Thesis for the Degree of Licentiate of Philosophy

Practical Unification for Dependent Type Checking

Víctor López Juan

CHALMERS

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2020

Practical Unification for Dependent Type Checking
Víctor López Juan

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Printed by Chalmers Print Service
Gothenburg, Sweden 2020

Practical Unification for Dependent Type Checking

Víctor López Juan
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract:
When using popular dependently-typed languages such as Agda, Idris or Coq to write a proof or a program, some function arguments can be omitted, both to decrease code size and to improve readability. Type checking such a program involves inferring a combination of these implicit arguments that makes the program type-correct.

Finding such a combination of implicit arguments entails solving a higher-order unification problem. Because higher-order unification is undecidable, our aim is to infer the omitted arguments for as many programs as possible with a reasonable use of computational resources. The extent to which these goals are achieved affect how usable a dependently-typed proof assistant or programming language is in practice.

Current approaches to higher-order unification are in some cases too inflexible, postponing unification of terms until their types have been unified (Coq, Idris). In other cases they are too optimistic, which sometimes leads to ill-typed terms that break internal invariants (Agda).

In order to increase the flexibility of our unifier without sacrificing soundness, we use the twin types technique by Gundry and McBride. We simplify their approach so that it can be used within an existing type theory without changes to the syntax of terms. We also extend it so that it can handle more classes of constraints. We show that the resulting solutions are correct and unique.

Finally, we implement the resulting unification algorithm on an existing type checker prototype for a smaller variant of the Agda language, developed by Mazzoli and Abel. We make a suitable choice of internal term representation, and use few, if any, example-specific optimizations. We obtain a type-checker which avoids ill-typed solutions, and is also able to handle some challenging examples in time and memory comparable to the existing Agda implementation.

Keywords: dependent types, type checking, unification.

A summary of this thesis was accepted for publication:

Víctor López Juan and Nils Anders Danielsson
Practical Dependent Type Checking using Twin Types
5th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2020)
doi:10.1145/3406089.3409030

Contents

Acknowledgements

I would first like to thank Nisse for his patience throughout this journey, and for his tireless proof-reading and copious feedback: the quality of this work would not be anywhere close without them. I would also like to thank Peter for his support and understanding when things were toughest.

I would like to thank Andreas Abel, Jesper Cockx, Thierry Coquand, Nils Anders Danielsson, Ulf Norell, Fabian Ruch and Andrea Vezzosi for the discussions on the subject of higher-order unification with dependent types, and its implementation in Agda. These discussions helped develop the theoretical underpinnings this work, put this work in a broader historical context, and produced examples which motivated our work and allowed us evaluate our progress. I also wish to thank Francesco Mazzoli for his previous work on the Tog project, and Adam Gundry for his clarifications about his thesis, upon both of which this work builds. Finally, I wish to thank Matthieu Sozeau for his interest in discussing this licentiate thesis.

I would like to dedicate this door stopper to my friends, including Manos, Inari, Fabian, Salvo, Franz, Herbert, Carlos, Irene, Stavros, Andrea, Stefan, and those of you whom I have undeservedly omitted: you have made my time as a PhD student more fulfilling than I could have imagined. And Carol, Borja, Carlos, Felipe and Ralph: when I have travelled back to Madrid, you have always made me feel as if I had never left.

A mis padres, por mantenerse a mi lado pese a lo difícil que se lo pongo.

Och till Henrik, mitt livs glädje.

Chapter 1

Introduction

Dependent type checking is at the heart of implementations of proof assistants and programming languages such as Agda, Idris or Coq. When writing programs and proofs in such tools, omitting information which is unequivocally determined by the surrounding context can make programs both easier to read and to write. These omitted values are replaced by metavariables, which are assigned values in the course of type checking. Inferring values for such metavariables is in general undecidable (see Section 3.3). However, for many specific programs and proofs that arise in practice (e.g. those where some of the resulting constraints are in Miller's pattern fragment [41]), unique solutions can be found.

In this work, we describe an algorithm for performing such an inference, with a focus on its practical implementability.

Following Mazzoli and Abel [36] with some minor optimizations, the entire type checking problem is reduced to a set of dependently-typed, higher-order unification constraints, of the form $\Gamma \vdash t : A \approx u : B$, where $\Gamma \vdash t : A$ (left side) and $\Gamma \vdash u : B$ (right side). These constraints are solved by first instantiating metavariables in such a way that i) A and B become definitionally equal as types, and ii) t and u become definitionally equal as terms.

