1 \le i \le n$, $(\theta_1 v_i)\, x_1\cdots x_m =_\lambda (\theta_2 v_i)\, x_1\cdots x_m$, and thus

$$\lambda v_1{:}A_1.\cdots\lambda v_m{:}A_m.\,(\theta_1 v_i)\, x_1\cdots x_m \quad =_\lambda \quad \lambda v_1{:}A_1.\cdots\lambda v_m{:}A_m.\,(\theta_2 v_i)\, x_1\cdots x_m$$

The result then follows from η reduction. □

Proposition 4.66 *For distinct $b, b' \in H$, θ_b and $\theta_{b'}$ have no common instances, i.e., there is no θ, θ' such that $\theta \circ \theta_b =_\lambda \theta' \circ \theta_{b'}$.*

Proof: Assume otherwise, so that for some distinct $b, b' \in H$ and substitutions θ, θ', we have $\theta \circ \theta_b =_\lambda \theta' \circ \theta_{b'}$. Then, in particular, $(\theta \circ \theta_b)v =_\lambda (\theta' \circ \theta_{b'})v$, *i.e.*, $\theta N_b =_\lambda \theta' N_{b'}$. However, N_b and $N_{b'}$ are both rigid bodies, so by Proposition 4.52, $\theta N_b \approx N_b$ and $\theta N'_{b'} \approx N_{b'}$, and by Proposition 4.32, $\theta N_b \approx \theta' N_{b'}$. We thus have $N_b \approx N_{b'}$, which is plainly false since $b \neq b'$. □

Now we will see how to use this fact to transform our unification problem. The following steps are mostly symbol manipulation, but the point to keep in mind is that we are trying to re-express $\mathcal{U}(Q)$ as $\bigcup_{b \in H} \mathcal{U}(Q_b)$ for some $\{ Q_b \mid b \in H \}$, since by Definition 3.12, this constitutes a *correct* transformation.

First, expand the definition of $\mathcal{U}(Q)$ in a somewhat more explicit form:

$$\{ \hat{\theta} \mid \exists \theta.\, \hat{\theta} =_\lambda \theta \circ \theta_0 \wedge \theta \in \Theta_\Gamma \wedge \forall \langle \Psi\, ,\, U,\, U' \rangle \in D.\, \theta U =_\lambda \theta U' \}$$

Now from Propositions 4.56 and 4.59, we know that the condition on θ implies the additional condition

$$\exists b \in H.\, \exists \theta'.\, \theta = \theta' \circ \theta_b$$

so we can conjoin this condition without changing the meaning of the set expression. Next perform some quantifier manipulation to get

$$\{ \hat{\theta} \mid \exists b \in H.\, \exists \theta'.\, \exists \theta.\, \hat{\theta} =_\lambda \theta \circ \theta_0 \wedge \theta = \theta' \circ \theta_b \wedge \theta \in \Theta_\Gamma \wedge \forall \langle \Psi\, ,\, U,\, U' \rangle \in D.\, \theta U =_\lambda \theta U' \}$$

Next, eliminate θ by replacing it by $\theta' \circ \theta_b$, and change the $\exists b$ into a set union

$$\bigcup_{b \in H} \{ \hat{\theta} \mid \exists \theta'.\, \hat{\theta} =_\lambda (\theta' \circ \theta_b) \circ \theta_0 \wedge (\theta' \circ \theta_b) \in \Theta_\Gamma \wedge \forall \langle \Psi\, ,\, U,\, U' \rangle \in D.\, (\theta' \circ \theta_b) U =_\lambda (\theta' \circ \theta_b) U' \}$$

Now, in a key step, re-associate the compositions

$$\bigcup_{b \in H} \{ \hat{\theta} \mid \exists \theta'.\, \hat{\theta} =_\lambda \theta' \circ (\theta_b \circ \theta_0) \wedge (\theta' \circ \theta_b) \in \Theta_\Gamma \wedge \forall \langle \Psi\, ,\, U,\, U' \rangle \in D.\, \theta'(\theta_b U) =_\lambda \theta'(\theta_b U') \}$$

This is almost in the form we want, *i.e.*, $\bigcup_{b \in H} \mathcal{U}(Q_b)$ for some family $\{ Q_b \mid b \in H \}$. The only obstacle is the condition $\theta' \circ \theta_b \in \Theta_\Gamma$, where we need a condition involving $\theta' \in \Theta_{\Gamma_b}$.

From Definition 2.42 of Θ_Γ, we know that the condition $\theta' \circ \theta_b \in \Theta_\Gamma$ means that for some context Γ', and all variables $u \in \mathsf{dom}(\Gamma)$ and types B,

$$u : B \in \Gamma \quad \Rightarrow \quad \Gamma' \vdash_\Sigma \theta'(\theta_b u) \in \theta'(\theta_b B)$$

There are three cases of interest, depending on the position of u relative to v in Γ:

- If $u \in \mathsf{dom}(\Gamma_1)$, then $\theta_b u = u$ and $\theta_b B = B$ (since, by weak validity of Γ, v does not occur in B). Therefore the type condition is equivalent to $\Gamma' \vdash_\Sigma \theta' u \in \theta' B$.
- If $u \in \mathsf{dom}(\Gamma_2)$, then $\theta_b u = u$, so the typing condition is equivalent to $\Gamma' \vdash_\Sigma \theta' u \in \theta'(\theta_b B)$, *i.e.*, $\Gamma' \vdash_\Sigma \theta' u \in \theta' \hat{B}$, where $\hat{B}$ is the type assigned to u by $\theta_b \Gamma_2$.
- If $u = v$ then also $B = A$ and $\theta_b u = N_b$ and $\theta_b B = B = A$. Thus the typing condition is equivalent to $\Gamma' \vdash_\Sigma \theta' N_b \in \theta' A$.

For analyzing the latter case, we will need the following:

Lemma 4.67 *Given a context Γ', a type A, and a (possibly ill-typed) substitution $\theta' \in \Theta^{\mathsf{dom}(\Gamma')}_{\mathsf{dom}(\Gamma_b)}$, we have $\Gamma' \vdash_\Sigma \theta' N_b \in \theta' A$ iff (a) $\Gamma' \vdash_\Sigma \theta' A \in \mathsf{Type}$, (b) $\theta' A =_\lambda \theta' C_b$, and (c) for $1 \leq j \leq n$, $\Gamma' \vdash_\Sigma \theta' v_j \in \theta' C_j$.*

Proof: Using the definition of N_b and properties of substitution, we have

$$\theta' N_b \quad = \quad \lambda x_1{:}\theta' A_1.\ \cdots \lambda x_m{:}\theta' A_m.\, b\,((\theta' v_1) x_1 \cdots x_m) \cdots ((\theta' v_n) x_1 \cdots x_m)$$

Recall that b is either a constant or one of the x_i, so

$$\Gamma' \oplus [\, x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m \,] \vdash_\Sigma b \in \theta' B$$

where B is as in Definition 4.58. Therefore, θN_b is well-typed in Γ' iff for $1 \leq j \leq n$,

$$\Gamma' \oplus [\, x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m \,] \vdash_\Sigma (\theta' v_j)\, x_1 \cdots x_m \in [\, (\theta' v_1)\, x_1 \cdots x_m / y_1, \ldots, (\theta' v_{j-1})\, x_1 \cdots x_m / y_{j-1} \,](\theta B_j)$$

By appealing to the abstraction typing rule m times, η-reducing, and recalling the definition of the C_j, we can see that this is equivalent to the conjunction, for $1 \leq j \leq n$, of

$$\Gamma' \vdash_\Sigma \theta' v_j \in \theta' C_j$$

Also, if this is the case, then

$$\Gamma' \vdash_\Sigma \theta' N_b \in \theta' C_b$$

We can now show the equivalence stated in the lemma.

Let A be an arbitrary type. First assume that $\Gamma' \vdash_\Sigma \theta' N_b \in \theta' A$. Then, since $\theta' N_b$ is well-typed in Γ', by the argument above, we have $\Gamma' \vdash_\Sigma \theta' v_j \in \theta' C_j$ for $1 \leq j \leq n$, and $\Gamma' \vdash_\Sigma \theta' N_b \in \theta' C_b$. Then by unicity of types (Proposition 2.30), $\theta' A =_\lambda \theta' C_b$. Finally, $\Gamma' \vdash_\Sigma \theta' A \in \mathsf{Type}$ by Proposition 2.34.

Next, assume that the conditions (a), (b), and (c) hold. By condition (c), we have $\Gamma' \vdash_\Sigma \theta' N_b \in \theta' C_b$, and the result follows by the conversion typing rule, given conditions (a) and (b). □

Combining the first two cases of interest above, with this analysis of the third case, results in the new condition that (a) $\theta' A =_\lambda \theta' C_b$, and (b) for all variables u and types B

$$u : B \in \Gamma_b \quad \Rightarrow \quad \Gamma' \vdash_\Sigma \theta' u \in \theta' B$$

This latter condition is just $\theta' \in \Theta^{\Gamma'}_{\Gamma_b}$. Since this applies to any Γ', we conclude that

$$\theta' \circ \theta_b \in \Theta_\Gamma \quad \Leftrightarrow \quad \theta' \in \Theta_{\Gamma_b} \wedge \theta' A =_\lambda \theta' C_b$$

Returning to our problem of re-expressing $\mathcal{U}(Q)$, we now have

$$\bigcup_{b \in H} \{\, \hat{\theta} \;\mid\; \exists \theta' . \hat{\theta} =_\lambda \theta' \circ (\theta_b \circ \theta_0) \wedge \theta' \in \Theta_{\Gamma_b} \wedge (\forall \langle \Psi \,,\, U,\, U' \rangle \in D . \theta'(\theta_b U) =_\lambda \theta'(\theta_b U')) \wedge \theta' A =_\lambda \theta' C_b \,\}$$

which can also be written as

$$\bigcup_{b \in H} \{\, \hat{\theta} \;\mid\; \exists \theta' . \hat{\theta} =_\lambda \theta' \circ (\theta_b \circ \theta_0) \wedge \theta' \in \Theta_{\Gamma_b} \wedge \theta' \text{ unifies } (\theta_b D \uplus \{\langle [\,] \,,\, A,\, C_b \rangle\}) \,\}$$

and then collapsing the definition of $\mathcal{U}$ gives the equivalent form

$$\bigcup_{b \in H} \mathcal{U}(\langle \Gamma_b, \theta_b \circ \theta_0, \theta_b D \uplus \{\langle [\,] \,,\, A,\, C_b \rangle\} \rangle)$$

These considerations motivate the following definition:

Definition 4.68 *For each $b \in H$ define the unification problem*

$$Q_b \quad = \quad \langle \Gamma_b, \theta_b \circ \theta_0, \theta_b D \uplus \{\langle [\,] \,,\, A,\, C_b \rangle\} \rangle$$

Transformation 4.4 *Let Q, H, and $\{\, Q_b \mid b \in H \,\}$ be as above. Then make the transition*

$$Q \quad \mapsto \quad \{\, Q_b \mid b \in H \,\}$$

Proposition 4.69 *Transformation 4.4 is valid.*

Proof: *Correctness* (in the sense of Definition 3.13) follows from the reasoning above.

To prove acceptability, we must first show how to construct new accountings out of an old one. Let $\sqsubset$ be an accounting for Q. For each $b \in H$, define $\sqsubset^b_{\text{fr}}$ by[15]

[15] Recall that $\Gamma = \Gamma_1 \oplus v{:}A \oplus \Gamma_2$, and $\Gamma_b = \Gamma_1 \oplus [\, v_1{:}C_1, \ldots, v_n{:}C_n \,] \oplus (\theta_b \Gamma_2)$.

- $\theta_b P \sqsubset^b_{\text{fr}} \theta_b X$ if $P \sqsubset X$ and $X \in D \cup \mathsf{ran}(\Gamma_2)$;
- $\theta_b P \sqsubset^b_{\text{fr}} B$ if $P \sqsubset B$ and $B \in \mathsf{ran}(\Gamma_1)$, in which case, $\mathcal{F}(P) \subseteq \mathsf{dom}(\Gamma_1)$, so $\theta_b P = P$;
- $\theta_b P \sqsubset^b_{\text{fr}} C_i$ for $1 \leq i \leq n$ if $P \sqsubset A$, in which case, $\theta_b P = P$;
- $\theta_b P \sqsubset^b_{\text{fr}} \langle [\,] , A, C_b \rangle$ if $P \sqsubset A$, in which case, $\theta_b P = P$;
- $\langle [\,] , A, C_b \rangle \sqsubset^b_{\text{fr}} \theta_b P$ if $P \not\sqsubset A$; and
- $\langle [\,] , A, C_b \rangle \sqsubset^b_{\text{fr}} \theta_b B$ for $B \in \mathsf{ran}(\Gamma_2)$.

It is straightforward to check that $\sqsubset^b_{\text{fr}}$ is a strict partial order.

To show that $\sqsubset^b_{\text{fr}}$ is an accounting, consider an arbitrary $X \in \theta_b D \uplus \{\langle [\,], A, C_b \rangle\} \uplus \mathsf{ran}(\Gamma_b)$. For the first part of Definition 4.37 of an accounting, we will need to show that for any unifier $\theta' \in \Theta_{\Gamma_b}$ of $D_X^{\sqsubset^b_{\text{fr}}}$, $\theta' X$ is well-typed. There are four cases:

- $X = \theta_b P$ for some $P \in D$ such that $P \sqsubset A$. Then $\mathcal{F}(P) \subseteq \mathsf{dom}(\Gamma_1)$ (by Definition 4.37), so $X = \theta_b P = P$, and $D_X^{\sqsubset^b_{\text{fr}}} = \theta_b D_P^{\sqsubset} = D_P^{\sqsubset}$. (By transitivity, $P' \sqsubset P \Rightarrow P' \sqsubset A$, so $\mathcal{F}(P') \subseteq \mathsf{dom}(\Gamma_1)$.) The result then follows, since $\sqsubset$ is an accounting.
- $X = \theta_b P$ for some $P \in D$ such that $P \not\sqsubset A$. Then $D_X^{\sqsubset^b_{\text{fr}}} = \theta_b D_P^{\sqsubset} \uplus \{\langle [\,] , A, C_b \rangle\}$. Let $\theta' \in \Theta_{\Gamma_b}$ be a unifier of $D_X^{\sqsubset^b_{\text{fr}}}$. Then $\theta' \circ \theta_b$ unifies $D_P^{\sqsubset}$, and, since $\theta' A =_\lambda \theta' C_b$, from the conclusion following Lemma 4.67, $\theta' \circ \theta_b \in \Theta_\Gamma$. Therefore, since $\sqsubset$ is an accounting, $(\theta' \circ \theta_b) P$ is well-typed. But $(\theta' \circ \theta_b) P = \theta'(\theta_b P) = \theta' X$, so the result follows.
- $X = \langle [\,] , A, C_b \rangle$ or $X = C_i$ for $1 \leq i \leq m$. Then $D_X^{\sqsubset^b_{\text{fr}}} = \theta_b D_A^{\sqsubset} = D_A^{\sqsubset}$, since $\mathcal{F}(D_A^{\sqsubset}) \subseteq \mathsf{dom}(\Gamma_1)$ by Definition 4.37. The result then follows from Lemma 4.60.
- $X = B \in \mathsf{ran}(\Gamma_1)$. Then, $D_X^{\sqsubset^b_{\text{fr}}} = \theta D_X^{\sqsubset} = D_X^{\sqsubset}$, so the result follows as in the first case.
- $X = \theta_b B$, for $B \in \mathsf{ran}(\Gamma_2)$. Then $D_X^{\sqsubset^b_{\text{fr}}} = \theta_b D_B^{\sqsubset} \uplus \{\langle [\,] , A, C_b \rangle\}$. The reasoning is the same as in the second case.

The permanence condition of Definition 4.38 follows from permanence of the θ_b (Proposition 4.65) and the closure of permanence under composition (Proposition 4.42).

The third condition of acceptability, *i.e.*, weak validity of Γ_b, follows easily from weak validity of Γ, with the additional considerations that (a) $\mathcal{F}(C_i) \subseteq \mathsf{dom}(\Gamma_1) \cup \{ v_1, \ldots, v_{i-1} \}$, and (b) letting $\Gamma_2 = [\, u_1 \colon \hat{B}_1, \ldots, u_k \colon \hat{B}_k \,]$, we have $\mathcal{F}(\theta \hat{B}_i) \subseteq \mathsf{dom}(\Gamma_1) \cup \{ v_1, \ldots, v_n \} \cup \{ u_1, \ldots, u_{i-1} \}$.

We will show minimality by contradiction: Assume that for some distinct $b, b' \in H$, there is some unifier $\hat{\theta} \in \mathcal{U}(Q_b) \cap \mathcal{U}(Q_{b'})$. Then there are θ' and θ'' such that $\theta' \circ (\theta_b \circ \theta_0) =_\lambda \hat{\theta} =_\lambda \theta'' \circ (\theta_{b'} \circ \theta_0)$, *i.e.*, $(\theta' \circ \theta_b) \circ \theta_0 =_\lambda (\theta'' \circ \theta_{b'}) \circ \theta_0$. But since θ_0 is permanent, $\theta' \circ \theta_b =_\lambda \theta'' \circ \theta_{b'}$. However, θ_b and $\theta_{b'}$ have no common instances, so this is a contradiction. □

Example 4.70 *Consider the unification problem* $Q = \langle \Gamma, \theta_\Gamma^{id}, \{\langle [\,] \,,\, f\ \top,\ \mathsf{triv}\ \top\rangle\}\rangle$ *in the signature*

$$\langle\, \mathsf{o}{:}\mathsf{Type}\ ,\ \vdash{:}\mathsf{o}{\to}\mathsf{Type}\ ,\ \top{:}\mathsf{o}\ ,\ \supset{:}\mathsf{o}{\to}\mathsf{o}{\to}\mathsf{o}\ ,\ \mathsf{triv}{:}\Pi p{:}\mathsf{o}.\ \vdash\ (\supset\ p\,p)\,\rangle$$

where the unification context Γ *is*

$$[\, f{:}\Pi p{:}\mathsf{o}.\ \vdash\ (\supset\ p\ \top)\,]$$

In $\lambda_\to$*, a unification problem like this (replacing the types by simple types) would have two solutions, with possible instantiations for* f *being* $\lambda z{:}\mathsf{o}.\ \mathsf{triv}\,\top$ *and* $\lambda z{:}\mathsf{o}.\ \mathsf{triv}\,z$*. However, neither of these terms has the type required by* Γ*. Our transformations correctly fail to find a solution. We can apply the flexible-rigid transformation (4.4). Trying the projection substitution* $\theta_p = [\,\lambda p{:}\mathsf{o}.\ p/f\,]$ *yields*

$$\langle [\,], \theta_p, \{\,\langle [\,]\,,\ \top,\ \mathsf{triv}\,\top\rangle, \langle [\,]\,,\ \Pi p{:}\mathsf{o}.\ \vdash\ (\supset\ p\,\top),\ \Pi p{:}\mathsf{o}.\,\mathsf{o}\rangle\,\}\rangle$$

for which applications of the rigid-rigid transformation (4.3) eventually indicate failure (i.e., make a transition to $\{\ \}$*). Trying instead the imitation substitution* $\theta_{\mathsf{triv}} = [\,\lambda p{:}\mathsf{o}.\ \mathsf{triv}\,(f_1\,p)/f\,]$ *yields* $\langle [\,f_1{:}\mathsf{o}{\to}\mathsf{o}\,], \theta_{\mathsf{triv}}, D_1\rangle$*, where* D_1 *is*

$$\{\,\langle [\,]\,,\ \mathsf{triv}\,(f_1\,\top),\ \mathsf{triv}\,\top\rangle\ ,\ \langle [\,]\,,\ \Pi p{:}\mathsf{o}.\ \vdash\ (\supset\ p\,\top),\ \Pi p{:}\mathsf{o}.\ \vdash\ (\supset\ (f_1\,p)\,(f_1\,p))\rangle\,\}$$

Four applications of the rigid-rigid transformation (4.3) lead to the the disagreement set

$$\{\,\langle [\,]\,,\ f_1\,\top,\ \top\rangle\ ,\ \langle [\,p{:}\mathsf{o}\,]\,,\ p,\ f_1\,p\rangle\ ,\ \langle [\,p{:}\mathsf{o}\,]\,,\ \top,\ f_1\,p\rangle\,\}$$

This will lead to failure, since f_1 *is constrained by the second disagreement pair to be* $\lambda p{:}\mathsf{o}.\ p$ *and by the third disagreement pair to be* $\lambda p{:}\mathsf{o}.\ \top$*.*

4.5 Completeness

In this section we show that the transformations developed in the previous section together form a complete transformation relation in the sense of Definition 3.17, and hence yield a complete pre-unification algorithm. The main ideas in our completeness argument are essentially the same as Huet's.

Definition 4.71 *Let the transformation relation ρ^{Π} be the union of Transformations 4.1 through 4.4.*

Proposition 4.72 *ρ^{Π} is valid (correct, minimal and acceptable).*

Proof: Immediate from validity of the transformations, using Proposition 3.14. □

More substantial is

Proposition 4.73 *ρ^{Π} is complete.*

Proof: We will use Proposition 3.20. Given a substitution $\hat{\theta}$, define the ordering $\succ_{\hat{\theta}}$ as follows. Let $Q = \langle \Gamma, \theta_0, D \rangle$ and $Q' = \langle \Gamma', \theta_0', D' \rangle$ be two unification problems having $\hat{\theta}$ as a solution, and let θ and θ' be such that $\hat{\theta} =_\lambda \theta \circ \theta_0$ and $\hat{\theta} =_\lambda \theta' \circ \theta_0'$. (These are uniquely determined up to convertibility because of the permanence of θ_0 and θ_0'.) Then $Q \succ_{\hat{\theta}} Q'$ iff either (a) $\theta \gg \theta'$, or (b) $\theta = \theta'$ and $\theta D >_{\mathsf{size}} \theta D'$, where $>_{\mathsf{size}}$ is the multiset ordering of disagreement sets based on size.[16] Then $\succ_{\hat{\theta}}$ is clearly a well founded ordering (being the lexicographic combination of two well-founded orderings).

Now, let $Q = \langle \Gamma, \theta_0, D \rangle$ and $Q' = \langle \Gamma', \theta_0', D' \rangle$ be such that for some $\mathcal{Q}'$, we have $Q \; \rho^{\Pi} \; \mathcal{Q}'$ and $Q' \in \mathcal{Q}'$, thus satisfying the conditions of Proposition 3.20. We will show that $Q \succ_{\hat{\theta}} Q'$, from which it follows that ρ is decreasing. Let θ and θ' be substitutions such that $\hat{\theta} =_\lambda \theta \circ \theta_0$ and $\hat{\theta} =_\lambda \theta' \circ \theta_0'$. (Again, these are unique up to convertibility.) There are two cases, depending on which of our transformations was used in making the transition from Q to $\mathcal{Q}'$:

- Other than the flexible-rigid transformation. Then for some D_1, P and $\hat{D}$, $D = D_1 \uplus \{P\}$, $\Gamma' = \Gamma$, $\theta_0' = \theta_0$, and $D' = D_1 \uplus \hat{D}$, where $\theta P \lhd \theta \hat{D}$. Thus $\theta =_\lambda \theta'$. Also, $\theta D'$ results from θD by replacing the disagreement pair θP by the disagreement pairs $\theta \hat{D}$ of strictly smaller size, and thus $\theta D >_{\mathsf{size}} \theta D'$.
- The flexible-rigid transformation. Then, by Proposition 4.59, there is a θ_b, such that $\theta_0' =_\lambda \theta_b \circ \theta_0$, $\theta =_\lambda \theta' \circ \theta_b$, and $\theta \gg \theta'$.

□

[16] The ordering $\succ_{\hat{\theta}}$ is only well-defined because the orderings $\gg$ and $>_{\mathsf{size}}$ are invariant under conversion. (See Proposition 4.62 and Definition 4.16.)

4.6 Unifiability of Solved Form Unification Problems

The value of pre-unification in $\lambda_{\rightarrow}$ is that solved form disagreement sets (ones containing only flexible-flexible pairs) are always unifiable, and so pre-unifiability implies unifiability [36]. By making vital use of the accounting in the definition of *acceptability*, we can generalize Huet's constructive proof of this fact to acceptable solved form unification problems in λ_{Π}. For the simply typed subset of λ_{Π}, the unifier that we construct specializes to Huet's.[17]

Definition 4.74 *For a weakly valid context* Γ*, the* canonical unifier θ^C_Γ *over* Γ *is the substitution assigning to each variable*

$$v \quad : \quad \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ \mathsf{c}\ N_1 \cdots N_n$$

in Γ*, the term*

$$\lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ h_{\mathsf{c}}\ N_1 \cdots N_n$$

where the kind assigned to the constant c *is* $\Pi y_1{:}B_1.\ \cdots \Pi y_n{:}B_n.\ \mathsf{Type}$*, and* h_{c} *is a variable of type*

$$\Pi y_1{:}B_1.\ \cdots \Pi y_n{:}B_n.\ \mathsf{c}\ y_1 \cdots y_n$$

(Note that in the simply typed subset of λ_{Π}, $n = 0$.)

Proposition 4.75 *If* Q *is a acceptable unification problem in solved form with unification context* Γ*, then* $\theta^C_\Gamma \in \mathcal{U}(Q)$.

Proof: Let $\sqsubset$ be an accounting for Q. Since disagreement sets are finite, $\sqsubset$ is a well founded ordering, and thus we will give an inductive argument. Let $P = \langle \Psi\ ,\ M,\ M' \rangle$ be an arbitrary member or our disagreement set. (There are no flexible-flexible type disagreement pairs.) We want to show that θ^C_Γ unifies P. By induction, we may assume that θ^C_Γ unifies $D^{\sqsubset}_P$, and so $\theta^C_\Gamma P$ is well-typed. Let

$$\begin{array}{rcl} M & = & v\ M_1 \cdots M_m \\ M' & = & v'\ M'_1 \cdots M'_{m'} \end{array}$$

where v and v' are variables in $\mathsf{dom}(\Gamma)$ with types

$$\begin{array}{rl} v & : \ \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ \Pi w_1{:}C_1.\ \cdots \Pi w_l{:}C_l.\ \mathsf{c}\ N_1 \cdots N_n \\ v' & : \ \Pi x'_1{:}A'_1.\ \cdots \Pi x'_{m'}{:}A'_{m'}.\ \Pi w'_1{:}C'_1.\ \cdots \Pi w'_l{:}C'_l.\ \mathsf{c}\ N'_1 \cdots N'_n \end{array}$$

[17] There is another notion of unifiability, sometimes called "closed" (as opposed to "open") unifiability, which requires the unifying substitutions to contain only closed substitution terms (*i.e.*, ones with no free variables). In our terminology, this would require a unifier $\theta \in \Theta^{[\,]}_\Gamma$. This problem is discussed for $\lambda_{\rightarrow}$ in [48], but is much more difficult in λ_{Π}, because determining the existence of closed terms of a given type is equivalent to general theorem proving [30].

for

$$\mathsf{c} \;:\; \Pi y_1{:}B_1.\ \cdots \Pi y_n{:}B_n.\ \mathsf{Type}$$

The reason that the types of both v and v' must involve the same type constant c, is that $\theta^C_\Gamma M$ and $\theta^C_\Gamma M'$ have the same type. To express these two instantiated terms, for $1 \le j \le n$, let

$$\begin{aligned} \hat{N}_j &= [\,(\theta^C_\Gamma M_1)/x_1, \ldots, (\theta^C_\Gamma M_m)/x_m\,]N_j \\ \hat{N}'_j &= [\,(\theta^C_\Gamma M'_1)/x_1, \ldots, (\theta^C_\Gamma M'_{m'})/x_{m'}\,]N'_j \end{aligned}$$

and, for $1 \le i \le l$, let

$$\begin{aligned} \hat{C}_i &= [\,(\theta^C_\Gamma M_1)/x_1, \ldots, (\theta^C_\Gamma M_l)/x_l\,]C_i \\ \hat{C}'_i &= [\,(\theta^C_\Gamma M'_1)/x'_1, \ldots, (\theta^C_\Gamma M'_l)/x'_l\,]C'_i \end{aligned}$$

Then we have

$$\begin{aligned} \theta^C_\Gamma M &= \lambda w_1{:}\hat{C}_1.\ \cdots \lambda w_l{:}\hat{C}_l.\ h_{\mathsf{c}}\,\hat{N}_1 \cdots \hat{N}_n \\ \theta^C_\Gamma M' &= \lambda w'_1{:}\hat{C}'_1.\ \cdots \lambda w'_l{:}\hat{C}'_l.\ h_{\mathsf{c}}\,\hat{N}'_1 \cdots \hat{N}'_n \end{aligned}$$

Since $\theta^C_\Gamma P$, which is $\langle \theta^C_\Gamma \Psi\ ,\ \theta^C_\Gamma M,\ \theta^C_\Gamma M' \rangle$, is well-typed, we have

$$\Pi w_1{:}\hat{C}_1.\ \cdots \Pi w_l{:}\hat{C}_l.\ \mathsf{c}\,\hat{N}_1 \cdots \hat{N}_n =_\lambda \Pi w'_1{:}\hat{C}'_1.\ \cdots \Pi w'_l{:}\hat{C}'_l.\ \mathsf{c}\,\hat{N}'_1 \cdots \hat{N}'_n$$

It then follows that each $\hat{N}_j =_\lambda \hat{N}'_j$, and also that each $\hat{C}_i =_\lambda \hat{C}'_i$, so $\theta^C_\Gamma M =_\lambda \theta^C_\Gamma M'$. □

4.7 Automatic Term Inference

It is well known that first-order unification provides for type inference in $\lambda_\to$ with type variables and in similar languages [51]. Recently, it has been shown that $\mathrm{HOU}_\to$ is the key ingredient for the corresponding problem in the ω-order polymorphic λ-calculus [59]. In λ_Π there is another problem of interest, namely *term inference*, which requires HOU_Π. This problem has two important applications. One is making our unification algorithm more widely applicable, by initially establishing the required invariant, as mentioned at the end of Section 4.3, and made precise below. The other is to provide automatic type inference in encoded languages, as described in Chapter 7. As in the type inference algorithms mentioned above, the basic idea is to combine type-checking and unification, in this case, HOU_Π. A similar problem is addressed by Huet [33] and by Pollack [61] under the name of "argument synthesis".

Given a signature Σ, context Γ, and a term M whose free variables are all typed by Γ, it may be the case that M is not well-typed, but it has well-typed substitution instances. The goal of term inference is to determine exactly which substitution instances (if any) of a

given term are well-typed. It does this by collecting pairs of types that have to be unified by any substitution instantiating M to a well-typed term.

We will construct the type checking/term inference algorithm using two mutually recursive operations expressed as an inference system. We define two judgments, "$\Gamma; \Psi \vdash_\Sigma M \in A$ **with** D" and "$\Gamma; \Psi \vdash_\Sigma A \in K$ **with** D", where D is a disagreement set. For the first judgment, M will be given and we will compute A and D. In the second, A will be given and we will compute K and D. Unifiers of D, if any, will lead to instantiations of M and A (or A and K), as will be made precise below. As usual, Γ is the unification context, and Ψ is a universal context.

There is a Standard ML [29] implementation based on this procedure extended to deal with type variables [19].

Definition 4.76 *Let the judgments "$\Gamma; \Psi \vdash_\Sigma A \in K$ **with** D" and "$\Gamma; \Psi \vdash_\Sigma M \in A$ **with** D" be defined by the following inference system.*

First for terms,

$$\frac{\mathsf{c}{:}A \in \Sigma}{\Gamma; \Psi \vdash_\Sigma \mathsf{c} \in A \textbf{ with } \{\,\}}$$

$$\frac{v{:}A \in \Gamma \oplus \Psi}{\Gamma; \Psi \vdash_\Sigma v \in A \textbf{ with } \{\,\}}$$

$$\frac{\Gamma; \Psi \vdash_\Sigma A \in \mathsf{Type} \textbf{ with } D \qquad \Gamma; \Psi \oplus v{:}A \vdash_\Sigma M \in B \textbf{ with } D'}{\Gamma; \Psi \vdash_\Sigma \lambda v{:}A.\, M \in \Pi v{:}A.\, B \textbf{ with } D \uplus D'}$$

$$\frac{\Gamma; \Psi \vdash_\Sigma M \in C \textbf{ with } D \qquad C \ \mathrm{wh}_\beta^* \ \Pi v{:}A.\, B \qquad \Gamma; \Psi \vdash_\Sigma N \in A' \textbf{ with } D'}{\Gamma; \Psi \vdash_\Sigma M\, N \in [\, N/v \,]B \textbf{ with } \{\langle \Psi\ ,\ A,\ A' \rangle\} \uplus D \uplus D'}$$

and then for types,

$$\frac{\mathsf{c}{:}K \in \Sigma}{\Gamma; \Psi \vdash_\Sigma \mathsf{c} \in K \textbf{ with } \{\,\}}$$

$$\frac{\Gamma; \Psi \vdash_\Sigma A \in \mathsf{Type} \textbf{ with } D \qquad \Gamma; \Psi \oplus v{:}A \vdash_\Sigma B \in \mathsf{Type} \textbf{ with } D'}{\Gamma; \Psi \vdash_\Sigma \Pi v{:}A.\, B \in \mathsf{Type} \textbf{ with } D \uplus D'}$$

$$\frac{\Gamma; \Psi \vdash_\Sigma A \in \mathsf{Type} \textbf{ with } D \qquad \Gamma; \Psi \oplus v{:}A \vdash_\Sigma B \in K \textbf{ with } D'}{\Gamma; \Psi \vdash_\Sigma \lambda v{:}A.\, B \in \Pi v{:}A.\, K \textbf{ with } D \uplus D'}$$

$$\frac{\Gamma; \Psi \vdash_\Sigma A \in \Pi v{:}B.\, K \textbf{ with } D \qquad \Gamma; \Psi \vdash_\Sigma M \in B' \textbf{ with } D'}{\Gamma; \Psi \vdash_\Sigma A\, M \in [\, M/v \,]K \textbf{ with } \{\langle \Psi\ ,\ B,\ B' \rangle\} \uplus D \uplus D'}$$

Once we construct A and D such that $\Gamma; \Psi \vdash_\Sigma M \in A$ **with** D, we will want to use our unification procedure to find unifiers of D, and so it is necessary that the unification problem $\langle \Gamma, \theta_\Gamma^{id}, D \rangle$ be acceptable (in the sense of Definition 4.38).

Proposition 4.77 *Let Σ be a valid signature, Γ and Ψ valid unification and universal contexts respectively, and M a (possibly ill-typed) term. Let A and D be such that $\Gamma;\Psi \vdash_\Sigma M \in A$* **with** *$D$, and let $Q = \langle \Gamma, \theta_\Gamma^{id}, D\rangle$. Then (a) Q is acceptable, and (b) for every context Γ' and solution $\theta \in \Theta_\Gamma^{\Gamma'}$ of Q, it is the case that $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in \theta A$ (and hence, by Proposition 2.34, also that $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta A \in \mathsf{Type}$). Similarly for types.*

Proof: We will argue by induction on the derivation $\mathcal{D}$ of $\Gamma;\Psi \vdash_\Sigma M \in A$ **with** D:

- If $\mathcal{D}$ consists of an instance of the first rule (for constants), then $M = \mathsf{c}$ for some constant $\mathsf{c}: A \in \Sigma$, and $D = \{\ \}$, so Q is vacuously acceptable. (Cf. Definition 4.38 of acceptability, noting that θ_Γ^{id} is trivially permanent.) Next, let Γ' be a context, and $\theta \in \Theta_\Gamma^{\Gamma'}$. (Since $D = \{\ \}$, θ unifies D.) Then $\theta\mathsf{c} = \mathsf{c}$, and $\mathcal{F}(A) = \{\ \}$, so we also have $\theta A = A$, and thus $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in \theta A$.

- If $\mathcal{D}$ consists of an instance of the second rule (for variables), then $M = v$ for some variable $v: A \in \Gamma \oplus \Psi$, and $D = \{\ \}$. The reasoning for part (a) is similar to the previous case. For part (b), if $v \in \mathsf{dom}(\Psi)$, then the reasoning is the same as the previous case. Otherwise, $v \in \mathsf{dom}(\Gamma)$, and, we simply rely on the definition of $\Theta_\Gamma^{\Gamma'}$.

- Assume that $\mathcal{D}$ ends in an instance of the third rule. Then M is an abstraction $\lambda v{:}\hat{A}.\ \hat{M}$, and A is a type $\Pi v{:}\hat{A}.\ \hat{B}$, where for some disagreement sets $\hat{D}$ and $\hat{D}'$,[18] $\mathcal{D}$ contains subderivations of (a) $\Gamma;\Psi \vdash_\Sigma \hat{A} \in \mathsf{Type}$ **with** $\hat{D}$, and (b) $\Gamma;\Psi \oplus v{:}\hat{A} \vdash_\Sigma \hat{M} \in \hat{B}$ **with** $\hat{D}'$, and $D = \hat{D} \uplus \hat{D}'$. By the induction hypothesis, $\hat{Q} = \langle \Gamma, \theta_\Gamma^{id}, \hat{D}\rangle$ and $\hat{Q}' = \langle \Gamma, \theta_\Gamma^{id}, \hat{D}'\rangle$ are acceptable, so there are accountings $\hat{\sqsubset}$ of $\hat{Q}$ and $\hat{\sqsubset}'$ of $\hat{Q}'$. Then the union $\sqsubset$ of $\hat{\sqsubset}$ and $\hat{\sqsubset}'$ is an accounting for Q, and thus Q is acceptable. To see the second condition, let Γ' be a context, and let $\theta \in \Theta_\Gamma^{\Gamma'}$ be a unifier of D. Since $D = \hat{D} \uplus \hat{D}'$, θ unifies $\hat{D}$ and $\hat{D}'$ as well. Thus, by the induction hypothesis, $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{A} \in \mathsf{Type}$ and $\Gamma' \oplus \theta\Psi \oplus v{:}\theta\hat{A} \vdash_\Sigma \theta\hat{M} \in \theta\hat{B}$. Then from the typing rule for abstractions, it follows that $\Gamma' \oplus \theta\Psi \vdash_\Sigma \lambda v{:}\theta\hat{A}.\ \theta\hat{M} \in \Pi v{:}\theta\hat{A}.\ \theta\hat{B}$, *i.e.*, $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in \theta A$.

- Assume $\mathcal{D}$ ends in an instance of the fourth rule. Then M is an application $\hat{M}\,\hat{N}$, $A = [\,\hat{N}/v\,]\hat{B}$, $D = \{\langle \Psi\ ,\ \hat{A},\ \hat{A}'\rangle\} \uplus \hat{D} \uplus \hat{D}'$, and $\mathcal{D}$ contains subderivations of the form $\Gamma;\Psi \vdash_\Sigma \hat{M} \in C$ **with** $\hat{D}$ and $\Gamma;\Psi \vdash_\Sigma \hat{N} \in \hat{A}'$ **with** $\hat{D}'$, where $C\ \mathrm{wh}_\beta{}^*\ \Pi v{:}\hat{A}.\ \hat{B}$. By the induction hypothesis, $\hat{Q} = \langle \Gamma, \theta_\Gamma^{id}, \hat{D}\rangle$ and $\hat{Q}' = \langle \Gamma, \theta_\Gamma^{id}, \hat{D}'\rangle$ are acceptable, having accountings $\hat{\sqsubset}$ and $\hat{\sqsubset}'$ respectively. Also by the induction hypothesis, for any context Γ' and unifier $\theta \in \Theta_\Gamma^{\Gamma'}$ of $\hat{D}$ and $\hat{D}'$, we have $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta C \in \mathsf{Type}$, and $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{A}' \in \mathsf{Type}$. By substitutivity of wh_β (Proposition 4.5), $\theta C\,\mathrm{wh}_\beta{}^*\,\theta(\Pi v{:}\hat{A}.\hat{B})$. Also, since wh_β preserves typing (Corollary 4.4), we have $\Gamma' \oplus \theta\Psi \vdash_\Sigma \Pi v{:}\theta\hat{A}.\theta\hat{B} \in \mathsf{Type}$, and thus $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{A} \in \mathsf{Type}$. In other words, $\hat{D} \uplus \hat{D}'$ "accounts for" $\langle \Psi\ ,\ \hat{A},\ \hat{A}'\rangle$, in the sense of Definition 4.37 of an accounting. Thus we can construct an accounting $\sqsubset$

[18] We are departing here from our convention of using "$\hat{D}$" for disagreement sequences.

for Q by $P \sqsubset X$ iff either (a) $P \hat{\sqsubset} X$, (b) $P \hat{\sqsubset}' X$, or (c) $P \in \hat{D} \uplus \hat{D}'$ and $X = \langle \Psi\,,\, \hat{A},\, \hat{A}' \rangle$, and hence Q is acceptable.

The second requirement is that for any unifier $\theta \in \Theta_\Gamma^{\Gamma'}$ of D, we have $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta(\hat{M}\,\hat{N}) \in \theta([\,\hat{N}/v\,]\hat{B})$. By the induction hypothesis, $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{M} \in \Pi v{:}\theta\hat{A}.\,\theta\hat{B}$ and $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{N} \in \theta\hat{A}'$. Since θ unifies D, it unifies, in particular, $\langle \Psi\,,\, \hat{A},\, \hat{A}' \rangle$, *i.e.*, $\theta\hat{A} =_\lambda \theta\hat{A}'$, and so by the conversion typing rule, $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{N} \in \theta\hat{A}$. Then, by the application typing rule, $\Gamma' \oplus \theta\Psi \vdash_\Sigma (\theta\hat{M})\,(\theta\hat{N}) \in [\,\theta\hat{N}/v\,](\theta\hat{B})$. However, by Lemma 2.23, we know that $[\,\theta\hat{N}/v\,](\theta\hat{B}) = \theta([\,\hat{N}/v\,]\hat{B})$, so the conclusion follows.

The cases for types are analogous. □

The previous proposition states that every solution of the constructed unification problem leads to a typing of the given term or type. The following says that all possible typings can be obtained this way.

Proposition 4.78 *Let Σ be a valid signature, Γ and Ψ valid contexts, and M a term with all of its free variables in $\Gamma \oplus \Psi$. For any context Γ', substitution $\theta \in \Theta_\Gamma^{\Gamma'}$, and type A', if $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in A'$, then there is a type A and a disagreement set D such that (a) $\Gamma; \Psi \vdash_\Sigma M \in A$ **with** D, (b) $\theta \in \mathcal{U}(\langle \Gamma, \theta_\Gamma^{id}, D \rangle)$, and (c) $A' =_\lambda \theta A$. Similarly for types.*

Proof: First consider the case that M is a variable $v \in \mathsf{dom}(\Gamma)$. Then for some $\hat{A}$, we have $v : \hat{A} \in \Gamma \oplus \Psi$. Therefore, $\Gamma; \Psi \vdash_\Sigma v \in \hat{A}$ **with** $\{\ \}$, and $\theta \in \mathcal{U}(\langle \Gamma, \theta_\Gamma^{id}, \{\ \} \rangle)$. By Definition 2.41 of $\Theta_\Gamma^{\Gamma'}$ and strengthening (Proposition 2.31), $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta v \in \theta\hat{A}$. Then by type unicity (Proposition 2.30), $A' =_\lambda \theta\hat{A}$, so the result follows with $A = \hat{A}$ and $D = \{\ \}$.[19]

We now proceed by induction on the derivation $\mathcal{D}$ of $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in A'$, under the assumption that $M \notin \mathsf{dom}(\Gamma)$.

- If $\mathcal{D}$ consists solely of an instance of the rule for typing constants, then θM is some constant c such that $\mathsf{c} : A' \in \Sigma$. Since $\theta M = \mathsf{c}$ and we are assuming $M \notin \mathsf{dom}(\Gamma)$, it follows that $M = \mathsf{c}$. (Note that we are considering $\theta M = \mathsf{c}$ and not $\theta M =_\lambda \mathsf{c}$.) Then we have (a) $\Gamma; \Psi \vdash_\Sigma M \in A'$ **with** $\{\ \}$, (b) $\theta \in \mathcal{U}(\langle \Gamma, \theta_\Gamma^{id}, \{\ \} \rangle$, and (c) $A' = \theta A'$ (since the types in a valid signature contain no free variables).
- If $\mathcal{D}$ consists solely of an instance of the variable typing rule, then θM is a variable, and hence M is a variable $u \in \mathsf{dom}(\Psi)$ (since $M \notin \mathsf{dom}(\Gamma)$). Then the result follows, with A being the type of u in Ψ and $D = \{\ \}$.

[19] Note that we have again relied on type unicity, which does not hold for the calculus $\lambda_{\Pi\Sigma}$ of the next chapter. We will not deal with term inference for $\lambda_{\Pi\Sigma}$.