Mazzoli and Abel [36] only consider constraints to be well-formed when the types of both sides coincide. Thus, the constraint $\Gamma \vdash A : \mathrm{Set} \approx B : \mathrm{Set}$ needs to be solved before $\Gamma \vdash t : A \approx u : B$ can be tackled. This prevents some programs from being type-checked at all (for instance, see Section 5.6).

Inspired by Gundry and McBride's twin types [25], we add rules that can handle constraints where the types of the side are distinct, by allowing each variable in the context to take up to two different types.

The original presentation of twin types requires annotations on variables to indicate which type they take. In our solution, the left (right) side of the constraint only refers to the left (right) type of the variables in the context. This means that the underlying term syntax and type theory remains essentially intact.

By keeping the term syntax and type theory intact, we can rely on common assumptions about the type theory when proving the correctness of our algorithm. Keeping the underlying theory unchanged also makes it easier to adapt an existing type checker to use our unification algorithm. This helps

us to demonstrate that an approach based on twin types can indeed work in practice.

1.1 Our contributions

- A high-level description of rules for higher-order unification (Section 4.5) for a dependent type theory with uninterpreted constants, metavariables, dependent products, dependent sums, and a boolean type (Chapter 2). The rules implement the ideas in Gundry and McBride's twin-type approach [25] without requiring changes to the underlying type theory (Section 4.1). We show that, under some postulates about the underlying type theory (page 216), the solutions produced by our unification rules are correct and unique (Theorem 4.31).

- An implementation of the unification algorithm for Mazzoli et al.'s prototype type-checker for a smaller variant of the Agda language, Tog [37]. This prototype implements a number of key features of Agda, such as induction-recursion, equality type with J, and records with η-equality (Section 5.3).

- Benchmarks of the implementation against programs that are typically challenging for existing approaches, and are not handled by other dependently-typed proof assistants such as Coq or Idris (Section 5.7.1). With a suitable term representation (Section 5.5.1), we can type check challenging examples in time and memory comparable to the existing Agda implementation, while preserving the internal invariants that the current Agda implementation occasionally breaks (Section 5.7.2).

1.2 Problem statement

We want to type-check terms with metavariables in an Agda-like dependently-typed language. Metavariables are stand-ins for terms that have been omitted by the language user. In the course of type checking a well-typed program, the metavariables are replaced by terms (that is, instantiated) in such a way that the resulting program is type-correct.

A program is only deemed type-correct if all metavariables can be instantiated. We are interested only in solutions which are *closed* (i.e. without uninstantiated metavariables) and *unique*.

First, we are interested in closed solutions because they correspond to well-typed programs. Our algorithm is executed stepwise, producing a sequence of intermediate metavariable assignments in which some metavariables are uninstantiated. However, we do not study the theoretical properties of these partial assignments beyond their well-typedness. We assume that the ultimate goal of the interaction will be to produce a closed solution.

The case for uniqueness requires more explanation. In our setting, many metavariables will occur in definitions, be they function bodies or theorem statements. Avoiding non-unique solutions limits the potential for ambiguity in what the define function does, or what theorem is being proved.

Consider the program in Listing 1.1. This pseudocode program creates, manipulates and prints integer vectors with statically-checked lengths. The function `replicate` produces a vector in which all the components have the same value. The first argument to `replicate`, which is implicit, determines the length of the resulting vector and, unless given explicitly, is inferred from the context where the result is used. In the second usage of `replicate` (line 16), the length of the resulting vector could be any natural number. We expect the type checker to ask the user to give the first argument explicitly, and not to fill in an arbitrary term.

Listing 1.1: Non-unique implicit argument

```
1  -- replicate 4 :: Vec 3 Int ≡ [4,4,4]
2  -- replicate {n = 5} 4 ≡ [4,4,4,4,4]
3  -- replicate {n = 0} 4 ≡ []
4  replicate :: {n : Nat} → Vec n Int
5
6  -- rotate90 [1,2] ≡ [-2,1]
7  rotate90 :: Vec 2 Int → Vec 2 Int
8
9  -- print [1,2,3]
10 -- > [1,2,3]
11 print :: {n : Nat} → Vec n Int → IO ()
12
13 main :: IO ()
14 main = do
15   print (rotate90 (replicate 1))   -- > [-1,1]
16   print (replicate 6)              --   (?)
```

From the user's point of view, avoiding ambiguity due to non-unique solutions is not only important for the well-defined behaviour of programs, but also in proof relevant settings such as homotopy type theory.