- Assume that $\mathcal{D}$ ends in an instance of the rule for typing abstractions. Then θM is an abstraction, and since we are assuming that $M \notin \mathsf{dom}(\Gamma)$, M must also be an abstraction, say $M = \lambda v{:}\hat{A}.\, \hat{M}$. Thus $\theta M = \lambda v{:}\theta\hat{A}.\, \theta\hat{M}$, $A' = \Pi v{:}\theta\hat{A}.\, B'$ for some B', and $\mathcal{D}$ contains subderivations of (1) $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{A} \in \mathsf{Type}$, and (2) $\Gamma' \oplus \theta\Psi \oplus v{:}\theta\hat{A} \vdash_\Sigma \theta\hat{M} \in B'$. By the induction hypothesis, there is a kind K, a type $\hat{B}'$, and disagreement sets $\hat{D}$ and $\hat{D}'$ such that (a) $\Gamma; \Psi \vdash_\Sigma \hat{A} \in K$ **with** $\hat{D}$ and $\Gamma; \Psi \oplus v{:}\hat{A} \vdash_\Sigma \hat{M} \in \hat{B}$ **with** $\hat{D}'$, (b) $\theta \in \mathcal{U}(\langle \Gamma, \theta^{id}_\Gamma, \hat{D}\rangle)$ and $\theta \in \mathcal{U}(\langle \Gamma, \theta^{id}_\Gamma, \hat{D}'\rangle)$, and (c) $B' =_\lambda \theta\hat{B}'$ and $\mathsf{Type} =_\lambda \theta K$ (*i.e.*, $K = \mathsf{Type}$). Thus we have (a) $\Gamma; \Psi \vdash_\Sigma \lambda v{:}\hat{A}.\, \hat{M} \in \Pi v{:}\hat{A}.\, \hat{B}$ **with** $\hat{D} \uplus \hat{D}'$ (by Definition 4.76), (b) $\theta \in \mathcal{U}(\langle \Gamma, \theta^{id}_\Gamma, \hat{D} \uplus \hat{D}'\rangle)$, and (c) $A' = \Pi v{:}\theta\hat{A}.\, B' =_\lambda \Pi v{:}\theta\hat{A}.\, \theta\hat{B} = \theta(\Pi v{:}\hat{A}.\, \hat{B})$. Thus, letting $D = \hat{D} \uplus \hat{D}'$ and $A = \Pi v{:}\hat{A}.\, \hat{B}$, the conclusion follows.

- Assume that $\mathcal{D}$ ends in an instance of the rule for typing applications. Then (since $M \notin \mathsf{dom}(\Gamma)$), M is an application $\hat{M}\,\hat{N}$, so θM is $(\theta\hat{M})\,(\theta\hat{N})$, where, for some v, $\hat{A}'$, $\hat{B}'$, $\mathcal{D}$ contains subderivations of $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{M} \in \Pi v{:}\hat{A}'.\, \hat{B}'$ and $\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta\hat{N} \in \hat{A}'$, where $A' = [\,\theta\hat{N}/v\,]\hat{B}'$. By the induction hypothesis, there are types $\hat{C}$ and $\hat{A}$, and disagreement sets $\hat{D}$ and $\hat{D}'$ such that (a) $\Gamma; \Psi \vdash_\Sigma \hat{M} \in \hat{C}$ **with** $\hat{D}$ and $\Gamma; \Psi \vdash_\Sigma \hat{N} \in \hat{A}$ **with** $\hat{D}'$, (b) $\theta \in \mathcal{U}(\langle \Gamma, \theta^{id}_\Gamma, \hat{D}\rangle)$ and $\theta \in \mathcal{U}(\langle \Gamma, \theta^{id}_\Gamma, \hat{D}'\rangle)$, and (c) $\Pi v{:}\hat{A}'.\, \hat{B}' =_\lambda \theta\hat{C}$ and $\hat{A}' =_\lambda \theta\hat{A}$. Then $\hat{C}$ $\mathrm{wh}_\beta{}^*$ $\Pi v{:}\hat{A}''.\, \hat{B}$ for some $\hat{A}''$ and $\hat{B}$, where $\theta\hat{A}'' =_\lambda \hat{A}'$ and $\theta\hat{B} =_\lambda \hat{B}'$. Therefore, we have $\Gamma; \Psi \vdash_\Sigma \hat{M}\,\hat{N} \in [\,\hat{N}/v\,]\hat{B}$ **with** $\{\langle \Psi\,,\, \hat{A}'',\, \hat{A}\rangle\} \uplus \hat{D} \uplus \hat{D}'$, and so the first condition follows, with $D = \{\langle \Psi\,,\, \hat{A}'',\, \hat{A}\rangle\} \uplus \hat{D} \uplus \hat{D}'$ and $A = [\,\hat{N}/v\,]\hat{B}$. The second condition follows since θ unifies $\hat{D}$ and $\hat{D}'$ by our induction hypothesis, and because $\theta\hat{A}'' =_\lambda \hat{A}' =_\lambda \theta\hat{A}$. The third condition, $A' =_\lambda \theta A$, holds because $\hat{B}' =_\lambda \theta\hat{B}$, and so $A' = [\,\theta\hat{N}/v\,]\hat{B}' =_\lambda [\,\theta\hat{N}/v\,](\theta\hat{B}) = \theta([\,\hat{N}/v\,]\hat{B}) = \theta A$.

- Finally (for terms), assume that $\mathcal{D}$ ends in an instance of the conversion typing rule, *i.e.*, $\mathcal{D}$ has the form

$$\frac{\begin{array}{ccc}\vdots & \vdots & \\ \Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in A'' & \Gamma' \oplus \theta\Psi \vdash_\Sigma A' \in \mathsf{Type} & A'' =_\lambda A'\end{array}}{\Gamma' \oplus \theta\Psi \vdash_\Sigma \theta M \in A'}$$

  By the induction hypothesis, there are A and D such that (a) $\Gamma; \Psi \vdash_\Sigma M \in A$ **with** D, (b) $\theta \in \mathcal{U}(\langle \Gamma, \theta^{id}_\Gamma, D\rangle)$, and (c) $A'' =_\lambda \theta A$. However, since $A'' =_\lambda A'$, we have $A' =_\lambda \theta A$, so the result follows with the same choice of A and D.

□

Given a term M in a context Γ we do type-checking/term inference as follows. If there is no A, D such that $\Gamma; [\,] \vdash_\Sigma M \in A$ **with** D, then, by the proposition, M has no well-typed instance, so we indicate failure. Otherwise, for such an A, D, let $\mathcal{Q}$ be a μCSP of $\langle \Gamma, \theta^{id}_\Gamma, D\rangle$. If $\mathcal{Q}$ is empty, then M has no well-typed instance. Otherwise, for each

$\langle \Gamma', \theta', D' \rangle \in \mathcal{Q}$, we return the instantiated term $\theta' M$ and the instantiated type $\theta' A$, together with the "constraint" D'. (Depending on the application, if $\mathcal{Q}$ has more than one element, and/or if D' is nonempty for some $\langle \Gamma', \theta', D' \rangle \in \mathcal{Q}$, it may be appropriate to request a user to provide a more constrained term.)

If on the other hand we have two terms M and M' to be unified in a context Γ, we can proceed as follows: If there is no A, D such that $\Gamma; [\,] \vdash_\Sigma M \in A$ **with** D or there is no A', D' such that $\Gamma; [\,] \vdash_\Sigma M' \in A'$ **with** D', then indicate a typing error. Otherwise, for such an A, D and A', D', construct a μCSP of $\langle \Gamma, \theta^{id}_\Gamma, \{\langle [\,],\ M,\ M' \rangle\} \uplus D \uplus D' \uplus \{\langle [\,],\ A,\ A' \rangle\} \rangle$.

In a practical implementation, it is better to exhaustively apply at least the non-branching transformations (4.1 through 4.3) to the type disagreement pairs produced in this process, rather than having completely separate collection and pre-unification phases as described above.

Chapter 5

Products

In this chapter, we extend the pre-unification algorithm developed in the previous chapter to the calculus "$\lambda_{\Pi\Sigma}$", which is λ_Π enriched with a dependent version of Cartesian product types, often called "strong sum types", or simply "Σ types".[1] The new algorithm then follows naturally from an analysis of weak head normal form terms and types, as guided by our development of HOU_Π. Finally, we present a commonly used notational variation that will make examples easier to read.

Many of the definitions and propositions carry over from Chapters 2 and 4 to this calculus. We will point out the extensions needed. In general, the degree of detail presented in this chapter is lower than the preceding one.

5.1 The Language Extension

Rather than stating the entire calculus, we will only state the new language constructs, typing rules and conversion rules. The new terms of $\lambda_{\Pi\Sigma}$ are for the construction and decomposition of pairs:

$$\begin{array}{rcl} M & ::= & M, N \\ & | & \mathsf{fst}\, M \\ & | & \mathsf{snd}\, M \end{array}$$

where "," associates to the right and binds less tightly than application.

Just as the type of the result of an application can depend on the value of the argument, the type of the second element of a pair can depend on the value of the first element. The new types of $\lambda_{\Pi\Sigma}$ are for these dependent pairs:

$$A \quad ::= \quad \Sigma v{:}A.\, B$$

[1] This use of Σ is not to be confused with the use of Σ for signatures.

We will often use the abbreviation "$A \times B$" for $\Pi v{:}A.\, B$ when v is not free in B, and then "$\times$" associates to the right and binds less tightly than "$\rightarrow$".

All of the typing rules from λ_Π carry over to $\lambda_{\Pi\Sigma}$. The new typing rules are

Types

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma \oplus v{:}A \vdash_\Sigma B \in \mathsf{Type}}{\Gamma \vdash_\Sigma \Sigma v{:}A.\, B \in \mathsf{Type}}$$

Terms

$$\frac{\Gamma \vdash_\Sigma M \in A \qquad \Gamma \vdash_\Sigma N \in [\, M/v \,]B}{\Gamma \vdash_\Sigma M, N \in \Sigma v{:}A.\, B}$$

$$\frac{\Gamma \vdash_\Sigma M \in \Sigma v{:}A.\, B}{\Gamma \vdash_\Sigma \mathsf{fst}\, M \in A}$$

$$\frac{\Gamma \vdash_\Sigma M \in \Sigma v{:}A.\, B}{\Gamma \vdash_\Sigma \mathsf{snd}\, M \in [\, \mathsf{fst}\, M/v \,]B}$$

An interesting feature of $\lambda_{\Pi\Sigma}$ not occurring in λ_Π is that well-typed terms do not have unique types (modulo $=_\lambda$), as illustrated in the following:

Example 5.1 *Let our signature Σ include*

$$\langle\, \mathsf{o}{:}\mathsf{Type}\ ,\ \vdash{:}\mathsf{o}{\rightarrow}\mathsf{Type}\ ,\ \top{:}\mathsf{o}\ ,\ \mathsf{I}_\top{:}\vdash\ \top \,\rangle$$

Then in Σ and the empty context, the term $(\top, \mathsf{I}_\top)$ has both types $\Sigma p{:}\mathsf{o}.\ \vdash p$ and $\mathsf{o} \times (\vdash\ \top)$.

5.2 Substitution

We need to extend Definition 2.7 of $\overline{\theta}$ given for λ_Π:

Definition 5.2 *Given variable sets V and V', and a substitution $\theta \in \Theta_\Gamma$, let $\overline{\theta}$ be the function from $\lambda^V_{\Pi\Sigma}$ to $\lambda^{V'}_{\Pi\Sigma}$ satisfying the properties listed in Definition 2.7, together with the following. First for terms,*

$$\begin{aligned} \overline{\theta}(M, N) &= (\overline{\theta}M, \overline{\theta}N) \\ \overline{\theta}(\mathsf{fst}\, M) &= \mathsf{fst}\, (\overline{\theta}M) \\ \overline{\theta}(\mathsf{snd}\, M) &= \mathsf{snd}\, (\overline{\theta}M) \end{aligned}$$

and then for types,

$$\overline{\theta}(\Sigma u{:}A.\, B) = \Sigma u{:}\overline{\theta}A.\, \overline{\theta^{+u}}B \qquad \text{if } u \notin V \cup V'$$

The properties of substitution proved in that section carry over directly to $\lambda_{\Pi\Sigma}$. We will continue to write "θ" in place of "$\overline{\theta}$", "$\theta' \circ \theta$" in place of "$\theta' * \theta$", and "θ", in place of θ^{+u}, where the u is clear from the context.

5.3 Conversion

In order for the η rules specified in Definition 2.16 to be meaningful in this calculus, the meaning of $\mathcal{F}(U)$ given in Definition 2.1 is extended to our new language constructs in the obvious way.

The new reduction relations are

$$\begin{array}{rcl} \mathsf{fst}\,(M,N) & \pi_1 & M \\ \mathsf{snd}\,(M,N) & \pi_2 & N \\ (\mathsf{fst}\,M, \mathsf{snd}\,M) & \pi & M \end{array}$$

The last rule is often referred to as "surjectivity".

Definition 5.3 *Given a binary relation ρ on $\lambda_{\Pi\Sigma}$, the relation $\to_\rho$ extends Definition 2.18, with the following additional cases, first for types,*

$$\frac{A \to_\rho A'}{\Sigma v{:}A.\ B \to_\rho \Sigma v{:}A'.\ B}$$

$$\frac{B \to_\rho B'}{\Sigma v{:}A.\ B \to_\rho \Sigma v{:}A.\ B'}$$

and then for terms

$$\frac{M \to_\rho M'}{(M,N) \to_\rho (M',N)}$$

$$\frac{N \to_\rho N'}{(M,N) \to_\rho (M,N')}$$

$$\frac{M \to_\rho M'}{\mathsf{fst}\,M \to_\rho \mathsf{fst}\,M'}$$

$$\frac{M \to_\rho M'}{\mathsf{snd}\,M \to_\rho \mathsf{snd}\,M'}$$

The relations $\to_\rho^*$ and $\leftrightarrow_\rho^*$ are again the reflexitive transitive closure, and the equivalence closure, respectively, of $\to_\rho$.

Then we extend our notion of convertibility:

Definition 5.4 *The convertibility relation $=_\lambda$ is $\leftrightarrow^*_{\beta\eta\pi_1\pi_2\pi}$.*

The well-typed terms in the analogous extension "$\lambda_{\to\times}$" of $\lambda_\to$ have the important normalization and Church-Rosser properties [62, 73]. Again, we will assume that the proof can be carried through to $\lambda_{\Pi\Sigma}$, and again, we could eliminate this assumption by redefining convertibility as discussed in Section 2.3.

We will also need to extend Proposition 2.22, which states the substitutivity of β and η:

Proposition 5.5 *The reduction relations π_1, π_2, and π are substitutive.*

Proof: Simple consequence of the definition of $\overline{\theta}$. □

It is also simple to extend the proof of Proposition 2.25 to show that for substitutive relations ρ, the relations $\to_\rho$, $\to_\rho^*$, and $\leftrightarrow_\rho^*$ are substitutive.

Finally, we must extend Proposition 2.36, which states that β and η preserve typing, to the new reduction relations π_1 and π_2.

Proposition 5.6 *The reduction relations π_1 and π_2 preserve typing.*

Proof: Of the two, π_2 is the trickier. Let $\mathcal{D}$ be a derivation of $\Gamma \vdash_\Sigma \mathsf{snd}\,(M, N) \in C$ for some type C. Then $\mathcal{D}$ ends in an instance of the typing rule for snd, followed by zero or more type conversions. Thus for some v, A, B, (a) $\mathcal{D}$ contains a subderivation $\mathcal{D}_1$ of $\Gamma \vdash_\Sigma (M, N) \in \Sigma v{:}A.\, B$, and (b) $C =_\lambda [\,\mathsf{fst}\,(M, N)/v\,]B$ and thus $C =_\lambda [\,M/v\,]B$. Also, $\mathcal{D}_1$ ends in an instance of the pair typing rule, followed by zero or more type conversions. Thus for some v', A', B', (a) $\mathcal{D}_1$ contains a subderivation of $\Gamma \vdash_\Sigma N \in [\,M/v'\,]B'$, and (b) $\Sigma v{:}A.\, B =_\lambda \Sigma v'{:}A'.\, B'$. By consideration of normal forms then, $A' =_\lambda A$ and $B' =_\lambda [\,v'/v\,]B$. Thus, $[\,M/v'\,]B' =_\lambda [\,M/v'\,]([\,v'/v\,]B) = [\,M/v\,]B =_\lambda C$, so $\Gamma \vdash_\Sigma N \in C$ by the conversion typing rule. □

5.4 Normal Forms

We need new normal forms to accomodate the new reduction rules. Recall that in HOU_Π we used the β rule to weakly head normalize terms and types. In $\mathrm{HOU}_{\Pi\Sigma}$ we will want to use the π_1 and π_2 rules also:

Definition 5.7 *The "weak head reduction" relation* $\mathrm{wh}_{\beta\pi_1\pi_2}$ *extends* wh_β *(Definition 4.1) as follows. In place of the first rule of Definition 4.1, we have*

$$\frac{U \beta\pi_1\pi_2 U'}{U\ \mathrm{wh}_{\beta\pi_1\pi_2}\ U'}$$

Then for types,

$$\frac{A\ \mathrm{wh}_{\beta\pi_1\pi_2}\ A'}{A\,M\ \mathrm{wh}_{\beta\pi_1\pi_2}\ A'\,M}$$

and for terms,

$$\frac{M\ \mathrm{wh}_{\beta\pi_1\pi_2}\ M'}{M\,N\ \mathrm{wh}_{\beta\pi_1\pi_2}\ M'\,N}$$

$$\frac{M\ \mathrm{wh}_{\beta\pi_1\pi_2}\ M'}{\mathsf{fst}\,M\ \mathrm{wh}_{\beta\pi_1\pi_2}\ \mathsf{fst}\,M'}$$

$$\frac{M\ \mathrm{wh}_{\beta\pi_1\pi_2}\ M'}{\mathsf{snd}\,M\ \mathrm{wh}_{\beta\pi_1\pi_2}\ \mathsf{snd}\,M'}$$

Proposition 5.8 *If* $U\ \mathrm{wh}_{\beta\pi_1\pi_2}\ V$ *then* $U \to_{\beta\pi_1\pi_2} V$.

Proof: A simple induction of the derivation of $U\ \mathrm{wh}_{\beta\pi_1\pi_2}\ V$. □

Corollary 5.9 $\mathrm{wh}_{\beta\pi_1\pi_2}$ *preserves types.*

As in Proposition 4.5, it is easy to show

Proposition 5.10 $\mathrm{wh}_{\beta\pi_1\pi_2}$ *is substitutive.*

Then we have the new version of *WHNF* and *body*:

Definition 5.11 *A term or type* U *is in* $(\beta\pi_1\pi_2)$ *weak head normal form (WHNF) iff there is no* V *such that* $U\ \mathrm{wh}_{\beta\pi_1\pi_2}\ V$.

Example 5.12 *We have the following reductions*

$$\mathsf{fst}\,(\mathsf{snd}\,(x, (\lambda w{:}\mathsf{i}.\, g\,w\,w))\,y)\,z \quad \mathrm{wh}_{\beta\pi_1\pi_2} \quad \mathsf{fst}\,((\lambda w{:}\mathsf{i}.\, g\,w\,w)\,y)\,z$$

$$\mathsf{fst}\,((\lambda w{:}\mathsf{i}.\, g\,w\,w)\,y)\,z \quad \mathrm{wh}_{\beta\pi_1\pi_2} \quad \mathsf{fst}\,(g\,y\,y)\,z$$

and the final term is in WHNF.

Definition 5.13 *A* body *is a (possibly ill-typed) WHNF term or type that is neither an abstraction nor pair.*

Similarly to Proposition 4.8, we will need the following.

Proposition 5.14 *A well-typed term body is either*

- *a variable,*
- *a constant,*
- $M\,N$ *for a well-typed body* M*, or*
- $\mathsf{fst}\,M$ *or* $\mathsf{snd}\,M$ *for a well-typed body* M*.*

and a well-typed type body is either

- *a type constant,*
- $\Pi v{:}A.\,B$ *or* $\Sigma v{:}A.\,B$*, or*
- $A\,M$ *for a well-typed body* A *that is neither a* Π *nor a* Σ *type,*

Proof: Follows easily from Definitions 5.11 and 5.7. The restriction of well-typedness ensures that, (a) in $M\,N$, M is not a pair, (b) in $\mathsf{fst}\,M$ or $\mathsf{snd}\,M$, M is not an abstraction, and (c) in $A\,M$, A is not a Π or Σ type. □

5.5 Some Useful Properties of Convertibility

As in Chapter 4, we present some methods for decomposing questions of convertibility of disagreement pairs into questions of simultaneous convertibility of "simpler" sets of constructed disagreement pairs.

Definition 5.15 *Extend the notion of the size of a $\beta\pi_1\pi_2$ normal form term or type U (Definition 4.16) by the following. First for types,*

$$\begin{array}{rcl} \mathsf{size}(\mathsf{fst}\, M) & = & 1 + \mathsf{size}(M) \\ \mathsf{size}(\mathsf{snd}\, M) & = & 1 + \mathsf{size}(M) \\ \mathsf{size}((M, N)) & = & \mathsf{size}(M) + \mathsf{size}(N) + 1 \end{array}$$

Then for types,

$$\mathsf{size}(\Sigma v{:}A.\, B) \quad = \quad \mathsf{size}(A) + \mathsf{size}(B) + 1$$

The definition is extended to well-typed terms and types (not necessarily in normal form), and to disagreement pairs, as in Definition 4.16.

Note that well-typedness is crucial for the notion of size to be well defined.

The meaning of "$P \lhd D$" is as given in Definition 4.18, given this new definition of size.

5.5.1 Weak Head Redices

This case is handled in analogy to Section 4.2.1, using $\mathrm{wh}_{\beta\pi_1\pi_2}$ instead of wh_β:

Definition 5.16 *The decomposition method $\rightsquigarrow_{\mathrm{wh}}$ is given by*

$$\frac{U\ \mathrm{wh}_{\beta\pi_1\pi_2}\ V}{\langle \Psi\ ,\ U,\ U' \rangle \rightsquigarrow_{\mathrm{wh}} \langle\!\langle\ \langle \Psi\ ,\ V,\ U' \rangle\ \rangle\!\rangle}$$

$$\frac{U'\ \mathrm{wh}_{\beta\pi_1\pi_2}\ V'}{\langle \Psi\ ,\ U,\ U' \rangle \rightsquigarrow_{\mathrm{wh}} \langle\!\langle\ \langle \Psi\ ,\ U,\ V' \rangle\ \rangle\!\rangle}$$

Then we have

Proposition 5.17 *Let P be a well-typed disagreement pair and $\hat{D}$ a disagreement sequence such that $P \rightsquigarrow_{\mathrm{wh}} \hat{D}$. Then $P \lhd \hat{D}$.*

Proof: The argument goes exactly as with of Proposition 4.20, replacing β by $\beta\pi_1\pi_2$. □

5.5.2 Abstractions and pairs

The treatment of pair terms is conceptually similar to the treatment given to abstractions in Section 4.2.2, and so we treat them together. This is the only form of η or π conversion used by the algorithm.