Implementation-wise, avoiding non-unique solutions means that all instantiations of metavariables during type checking are final. In those cases when a program type-checks, the result of the unification is predictable and independent of implementation details; in particular, the order in which constraints are solved. The algorithm can tackle the constraints in the order that the implementer considers most efficient or convenient, and does not need to implement a mechanism for backtracking. Furthermore, as Andreas Abel pointed out (personal communication), making all instantiations final is also helpful in an interactive setting, where backtracking on solutions which have already been output might be confusing to the user.

1.3 A brief history of higher-order unification with dependent types

Higher-order unification is a key component in type checking programs with metavariables. In 1974, Huet [29] described a semi-decision algorithm which given a higher-order unification problem in the simply typed λ-calculus, finds

a unifier if one exists, albeit not necessarily a most general one. This is known as a *pre-unification* algorithm. As shown by Huet [28], because higher-order unification is undecidable, any algorithm which can always find a solution when it exists must in some cases not terminate when a solution does not exist.

In 1989, with Huet's work [29] as a starting point, Elliott [20] shows an algorithm for the λ_Π calculus, only constructs *approximately well-typed* terms, which do at least have long $\beta\eta$ head normal forms. Pym [51] arrives at a similar solution, allowing variables to be substituted for terms of *similar type*, which guarantees the existence of head normal forms. In both cases, this is enough to prevent ill-typed terms from throwing the unification algorithm into a loop. However, the problem of higher-order unification remains undecidable, so the algorithm may not terminate.

In 1990, Elliott [21] extends his previous work into a pre-unification algorithm for calculi with dependent function types (Π-types) and dependent pair types (Σ-types). Higher-order unification with dependent types may result in ill-typed terms, which may not be strongly normalizing and thus cause non-termination. However, in this case, the presence of non-normalizing terms implies that no unifiers exist, in which case non-termination is already one of the expected outcomes of the algorithm.

In 1994, Magnusson [34] implements ALF, a precursor to Agda. Before ALF, all implementations of type theory (among them NuPRL [13], Petersson's system [49] and Coq [15, 18]) would instead have the user build proof-trees using tactics until eventually a complete term is produced [42, section 1.4], in the style of LCF [23]. The complete term (which may or may not be well-typed) would then be type-checked by the proof assistant.

A key innovation in ALF, was the use of metavariables to allow the user to refine the terms themselves interactively. Typechecking of the incomplete terms in ALF is done by using a simplified version of the ideas by Elliott [20] and Pym [51]. The type checker would produce a list of constraints; the user would refine the metavariables until all constraints were satisfied. Once all constraints are satisified, all terms are known to be type correct. An advantage of the ALF approach is that terms are manipulated directly. The user can then have several incomplete terms simultaneously, add new definitions, and manipulate everything in any order they wish, without needing to type check the terms again.

The underlying type theory of ALF has Π-types, two universe levels Set and Type, inductive data types, explicit substitutions, and metavariables as placeholders for open terms. In contrast with the approaches discussed so far, only first order constraints are solved, and the remainder are postponed. The unification algorithm may introduce terms which are not well-typed in general, but are only well-typed modulo the unsolved constraints. In the case of ALF, it is conjectured [34, section 8.4.1] that all terms involved in the execution of the unification algorithm, even those which are ill-typed, are well-typed in the simply-typed λ-calculus, and thus normalizing, so the unification algorithm will terminate regardless. However, it is not clear how this line of reasoning extends to a full dependent type theory.

In 1997, Cesar Muñoz [42, 43] put Magnusson's [34] approach on a formal footing, in such a way that all intermediate terms are well-typed, including those with placeholders. As in the work by Huet [29], Elliott [20] and Pym [51],

the focus is on the completeness of the algorithm; thus it is still possible that the algorithm may not terminate when a solution does not exist.

Coquand built a dependent type checker called Agda [14], which implements Martin-Löf type theory. Agda was further improved by Norell and Coquand [48], introducing a metavariable solving mechanism based on the one in Epigram [38]. Metavariable solving involves checking the equality of terms. As Norell and Coquand [48] explain, normalizing terms in order to check for equality may make the type checker loop if those terms are only well-typed modulo a set of constraints. With the aim of ensuring normalization even in those cases where the program being type-checked is not well-typed as a whole, potentially ill-typed subterms are replaced by *guarded constants* of an appropriate type. These guarded constants are such that they only reduce to the corresponding subterm once the constraints that ensure the well-typedness of the subterm are solved. According to Norell and Coquand [48], an improvement in Agda with respect to Muñoz's [43] approach is that both sides of each constraint have the same type. This means that no additional type checking is needed when instantiating a metavariable, which may otherwise be costly.