Definition 5.18 *Let the syntactic decomposition relation "$\rightsquigarrow_{\Pi\Sigma}$" be defined by the union of the relation $\rightsquigarrow_{\Pi}$ (Definition 4.21) and the relation defined by the following rules:*

$$\langle \Psi\ ,\ (M, N),\ (M', N') \rangle \rightsquigarrow_{\Pi\Sigma} \langle\!\langle\ \langle \Psi\ ,\ M,\ M' \rangle\ ,\ \langle \Psi\ ,\ N,\ N' \rangle\ \rangle\!\rangle$$

$$\frac{M' \text{ is a body}}{\langle \Psi\ ,\ (M, N),\ M' \rangle \rightsquigarrow_{\Pi\Sigma} \langle\!\langle\ \langle \Psi\ ,\ M,\ \mathsf{fst}\, M' \rangle\ ,\ \langle \Psi\ ,\ N,\ \mathsf{snd}\, M' \rangle\ \rangle\!\rangle}$$

$$\frac{M \text{ is a body}}{\langle \Psi\ ,\ M,\ (M', N') \rangle \rightsquigarrow_{\Pi\Sigma} \langle\!\langle\ \langle \Psi\ ,\ \mathsf{fst}\, M,\ M' \rangle\ ,\ \langle \Psi\ ,\ \mathsf{snd}\, M,\ N' \rangle\ \rangle\!\rangle}$$

Proposition 5.19 *Let P be a well-typed disagreement pair and $\hat{D}$ a disagreement sequence such that $P \rightsquigarrow_{\Pi\Sigma} \hat{D}$. Then $P \lhd \hat{D}$.*

(Proof below.)

Example 5.20 *Let our signature $\Sigma = \langle\, \mathsf{i}{:}\mathsf{Type}\ ,\ \mathsf{a}{:}\mathsf{i}\ ,\ \mathsf{b}{:}\mathsf{i}\ ,\ \mathsf{c}{:}\mathsf{i} \times \mathsf{i}\, \rangle$, and our unification context $\Gamma = [\, x{:}\mathsf{i}\ ,\ y{:}\mathsf{i}\,]$. Let $P_1 = \langle [\ ]\ ,\ (x, \mathsf{a}),\ \mathsf{c} \rangle$. Then*

$$P_1 \quad \rightsquigarrow_{\Pi\Sigma} \quad \langle\!\langle\ \langle [\ ]\ ,\ x,\ \mathsf{fst}\, \mathsf{c} \rangle, \langle [\ ]\ ,\ \mathsf{a},\ \mathsf{snd}\, \mathsf{c} \rangle\ \rangle\!\rangle$$

Next, let $P_2 = \langle [\ ]\ ,\ (x, \mathsf{c}),\ (\mathsf{b}, y) \rangle$. Then

$$P_2 \quad \rightsquigarrow_{\Pi\Sigma} \quad \langle\!\langle\ \langle [\ ]\ ,\ x,\ \mathsf{b} \rangle, \langle [\ ]\ ,\ \mathsf{a},\ y \rangle\ \rangle\!\rangle$$

Proof of Proposition 5.19: The abstraction cases have already been proved in Proposition 4.22. We will prove the claim for the second pair case. Thus $P = \langle \Psi\ ,\ (M, N),\ M' \rangle$ and $\hat{D} = \langle\!\langle\ \langle \Psi\ ,\ M,\ \mathsf{fst}\, M' \rangle, \langle \Psi\ ,\ N,\ \mathsf{snd}\, M' \rangle\ \rangle\!\rangle$, where M' is a body. To prove that the first condition of Definition 4.18 of $\lhd$ is satisfied, first assume that $(M, N) =_\lambda M'$. Then $M =_\lambda \mathsf{fst}\, (M, N) =_\lambda \mathsf{fst}\, M'$ and $N =_\lambda \mathsf{snd}\, (M, N) =_\lambda \mathsf{snd}\, M'$. Next, assume that $M =_\lambda \mathsf{fst}\, M'$ and $N =_\lambda \mathsf{snd}\, M'$. Then $(M, N) =_\lambda (\mathsf{fst}\, M', \mathsf{snd}\, M') =_\lambda M'$.

For the second part, we must show that $P_1' = \langle \Psi\ ,\ M,\ \mathsf{fst}\, M' \rangle$ is well-typed, and that if $M =_\lambda \mathsf{fst}\, M'$ then $P_2' = \langle \Psi\ ,\ N,\ \mathsf{snd}\, M' \rangle$ is well-typed. Since P is well-typed, there must be some B such that (M, N) and M' both have type B in $\Gamma \oplus \Psi$. But then B must be convertible to the form $\Sigma v{:}\hat{A}.\ \hat{B}$, where (a) $\Gamma \oplus \Psi \vdash_\Sigma M \in \hat{A}$, and (b) $\Gamma \oplus \Psi \vdash_\Sigma N \in [\, M/v\,]\hat{B}$. But

also (a) $\Gamma \oplus \Psi \vdash_\Sigma \mathsf{fst}\, M' \in \hat{A}$, which shows that P_1' is well-typed, and (b) $\Gamma \oplus \Psi \vdash_\Sigma \mathsf{snd}\, M' \in [\,\mathsf{fst}\, M'/v\,]\hat{B}$, which shows that if $M =_\lambda \mathsf{fst}\, M'$ then P_2 is well-typed.

The size requirement follows since

$$\mathsf{size}(\langle \Psi\ ,\ (M, N),\ M' \rangle) \quad = \quad \mathsf{size}(M) + \mathsf{size}(N) + 1 + \mathsf{size}(M')$$

while

$$\mathsf{size}(\langle \Psi\ ,\ M,\ \mathsf{fst}\, M' \rangle) \quad = \quad \mathsf{size}(M) + \mathsf{size}(M') + 1$$

(recalling that $\mathsf{size}(N) > 0$ for all N), and similarly for $\langle \Psi\ ,\ N,\ \mathsf{snd}\, M' \rangle$.

The third pair decomposition case is analogous, and the first case is simpler. □

5.5.3 Bodies

To treat bodies, we will need a new version of "$\rightsquigarrow_{\mathrm{rr}}$":

Definition 5.21 *Given a disagreement pair P relating bodies, and a disagreement sequence $\hat{D}$, P* rigidly decomposes to *D, written "$P \rightsquigarrow_{\mathrm{rr}} \hat{D}$", according to the inference system of Definition 4.28, plus the following new inference rules. First for terms,*

$$\frac{\langle \Psi\ ,\ M,\ M' \rangle \rightsquigarrow_{\mathrm{rr}} \hat{D}}{\langle \Psi\ ,\ \mathsf{fst}\, M,\ \mathsf{fst}\, M' \rangle \rightsquigarrow_{\mathrm{rr}} \hat{D}}$$

$$\frac{\langle \Psi\ ,\ M,\ M' \rangle \rightsquigarrow_{\mathrm{rr}} \hat{D}}{\langle \Psi\ ,\ \mathsf{snd}\, M,\ \mathsf{snd}\, M' \rangle \rightsquigarrow_{\mathrm{rr}} \hat{D}}$$

and then for types

$$\langle \Psi\ ,\ \Sigma v{:}A.\, B,\ \Sigma v{:}A'.\, B' \rangle \rightsquigarrow_{\mathrm{rr}} \langle\!\langle\ \langle \Psi\ ,\ A,\ A' \rangle, \langle \Psi \oplus v : A\ ,\ B,\ B' \rangle\ \rangle\!\rangle$$

*We define "$U \approx U'$" and "*topeq(P)*" as before.*

Again we will use

Proposition 5.22 *Let P be a disagreement pair relating bodies. If* $\mathrm{eq}_\lambda(P)$*, then* topeq(P)*. Otherwise, let $P \rightsquigarrow_{\mathrm{rr}} \hat{D}$. Then $P \lhd \hat{D}$.*

Proof of Proposition 5.22: Similar to the proofs of Propositions 4.32 and 4.34. □

Example 5.23 *Let the disagreement pair*

$$P \quad = \quad \langle [\,] \,,\, \mathsf{snd}\,((\mathsf{fst}\,\mathsf{g})\,x\,y)\,z,\, \mathsf{snd}\,((\mathsf{fst}\,\mathsf{g})\,u\,v)\,w\rangle$$

Then

$$P \quad \leadsto_{\mathrm{rr}} \quad \langle\!\langle\, \langle [\,] \,,\, x,\, u\rangle \,,\, \langle [\,] \,,\, y,\, v\rangle \,,\, \langle [\,] \,,\, z,\, w\rangle \,\rangle\!\rangle$$

Example 5.24 *Changing the previous example, let*

$$P \quad = \quad \langle [\,] \,,\, \mathsf{snd}\,((\mathsf{fst}\,\mathsf{g})\,x\,y)\,z,\, \mathsf{fst}\,((\mathsf{snd}\,\mathsf{g})\,u\,v)\,w\rangle$$

Then $\neg\mathrm{topeq}(P)$.

5.6 The Transformations

The meaning of *solved form* carries over without change from Definition 4.46.

As in Section 4.4, the following fact will play an important role

Proposition 5.25 *Let* $Q = \langle \Gamma, \theta_0, D \uplus \{\, P \,\}\rangle$ *be acceptable,* $\hat{D}$ *be such that for any* $\theta \in \Theta_\Gamma$ *such that* θP *is well-typed,* $\theta P \lhd \theta\hat{D}$*, and* Q' *be* $\langle \Gamma, \theta_0, D \uplus \hat{D}\rangle$*. Then the transition* $Q \mapsto \{Q'\}$ *is valid.*

Proof: The proof of Proposition 4.47, which did not depend on the particulars of λ_Π, applies here. □

We will also need an extended notion of head:

Definition 5.26 *The* head *of a body, which is an atom or one of the symbols* Π *or* Σ*, is given by, first for terms,*

- $head(v) = v$,
- $head(\mathsf{c}) = \mathsf{c}$,
- $head(M\,N) = head(M)$,
- $head(\mathsf{fst}\,M) = head(\mathsf{snd}\,M) = head(M)$,

and then for types

- $head(\mathsf{c}) = \mathsf{c}$,
- $head(A\,M) = head(A)$,
- $head(\Pi v{:}A.\,B) = \Pi$,
- $head(\Sigma v{:}A.\,B) = \Sigma$,

Example 5.27 *The head of* $\mathsf{snd}\,((\mathsf{fst}\,f)\,x\,y)\,z$ *is* f.

The meanings of *flexible* and *rigid* are as in Definition 4.45 (extending *rigid* to cover the possibility of Σ as a head), given this new notion of head. Another change is to Definition 4.43 of weak validity of a context $\Gamma = [\,u_1{:}B_1, \ldots, u_n{:}B_n\,]$. We will require that each B_i is of the form $Qx_1{:}A_1.\ \cdots Qx_m{:}A_m.\ A_0$, where (a) each Q_i is Π or Σ, and (b) A_0 is atomic.

5.6.1 Redices

The treatment of weak head redices is the same as in HOU_Π, given the new definition of $\rightsquigarrow_{\mathrm{wh}}$:

Transformation 5.1 *Let* $Q = \langle \Gamma, \theta_0, D \uplus \{\,P\,\}\rangle$ *and let* $\hat{D}$ *be such that* $P \rightsquigarrow_{\mathrm{wh}} \hat{D}$. *Then make the transition*

$$Q \quad \mapsto \quad \{\langle \Gamma, \theta_0, D \uplus \hat{D}\rangle\}$$

Proposition 5.28 *Transformation 5.1 is valid.*

Proof: The argument goes as for Transformation 4.1. □

5.6.2 Abstractions and Pairs

We will treate these two together, extending the treatment given for HOU_{Π}.

Proposition 5.29 *Let P be a disagreement pair and $\hat{D}$ a disagreement sequence such that $P \leadsto_{\Pi\Sigma} \hat{D}$. Then for any $\theta \in \Theta_\Gamma$, $\theta P \leadsto_{\Pi\Sigma} \theta\hat{D}$, and in particular if θP is well-typed then $\theta P \lhd \theta\hat{D}$.*

Proof: We have already treated the abstraction cases in Proposition 4.50. To treat a disagreement pair involving a pair let $P = \langle \Psi, (M, N), M' \rangle$ and $\hat{D} = \langle\!\langle\, \langle \Psi, M, \mathsf{fst}\, M' \rangle, \langle \Psi, N, \mathsf{snd}\, M' \rangle \,\rangle\!\rangle$, and reason

$$\begin{array}{rcl}
\theta P & = & \theta\langle \Psi\ ,\ (M, N),\ M' \rangle \\
& = & \langle \theta\Psi\ ,\ \theta(M, N),\ \theta M' \rangle \\
& = & \langle \theta\Psi\ ,\ (\theta M, \theta N),\ \theta \rangle M' \\
& \leadsto_{\Pi\Sigma} & \langle\!\langle\, \langle \theta\Psi\ ,\ \theta M,\ \mathsf{fst}\,(\theta M') \rangle, \langle \theta\Psi\ ,\ \theta N,\ \mathsf{snd}\,(\theta M') \rangle \,\rangle\!\rangle \\
& = & \langle\!\langle\, \langle \theta\Psi\ ,\ \theta M,\ \theta(\mathsf{fst}\, M') \rangle, \langle \theta\Psi\ ,\ \theta N,\ \theta(\mathsf{snd}\, M') \rangle \,\rangle\!\rangle \\
& = & \theta\hat{D}
\end{array}$$

The final conclusion follows from Proposition 5.19.

The other pair decomposition cases are similar. □

Transformation 5.2 *Let $Q = \langle \Gamma, \theta_0, D \uplus \{\, P \,\} \rangle$, and let $\hat{D}$ be such that $P \leadsto_{\Pi\Sigma} \hat{D}$. Then make the transition*

$$Q \quad \mapsto \quad \{\langle \Gamma, \theta_0, D \uplus \hat{D} \rangle\}$$

Proposition 5.30 *Transformation 5.2 is valid.*

Proof: Immediate from Propositions 5.25 and 5.29. □

5.6.3 Rigid-rigid

The treatment of this case is exactly as in Section 4.4.4, given the extended meaning of "$\rightsquigarrow_{\text{rr}}$":

Proposition 5.31 *Let P be a rigid-rigid disagreement pair. If $\neg\text{topeq}(P)$, then P is nonunifiable. If $P \rightsquigarrow_{\text{rr}} \hat{D}$, then for all $\theta \in \Theta_\Gamma$, $\theta P \rightsquigarrow_{\text{rr}} \theta\hat{D}$, and in particular if θP is well-typed, then $\theta P \lhd \theta\hat{D}$, and hence $\theta P \lhd \theta\hat{D}$.*

Proof: Refer to proof of Proposition 4.53. The new cases (fst, snd, and Σ) present no new difficulties. □

Transformation 5.3 *Let $Q = \langle \Gamma, \theta_0, D \uplus \{ P \} \rangle$ be acceptable, where P is rigid-rigid. If $\neg\text{topeq}(P)$ then make the transition*

$$Q \quad \mapsto \quad \{ \}$$

Otherwise, let $P \rightsquigarrow_{\text{rr}} \hat{D}$, and make the transition

$$Q \quad \mapsto \quad \{\langle \Gamma, \theta_0, D \uplus \hat{D} \rangle\}$$

Proposition 5.32 *Transformation 5.3 is valid.*

Proof: Immediate from Propositions 5.25 and 5.31. □

5.6.4 Pair-producing Variables

The purpose of this case is just to simplify treatment of the flexible-rigid case, by eliminating certain types of variables. It is different from the other transformations, in that it is possible for a unification problem to be in solved form even when this transformation applies.

Definition 5.33 *A* pair-producing type *is one that is convertible to the form*

$$\Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ \Sigma u{:}\hat{A}.\ \hat{B}$$

where m is possibly zero.

Consider a unification problem $Q = \langle \Gamma, \theta_0, D \rangle$, where Γ contains a variable of pair-producing type. We will treat this case using a simplification of the ideas in Section 4.4.5, introducing an approximating substitution, having the effect of replacing a unification variable of pair-producing type by two new variables of simpler type.

Definition 5.34 *Given a unification context $\Gamma = \Gamma_1 \oplus v{:}A \oplus \Gamma_2$, where A is a pair-producing type convertible to $\Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ \Sigma u{:}\hat{A}.\ \hat{B}$, let*

$$N_\pi \quad = \quad \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ (v_1\, x_1 \cdots x_m, v_2\, x_1 \cdots x_m)$$

where $\{\, v_1, v_2 \,\} \cap (\mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)) = \{\ \}$. Then define the approximating substitution

$$\theta_\pi \quad = \quad [\, N_\pi / v \,]_{\mathsf{dom}(\Gamma)}^{\mathsf{dom}(\Gamma_1) \cup \{\, v_1, v_2 \,\} \cup \mathsf{dom}(\Gamma_2)}$$

For the types of the new variables v_1, v_2, let

$$\begin{array}{rcl} C_1 & = & \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ \hat{A} \\ C_2 & = & \Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ [\, (v_1\, x_1 \cdots x_m)/u \,]\hat{B} \end{array}$$

and define the new context

$$\Gamma_\pi \quad = \quad \Gamma_1 \oplus [\, v_1{:}C_1\ ,\ v_2{:}C_2 \,] \oplus (\theta_\pi \Gamma_2)$$

(Note that these are all well-defined up to convertibility.)

Similarly to Propositions 4.59 and 4.64, we will need the following:

Proposition 5.35 *Let $Q = \langle \Gamma, \theta_0, D \rangle$, where Γ is as above. For any $\theta \in \Theta_\Gamma$ (and in particular for any such θ that unifies D), θ is an instance of θ_π, i.e., there is a θ' such that $\theta =_\lambda \theta' \circ \theta_\pi$. Furthermore, we can choose such a θ' such that $\theta \gg \theta'$ (see Definition 4.63).*

Proof: The key observation is that for any $\theta \in \Theta_\Gamma$, θv is convertible to a term of the form

$$\lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ (N_1, N_2)$$

and that this is equivalent to saying that $\theta =_\lambda \theta' \circ \theta_\pi$, where

$$\begin{array}{rcl} \theta' v_1 & = & \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ N_1 \\ \theta' v_2 & = & \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ N_2 \end{array}$$

and θ' agrees with θ on $\mathsf{dom}(\Gamma) - \{v\}$. In particular, this is true for any unifier θ of D. $\Box$

Now we can state our transformation:

Transformation 5.4 *Let $Q = \langle \Gamma, \theta_0, D \rangle$ be a unification problem, with Γ and θ_π as above. Then make the transition*

$$Q \quad \mapsto \quad \{\langle \Gamma_\pi, \theta_\pi \circ \theta_0, \theta_\pi D \rangle\}$$

Example 5.36 *Let the signature be*

$$\Sigma \quad = \quad \langle\, \mathsf{a}\text{:}\mathsf{Type}\ ,\ \mathsf{b}\text{:}\mathsf{Type}\ ,\ \mathsf{c}\text{:}\mathsf{Type}\ ,\ \mathsf{d}\text{:}\mathsf{Type}\ ,\ \mathsf{g}\text{:}\mathsf{c}\to\mathsf{a}\to\mathsf{d}\,\rangle$$

and the unification context be

$$\Gamma \quad = \quad [\, f\text{:}\mathsf{a}\to\mathsf{b} \times (\mathsf{c}\to\mathsf{d})\,]$$

Let our unification problem be

$$Q \quad = \quad \langle \Gamma, \theta_\Gamma^{id}, \{\langle [\, x\text{:}\mathsf{a}\ ,\ y\text{:}\mathsf{c}\,]\ ,\ \mathsf{snd}\,(f\,x)\,y,\ \mathsf{g}\,y\,x\rangle\}\rangle$$

Applying Transformation 5.4 gives $\{\langle \Gamma_\pi, \theta_\pi, \{P'\}\rangle\}$, where

$$\begin{array}{rcl} \Gamma_\pi & = & [\, f_1\text{:}\mathsf{a}\to\mathsf{b}\ ,\ f_2\text{:}\mathsf{a}\to\mathsf{c}\to\mathsf{d}\,] \\ \theta_\pi & = & [\, (\lambda z\text{:}\mathsf{a}.\ f_1\,z, f_2\,z)/f\,] \\ P' & = & \langle [\, x\text{:}\mathsf{a}\ ,\ y\text{:}\mathsf{c}\,]\ ,\ \mathsf{snd}\,((\lambda z\text{:}\mathsf{a}.\ f_1\,z, f_2\,z)\,x)\,y,\ \mathsf{g}\,y\,x\rangle \end{array}$$

Then applying Transformation 5.1 for weak head reduction gives the disagreement pair

$$\langle [\, x\text{:}\mathsf{a}\ ,\ y\text{:}\mathsf{c}\,]\ ,\ f_2\,x\,y,\ \mathsf{g}\,y\,x\rangle$$

The proof of validity of this transformation is similar to but simpler than the flexible-rigid transformation proof in Section 4.4.5. Recall the definition of $\mathcal{U}(Q)$ from Definition 3.5:

$$\{\, \hat{\theta} \ \mid\ \exists\theta.\, \hat{\theta} =_\lambda \theta \circ \theta_0 \wedge \theta \in \Theta_\Gamma \wedge \theta \text{ unifies } D\,\}$$

From Proposition 5.35, we know that the condition $\theta \in \Theta_\Gamma$ implies the additional condition

$$\exists\theta'.\, \theta =_\lambda \theta' \circ \theta_\pi$$

As before, we can thus add this condition without changing the meaning of the set expression. Then, by steps similar to those on page 66, we get the equivalent form of $\mathcal{U}(Q)$:

$$\{\, \hat{\theta} \ \mid\ \exists\theta'.\, \hat{\theta} =_\lambda \theta' \circ (\theta_\pi \circ \theta_0) \wedge \theta' \circ \theta_\pi \in \Theta_\Gamma \wedge \theta' \text{ unifies } \theta_\pi D\,\}$$

Again, this is almost in the right form to be collapsed into $\langle \Gamma_\pi, \theta_\pi \circ \theta_0, \theta_\pi D\rangle$. The only problem is the condition $\theta' \circ \theta_\pi \in \Theta_\Gamma$, instead of $\theta' \in \Theta_{\Gamma_\pi}$. Reasoning as before, we can see that $\theta' \circ \theta_\pi \in \Theta_\Gamma$ iff (a) for each $u\text{:}\hat{B} \in \Gamma_\pi$, excepting $u \in \{\, v_1, v_2\,\}$, we have $\Gamma' \vdash_\Sigma \theta' u \in \theta'\hat{B}$, and (b) $\Gamma' \vdash_\Sigma \theta' N_\pi \in \theta' A$. The following lemma shows that this is exactly what we need. (Note that, unlike Proposition 4.67, we do *not* also need to add a type pair.)