When using higher-order unification to build a type checker for dependent types with implicit arguments, an algorithm that is not complete, but is instead guaranteed to always give an answer (even if it is more often a negative one) may provide for a more predictable user experience. Miller [41] discovered that when the constraints are restricted to a specific form (the pattern fragment), higher-order unification becomes decidable, and thus a terminating algorithm is possible. Even if not all the constraints are in the pattern fragment, one may solve those which are, and postpone the remainder with the hope that they will become part of the pattern fragment when other constraints are solved. This technique is known as *dynamic pattern unification*.

In 2009, Reed [52] showed a terminating algorithm for dynamic pattern unification, and Abel and Pientka [3] extended the dynamic pattern unification technique to handle Σ-types and η-equality. In both cases, the terms are well-typed only modulo the unsolved constraints. Normalization is ensured by limiting how types may depend on terms.

Dynamic pattern unification [52, 3] with guarded constants [48] is used for solving metavariables in the current version of Agda (2.6.1). The guarded constants technique seems effective to ensure normalization of terms, but it has some issues when implemented in practice [33, 16]. In the course of this work we discovered that the first of these issues [33] can be solved using further application of the technique of guarded constants, and proposed a fix to the Agda developers. However, this fix was reverted after Abel et al. [9] found out that it broke some existing code. For the other open issue [16] we are not aware of any fix that does not involve a heterogeneous approach to unification such as the one discussed in this work (but that does not mean that there are none).

1.4 Recent approaches to dependent type checking with metavariables

As Mazzoli and Abel [36] show, elaboration of an Agda-like language (an extension of Martin-Löf type theory) can be completely reduced into a higher-order unification problem with dependent types. Their elaboration algorithm (Algorithm 1) produces constraints of the form $\Gamma \vdash t : A \approx u : B$. Such a constraint becomes solved when both $\Gamma \vdash A \equiv B$ **type** and $\Gamma \vdash t \equiv u : A$.

Because A and B are potentially different types, solving these constraints is more complex than solving those with a single type for both sides ($\Gamma \vdash t \approx u : A$).

For instance, when solving a constraint by assigning a term to a metavariable (e.g. solving $\cdot \vdash \alpha : A \approx t : B$ by assigning $\alpha := t : A$), it is important that the type of the right-hand side (here, B) matches the type of the metavariable (A). Otherwise, when the newly-assigned value of the metavariable is substituted elsewhere, ill-typed terms may be produced, which breaks important invariants of the type checker, possibly resulting in run-time errors [4].

Checking whether the types A and B match is complicated by the fact that the types themselves may contain metavariables. Mazzoli and Abel's [36] implementation unifies the types of both sides of the constraint before working on the constraint itself. That is, first they will solve the constraint $\Gamma \vdash A \approx B : \mathrm{Set}$ and then solve $\Gamma \vdash t \approx u : A$. This may prove too restrictive, as explained in Section 3.9.

Agda [48] avoids creating constraints where the two types are distinct during elaboration, using *guarded constants*. A guarded constant is a definition that is blocked from being unfolded until the constraints that *guard* the body of the constant are solved. These constraints, when solved, ensure that the body of the constant has the same type as the constant itself. For example, instead of producing a constraint $\Gamma \vdash t : A \approx u : B$, it would produce a constraint $\Gamma \vdash t \approx p : A$, using the guarded constant $p : A$ such that $p = u$ when $\Gamma \vdash A \approx B : \mathrm{Set}$.

Gundry and McBride [27] allow constraints with two different types. To work around issues such as the one presented in Section 3.8, they introduce *twin variables*. A twin variable $\hat{x}$ in a context Γ can have two possible types A_1 and A_2, i.e. $\hat{x} : A_1 \ddagger A_2 \in \Gamma$. The type of the variable when it appears in a term depends on which of the two forms of the variable is used: $\Gamma \vdash \acute{x} : A_1$, and $\Gamma \vdash \grave{x} : A_2$.

In all three approaches, the use of the solution to a metavariable is in effect blocked until we can ensure that the types of the metavariable and its potential body match. This aims to guarantee that constraints are always well-formed.