Lemma 5.37 *Given θ_π, C_1, C_2, ... as above, we have $\Gamma' \vdash_\Sigma \theta' N_\pi \in \theta' A$ iff $\Gamma' \vdash_\Sigma \theta' v_1 \in \theta' C_1$ and $\Gamma' \vdash_\Sigma \theta' v_2 \in \theta' C_2$.*

Proof: Note that

$$\theta N_\pi \quad = \quad \lambda x_1{:}\theta' A_1.\ \cdots \lambda x_m{:}\theta' A_m.\ ((\theta' v_1)\, x_1 \cdots x_m, (\theta' v_2)\, x_1 \cdots x_m)$$

and

$$\theta' A \quad = \quad \Pi x_1{:}\theta' A_1.\ \cdots \Pi x_m{:}\theta' A_m.\ \Sigma u{:}\theta' \hat{A}.\ \theta' \hat{B}$$

thus $\Gamma' \vdash_\Sigma \theta' N_\pi \in \theta' A$ iff

$$\Gamma' \oplus [\ x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m\ ] \vdash_\Sigma (\theta' v_1)\, x_1 \cdots x_m \in \theta' \hat{A}$$

and

$$\Gamma' \oplus [\ x_1{:}\theta' A_1, \ldots, x_m{:}\theta' A_m\ ] \vdash_\Sigma (\theta' v_2)\, x_1 \cdots x_m \in [\ (\theta' v_1)\, x_1 \cdots x_m / u\ ](\theta' \hat{B})$$

By a simple inductive argument, the latter type is the same as $\theta'([\ v_1\, x_1 \cdots x_m / v\ ]\hat{B})$. Then using the abstraction rule and η reduction m times, this is equivalent to

$$\Gamma' \vdash_\Sigma \theta' v_1 \in \Pi x_1{:}\theta' A_1.\ \cdots \Pi x_m{:}\theta' A_m.\ \theta' \hat{A}$$

and

$$\Gamma' \vdash_\Sigma \theta' v_2 \in \Pi x_1{:}\theta' A_1.\ \cdots \Pi x_m{:}\theta' A_m.\ \theta'([\ v_1\, x_1 \cdots x_m / \hat{A}\ ]\hat{B})$$

but given the definitions of C_1 and C_2, this is just $\Gamma' \vdash_\Sigma \theta' v_1 \in \theta C_1$ and $\Gamma' \vdash_\Sigma \theta' v_2 \in \theta C_2$. □

Now we are ready to show the validity of our transformation.

Proposition 5.38 *Transformation 5.4 is valid.*

Proof: The reasoning is much the same as in the flexible-rigid case for HOU_Π, but simpler since there is no branching. Correctness then follows from Proposition 5.35 and Lemma 5.37. Acceptability is shown as for Transformation 4.4. Finally, minimality is vacuous. □

5.6.5 Flexible-rigid

Because of the previous transformation, we may specialize the flexible-rigid case to handle only flexible heads not of pair-producing type. As in Section 4.4.5, assume that our acceptable unification problem Q is $\langle \Gamma, \theta_0, D \rangle$, where D contains a flexible-rigid disagreement pair $P = \langle \Psi\ ,\ M,\ M' \rangle$ or a rigid-flexible disagreement pair $P = \langle \Psi\ ,\ M',\ M \rangle$. Let the unification variable v be the head of M. Let $\Gamma = \Gamma_1 \oplus v{:}\, A \oplus \Gamma_2$, where, by weak validity of Γ, A is convertible to the form $\Pi x_1{:}\, A_1.\ \cdots \Pi x_m{:}\, A_m.\ A_0$, for an atomic A_0. (Recall we are assuming A not to be a pair producing type.) Then, for every substitution $\theta \in \Theta_\Gamma$, θv is convertible to a term of the form

$$\theta v =_\lambda \lambda x_1{:}\, \hat{A}_1.\ \cdots \lambda x_m{:}\, \hat{A}_m.\ N$$

for some body N. As in Section 4.4.5, our analysis is based on examining the possible top level structure of N. In $\lambda_{\Pi\Sigma}$, we cannot simply express N in the form $b\, N_1 \cdots N_n$ because there may also be occurrences of fst and snd involved (as in Example 5.12). Let $\Gamma_x = [\, x_1{:}\, A_1, \ldots, x_m{:}\, A_m \,]$ and assume that θ unifies P. With respect to Γ_x, N is either rigid or flexible. If rigid (*i.e.*, $\mathrm{head}(N) \notin \{\, x_1, \ldots, x_m \,\}$), then

$$\begin{array}{rcl} \theta M & =_\lambda & (\lambda x_1{:}\, \hat{A}_1.\ \cdots \lambda x_m{:}\, \hat{A}_m.\ N)\,(\theta M_1) \cdots (\theta M_m) \\ & =_\lambda & [\, \theta M_1/x_1, \ldots, \theta M_m/x_m \,] N \approx N \end{array}$$

But also

$$\theta M =_\lambda \theta M' \approx M'$$

by Proposition 4.52, since M' is rigid, so $N \approx M'$. (The other possibility is that $\mathrm{head}(N) = x_i$ for some $1 \le i \le m$.) We will now show how to construct an approximating *imitation* substitution reflecting the restriction that $N \approx M'$. (An example follows.)

Definition 5.39 *Given Γ_1, Γ_2 and $\Gamma_x = [\, x_1{:}\, A_1, \ldots, x_m{:}\, A_m \,]$, define the relation "$\hat{M} \Longrightarrow \Gamma_{\mathrm{new}}\ ;\ \tilde{M} \in B$" as follows. (In practice, $\hat{M}$ will be given, and we will construct the "template" $\tilde{M}$ and its type B, in which some subterms of $\hat{M}$ have been replaced by placeholders of the form $(w\, x_1 \cdots x_m)$. The constructed context Γ_{new} accumulates typings for these new w variables.)*

$$\frac{\Gamma_x \vdash_\Sigma v \in B}{v \Longrightarrow [\,]\ ;\ v \in B}$$

$$\frac{[\,] \vdash_\Sigma \mathsf{c} \in B}{\mathsf{c} \Longrightarrow [\,]\ ;\ \mathsf{c} \in B}$$

$$\frac{\hat{M} \Longrightarrow \Gamma_{\mathrm{new}}\ ;\ \tilde{M} \in \hat{A} \qquad \hat{A}\, \mathrm{wh}_{\beta\pi_1\pi_2}\, \Pi v{:}\, A.\ B \qquad w \notin \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_{\mathrm{new}}) \cup \mathsf{dom}(\Gamma_2)}{\hat{M}\, \hat{N} \Longrightarrow \Gamma_{\mathrm{new}} \oplus w{:}\, \Pi x_1{:}\, A_1.\ \cdots \Pi x_m{:}\, A_m.\ A\ ;\ \tilde{M}\, (w\, x_1 \cdots x_m) \in [\, (w\, x_1 \cdots x_m)/v \,] B}$$

$$\frac{\hat{M} \Longrightarrow \Gamma_{\mathrm{new}}\ ;\ \tilde{M} \in \Sigma v{:}\, A.\ B}{\mathsf{fst}\, \hat{M} \Longrightarrow \Gamma_{\mathrm{new}}\ ;\ \mathsf{fst}\, \tilde{M} \in A}$$

$$\frac{\hat{M} \Longrightarrow \Gamma_{\mathrm{new}}\ ;\ \tilde{M} \in \Sigma v{:}\, A.\ B}{\mathsf{snd}\, \hat{M} \Longrightarrow \Gamma_{\mathrm{new}}\ ;\ \mathsf{snd}\, \tilde{M} \in [\, \mathsf{fst}\, \tilde{M}/v \,] \hat{B}}$$

Given $\Gamma_1, \Gamma_2, \Gamma_x$, and $\hat{M}$, this definition suggests a simple recursive procedure for constructing $\tilde{M}, B$, and Γ_{new} such that $\hat{M} \Longrightarrow \Gamma_{\text{new}}\ ;\ \tilde{M} \in B$. Moreover, $\tilde{M}, B$, and Γ_{new} are uniquely determined, modulo type convertibility and the choice of new variable names in Γ_{new}.

Example 5.40 *Let our signature include* $\mathsf{q}\colon (\mathsf{a}\to\mathsf{b}\to\mathsf{c} \times (\mathsf{d}\to\mathsf{e})) \times \mathsf{f}$. *Let* $\Gamma_x = [\, y\colon\mathsf{i}\ ,\ z\colon\mathsf{o}\,]$ *and* $\hat{M} = \mathsf{snd}\,((\mathsf{fst}\,\mathsf{q})\,M_1\,M_2)\,M_3$, *for some terms* M_1, M_2, M_3. *Then*

$$\hat{M} \Longrightarrow \Gamma_{\text{new}}\ ;\ \mathsf{snd}\,((\mathsf{fst}\,\mathsf{q})\,(w_1\,y\,z)\,(w_2\,y\,z))\,(w_3\,y\,z) \in \mathsf{e}$$

where

$$\Gamma_{\text{new}} \quad = \quad [\, w_1\colon\mathsf{i}\to\mathsf{o}\to\mathsf{a}\ ,\ w_2\colon\mathsf{i}\to\mathsf{o}\to\mathsf{b}\ ,\ w_3\colon\mathsf{i}\to\mathsf{o}\to\mathsf{d}\,]$$

The property we will make use of is

Proposition 5.41 *Let* $\Gamma = \Gamma_1 \oplus v\colon A \oplus \Gamma_2$, *where* $A = \Pi x_1\colon A_1.\ \cdots \Pi x_m\colon A_m.\, A_0$ *for an atomic* A_0. *Let* $\theta \in \Theta_\Gamma$ *and suppose*

$$\theta v =_\lambda \lambda x_1\colon A'_1.\ \cdots \lambda x_m\colon A'_m.\, N$$

where N *is a body, and for a given body* $\hat{M}$, $N \approx \hat{M}$. *Suppose for some* $\Gamma_{\hat{M}}$, $\tilde{M}$, *and* $B_{\hat{M}}$, *we have (with respect to* Γ_1, Γ_2 *and* $[\, x_1\colon A_1, \ldots, x_m\colon A_m\,]$*)*

$$\hat{M} \Longrightarrow \Gamma_{\text{new}}\ ;\ \tilde{M} \in B_{\hat{M}}$$

Let

$$\theta_{\hat{M}} \quad = \quad [\, (\lambda x_1\colon A_1.\ \cdots \lambda x_m\colon A_m.\, \tilde{M})/v\,]^{\mathsf{dom}(\Gamma_1)\cup\mathsf{dom}(\Gamma_{\text{new}})\cup\mathsf{dom}(\Gamma_2)}_{\mathsf{dom}(\Gamma)}$$

and

$$\Gamma_{\hat{M}} \quad = \quad \Gamma_1 \oplus \Gamma_{\text{new}} \oplus \theta_{\hat{M}}\Gamma_2$$

Then there is a θ' *such that* $\theta =_\lambda \theta' \circ \theta_{\hat{M}}$. *Furthermore* θ' *can be chosen such that* $\theta \gg \theta'$.

Proof: Similar to the proofs of Propositions 4.59 and 4.64, although the actual construction of θ' is somewhat more complicated. Let $\{\, v_1, \ldots, v_n \,\} = \mathsf{dom}(\Gamma_{\text{new}})$. (The order will be unimportant.) In the derivation of $\hat{M} \Longrightarrow \Gamma_{\text{new}}\ ;\ \tilde{M} \in B_{\hat{M}}$, each v_j is introduced in an instance of the third rule of Definition 5.39, which applies to an application $\hat{M}_j\ \hat{N}_j$. (This $\hat{N}_j$ is replaced by $(w_j\,x_1\cdots x_m)$.) As before, let $\{\, \hat{v}_1, \ldots, \hat{v}_l \,\} = \mathsf{dom}(\Gamma_1)\cup\mathsf{dom}(\Gamma_2)$ and $\theta\hat{v}_i = \hat{M}_i$, for $1 \le i \le l$. Then the following suffices for θ':

$$[\, (\lambda x_1\colon A'_1.\ \cdots \lambda x_m\colon A'_m.\, N_1)/v_1, \ldots, (\lambda x_1\colon A'_1.\ \cdots \lambda x_m\colon A'_m.\, N_n)/v_n\ ,\ \hat{M}_1/\hat{v}_1, \ldots, \hat{M}_l/\hat{v}_l\,]$$

□

The remaining possibilities are where the head of θv is among $\{ x_1, \ldots, x_m \}$. In HOU_Π, each of these possible heads leads to a single projection substitution. This is because in λ_Π, every well-typed body of atomic type with the same head has the same top level structure. In $\lambda_{\Pi\Sigma}$, however, this is not the case. As a simple example, given a variable $x : \mathsf{i} \times \mathsf{i}$, the two terms $\mathsf{fst}\, x$ and $\mathsf{snd}\, x$ have the same head but different top level structure. Fortunately though, there is always a finite set of "top level structures", as the following proposition makes precise.

Proposition 5.42 *Given contexts Γ and Γ_x, with Γ_x containing a typing $x_i : A_i$, there is a finite set $\mathcal{M}_i$ of terms such that for any body N of atomic type in $\Gamma \oplus \Gamma_x$ having head x_i, there is exactly one $N' \in \mathcal{M}_i$ for which $N \approx N'$. Moreover, we can effectively construct such a $\mathcal{M}_i$.*

Proof: We use $\mathcal{M}_i = \mathsf{bt}(x_i, A_i)$, where the function bt ("build template") is defined as follows. First let $\Box$ be an arbitrary term (not necessary well-typed). The idea here is that we are building up templates, representing top level structure equivalence classes, while reducing type.

$$\begin{array}{rcll} \mathsf{bt}(M, A) & = & \mathsf{bt}(M, A') & \text{if } A \text{ wh}_{\beta\pi_1\pi_2} A' \\ \mathsf{bt}(M, A_0) & = & \{M\} & \text{if } A_0 \text{ is atomic} \\ \mathsf{bt}(M, \Pi x{:}A.\, B) & = & \mathsf{bt}(M\, \Box, B) & \\ \mathsf{bt}(M, \Sigma x{:}A.\, B) & = & \mathsf{bt}(\mathsf{fst}\, M, A) \cup \mathsf{bt}(\mathsf{snd}\, M, B) & \end{array}$$

Then we reason by induction on the structure of A_i. The reason we can use an arbitrary term $\Box$ here is that we will the members $\hat{M}$ of these constructed $\mathcal{M}_i$ in applications of Definition 5.39, where the occurrences of $\Box$ will be ignored. □

Example 5.43 *Let $\Gamma_x = [\, y{:}\mathsf{a} \times \mathsf{b}\ ,\ h{:}\mathsf{a}{\to}\mathsf{b} \times (\mathsf{c}{\to}\mathsf{d})\,]$. Then we can use*

$$\begin{array}{rcl} \mathcal{M}_1 & = & \{\, \mathsf{fst}\, y, \mathsf{snd}\, y \,\} \\ \mathcal{M}_2 & = & \{\, \mathsf{fst}\, (h\, \Box), \mathsf{snd}\, (h\, \Box)\, \Box \,\} \end{array}$$

Definition 5.44 *Let the set H of terms be $\bigcup_{1 \le i \le m} \mathcal{M}_i$, together with M' if head(M') is a constant. (Recall that M' is the rigid body in the chosen disagreement pair.) For each $\hat{M} \in H$, let $\theta_{\hat{M}}, B_{\hat{M}}$, and $\Gamma_{\hat{M}}$ be as in Proposition 5.41, and define the unification problem*

$$Q_{\hat{M}} \quad = \quad \langle \Gamma_h M, \theta_{\hat{M}} \circ \theta_0, D \cup \{\langle [\,]\ ,\ A,\ B_{\hat{M}} \rangle\} \rangle$$

Now we can state our transformation:

Transformation 5.5 *Let Q and H be as above. Then make the transition*

$$Q \quad \mapsto \quad \{\, Q_{\hat{M}} \mid \hat{M} \in H \,\}$$

Proposition 5.45 *Transformation 5.5 is valid.*

Proof: Similar to the validity proof of Transformation 4.4. □

Example 5.46 *Let* $\Gamma = [\, f{:}\mathsf{a} \times \mathsf{b} \to \mathsf{a} \,]$, *and*

$$Q \quad = \quad \langle \Gamma, \theta_\Gamma^{id}, \{\langle [\, y{:}\mathsf{a} \;,\; z{:}\mathsf{b} \,] \;,\; f\,(y,z),\; y \rangle\} \rangle$$

Let $\Gamma_x = [\, x{:}\mathsf{a} \times \mathsf{b} \,]$ *Then* $H = \{\, \mathsf{fst}\, x, \mathsf{snd}\, x \,\}$, *and our set of approximating substitutions is*

$$\{\, [\, \lambda x{:}\mathsf{a} \times \mathsf{b}.\, \mathsf{fst}\, x / f \,], [\, \lambda x{:}\mathsf{a} \times \mathsf{b}.\, \mathsf{snd}\, x / f \,] \,\}$$

The first of these is well-typed and will lead to a solution. The second is ill-typed and does not.

5.7 Completeness and Unifiability

Now that we have our collection of valid transformations, we have two remaining issues: completeness of the combined transformation, and unifiability of solved form unification problems. The completeness proof is similar to the proof of Proposition 4.73.

For unifiability, we have an additional complication over the proof of Proposition 4.75: Flexible bodies are not necessarily of the form $v\, M_1 \cdots M_m$, because v might be of pair-producing type. Fortunately however, we may appeal to the validity of Transformation 5.4. Let $Q = \langle \Gamma, \theta_0, D \rangle$ be in solved form, and suppose that Γ contains a variable of pair-producing type. Let $\{Q'\}$ result from Q by Transformation 5.4. By correctness of that transformation, Q has a solution iff Q' does. (In fact they have the same solutions, but this is not relevant.) We then appeal to this argument until there are no remaining variables of pair-producing type. At this point, the proof of Proposition 4.75 applies.

Chapter 6

Polymorphism

In this chapter, we informally sketch an extension of the pre-unification algorithm for $\lambda_{\Pi\Sigma}$ to a calculus $\lambda_{\Pi\Sigma\delta}$ with implicit polymorphism, *i.e.*, type variables but no explicit type abstraction, and a very limited form of type application. The resulting algorithm is, unfortunately, incomplete. However, considerable experience has shown it to be useful in practice. For many unification problems of interest, the algorithm does indeed construct minimal complete sets of pre-unifiers, and the cases in which the algorithm is incomplete can always be detected.

6.1 The Language Extension

There are two changes in the language of types. The first is the presence of type variables, which we denote by ν and α. The second is that we generalize type constants to instantiated *type constructors*, which we notate by subscripting. To reduce confusion between terms and types, we will often use "a" instead of "c" for type constructors.

$$A \quad ::= \quad \mathsf{a}_{A_1 \cdots A_n}$$

Similarly, constant terms result from type-instantion of *polymorphic constants*:

$$M \quad ::= \quad \mathsf{c}_{A_1 \cdots A_n}$$

Of course, this calculus could be made more uniform by allowing general application of types to types and terms to types, and by having corresponding abstractions. The resulting calculus, which would resemble the second- or ω-order polymorphic λ-calculus [25, 24, 65, 45], or the Calculus of Constructions [11], is very powerful computationally. However, unification is a topic for future research.

6.1.1 Substitution and Conversion

The meaning of applying substitutions to terms, types, and kinds carries over from Definition 2.7, with the obvious extensions. In particular, for instantiated polymorphic constants,

$$\overline{\theta}(\mathsf{c}_{A_1 \cdots A_n}) \quad = \quad \mathsf{c}_{\overline{\theta} A_1 \cdots \overline{\theta} A_n}$$

As always, we will generally write "θ" in place of "$\overline{\theta}$".

There are no new reduction rules. The meanings of (a) $\rightarrow_\rho^*$ for a reduction relation ρ, and (b) $\mathrm{wh}_{\beta\pi_1\pi_2}$, are extended in the obvious way. Weak head normal forms are the same as before, except that a type body may also be of the form $\mathsf{a}_{A_1 \cdots A_n}$, and a term body may also be of the form $\mathsf{c}_{A_1 \cdots A_n}$.

6.1.2 Typing Rules

Signatures for this language assign kind and type *schemas* to type constructors and polymorphic constants; we write these as "$\Delta\nu_1 \cdots \nu_n.\ K$" and "$\Delta\nu_1 \cdots \nu_n.\ A$" where the only type variables occurring in K or A are among the ν_i. The instantiation rules show how these schema are interpreted. Similarly to our abbreviations with Π types and kinds, we will, *e.g.*, write "$\mathsf{Type} \rightarrow L$" in place of "$\Delta\nu.\ L$" if $\nu \notin \mathcal{F}(L)$.

Valid Signatures In place of the rules in Definition 2.29 for adding kind and type constants, we have

$$\frac{\vdash \Sigma \text{ sig} \qquad [\,\nu_1{:}\mathsf{Type}\ ,\ \ldots\ ,\ \nu_k{:}\mathsf{Type}\,] \vdash_\Sigma K \in \mathsf{Type} \qquad \mathsf{a} \notin \mathsf{dom}(\Sigma)}{\vdash \Sigma \oplus \mathsf{a}{:}\Delta\nu_1 \cdots \nu_k.\ K \text{ sig}}$$

$$\frac{\vdash \Sigma \text{ sig} \qquad [\,\nu_1{:}\mathsf{Type}\ ,\ \ldots\ ,\ \nu_k{:}\mathsf{Type}\,] \vdash_\Sigma A \in \mathsf{Type} \qquad \mathsf{c} \notin \mathsf{dom}(\Sigma)}{\vdash \Sigma \oplus \mathsf{c}{:}\Delta\nu_1 \cdots \nu_k.\ A \text{ sig}}$$

Valid Types

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad \mathsf{a}{:}\,\Delta\nu_1 \cdots \nu_k.\ K \in \Sigma \qquad \Gamma \vdash_\Sigma A_1 \text{ kind} \qquad \ldots \qquad \Gamma \vdash_\Sigma A_n \text{ kind}}{\Gamma \vdash_\Sigma \mathsf{a}_{A_1 \cdots A_n} \in [\,A_1/\nu_1\ ,\ \ldots\ ,\ A_n/\nu_n\,]K}$$

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad \nu{:}\mathsf{Type} \in \Gamma}{\Gamma \vdash_\Sigma \nu \in \mathsf{Type}}$$

Valid Terms

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad \mathsf{c}{:}\Delta\nu_1 \cdots \nu_k.\, A \in \Sigma \qquad \Gamma \vdash_\Sigma A_1 \text{ kind} \qquad \ldots \qquad \Gamma \vdash_\Sigma A_n \text{ kind}}{\Gamma \vdash_\Sigma \mathsf{c}_{A_1 \cdots A_n} \in [\, A_1/\nu_1 \;,\; \ldots \;,\; A_n/\nu_n \,]A}$$

6.2 The Transformations

Most of the transformations developed for $\text{HOU}_{\Pi\Sigma}$ are valid for $\text{HOU}_{\Pi\Sigma\delta}$ as well. Rather than formally restating and proving every transformation, we will focus on what has to be changed.

The transformations for handling $\text{wh}_{\beta\pi_1\pi_2}$ redices, abstractions and pairs, and pair-producing types, *i.e.*, Transformations 5.1, 5.2, and 5.4, carry over with their proofs unchanged.