1.5 Uniqueness of solutions

Implementation-wise, finding solutions to metavariables is typically done iteratively, as explained in Section 3.5 (dynamic pattern unification). At each step of the algorithm, one or more metavariables may be instantiated with a term. Giving a value to a metavariable may in turn make further constraints mentioning that metavariable solvable.

A big point of distinction among implementations of unification for dependently-typed languages is the uniqueness of solutions. In our approach, a metavariable may only be instantiated with a term if the metavariable is equal to that term in all possible solutions to the problem. This way, all metavariable assignments are final, and so backtracking is not needed. Because all instantiations are unique (i.e. there are no choices forced by implementation heuristics), this approach is also less sensitive to the order in which constraints are tackled, or how far terms are normalized. Restricting instantiations to unique solutions is particularly advantageous when these metavariables occur in definitions, so that changes in the unification algorithm do not change the statement of a theorem or the behaviour of a program.

Other implementations of dependently-typed calculi, such as Coq or Idris, will produce non-unique solutions. This allows for programs to type-check which otherwise would not. Additionally, as Ziliani and Sozeau [55] explain, tolerating non-unique solutions, when combined with a careful reduction strategy and a strict ordering of constraints, works particularly well with overloading mechanisms such as canonical structures. Furthermore, the lack of unique solutions is not a big issue for those metavariables which occur in proofs of a proof-irrelevant type.

In principle, allowing non-unique solutions makes more unification problems solvable. However, the increase in the problems that can be solved by renouncing uniqueness of solutions is not strict in practice. As we note in Section 5.7.1, the additional rigidity imposed in the ordering of constraints might actually prevent some unification problems from being solved.

1.6 Design choices

We build on the existing unification algorithms by Abel and Pientka [3], Gundry and McBride [25] and Mazzoli and Abel [36]. Together with some modifications of our own, we obtain a type-checker which can handle a range of examples.

We first note that the type theory which we use includes Type : Type as an axiom, which means that not only the theory itself, but also the assumptions that we make regarding the theory (e.g. that all terms have a normal form) are inconsistent. We believe however that these issues are largely orthogonal to the treatment of unification; and thus that our arguments will hold in a properly stratified theory.

In this section we explain the design choices that we made in the course of our work.

Type checking through unification We use Mazzoli and Abel's [36] approach for reducing a type checking problem with metavariables to a higher-order unification problem.

As in Mazzoli and Abel's [36] work, we use metavariables as place-holders for both implicit arguments and for subterms which are not yet known to be of the appropriate type. In the latter case, these metavariables are constrained to be equal to the subterm they are replacing, and will be assigned to this subterm once the types have been established to match.

Each placeholder introduces additional constraints that need to be solved. To reduce this overhead, we add some shortcuts to the Tog elaboration process which avoids inserting these placeholders in some cases where the types are already known to match.

Type theory In the formal description, we use a dependent type theory with Π-types, Σ-types, and booleans. Having booleans in the theory allows us to describe situations where the final form of a type is non-trivially dependent upon a metavariable instantiation (Example 5.3).

Pattern unification for λΠΣ with postponing At its core, the algorithm used is Miller pattern unification with records and postponing of constraints. We follow Abel and Pientka's approach [3] for solving unification problems, including the requirement that solutions to metavariables are unique. The latter choice is motivated in Section 1.5.

Solutions as closed metasubstitutions We consider only those solutions in which all metavariables are instantiated in our correctness proof, following Abel and Pientka's approach [3] of grounding metasubstitutions. How this compares with an approach based on most-general unifiers is described in Section 5.7.3.

Constraint unblocking We extend the blocking mechanism from the original implementation of Tog by Mazzoli and Abel [36], which allows for quickly narrowing down whether a constraint can potentially make progress (Section 5.2.1).

Term representation Term representation affects the performance of term normalization and other aspects of unification. The implementation uses de Bruijn indices, just like both Agda and Tog. The theoretical representation also uses de Bruijn indices in order to stay close to the implementation, although we sometimes denote them with names for clarity (Section 2.7).

On top of the de Bruijn index based representation, we implement hash-consing for terms. This increases the sharing of both the memory used to store the terms, and of the work performed when normalizing them. Hash consing is implemented in a way that is transparent to the unification algorithm.

Term normalization We consider terms in β-normal form, both in the base theory and in the implementation. This keeps us close to existing implementations such as Agda.

Overly eager normalization of terms may have adverse effects on performance. Therefore, like Agda, but unlike Gundry and McBride [25], we allow δ-redexes in terms.

Heterogeneous constraints In order to make more problems solvable, we perform operations on constraints even when the types of the two sides have not been shown to be equal (see Section 3.10 and Chapter 4).

More specifically, we use the twin type approach by Gundry and McBride [25] for unification, but enforcing the additional invariant that

each side of an equality constraint (left or right) only references the corresponding side of the context. In practice, this means that the term syntax does not need to be extended with twin variables. This avoids the need for extending the underlying type theory, reduces visual clutter in the presentation, and saves the implementation cost of removing the annotations on twin variables once an equation finally becomes solvable.

Closed metavariables The types and values of our metavariables are all typed in the empty context. As Ziliani and Sozeau [56] observe, this has the advantage of being easy to implement. However, because metavariables will often appear applied to a series of variables in the context, this leads to unnecessary β-reductions when the solution is substituted in. We follow Mazzoli and Abel [36] in mitigating this issue in the implementation by removing the leading λ-abstractions from the bodies of the metavariables (Section 5.5.1).

A second issue with closed metavariables observed by Ziliani and Sozeau [56] is that, when substituting their values into terms, many unsightly lambdas will occur, unless the term is β-normalized afterwards. This is not a concern for us, as we only consider terms in β-normal form (see "Term representation" above).

Unique solutions As explained in Section 1.5, we choose to restrict our unifier to producing those solutions which are unique. This gives us more freedom in the order in which constraints can be solved, while preserving the suitability of the language for both programs and proofs. The additional freedom in the ordering of constraints allows in practice for more complex unification problems to be solved (Section 5.7). However, due to the restriction to unique solutions, problems with more than one solution will not be solvable. As discussed above, other proof assistants may have heuristics which choose one of those solutions as the more likely to be intended by the user, and are thus able to deal with such unification problems.

These choices constitute only one of the possible approaches to higher-order unification for dependent type checking. In Section 5.7 we evaluate the practical impact of some of these choices by looking at certain existing systems in which these choices were made in certain ways.

1.7 Structure of the thesis

The next chapter (Chapter 2) describes a dependently-typed theory with metavariables and its properties. Chapter 3 discusses pre-existing work on how type checking terms in such a theory corresponds to a higher-order unification problem, and the challenges that arise when performing higher-order unification on dependently-typed terms. Chapter 4 provides a toolkit of rules to tackle these higher-order unification problems. Finally, Chapter 5 discusses how the rules in Chapter 4 can be used to implement a unification algorithm, and evaluates its performance.

Chapter 2

A dependently-typed language

The theory presented below contains the same constructs as Mazzoli and Abel [36], with the addition Σ-types and their corresponding equality rules. The result is a dependent type theory with dependent function and sum types, η-equality, and large elimination. These features allow us to address the issues that we describe in Chapter 3.

Our theory is similar to the one used by Gundry and McBride [25], with the distinction that they specify the recursor for booleans as an eliminator, instead of as a term head.

2.1 Term syntax

The syntax of terms in the language is described in Figure 2.1.

We include some typical constructions from Martin-Löf Type Theory, along with metavariables (which take the place of terms omitted by the user which we hope to infer), and atoms, which are irreducible constants (corresponding to postulates in Agda) that we use to write more interesting examples.

2.2 Notational preliminaries

When discussing unification, it is common to work with lists of variables, terms, and other syntactic constructs. Throughout the document, we use $\vec{t}$ as a succinct way to denote a sequence of elements $t_1, ..., t_n$. Several variations on this notation are described below.

Notation (Vector notation: $\vec{t}$). Vector notation is a shorthand for sequences of possibly-distinct elements sharing a common form:

- $\overrightarrow{\square}$ denotes a sequence of zero or more possibly-distinct elements of the form $\square$. *Example:* $\vec{x}$ denotes sequence of zero or more possibly-distinct variables.

x, y, z	::=		*variables*
X, Y, Z	|	$0, 1, 2, ...$	de Bruijn indices
α, β, γ			*metavariables*
$\mathbb{a}, \mathbb{b}, \mathbb{c},$ $\mathbb{A}, \mathbb{B}, \mathbb{C}$			*atoms*
$t, u, v, r,$ T, U, A, B	::=		*terms and types*
	|	Bool	boolean type
	|	ΠAB	function type
	|	ΣAB	record type
	|	Set	universe
	|	c	data constructor
	|	$\lambda.t$	λ-abstraction
	|	$\langle t, u \rangle$	pair constructor
	|	f	neutral terms
f, g	::=		*neutral terms*
	|	h	elimination head
	|	$f\, e$	eliminator
h	::=		*elimination heads*
	|	$x, X, ...$	variable head
	|	$\alpha, \beta, ...$	metavariable head
	|	$\mathbb{a}, \mathbb{b}, ...$	atom head
	|	if	boolean recursor head
e	::=		*eliminators*
	|	t	term application
	|	$.\pi_1 \quad \| \quad .\pi_2$	projections
c	::=		*data constructors*
	|	true | false	booleans

Figure 2.1: Syntax for terms. Metavariables (α, β, …) and atoms (𝕒, 𝕓, …) are drawn from disjoint and countably infinite sets of names.