6.2.1 Rigid-rigid

The rigid-rigid case carries over with one change, which is to the definition of the rigid decomposition relation $\rightsquigarrow_{\text{rr}}$ (Definition 5.21). The new rules are, for types,

$$\langle \Psi \;,\; \mathsf{a}_{A_1 \cdots A_n} \;,\; \mathsf{a}_{A'_1 \cdots A'_n} \rangle \rightsquigarrow_{\text{rr}} \{\, \langle \Psi \;,\; A_1,\; A'_1 \rangle, \ldots, \langle \Psi \;,\; A_n,\; A'_n \rangle \,\}$$

and, for terms,

$$\langle \Psi \;,\; \mathsf{c}_{A_1 \cdots A_n} \;,\; \mathsf{c}_{A'_1 \cdots A'_n} \rangle \rightsquigarrow_{\text{rr}} \{\, \langle \Psi \;,\; A_1,\; A'_1 \rangle, \ldots, \langle \Psi \;,\; A_n,\; A'_n \rangle \,\}$$

It is then simple to extend the proof of Propositions 4.34 and 5.22 to this new definition of $\rightsquigarrow_{\text{rr}}$.

It is worth pointing out that we benefit here from the use of weak head normal form as opposed to the long (η and π expanded) head normal form. The reason for this is evident in the following example:

Example 6.1 *Consider the rigid-rigid disagreement pair*

$$\langle [\, g{:}\mathsf{i}{\to}\nu \,] \;,\; g\, M,\; g\, M' \rangle$$

over a unification context Γ *with* $\nu \in \mathsf{dom}(\Gamma)$*. If our notion of top level structure depended on the long head normal form, we could not say that applying a substitution to these rigid terms leaves the top level structure unchanged. Instantiating* ν *to a functional type, say,* $\mathsf{i}{\to}\mathsf{i}$*, followed by the required* η*-expansion, would result in*

$$\langle [\, g{:}\mathsf{i}{\to}\mathsf{i}{\to}\mathsf{i} \,] \;,\; \lambda x{:}\mathsf{i}.\, g\, M\, x,\; \lambda x{:}\mathsf{i}.\, g\, M'\, x \rangle$$

6.2.2 Type Flexible-rigid

With the addition of type variables, we now have flexible types. Our unification algorithm must then handle the type flexible-rigid case. This case is simpler than the corresponding case for terms, because we need only to consider imitations and not projections.

Given a unification problem $\langle \Gamma, \theta_0, D \rangle$, suppose D contains a flexible-rigid type disagreement pair $\langle \Psi, A, A' \rangle$ or rigid-flexible type disagreement pair $\langle \Psi, A', A \rangle$. Call this disagreement pair P. Then A has the form $\nu\, M_1 \cdots M_m$, where $\Gamma = \Gamma_1 \oplus \nu{:}\Pi x_1{:}A_1.\ \cdots \Pi x_m{:}A_m.\ K \oplus \Gamma_2$. Also, A' is either a Π or Σ type or has the form $\mathsf{a}'_{B'_1 \cdots B'_{l'}}\ M'_1 \cdots M'_{m'}$. For now, we will assume the latter. Let $\theta \in \Theta_\Gamma$ and let $\theta\nu$ be

$$\lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ \mathsf{a}_{B_1 \cdots B_l}\ N_1 \cdots N_n$$

If θ unifies D, and hence P, then $\theta A =_\lambda \theta A' \approx A'$, so the WHNF of θA has head a'. However, because of the form of $\theta\nu$, the head of θA is a whether or not θ is a unifier, so $\mathsf{a} = \mathsf{a}'$. Therefore, a single approximating substitution (an imitation) handles this case. Let

$$\begin{aligned} C_I &= \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ \mathsf{a}'_{\hat{B}_1 \cdots \hat{B}_l}\ \hat{M}_1 \cdots \hat{M}_{m'} \\ \theta_I &= [\, C_I/\nu \,]^{\mathsf{dom}(\Gamma_1) \cup \{\, \nu_1, \ldots, \nu_n, v_1, \ldots, v_{m'} \,\} \cup \mathsf{dom}(\Gamma_2)}_{\mathsf{dom}(\Gamma)} \end{aligned}$$

for "new" distinct type variables $\nu_1, \ldots, \nu_n \notin \mathsf{dom}(\Gamma) - \{\nu\}$, where for $1 \le k \le l$

$$\hat{B}_k = \nu_k\, x_1 \cdots x_m$$

and for new distinct term variables $v_1, \ldots, v_m \notin \mathsf{dom}(\Gamma)$, where, for $1 \le j \le m'$,

$$\hat{M}_j = v_j\, x_1 \cdots x_m$$

The kinds $K_1, \ldots, K_l$ of $\nu_1, \ldots, \nu_l$, the types $C_1, \ldots, C_{m'}$ of $v_1, \ldots, v_{m'}$, and the kind K_I of C_I are constructed similarly to Definition 4.58. Then we define our new context as

$$\Gamma_I = \Gamma_1 \oplus [\, \nu_1{:}K_1, \ldots, \nu_l{:}K_l, v_1{:}C_1, \ldots, v_{m'}{:}C_{m'} \,] \oplus \Gamma_2$$

For the case that A' is a Π or Σ type, the construction is simpler. For instance, suppose that $A' = \Pi u{:}B'.\ C'$. Then we would use

$$C_I = \lambda x_1{:}A_1.\ \cdots \lambda x_m{:}A_m.\ \Pi u{:}\nu_1\, x_1 \cdots x_m.\ \nu_2\, u\, x_1 \cdots x_m$$

Following the same reasoning as in Section 4.4.5, we get

Transformation 6.1 *Let A, θ_I, K_I, and Γ_I be as above. Then make the transition*

$$Q \mapsto \{\langle \Gamma_I, \theta_I \circ \theta_0, \theta_I D \uplus \{\langle [\,], K, K_I \rangle\} \rangle\}$$

(Note that we are now adding a kind *disagreement pair. This requires an obvious and very simple extension to the rigid-rigid case, which handles these disagreement pairs immediately.)*

6.2.3 Term Flexible-rigid

We adopt the flexible-rigid transformation from $\text{HOU}_{\Pi\Sigma}$, although it is incomplete in the presence of polymorphism. In this section, we show under what conditions completeness is lost. Extensive experience with λProlog [54], which uses a similarly incomplete algorithm (without dependent types), has shown that these conditions are rare in practice, but do occur.

Let our disagreement pair be $\langle \Psi\ ,\ M,\ M' \rangle$ where M is a flexible term body with head v and M' is a rigid term body. Let the unification context $\Gamma = \Gamma_1 \oplus v\colon A \oplus \Gamma_2$, where $A = \Pi x_1\colon A_1.\ \cdots \Pi x_m\colon A_m.\ A_0$. In $\lambda_{\Pi\Sigma}$, we made the assumption that A_0 is atomic,[1] but it is no longer helpful to assume this, since A_0 might still be instantiated to a Π type. For any $\theta \in \Theta_\Gamma$, θv is convertible to a term

$$\lambda x_1\colon A'_1.\ \cdots \lambda x_{m+k}\colon A'_{m+k}.\ N$$

where N is a body or pair, and $\theta A =_\lambda \Pi x_1\colon A'_1.\ \cdots \Pi x_{m+k}\colon A'_{m+k}.\ A'_0$ where A'_0 is atomic. Again, we consider the possibilities for N when θ is a unifier: If a pair, then since M' is a rigid body (not a pair), and θ unifies M and M', N must be convertible to a body. Thus without loss of generality we can assume that N is a body. Then as in Section 5.6.5, either (a) $N \approx M'$ and $\text{head}(M')$ is a constant, or (b) $\text{head}(N) = x_j$ where $1 \le j \le m+k$. However for $m < j \le m+k$, θ would not be a unifier, since β reduction leaves

$$\lambda x_{m+1}\colon A'_{m+1}.\ \cdots \lambda x_{m+k}\colon A'_{m+k}.\ x_j\ \hat{N}_1 \cdots \hat{N}_n$$

but $\text{head}(M')$ is not x_j. Thus it is sufficient to consider only the first m projections.

The imitation substitution must be generalized to accomodate polymorphic type constants. The construction is as in the type flexible-rigid case above.

In $\text{HOU}_{\Pi\Sigma}$, we were able to re-express these possibilities as an equivalent condition on unifiers θ that $\theta =_\lambda \theta' \circ \theta_b$ for some constructed θ_b (Proposition 5.41). A simple example shows why this is harder with polymorphism.

Example 6.2 *Let our signature and unification context be*

$$\begin{array}{rcl} \Sigma & = & \langle\, \mathsf{c}\colon \Delta\alpha.\ \alpha \to \alpha \,\rangle \\ \Gamma & = & [\, \alpha\colon \mathsf{Type}\ ,\ f\colon (\mathsf{i} \to \mathsf{i}) \to \alpha \,] \end{array}$$

and consider the disagreement set $\langle [\,]\ ,\ f\ (\lambda z\colon \mathsf{i}.\ z),\ \mathsf{c}_\alpha \rangle$. *For projections, as we have said, we need only consider the first. However the following approximating substitution*

$$[\, (\lambda g\colon \mathsf{i} \to \mathsf{i}.\ g\ (f_1\ g))/f\ ]$$

[1] We extend "atomic" to include the form $\nu\, M_1 \cdots M_m$.

is not sufficient, because then we would fail to find the following unifier:

$$[\,\mathsf{i}{\to}\mathsf{i}/\alpha\ ,\ (\lambda g{:}\mathsf{i}{\to}\mathsf{i}.\ \lambda y{:}\mathsf{i}.\ g\,(\mathsf{c}_{\mathsf{i}}\,y))/f\,]$$

Similarly, if our approximating substitution were

$$[\,(\lambda g{:}\mathsf{i}{\to}\mathsf{i}.\ \mathsf{c}_{(\beta\,g)}\,(f_1\,g))/f\,]$$

then we would miss the solution

$$[\,\mathsf{i}{\to}\mathsf{i}/\alpha\ ,\ (\lambda g{:}\mathsf{i}{\to}\mathsf{i}.\ \lambda y{:}\mathsf{i}.\ \mathsf{c}_{\mathsf{i}}\,(g\,y))/f\,]$$

Thus we see that the flexible-rigid transformation for $\mathrm{HOU}_{\Pi\Sigma}$ can cause loss of unifiers in $\lambda_{\Pi\Sigma\delta}$. Fortunately however, no incorrect unifiers are introduced, and minimality still holds. Furthermore, we can easily detect the cases that can lead to loss of unifiers. One case that causes loss of unifiers is as in the example, in which the *target type* A_0 is flexible (has a type variable as head. The other is when the head b being used to build up the approximating substitution has a flexible target type.

Chapter 7

Applications

This chapter explores applications of the pre-unification procedures we developed in the preceding chapters. These applications all have in common that they use a typed λ-calculus as a *meta-language*, *i.e.*, a calculus in which to encode other languages, which we will call *object-languages*. The rich structure of a typed λ-calculus as opposed to the traditional, first-order abstract syntax trees allows us to express rules, *e.g.*, program transformation and logical inference rules, that are more succinct, more powerful, and easier to reason about. We can then use unification in the meta-language to mechanize application of these rules.

As we will demonstrate in the examples below, there are three primary benifits of this kind of meta-language.

- By exploiting the λ of the meta-language, and its corresponding functional type, one can directly capture the scoping rules of many object-languages. As we will see, this allows for object-language independent mechanisms (β-reduction and α-conversion) for substitution and bound variable renaming that work correctly for even the binding constructs of a (correctly encoded) object-language.

- Using dependent types, when the object-language is a logic (an *object-logic*) one can capture the theorem/proof relationship, as convincingly demonstrated by the work on the Logical Framework [30]. This allows for object-logic independent proof checking and interactive proof construction, but also some degree of automated theorem proving, given a suitable unification-based language, such as Elf [58].

- Again, using the dependent features of the type system, we can internalize object-language typing rules, so that only object-language terms that are well-typed according the the object-language typing rules have meta-language representations that are well-typed acording to the meta-language typing rules. This property allows for object-language independent mechanisms for object-language type checking and inference.

We will often refer to this kind of encoding as "higher-order abstract syntax" (HOAS) in constrast with (first-order) abstract syntax trees, when we are emphasizing the first of these benefits, and "dependent HOAS" when emphasing the others.

The general idea goes back to Church, who expressed all of the binding constructs of higher-order logic in terms of the λ of the simply typed λ-calculus [8]. The use of second-order matching and substitution for transformation of programs represented in the simply typed λ-calculus was suggested in [37]. In recent years the idea has appeared in several guises: In Isabelle [56, 57] the syntax of logics are encoded as simply typed terms and their inference rules are encoded as formulas in intuitionistic higher-order logic. In λProlog the representation is enriched to include implicit polymorphism, and the logic programming framework allows one to program control of the selection and application of rules [50]. In LF [30] a λ-calculus with dependent types is used as a meta-logic to encode the "language of a logic, its axiom and rule schemes, and its proofs", but unification is not used. In [60] the value of products together with polymorphism is demonstrated. The basic idea is also present in Martin-Löf's system of arities [43].

7.1 Some Motivating Examples

In this section we highlight some of the problems that arise in matching and substitution due to the presence of binding constructs in a language. Almost all languages have these binding constructs, though sometimes they are not immediately apparent. For example, in Prolog the "free" variables in a clause are actually bound over that clause, since they are clearly distinct from variables with the same name in other clauses. A function definition stated as $f(x) = b$ actually binds x and f (see the beginning of Section 7.3.2).

The rules we present throughout this paper are stated without any *semantic* side conditions such as strictness or termination. Depending on the language semantics (in particular, call-by-name *vs* call-by-value), such conditions may still be necessary to ensure semantic equivalence between the transformed programs. However, it should be noted that in all the examples the *syntactic* side conditions on the rules disappear without compromising the validity of the rule.

7.1.1 Correct Matching and Substitution

This problem of variable capture is very common. It appears in two different forms: during matching and during substitution. Consider the rule of **let**-conversion[1]:

$$\text{“}\mathbf{let}\ x = e\ \mathbf{in}\ b\text{”} \quad \Leftrightarrow \quad \text{“}[\, e/x\,]b\text{”}$$

[1] We use quotation marks, “...”, to distinguish concrete syntax from representations.

Here are two incorrect applications of this rule. Note that reading them from right to left shows the problem of doing correct matching against "$[\, e/x \,]b$".

$$\text{“}\textbf{let } x = y \textbf{ in let } y = 5 \textbf{ in } x * y\text{”} \quad \not\Leftrightarrow \quad \text{“}\textbf{let } y = 5 \textbf{ in } y * y\text{”}$$

$$\text{“}\textbf{let } x = 5 \textbf{ in let } x = x * x \textbf{ in } x\text{”} \quad \not\Leftrightarrow \quad \text{“}\textbf{let } x = 5 * 5 \textbf{ in } 5\text{”}$$

What is required for correct substitution is recognition of name conflicts and renaming of bound variables. If this rule is read from right-to-left, it is clear that there are many possible ways of abstracting an expression from a program, and that therefore straightforward matching on *any* representation would be very non-deterministic. In a situation like this the solution is to partially instantiate the pattern before matching.

7.1.2 Variable Occurrence Restriction

Variable occurrence restrictions again require renaming of bound variables during substitution, or failure of matching. The following example is taken from a formalization of a natural deduction system to show the variety of circumstances in which these problems occur.

$$\text{“}\ \frac{\Gamma \vdash P}{\Gamma \vdash \forall x P}\ \forall I \quad \text{where } x \text{ not free in } \Gamma\ \text{”}$$

If this rule is used by matching against the lower line, the restriction on x must be checked separately. Ideally, x would be renamed to a new variable if x is already free in Γ. If the rule is used in the other direction, it should simply not match if x appears in Γ. As we will see in Section 7.5, rules incorporating occurrence conditions can be formulated easily and applied correctly using higher-order abstract syntax.

Note that in a system that uses first-order abstract syntax, not only must the rule be conditional, but the language implementor must somehow define a predicate not-free-in for the language in question.

7.1.3 Correct Treatment of Contexts

Many program transformation rules can be stated naturally through the use of *contexts*. Correct applications of these rules, however, is tricky. For example, a rule propagating computation into the branches of an **if** expression could be written as

$$\text{“}C[\textbf{if } p \textbf{ then } a \textbf{ else } b]\text{”} \quad \Leftrightarrow \quad \text{“}\textbf{if } p \textbf{ then } C[a] \textbf{ else } C[b]\text{”}$$

Consider the following incorrect application.

"**let** $p =$ **false in if** p **then** 1 **else** 2" $\not\Leftrightarrow$
"**if** p
then let $p =$ **false in** 1
else let $p =$ **false in** 2"

As noted in [52] syntactic conditions on C are difficult to formulate if one wishes to eliminate the possibility of incorrect rule application as in the example. The use of higher-order abstract syntax solves this problem by allowing the statement of the rule as above, but automatically prohibiting the incorrect use below without any additional conditions.

7.1.4 Object-language Typing

For typed object-languages, correct formulation of rules often require taking object-types into account. For example, consider the following rule of existential introduction:

$$\text{“}\ \frac{\mathcal{A} \vdash [\,t/x\,]P}{\mathcal{A} \vdash \exists x{:}\alpha.\ P}\ \exists I \quad \text{where } t \text{ has type } \alpha\ \text{”}$$

To formalize a rule like this without dependent types would require a mechanism for defining and checking object-language typing. However, even this wold be insufficient in practice. A very convenient way to use this kind of rule, in a unification based language as discussed in [23, page 8], is to introduce a new unification variable, which will be instantiated later, when the desired value for t becomes known. We would like the condition that the instantiation term t must have object-type α to act as a constraint to reduce the possible instantiations rather than as a filter to reject the choices later, as the latter can lead to much more search.

7.2 A Convenient Notation

The calculus notation used in the previous chapters works very well for formal manipulation, but can be improved on for ease of reading and writing. Consider the following the following function for reversing a pair:

$$\lambda z{:}\mathsf{i} \times \mathsf{o}.\ \mathsf{snd}\ z, \mathsf{fst}\ z$$

Many modern functional programming languages provide a convert notation for functions that operate on structured information [6, 29, 74]. In the fashion of these languages, we can write the above as

$$\lambda(x, y){:}\mathsf{i} \times \mathsf{o}.\ y, x$$

We will use such expressions in the examples of this chapter, as an abbreviation for terms like the previous one (that binds z).

7.3 Language Representation

In this section we demonstrate how to use $\lambda_{\Pi\Sigma\delta}$ as a meta-language for encoding typed programming languages. (It is simpler to represent untyped languages, which is a degenerate case.)

7.3.1 A Simple Expression Language

We begin with a simple language of expressions given by the following grammar:

$$\begin{array}{rcl} e & ::= & v \\ & | & 0 \mid 1 \mid \ldots \\ & | & \textbf{true} \mid \textbf{false} \mid \textbf{nil} \\ & | & e + e \mid e * e \mid e :: e \mid e = e \mid \ldots \\ & | & \textbf{hd}(e) \mid \textbf{tl}(e) \mid \ldots \\ & | & \textbf{if } e \textbf{ then } e \textbf{ else } e \end{array}$$

For now, types in this language will be simply integer, boolean, and lists of elements of the same type:

$$\begin{array}{rcl} tp & ::= & \textbf{int} \\ & | & \textbf{bool} \\ & | & tp \textbf{ list} \end{array}$$

There is a lot of flexibility in choosing how to represent such a language, and we give one possibility. What we mean by choosing a representation for an object-language L is constructing a signature Σ_L and a correspondence between expressions in L and terms in the meta-language (in this case $\lambda_{\Pi\Sigma\delta}$). We will be informal about what this correspondence is, prefering to describe it by means of examples. First we must choose a representation of object-language types, and so introduce a (meta-language) type constant

$$\mathsf{tp} \;:\; \mathsf{Type}$$

Then, to represent the object-types, add the following new constants[2]

$$\begin{array}{rcl} \mathsf{int}, \mathsf{bool} & : & \mathsf{tp} \\ \mathsf{list} & : & \mathsf{tp} \to \mathsf{tp} \end{array}$$

For example, the object-type "**int list**" is represented by $(\mathsf{list}\,\mathsf{int})$.

[2]For brevity in specifying signatures, we use "$\mathsf{c}, \mathsf{c}' : A$" as an abbreviation for "$\mathsf{c} : A, \mathsf{c}' : A$". Also, for readability, we stack these typings vertically rather than presenting them linearly and separating them by commas.

Next we need a type to represent expressions. In fact, we will use a type family, indexed by object-type:

$$\mathsf{e} \;:\; \mathsf{tp}\to\mathsf{Type}$$

The idea is that the meta-language type $(\mathsf{e}\,s)$ is the type of terms representing well-typed object-language expressions whose object-type is represented by s. As it turns out, it is important to represent object-language variables directly as meta-language variables. To construct representations of other expressions, we begin by introducing constants for basic integer and boolean values. Booleans are straightforward:

$$\mathsf{true}, \mathsf{false} \;:\; \mathsf{e}\,\mathsf{bool}$$

Number expressions are constructed by a "coercion" constant:

$$\mathsf{num} \;:\; \mathsf{integer}\to\mathsf{e}\,\mathsf{int}$$

where integer is the type of integers (as opposed to integer expressions). The other basic value, the empty list, exists for all list object-types, so we will use a constant of functional type. Here we have our first use of *dependent* function types:

$$\mathsf{nil} \;:\; \Pi s{:}\mathsf{tp}.\,\mathsf{e}\,(\mathsf{list}\,s)$$

For instance, the empty integer list is represented by $(\mathsf{nil}\,\mathsf{int})$. Next we add constants for all of the "built-in" operations of the language, *e.g.*,

$$\begin{aligned} \mathsf{plus}, \mathsf{times} &: \mathsf{e}\,\mathsf{int}\to\mathsf{e}\,\mathsf{int}\to\mathsf{e}\,\mathsf{int} \\ \mathsf{gtr} &: \mathsf{e}\,\mathsf{int}\to\mathsf{e}\,\mathsf{int}\to\mathsf{e}\,\mathsf{bool} \\ \mathsf{and}, \mathsf{or} &: \mathsf{e}\,\mathsf{bool}\to\mathsf{e}\,\mathsf{bool}\to\mathsf{e}\,\mathsf{bool} \end{aligned}$$

Again, we must handle the object-type polymorphism of operators like equality and list cons:

$$\begin{aligned} \mathsf{equal} &: \Pi s{:}\mathsf{tp}.\,\mathsf{e}\,s\to\mathsf{e}\,s\to\mathsf{e}\,\mathsf{bool} \\ \mathsf{cons} &: \Pi s{:}\mathsf{tp}.\,\mathsf{e}\,s\to\mathsf{e}\,(\mathsf{list}\,s)\to\mathsf{e}\,(\mathsf{list}\,s) \end{aligned}$$

Similarly for conditional expressions:

$$\mathsf{ite} \;:\; \Pi s{:}\mathsf{tp}.\,\mathsf{e}\,\mathsf{bool}\to\mathsf{e}\,s\to\mathsf{e}\,s\to\mathsf{e}\,s$$

Example 7.1 *The expression "***if** $a > b$ **then** a **else** b*", where a and b are integer variables, is represented*

$$\mathsf{ite}\,\mathsf{int}\,(\mathsf{gtr}\,a\,b)\,a\,b$$

By using dependent types in our representation, we have *internalized* the typing rules of the object-language. Thus for each well-typed term s of type tp, there is a one-to-one correspondence between long normal form terms of type $(\mathsf{e}\,s)$ and well-typed object-language expressions whose type is represented by s. (See [30] for a definition of these normal forms and a rigorous account of the correspondence.) A fortunate consequence is that when these techniques apply, object-language type checking is reduced to meta-language type checking, which may be implemented once, independently of any object-language. However, in order to accomplish this, the representation has to contain much more information.