- $\overrightarrow{\square}^n$ denotes a sequence of n possibly-distinct elements of the form $\square$. *Note:* When specifying the length of a vector in this way, only one of the occurrences needs to contain the length superscript. For instance, the two sides of the equality $\alpha\,\vec{x}^n = \alpha\,\vec{x}$ denote identical terms.

- $\square_1 \cdots \square_n$ denotes a sequence of n possibly-distinct elements of the form $\square$, numbered from 1 to n.

- $\square \cdots_n \square$ denotes a sequence of n *identical* elements of the form $\square$.

- $\square_1 \diamond \ldots \diamond \square_n$ denotes a sequence of n possibly distinct elements of the form $\square$, such that the operator $\diamond$ is interspersed between each consecutive pair of elements in the vector.

- $\square \diamond \ldots_n \diamond \square$ denotes a sequence of n *identical* elements of the form $\square$ such that the operator $\diamond$ is interspersed between each consecutive pair of elements in the vector.

Remark. Within a given context (e.g. an example, lemma or theorem and its proof, or a single paragraph), vector notations with the same name refer to the same sequences of elements. For example, the two sides of the equality $\alpha\,\vec{x} = \alpha\,\vec{x}$ denote identical terms.

Notation (Neutral terms in vector form: $h\,\vec{e}$). A neutral term can be viewed as a head h followed by a vector $\vec{e}$ of eliminators.

When manipulating neutral terms, we will use the recursive structure in Figure 2.1 and the vector form given above interchangeably.

Notation (Vector elements: t_i). Subindices can be used to pick out individual elements of a vector. Indices start at 1, unless otherwise noted.

- x_i denotes the ith element of a vector $\vec{x}$.

- $x_{i,j}$ denotes the jth element of a vector $\overrightarrow{x_i}$.

Notation (Vector slices: $\vec{t}_{i,\ldots,j}$). Subindices can be used to pick out a sequence of consecutive elements from a vector.

Let $\vec{x}^n$ be a vector of n elements. Then, given i,j such that $1 \leq i \leq j \leq n$, $\vec{x}_{i,\ldots,j}$ is a vector of length $j - i + 1$ whose kth element is the $(i + k - 1)$th element of $\vec{x}$. Whenever $i > j$, the expression $\vec{x}_{i,\ldots,j}$ denotes a vector of length 0.

Notation (Vector membership: $\_ \in \_$). We overload the notation $\in$ for set membership to also denote membership in vectors, or vector-like objects.

For example, Section 2.3 defines signatures, which we consider as a vector-like object. If $\Sigma = \alpha : \mathrm{Bool}, \mathbb{A} : \mathrm{Set}, \mathfrak{a} : \mathbb{A}$, we say that $\alpha : \mathrm{Bool} \in \Sigma$.

Notation (Ungrammatical terms: $\lceil t \rfloor$). We use $\lceil t \rfloor$ to clarify that t is syntactically invalid, or otherwise ill-formed.

Notation (Partial functions: $F \Downarrow y$, $F\Downarrow$, F). In this development, we specify procedures on syntax that may only be well-defined under certain conditions. Some examples are Definition 2.31 (hereditary substitution) and Definition 2.143 (closing metasubstitution).

We specify these procedures as relations $F \Downarrow y$ between two sides F and y, where F is the computation begin defined, and y is the result of the computation (if it exists). The definition is such that there is always at most one y such that $F \Downarrow y$.

We say $F\Downarrow$ if and only if there exists a y such that $F \Downarrow y$. We denote such a y by F itself.

2.3 Signatures (Σ sig)

A signature contains declarations which are available globally. In an extended implementation, it could also include function and data type definitions.