Example 7.2 *The representation of "*$(1 :: 2 :: \mathbf{nil}) = (2 :: 1 :: \mathbf{nil})$*" is*

$$\begin{array}{l}\mathsf{equal}\,(\mathsf{list}\,\mathsf{int})\\ \quad(\mathsf{cons}\,\mathsf{int}\,(\mathsf{num}\,1)\,(\mathsf{cons}\,\mathsf{int}\,(\mathsf{num}\,2)\,(\mathsf{nil}\,\mathsf{int})))\\ \quad(\mathsf{cons}\,\mathsf{int}\,(\mathsf{num}\,2)\,(\mathsf{cons}\,\mathsf{int}\,(\mathsf{num}\,1)\,(\mathsf{nil}\,\mathsf{int})))\end{array}$$

Fortunately, as we will see in Section 7.4, this extra information can usually be automatically generated.

Now we will add a statically scoped variable binding expression:

$$e \quad ::= \quad \mathbf{let}\ v = e\ \mathbf{in}\ e$$

It will be important to directly capture the scope of this variable binding. In general, we do this by using the meta-language λ for all statically scoped binding constructs. In the representation of a **let** expression, we will need to take into account the object-type of the expression being bound to the variable and of the whole expression:

$$\mathsf{let} \quad : \quad \Pi s{:}\mathsf{tp}.\ \Pi t{:}\mathsf{tp}.\ (\mathsf{e}\,s{\to}\mathsf{e}\,t){\to}\mathsf{e}\,s{\to}\mathsf{e}\,t$$

Example 7.3 *The expression "***let** $x = 2$ **in** $x > 1$*" is represented*

$$\mathsf{let}\,\mathsf{int}\,\mathsf{bool}\,(\lambda x{:}\,\mathsf{e}\,\mathsf{int}.\ \mathsf{gtr}\,x\,(\mathsf{num}\,1))\,(\mathsf{num}\,2)$$

Note that although this **let** is (object language) polymorphic, it is not "genericly polymorphic" as is ML [51, 7].

In languages with this kind of binding construct, such as ML [29], Scheme [72], and Lisp [71], it is often possible to bind many variables in parallel, so that the general form is instead

$$e \quad ::= \quad \mathbf{let}\ v = e, \ldots, v = e\ \mathbf{in}\ e$$

One way of dealing with this flexibility is to have an infinite (or for practical purposes, reasonably large) family of constants. For each $n \geq 0$, we would declare

$$\mathsf{let}_n \quad : \quad \Pi s_1{:}\mathsf{tp}.\ \cdots \Pi s_n{:}\mathsf{tp}.\ \Pi t{:}\mathsf{tp}.\ (\mathsf{e}\,s_1{\to}\cdots{\to}\mathsf{e}\,s_n{\to}\mathsf{e}\,t){\to}\mathsf{e}\,s_1{\to}\cdots{\to}\mathsf{e}\,s_n{\to}\mathsf{e}\,t$$

Aside from the general awkwardness of this approach, it has the serious drawback that each rule for manipulating **let** expressions must also have an infinite number of versions. These problems may be avoided by using pair types and polymorphism in our representation:

$$\mathsf{let} \quad : \quad \Delta\alpha.\ \Pi t{:}\mathsf{tp}.\ (\alpha{\to}\mathsf{e}\,t){\to}\alpha{\to}\mathsf{e}\,t$$

where we will instantiate α to a type of the form $(\mathsf{e}\,s_1 \times \cdots \times \mathsf{e}\,s_n)$. (This ability was our original motivation to explore higher-order unification with pair types.)

Example 7.4 *The expression "***let** $n = 1, b =$ **true in** b **and** $n > 0$*" is represented (using the varstruct notation introduced in Section 7.2)*

$$\mathsf{let}_{\mathsf{e\,int}\times\mathsf{e\,bool}}\ \mathsf{bool}\ (\lambda(n,b){:}\,\mathsf{e\,int}\times\mathsf{e\,bool}.\ \mathsf{gtr}\,n\,(\mathsf{num}\,0))\ (\mathsf{num}\,1, \mathbf{true})$$

There is a theoretical problem with this representation, however. In the type of let, there is no simple way to restrict the type parameter α to be instantiated to types of the form $(\mathsf{e}\,s_1 \times \cdots \times \mathsf{e}\,s_n)$. As a consequence, there are well-typed LNF meta-language terms of type $(\mathsf{e}\,s)$ that do not correspond to terms in our object-language. We will show how to eliminate this problem in Section 7.3.3.

7.3.2 Adding Programs

In the previous section we developed a simple language of typed expressions. We now extend it to a language of recursive function definitions of a simple form:

$$p \quad ::= \quad \mathbf{rec}\ v(v,\ldots,v) = e, \ldots, v(v,\ldots,v) = e$$

and add a form of expression for invoking a defined function

$$e \quad ::= \quad v(e,\ldots,e)$$

Observe that a program may involve any number of function definitions, and each function may take any number of arguments. Again we will use polymorphism:

$$\mathsf{rec} \quad : \quad \Delta\alpha.\ (\alpha{\to}\alpha){\to}\alpha$$

Example 7.5 *Consider the following simple list reversal program*

$$\begin{array}{l}\mathbf{rec}\ \mathit{rev}(l) = \mathit{ra}(l, \mathbf{nil})\ , \\ \quad \mathit{ra}(l,z) = \mathbf{if}\ \mathbf{null}(l)\ \mathbf{then}\ z\ \mathbf{else}\ \mathit{ra}(\mathbf{tl}(l), \mathbf{hd}(l) :: z)\end{array}$$

The complete representation (see below for the values of A and B) is

$$\begin{array}{l}\mathsf{rec}_A\,(\lambda(\mathit{rev},\mathit{ra}){:}\,A.\ ((\lambda l{:}\,B.\ \mathit{ra}(l, \mathsf{nil}\,s))\ , \\ \qquad\qquad (\lambda(l,z){:}\,B\times B.\ \mathsf{ite}\,(\mathsf{list}\,s)\,(\mathsf{null}\,s\,l)\,z\,(\mathit{ra}(\mathsf{tl}\,s\,l, \mathsf{cons}\,s\,(\mathsf{hd}\,s\,l)\,z))))\end{array}$$

where s is an arbitrary term of type tp*, and*

$$\begin{array}{rcl} A & = & (B{\to}B)\times(B\times B{\to}B) \\ B & = & \mathsf{e}\,(\mathsf{list\,int})\end{array}$$

7.3.3 Syntactic Judgments

The encodings in the previous section that use polymorphism are too liberal, in that they allow for well-typed terms that do not correspond to legitimate expressions. Consider again the constant used in representing parallel binding **let** expressions:

$$\mathsf{let} \quad : \quad \Delta\alpha.\, \Pi t{:}\mathsf{tp}.\, (\alpha \to \mathsf{e}\, t) \to \alpha \to \mathsf{e}\, t$$

Our intent is that the type argument to let be of the form $(\mathsf{e}\, s_1 \times \cdots \times \mathsf{e}\, s_n)$, for object-types $s_1, \ldots, s_n$.[3] How can this intent be enforced, so that non-legitimate **let** expressions do not have well-typed encodings?

Our solution is similar to Mason's technique of "syntactic judgments" used in the encoding of Hoare logic [2, 44]. One problem in representing Hoare logic is that there are boolean expressions (used in constructing statements of the imperative programming language) and first-order formulas (used in assertions), and the boolean expressions are identified with the quantifier-free first-order formulas. Mason's representation uses a (syntactic) judgment QF indexed over the type o of formulas, and declares, *e.g.*, the conditional statement constructor as

$$\mathsf{if} \quad : \quad \Pi e{:}\mathsf{o}.\, \mathsf{QF}\, e \to \mathsf{w} \to \mathsf{w} \to \mathsf{w}$$

where w is the type used for representing statements. That is, if now takes an additional argument, which is a proof that the first argument is a quantifier-free expression. The signature is extended to include constant declarations that encode an inference system for proving formulas to be quantifier-free, *e.g.*,

$$\mathsf{QF}_2 \quad : \quad \Pi e{:}\mathsf{o}.\, \mathsf{QF}\, e \to \mathsf{QF}\, (\neg e)$$

This constant represents the rule that $\neg e$ is quantifier-free if e is. He also uses a syntactic judgment for non-interference conditions.

Our situation is slightly different, because we want to restrict type arguments, rather than term arguments, but the essential idea is the same. We introduce a syntactic judgment

$$\mathsf{ok} \quad : \quad \mathsf{Type} \to \mathsf{Type}$$

The intent is that, for a given type α, there is a term of type ok_α iff α is of the appropriate form. This intent is formalized via the following declarations

$$\begin{array}{lcl} \mathsf{oke} & : & \Pi s{:}\mathsf{tp}.\, \mathsf{ok}_{(\mathsf{e}\, s)} \\ \mathsf{okx} & : & \Delta\alpha\, \beta.\, \mathsf{ok}_\alpha \to \mathsf{ok}_\beta \to \mathsf{ok}_{\alpha \times \beta} \end{array}$$

We would then replace the old typing for let by

$$\mathsf{let} \quad : \quad \Delta\alpha.\, \mathsf{ok}_\alpha \to \Pi t{:}\mathsf{tp}.\, (\alpha \to \mathsf{e}\, t) \to \alpha \to \mathsf{e}\, t$$

[3] For uniformity, we might like to allow $n = 0$. This would be neatly handled by introducing the unit type.

Example 7.6 *Returning to Example 7.4, the expression*

$$\text{“}\mathbf{let}\ n = 1, b = \mathbf{true}\ \mathbf{in}\ (b\ \mathbf{and}\ n > 0)\text{”}$$

is now represented as

$$\begin{array}{l}\mathsf{let}_{\mathsf{e\,int}\times\mathsf{e\,bool}}\ (\mathsf{okx}_{\mathsf{e\,int},\mathsf{e\,bool}}\ (\mathsf{ok\,e\,int})\ (\mathsf{ok\,e\,bool}))\\ \qquad \mathsf{bool}\\ \qquad (\lambda(n,b){:}\,\mathsf{e\,int}\times\mathsf{e\,bool}.\ \mathsf{and}\ b\ (\mathsf{gtr}\ n\ (\mathsf{num}\ 0)))\\ \qquad (\mathsf{num}\ 1, \mathbf{true})\end{array}$$

As discussed in [44] in reference to the representation of quantifier-freeness and interference, all proofs of a given syntactic judgment (*i.e.*, terms of the representing type) are convertible, and moreover could be constructed automatically. This is significantly different from the process of *term inference* defined in Section 4.7 and illustrated in the next section, where unification is sufficient for inferring terms. Automatically proving syntactic judgments requires searching for constants of relevant types and recursively trying to construct their argument terms. A very elegant framework for this kind of automatic proof term construction is provided by Pfenning's programming language *Elf*, which "unifies logic definition (in the style of LF) with logic programming (in the style of λProlog)" [58]. (As described in [58], Elf does not allow polymorphism. However, the implementation currently under development at Carnegie Mellon University is based on our prototype implementation of $\mathrm{HOU}_{\Pi\delta}$ ($\mathrm{HOU}_{\Pi\Sigma\delta}$ without Σ types), and so does allow polymorphism.)

We can use a similar syntactic judgment for restricting the type argument of **rec**:

$$\mathsf{okr} \ : \ \mathsf{Type}\to\mathsf{Type}$$

The restriction we want to enforce here is that a given type is of the form $\alpha_1 \times \cdots \times \alpha_m$, where each α_i is of the form $(\mathsf{e}\,s_1 \times \cdots \times \mathsf{e}\,s_n)\to\mathsf{e}\,t$. Our formalization makes use of the previous syntactic judgment ok:

$$\begin{array}{rcl}\mathsf{okro} & : & \Delta\alpha.\ \mathsf{ok}_\alpha\to\Pi t{:}\,\mathsf{tp}.\ \mathsf{okr}_{\alpha\to\mathsf{e}\,t}\\ \mathsf{okrx} & : & \Delta\alpha\beta.\ \mathsf{okr}_\alpha\to\mathsf{okr}_\beta\to\mathsf{ok}_{\alpha\times\beta}\end{array}$$

The new version of rec is then

$$\mathsf{rec} \ : \ \Delta\alpha.\ \mathsf{okr}_\alpha\to(\alpha\to\alpha)\to\alpha$$

7.4 Object-language Type Checking and Inference

The example language encodings in the previous section contain a lot of information that is not directly present in the concrete syntax. This extra information is in the form of arguments of type tp, *i.e.*, object-types. Fortunately, these arguments can be synthesized automatically by using the term inference algorithm defined in Section 4.7. The following very simple example illustrates how and why this process works.

Example 7.7 *Consider the expression "*$1 :: \mathbf{nil}$*". We begin by constructing a "partial representation", in which the object-type components are just new distinct unification variables of type* tp*:*

$$\mathsf{cons}\, s_1\, (\mathsf{num}\, 1)\, (\mathsf{nil}\, s_2)$$

We then apply term inference, which, recall, is a combination of type-checking and unification. In this process, we discover the typings

$$\begin{array}{l} \vdash_\Sigma \mathsf{cons}\, s_1 \in \mathsf{e}\, s_1 \to \mathsf{e}\, (\mathsf{list}\, s_1) \to \mathsf{e}\, (\mathsf{list}\, s_1) \\ \vdash_\Sigma \mathsf{num}\, 1 \in \mathsf{e}\, \mathsf{int} \end{array}$$

so we unify $(\mathsf{e}\, s_1)$ *with* $(\mathsf{e}\, \mathsf{int})$*. Next the process discovers that*

$$\begin{array}{l} \vdash_\Sigma \mathsf{cons}\ \mathsf{int}\ (\mathsf{num}\, 1) \in \mathsf{e}\, (\mathsf{list}\, \mathsf{int}) \to \mathsf{e}\, (\mathsf{list}\, \mathsf{int}) \\ \vdash_\Sigma \qquad\qquad\qquad \mathsf{nil}\, s_2 \in \mathsf{e}\, (\mathsf{list}\, s_2) \end{array}$$

and so we unify $\mathsf{e}\, (\mathsf{list}\, \mathsf{int})$ *with* $\mathsf{e}\, (\mathsf{list}\, s_2)$*. We then instantiate our original encoding to*

$$(\mathsf{cons}\, \mathsf{int}\, (\mathsf{num}\, 1)\, (\mathsf{nil}\, \mathsf{int}))$$

which has type $(\mathsf{list}\, \mathsf{int})$*.*

The previous example required only very simple first-order unifications. An object-language whose type inference problem requires truly higher-order unification is the polymorphic λ-calculus [25, 24, 65, 45], which we will refer to as "λ_Δ".[4] The undecidability of this type inference problem for even the second-order polymorphic λ-calculus was shown by Boehm in [4], by reducing it to second-order unification. In [59], Pfenning shows a more general converse, namely that partial type inference for the nth order polymorphic λ-calculus reduces to nth order unification. He then gives an implementation in λProlog, based on an encoding of λ_Δ in $\lambda_{\to\delta}$ ($\lambda_\to$ plus implicit polymorphism). If instead we encode λ_Δ in $\lambda_{\Pi\delta}$, we can use $\lambda_{\Pi\delta}$ term and type inference to do λ_Δ type inference. (To encode the second-order polymorphic λ-calculus, meta-language polymorphism is not necessary, so λ_Π term inference suffices.)

7.5 Program Transformation

Now that we have explored representing languages, we want to construct and apply program transformation rules. To do so, we will also need to introduce concrete syntax for

[4]Note that we mean what is sometimes referred to as "partial type inference", in which we are only allowed to fill in types, but not construct new type abstractions and applications.

the λ-calculus abstraction, application, and pairing. We will use the following additional expressions:

$$\begin{array}{rcl} e & ::= & \lambda v_1, \ldots, v_n.\, e \\ & | & e_1[e_2] \\ & | & e_1, e_2 \end{array}$$

We express the concrete syntax of rules in the form

$$\text{“}LHS\text{”} \quad \Leftrightarrow \quad \text{“}RHS\text{”}$$

and their meta-language representation as

$$\Gamma \vdash M \;\Leftrightarrow\; N$$

where Γ provides types for the variables in M and N, which are the representations of *LHS* and *RHS*.

Let Conversion

We start with the rule of let-conversion. It is simply

$$\text{“}\mathbf{let}\ x = a\ \mathbf{in}\ b[x]\text{”} \quad \Leftrightarrow \quad \text{“}b[a]\text{”}$$

which is represented as

$$[\,\alpha{:}\mathsf{Type}\ ,\ q{:}\mathsf{ok}_\alpha\ ,\ s{:}\mathsf{tp}\ ,\ b{:}\alpha{\to}\mathsf{e}\,s\ ,\ a{:}\alpha\,] \vdash \mathsf{let}_\alpha\, q\, s\, (\lambda x{:}\alpha.\, b\, x)\, a \;\Leftrightarrow\; b\, a$$

We can match the left hand side of this rule against, for example, the term from Example 7.6 giving the substitution

$$\begin{array}{l} [\,\mathsf{e\,int} \times \mathsf{e\,bool}/\alpha\ , \\ \ (\mathsf{okx}_{\mathsf{e\,int},\mathsf{e\,bool}}\,(\mathsf{oke\,int})\,(\mathsf{oke\,bool}))/q\ , \\ \ \mathsf{bool}/s\ , \\ \ (\lambda(n,b){:}\mathsf{e\,int} \times \mathsf{e\,bool}.\ \mathsf{gtr}\,n\,(\mathsf{num}\,0))/b\ , \\ \ (\mathsf{num}\,1, \mathbf{true})/a\,] \end{array}$$

We could even apply this rule from right to left, but we would have to specialize it first to reduce the nondeterminism to a manageable level. For instance, we could instantiate the type variable α to e, saying that we want to introduce one variable in the **let**. (Leaving s uninstantiated, we leave open the object-type of the introduced variable.) Matching this

specialization of the RHS against the term $2 + 3 * 2$ gives several substitutions, resulting in the following corresponding instances of the LHS:

$$\begin{array}{l}
\textbf{let } x = 2 + 3 * 2 \textbf{ in } x \\
\textbf{let } x = 2 \textbf{ in } x + 3 * x \\
\textbf{let } x = 2 \textbf{ in } x + 3 * 2 \\
\textbf{let } x = 3 * 2 \textbf{ in } 2 + x \\
\textbf{let } x = 3 \textbf{ in } 2 + x * 2 \\
\textbf{let } x = 2 \textbf{ in } 2 + 3 * x \\
\textbf{let } x = a \textbf{ in } 2 + 3 * 2
\end{array}$$

Of course this rule assumes a call-by-name semantics. A call-by-value version would have an attached semantic condition that a terminates or b is strict. Otherwise the rule can gain termination applied from left to right.

Context Propagation

Let us now return to the example from Section 7.1.3:

$$\text{“}c[\textbf{if } p \textbf{ then } a \textbf{ else } b]\text{”} \quad \Leftrightarrow \quad \text{“}\textbf{if } p \textbf{ then } c[a] \textbf{ else } c[b]\text{”}$$

Using higher-order abstract syntax, the variable c simply becomes a second-order variable. Higher-order matching ensures that bound variables cannot leave their scope. The representation of this rule is

$$[\, s{:}\mathsf{tp}\ ,\ t{:}\mathsf{tp}\ ,\ a{:}\mathsf{e}\,s\ ,\ b{:}\mathsf{e}\,s\ ,\ p{:}\mathsf{e}\,\mathsf{bool}\ ,\ c{:}\mathsf{e}\,s{\to}\mathsf{e}\,t\,] \vdash c\,(\mathsf{ite}\,s\,p\,a\,b) \ \Leftrightarrow\ \mathsf{ite}\,s\,p\,(c\,a)\,(c\,b)$$

On the other hand, there are similar conversions that we would like to do, but which are not covered by this rule. For instance, it is correct to transform the expression

```
let x = y * y in
  let z = x * y in
    if y > 0 then z else x
```

into

```
if y > 0
  then let x = y * y in let z = x * y in z
  else let x = y * y in let z = x * y in x
```

This does not match the context propagation rule from above, since a substitution like $[\, z/a \,]$ would be captured by the binding on z. A general solution in a case like this is to *raise* the order of the rule through explicit abstraction. This solution is inspired by Paulson's $\forall$-lifting [56], which was discovered independently by Miller and called raising [48]. Both increase the order of some of the variables involved. Raising requires that we be able to explicitly mention the "l" of the l-calculus representation in the pattern. The following is a raised version of context propagation.

$$\text{“}c[\lambda u.\, \textbf{if } p \textbf{ then } a[u] \textbf{ else } b[u]]\text{”} \quad \Leftrightarrow \quad \text{“}\textbf{if } p \textbf{ then } c[\lambda u.\, a[u]] \textbf{ else } c[\lambda u.\, b[u]]\text{”}$$

A match of this pattern against its motivating example is given through the substitution

$$[\, \mathsf{eint} \times \mathsf{eint}/\alpha \;,\; y > 0/p \;,\; \lambda u, v.\, v/a \;,\; \lambda u, v.\, u/b \;,\; \lambda f.\, \text{“}\textbf{let } x = y * y \textbf{ in let } z = x * y \textbf{ in } f[x, y]\text{”}/c \,]$$

Again note the use of polymorphism in the rule description, and the instantiation of type variables to product types, to capture the fact that this rule should apply to any number of bound variables that may appear in the branches of the **if**, but not in the test p.

This also illustrates how variable occurrence conditions become unnecessary. The fact that p could not depend on any variable bound in the context is implicit in the formulation of the rule.

Condition Propagation

Another useful rule that is handled easily in our framework is that of *condition propagation*: when evaluating the **then** branch of a conditional expression, we know that the test succeeded, and when evaluating the **else** branch, we know it failed. We can use this to simplify the subexpressions.

The way we use this idea in an expression "**if** p **then** a **else** b" is to replace some instances of p in a by **true** and some instances of p in b by **false**.[5] The higher-order term structure lets us do this very naturally:

$$\text{“}\textbf{if } p \textbf{ then } a[p] \textbf{ else } b[p]\text{”} \quad \Leftrightarrow \quad \text{“}\textbf{if } p \textbf{ then } a[\mathsf{true}] \textbf{ else } b[\mathsf{false}]\text{”}$$

Unfolding

The *unfold* rule, described in [5], transforms a set of mutually recursive function definitions, by replacing a call to one of the functions by that function's body, with formal parameters

[5] This formulation is due to Tim Freeman.

replaced by actual parameters. The rule below is a very general version of unfold that can simultaneously unfold any subset of the defined functions at any subset of their calling instances. In practice, we would probably want to specialize this rule before applying it. As always, the validity of all specializations are insured by the validity of the general rule. In concrete syntax, it is

$$\text{“}\mathbf{rec}\ f = b[f][f]\text{”} \quad \Leftrightarrow \quad \text{“}\mathbf{rec}\ f = b[b[f][f]][f]\text{”}$$

(In a call-by-value semantics, this rule might gain termination.) The way this works is that the uses of all of the functions (represented *en masse* by f) will be partitioned in any match between the first and second argument of b. Its representation is

$$[\,\alpha{:}\mathsf{Type}\ ,\ q{:}\mathsf{ok}_\alpha\ ,\ b{:}\alpha{\to}\alpha{\to}\alpha\,] \vdash \mathsf{rec}_\alpha\, q\,(\lambda f{:}\alpha.\, b\, f\, f) \ \Leftrightarrow\ \mathsf{rec}_\alpha\, q\,(\lambda f{:}\alpha.\, b\,(b\, f\, f)\, f)$$

As an example, consider the following program, which arises partway into the derivation of an efficient program for list-reversal. We use “$l \mathbin{@} l'$” for the result of appending the lists l and l', and “$a :: l$” for the new list made up of a followed by the elements of the list l.

$$\begin{aligned}&\mathbf{rec}\ rev(u) = \mathbf{if}\ \mathsf{null}(u)\ \mathbf{then}\ \mathsf{nil}\ \mathbf{else}\ rev(\mathsf{tl}(u)) \mathbin{@} (\mathsf{hd}(u) :: \mathsf{nil})\ ,\\ &\quad ra(u,v) = rev(u) \mathbin{@} v\end{aligned}$$

The representation of this program is

$$\begin{aligned}&\mathsf{rec}_A\, M\, (\lambda(rev, ra){:}A.\ (\lambda u{:}\mathsf{e}.\, \mathsf{ite}\,(\mathsf{null}\, u)\,\mathsf{nil}\,(\mathsf{app}\,(rev\,(\mathsf{tl}\, a))(\mathsf{cons}\,(\mathsf{hd}\, u)\mathsf{nil}))),\\ &\qquad\qquad (\lambda u{:}\mathsf{e}, v{:}\mathsf{e}.\, \mathsf{app}\,(rev\, u) v))\end{aligned}$$

Where

$$\begin{aligned}A &= (B{\to}B) \times (B \times B{\to}B)\\ B &= \mathsf{e}\,(\mathsf{list}\, s)\\ M &= \mathsf{okrx}_{(B\to B),(B\times B\to B)}\,(\mathsf{okro}_B\,(\mathsf{oke}\, s)\, s)\,(\mathsf{okro}_{B\times B}\,(\mathsf{okx}_{B,B}\,(\mathsf{oke}\, s)\,(\mathsf{oke}\, s))\, s)\end{aligned}$$

and s is a variable of type tp. We can tell at once that the type part of the unifying substitution must be

$$[\,(B{\to}B) \times (B \times B{\to}B)/\alpha\,]$$

There are two applications of *rev* and none of *ra*. There are thus four possible rewritings, depending on whether each of the two calls is unfolded. The one that unfolds just the second call gives the new program

$$\begin{aligned}&\mathbf{rec}\ rev(u) = \mathbf{if}\ \mathsf{null}(u)\ \mathbf{then}\ \mathsf{nil}\ \mathbf{else}\ rev(\mathsf{tl}(u)) \mathbin{@} (\mathsf{hd}(u) :: \mathsf{nil})\ ,\\ &\quad ra(u,v) = (\mathbf{if}\ \mathsf{null}(u)\ \mathbf{then}\ \mathsf{nil}\ \mathbf{else}\ rev(\mathsf{tl}(u)) \mathbin{@} (\mathsf{hd}(u) :: \mathsf{nil})) \mathbin{@} v\end{aligned}$$

We can then simplify the body of *ra* using simplification rules for lists.

7.5.1 Subterm Rewriting

A very useful ability in program transformation and theorem proving is rewriting one or more subexpressions of a given expression according to a set of simplification rules. For example, simple program derivations are often unfolding, followed by simplification, followed by folding. Similarly, simple proofs are often induction accompanied by simplification [55]. Higher-order abstract syntax and unification provides a simple way to do such subexpression rewriting. Suppose our simplification rule set consists of the following facts about list appending:

$$\begin{array}{rcl}(a :: l) \mathbin{@} l' & = & a :: (l \mathbin{@} l') \\ \mathbf{nil} \mathbin{@} l & = & l\end{array}$$

We can make these into a single rule, capable of rewriting subexpressions as follows:

$$\text{“}f[(a :: l) \mathbin{@} l'][\mathbf{nil} \mathbin{@} l'']\text{”} \quad \Leftrightarrow \quad \text{“}f[a :: (l \mathbin{@} l')][l'']\text{”}$$

Note that we had to rename l to l'' from the second simplification rule to avoid interference with the first.

This method works, but it has two drawbacks. The first is that it cannot rewrite different subexpressions of the same LHS pattern. For instance, in the expression

$$\text{“}\mathsf{length}(\mathbf{nil} \mathbin{@} u) - \mathsf{length}(\mathbf{nil} \mathbin{@} v)\text{”}$$

we could rewrite “$\mathbf{nil} \mathbin{@} u$” to “$u$” or “$\mathbf{nil} \mathbin{@} v$” to “$v$”, but not both. This problem is not very serious, since we could just do the subexpression rewriting repeatedly. The second problem, however, is more serious, and is related to the problem with the simple version of context propagation for conditionals (Section 7.1.3). Consider an expression like

$$\text{“}\mathbf{let}\ u = 1 :: 2 :: \mathbf{nil}\ \mathbf{in}\ \mathbf{nil} \mathbin{@} u\text{”}$$

Our rule above cannot match this expression nontrivially because the subexpression “$\mathbf{nil} \mathbin{@} u$” contains a variable, u, that is bound between the top of this expression and the top of the subexpression. The solution to both problems is, again, to use a third-order rule:

$$\text{“}f[\lambda a.\, \lambda l.\, \lambda l'.\, (a :: l) \mathbin{@} l'][\lambda l.\, \mathbf{nil} \mathbin{@} l]\text{”} \quad \Leftrightarrow \quad \text{“}f[\lambda a.\, \lambda l.\, \lambda l'.\, a :: (l \mathbin{@} l')][\lambda l.\, l]\text{”}$$

We call this the *raising* of the two first-order rules.[6] To see how a rule like this works, consider matching its LHS against the expression

$$\text{“}\mathbf{nil} \mathbin{@} (\mathbf{let}\ u = \mathbf{nil} \mathbin{@} (1 :: 2 :: \mathbf{nil})\ \mathbf{in}\ (1 :: u) \mathbin{@} (4 :: \mathbf{nil}))\text{”}$$

We would get eight possible values for f, one of which is

$$\text{“}\lambda g.\, \lambda h.\, h[\mathbf{let}\ u = h[1 :: 2 :: \mathbf{nil}]\ \mathbf{in}\ g[1][u][4 :: \mathbf{nil}]]\text{”}$$

[6] The representation of this rule would require f to have three additional `tp` arguments, which we leave implicit in the concrete syntax.

which would cause the RHS to be instantiated to

$$\text{“}\mathbf{let}\ u = 1 :: 2 :: \mathbf{nil}\ \mathbf{in}\ 1 :: (u \mathbin{@} 4 :: \mathbf{nil})\text{”}$$

The other seven unifiers result in less simplification. The last of these is a trivial unifier that leaves the expression unchanged.

Clearly, these raised rules can be highly nondeterministic. In practice, we would want to interactively specialize them before application.

7.5.2 Generalized Rewriting via Unification

The conventional view of rewriting, as matching against a rule's LHS, followed by substitution into its RHS, is subsumed and generalized by unification. For example, take the rule of associativity of addition:

$$\text{“}(x + y) + z\text{”} \quad \Leftrightarrow \quad \text{“}x + (y + z)\text{”}$$

We can internalize the rewriting relation, $\Leftrightarrow$, into the expression language and use the single expression

$$\text{“}(x + y) + z \Leftrightarrow x + (y + z)\text{”}$$

Then, to rewrite a given expression, say “$(3 + 4) + 5$”, we unify our rule expression against the expression

$$\text{“}(3 + 4) + 5 \Leftrightarrow z\text{”}$$

where z is a new variable that will get bound by unification to the desired result.

In this simple use of unification, the second expression is always of the form “$e_1 \Leftrightarrow e_2$” where e_1 is completely instantiated and e_2 is completely uninstantiated (*i.e.*, a variable). But now we have the freedom to make e_1 and e_2 instantiated or uninstantiated to any degree. Taking the opposite extreme (e_1 uninstantiated and e_2 instantiated) is equivalent to using a rewrite rule backwards.

One possibility this approach suggests is doing transformation on *program schema*, *i.e.*, programs with free variables that are subject to instantiation. Then the two-way nature of unification, as opposed to matching, would allow the application of a transformation or simplification rule to partially instantiate the program it is transforming as well as the rule it is using. A natural example of how program schema come into existence is the rule of case-introduction:

$$\text{“}w \Leftrightarrow \mathbf{if}\ p\ \mathbf{then}\ w\ \mathbf{else}\ w\text{”}$$

Note that the variable p is not mentioned on the LHS. This rule is useful when the computation of a expression can be optimized in the presence of an assertion p or its negation. (Interestingly, this rule turns out to be a instance of the context propagation rule

$$\text{“}c[\mathbf{if}\ p\ \mathbf{then}\ a\ \mathbf{else}\ b] \Leftrightarrow \mathbf{if}\ p\ \mathbf{then}\ c[a]\ \mathbf{else}\ c[b]\text{”}$$

by taking c to be $\lambda x.\, w$ for a variable w.)

7.6 Theorem Proving

Another general area of application for the unification algorithms developed in the preceding chapters is theorem proving in various logics. In [30], the *Logical Framework* (*LF*) is presented as a "first step towards a general theory of interactive proof checking and proof construction." The key ingredients are the calculus λ_Π and the *judgments as types* principle, forming the basis of a very elegant and expressive system encompassing the syntax, rules, and proofs for a wide class of *object-logics*. Our unification algorithms allow us to go beyond purely interactive theorem proving, to do automated (and semi-automated) theorem proving in LF encoded logics. These theorem provers could be expressed, *e.g.*, in a λProlog based on λ_Π (or $\lambda_{\Pi\Sigma}$ or $\lambda_{\Pi\Sigma\delta}$), or in Elf, a language for logic definition and verified meta-programming [58].

Bibliography

[1] Peter B. Andrews. Theorem proving via general matings. *Journal of the ACM*, 28:193–214, 1981.

[2] Arnon Avron, Furio A. Honsell, and Ian A. Mason. *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* Technical Report ECS-LFCS-87-31, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

[3] Hendrik P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics.* North-Holland, 1980.

[4] Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345, IEEE, October 1985.

[5] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[6] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. *HOPE: an Experimental Applicative Language.* Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, Edinburgh, U.K., 1981.

[7] Luca Cardelli. Basic polymorphic typechecking. *Polymorphism Newsletter*, 1986.

[8] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[9] A. Colmerauer, H. Kanoui, and M. van Caneghem. *Un Systeme de Communication Homme-machine en Francais.* Research Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1973.

[10] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[11] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[12] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL85*, Springer-Verlag LNCS 203, 1985.

[13] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[14] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[15] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980.

[16] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. In Alberto Maria Segre, editor, *Sixth International Workshop on Machine Learning*, pages 447–449, Morgan Kaufmann Publishers, San Mateo, California, June 1989. Expanded version available as Technical Report CMU–CS–89–160, Carnegie Mellon University, Pittsburgh.

[17] Michael R. Donat and Lincoln A. Wallen. Learning and applying generalised solutions using higher order resolution. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 41–60, Springer-Verlag LNCS 310, Berlin, May 1988.

[18] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136, Springer-Verlag LNCS 355, April 1989.

[19] Conal Elliott and Frank Pfenning. A Standard ML implementation of extended higher-order unification, λProlog and Elf. 1990. Send mail to fp@cs.cmu.edu on the Internet for further information.

[20] François Fages and Gérard Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(2,3):189–200, 1986.

[21] William M. Farmer. *A Unification Algorithm for Second-Order Monadic Terms*. Technical Report, Mitre Corporation, Bedford, Massachusetts, June 1986. To appear in the Journal of Pure and Applied Logic.

[22] Amy Felty. *Implementing Theorem Provers in Logic Programming*. Technical Report MS-CIS-87-109, University of Pennsylvania, Philadelphia, December 1987.

[23] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International*

Conference on Automated Deduction, Argonne, Illinois, pages 61–80, Springer-Verlag LNCS 310, Berlin, May 1988.

[24] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[25] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, North-Holland Publishing Co., Amsterdam, London, 1971.

[26] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[27] J. R. Guard. *Automated Logic for Semi-Automated Mathematics*. Scientific Report 1, AFCRL 64–411, 1964.

[28] John J. Hannan. *Proof-theoretic Methods for Analysis of Functional Programs*. Technical Report MS-CIS-89-07, University of Pennsylvania, Philadelphia, January 1989. Dissertation Proposal.

[29] Robert Harper. *Standard ML*. Technical Report ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[30] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. January 1989. Submitted for publication. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[31] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Letrres de Varsovic*, 33, 1930.

[32] W. A. Howard. The formulae-as-types notion of construction. Unpublished manuscript, 1969. Reprinted in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980.

[33] Gérard Huet. Formal structures for computation and deduction. May 1986. Lecture notes for a graduate course at Carnegie Mellon University.

[34] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[35] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

[36] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[37] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[38] D. C. Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, 3:123–171, 1976.

[39] Kevin Knight. Unification: a multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[40] C. L. Lucchesi. *The Undecidability of the Unification Problem for Third Order Languages*. Report CSRR 2059, University of Waterloo, Waterloo, Canada, 1972.

[41] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Lanugaes and Systems*, 4(2):258–282, April 1982.

[42] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175, North-Holland, 1980.

[43] Per Martin-Löf. *On the Meanings of the Logical Constants and the Justifications of the Logical Laws*. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[44] Ian A. Mason. *Hoare's Logic in the LF*. Technical Report ECS-LFCS-87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

[45] Nancy McCracken. The typechecking of programs with implicit type structure. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 301–315, Springer-Verlag LNCS 173, 1984.

[46] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Anual Meeting of the Associtation for Computational Linguistics*, pages 247–255, 1986.

[47] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1988. To appear. Available as Ergo Report 88–055, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[48] Dale A. Miller. Unification under mixed prefixes. 1987. Unpublished manuscript.

[49] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, July 1986.

[50] Dale A. Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Symposium on Logic Programming, San Francisco*, IEEE, September 1987.

[51] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[52] B. Möller. *A survey of the project CIP: Computer-aided, intuition-guided programming.* Technical Report TUM–18406, Institut für Informatik der TU München, Munich, West Germany, 1984.

[53] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, 1987.

[54] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.

[55] Lawrence Paulson. A higher-order implementation of rewriting. *Science of Computer Programing*, 3:119–149, 1983.

[56] Lawrence Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[57] Lawrence C. Paulson. *The Representation of Logics in Higher-Order Logic.* Technical Report 113, University of Cambridge, Cambridge, England, August 1987.

[58] Frank Pfenning. Elf: a language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, IEEE, June 1989. Also available as Ergo Report 89–067, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[59] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163, ACM Press, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[60] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[61] Randy Pollack. The theory of LEGO. October 1988. Unpublished manuscript and documentation.

[62] Garrel Pottinger. The Church-Rosser theorem for the typed λ-calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, July 1981.

[63] Garrel Pottinger. Proof of the normalization and Church-Rosser theorems for the typed λ-calculus. *Notre Dame Journal of Formal Logic*, 19(3):445–451, July 1978.

[64] D. J. Pym. *Proof, Search and Computation in General Logic.* PhD thesis, University of Edinburgh, 1990. forthcoming.

[65] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, Springer-Verlag LNCS 19, New York, 1974.

[66] J. A. Robinson. Computational logic: the unification computation. *Machine Intelligence*, 6:63–72, 1971.

[67] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[68] Anne Salvesen. *The Church-Rosser Theorem for LF with $\beta\eta$ Reduction.* Technical Report forthcoming, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[69] Wayne Snyder. *Complete Sets of Transformations for General Unification.* PhD thesis, University of Pennsylvania, 1988.

[70] Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, 1988. To appear in the special issue on unification.

[71] Guy L. Steele. *Common Lisp: The Language.* Digital Press, 1984.

[72] Guy Lewis Steele and Gerald Jay Sussman. *The Revised Report on SCHEME—A Dialect of LISP.* AI Memo 452, MIT, Cambridge, January 1978.

[73] Anne S. Troelstra. Strong normalization for typed terms with surjective pairing. *Notre Dame Journal of Formal Logic*, 27(4):547–550, October 1986.

[74] David A. Turner. Miranda: a non-strict functional lanugage with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Springer-Verlag, Berlin, September 1985.

[75] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–743, October 1976.

Glossary

λ_Π	Dependent calculus without products or polymorphism, 9
M, N	Meta-variables ranging over terms, 9
A, B	Meta-variables ranging over types, 9
K	Meta-variable ranging over kinds, 10
U	Meta-variable for terms and types, and occasionally kinds, 10
$\mathcal{F}(U)$	Set of free variables in U, 10
λ_Π^V	Terms, types, and kinds with free variables in V, 11
V	Set of variables, 11
$\Theta_V^{V'}$	Substitutions from V to V', 11
θ	Meta-variable ranging over substitutions, 11
Θ_V	The set of substitutions over variable set V, 11
θ_V^{id}	Identity substitution over V, 11
θ^{+u}	Extension of θ to map u to u, 12
$\overline{\theta}$	Extension of θ to terms, types, and kinds, 12
$\theta' * \theta$	Composition of substitutions, 12
$[\, M_1/x_1, \ldots, M_m/x_m \,]_V^{V'}$	Substitution specified by images of a subset of its domain, 15
β, η	Conversion rules for λ_Π, 17
$\rightarrow_\rho$	The one step subterm rewriting extension of ρ, 17
$\rightarrow_\rho^*$	Transitive closure of $\rightarrow_\rho$, 18

$\leftrightarrow_\rho^*$	Equivalence closure of $\rightarrow_\rho$, 18
$=_\lambda$	Convertibility, 18
$\langle\, \mathsf{c}_1{:}U_1, \ldots, \mathsf{c}_n{:}U_n \,\rangle$	Signature, 20
$\mathsf{dom}(\Sigma)$	Domain of a signature, 20
$\Sigma \oplus v{:}A$	Signature extension, 20
$[\, v_1{:}A_1, \ldots, v_n{:}A_n \,]$	Context, 21
$\mathsf{dom}(\Gamma)$	Domain of a context, 21
$\mathsf{ran}(\Gamma)$	Range of a context, 21
$\Gamma \oplus v{:}A$	Context extension, 21
$\vdash \Sigma$ sig	Σ is a valid signature, 21
$\vdash_\Sigma \Gamma$ context	Γ is a valid context given Σ, 21
$\Gamma \vdash_\Sigma K$ kind	K is a valid kind given Σ and Γ, 21
$\Gamma \vdash_\Sigma A \in K$	A has kind K given Σ and Γ, 21
$\Gamma \vdash_\Sigma M \in A$	M has type A given Σ and Γ, 21
$\Theta_\Gamma^{\Gamma'}$	The set of well-typed substitutions from Γ to Γ', 28
Θ_Γ	The set of well-typed substitutions over Γ, 28
P	Disagreement pair, 29
Ψ	Universal context, 29
$\langle \Psi\ ,\ U,\ U' \rangle$	Disagreement pair, 29
D	Disagreement set, 30
Q	Unification problem, 30
$\langle \Gamma, \theta_0, D \rangle$	Unification problem, 30
$\mathcal{A}(Q)$	Q is acceptable, 31
$\mathcal{U}(Q)$	Set of solutions of Q, 31
$\hat{\theta}$	Potential solution of a unification problem, 31
μCSP	Minimal complete set of pre-unifiers, 32

ρ^{**}	Transformation relation closure, 34
wh_β	β weak head reduction, 37
WHNF	Weak head normal form, 38
$\theta\Psi, \theta\Gamma$	Substitution applied to context, 39
$\mathcal{F}(P)$	Free variables of a disagreement pair, 39
$\mathcal{F}(D)$	Free variables of a disagreement set, 39
$\theta P, \theta D$	Substitution applied to disagreement pair or disagreement set, 39
$\mathrm{eq}_\lambda(P)$	P relates convertible terms or types, 39
$\mathsf{size}(\ldots)$	Size of a term, type, *etc.*, 40
$\mathsf{dn}(\ldots)$	Measure of distance to weak head normalization, 40
$P \lhd\!\!\cdot \hat{D}$	P is decomposable to $\hat{D}$, 41
$P \rightsquigarrow_{\mathrm{wh}} \hat{D}$	Decomposition via weak head reduction, 42
$P \rightsquigarrow_{\Pi} \hat{D}$	Decomposition based on an abstraction., 42
$P \rightsquigarrow_{\mathrm{rr}} \hat{D}$	P rigidly decomposes to $\hat{D}$, 47
$\hat{D} \oplus P$	Appending to a disagreement sequence, 47
$\mathrm{topeq}(P)$	P relates terms or types with the same top level structure, 47
$U \approx U'$	U and U' have the same top level structure, 47
$D^{\sqsubset}_P$	Members of D that "account for" P, 51
$\sqsubset$	Strict partial order forming an accounting, 51
$\mathrm{head}(U)$	The head of the term or type body U., 54
$\mathsf{height}_\Gamma(M)$	The height of a term., 64
$\theta \gg \theta'$	Substitution height comparison, 65
$\lambda_{\Pi\Sigma}$	λ_Π enriched with Σ types, 79
π_1, π_2, π	Additional conversion rules for $\lambda_{\Pi\Sigma}$, 81
$P \rightsquigarrow_{\Pi\Sigma} \hat{D}$	Decomposition based on an abstraction or pair, 86

Index