Σ	::=	$\cdot$	empty signature
	$\mid$	$\Sigma, \mathbb{a} : A$	atom
	$\mid$	$\Sigma, \alpha : A$	metavariable declaration
	$\mid$	$\Sigma, \alpha := t : A$	metavariable instantiation

Definition 2.1 (Fresh declaration)**.** We say that $\mathbb{a}$ is fresh in Σ (or, that α is fresh in Σ) if there is no B such that $\mathbb{a} : B \in \Sigma$ (respectively, if there is no B such that $\alpha : B \in \Sigma$ or no t and B such that $\alpha := t : B \in \Sigma$).

Definition 2.2 (Instantiated metavariable, body of a metavariable)**.** When a signature Σ contains an element of the form $\alpha := t : A$, we say that the metavariable α is *instantiated* in the signature Σ. The term t is the *body* of α in this signature.

Definition 2.3 (Uninstantiated metavariable)**.** Conversely, given $\alpha : A \in \Sigma$, we say that α is uninstantiated in Σ if there is no t and B such that $\alpha := t : B \in \Sigma$.

Instantiated metavariables "expand" to their bodies. For example, in a signature containing $\alpha := \lambda x.\lambda y.y : B$ (for some type B), the term $\mathbb{A}\,\alpha$ expands to the term $\mathbb{A}\,(\lambda x.\lambda y.y)$. Section 2.13 gives a full account of computation in terms, which includes metavariable expansion.

Definition 2.4 (Well-formed signature: Σ **sig**)**.** A signature Σ is well-formed (written Σ **sig**) if each declaration is well-typed with respect to the preceding declarations.

$$\frac{}{\cdot\ \textbf{sig}}\ \textsc{empty}$$

$$\frac{\Sigma\ \textbf{sig} \qquad \mathbb{a} \text{ is fresh in } \Sigma \qquad \Sigma; \cdot \vdash A\ \textbf{type}}{\Sigma, \mathbb{a} : A\ \textbf{sig}}\ \textsc{atom-decl}$$

$$\frac{\Sigma\ \textbf{sig} \qquad \alpha \text{ is fresh in } \Sigma \qquad \Sigma; \cdot \vdash A\ \textbf{type}}{\Sigma, \alpha : A\ \textbf{sig}}\ \textsc{meta-decl}$$

$$\frac{\Sigma\ \textbf{sig} \qquad \alpha \text{ is fresh in } \Sigma \qquad \Sigma; \cdot \vdash A\ \textbf{type} \qquad \Sigma; \cdot \vdash t : A}{\Sigma, \alpha := t : A\ \textbf{sig}}\ \textsc{meta-inst}$$

The typing relations $\Sigma; \cdot \vdash A$ **type**, and $\Sigma; \cdot \vdash t : A$ are defined in Section 2.5.

Remark 2.5 (Signature inversion). Let $\Sigma = \Sigma_1, \Sigma_2$, with Σ **sig**. Then Σ_1 **sig**, and:

- If $\Sigma_2 = \mathbb{a} : A, \Sigma_2'$, then $\Sigma_1; \cdot \vdash A$ **type**.
- If $\Sigma_2 = \alpha : A, \Sigma_2'$, then $\Sigma_1; \cdot \vdash A$ **type**.
- If $\Sigma_2 = \alpha := t : A, \Sigma_2'$, then $\Sigma_1; \cdot \vdash A$ **type**. and $\Sigma_1; \cdot \vdash t : A$.

Definition 2.6 (Support of a signature: $\textsc{support}(\Sigma)$)**.** The support of a signature is the set of metavariables that it declares.

$$\begin{array}{rcl} \textsc{support}(\cdot) & = & \varepsilon \\ \textsc{support}(\Sigma, \mathbb{a} : A) & = & \textsc{support}(\Sigma) \\ \textsc{support}(\Sigma, \alpha : A) & = & \textsc{support}(\Sigma) \cup \{\alpha\} \\ \textsc{support}(\Sigma, \alpha := t : A) & = & \textsc{support}(\Sigma) \cup \{\alpha\} \end{array}$$

Notation (Signature concatenation: Σ_1, Σ_2). Signatures can be syntactically viewed as lists. The concatenation of two signatures Σ_1 and Σ_2 is written Σ_1, Σ_2. Note that this is a purely syntactic operation. It is not implied that Σ_2 is well-formed on its own, even if Σ_1 and Σ_1, Σ_2 are.

Definition 2.7 (Atom declarations of a signature: $\textsc{AtomDecls}(\Sigma)$)**.** The atom declarations of a signature Σ (written $\textsc{AtomDecls}(\Sigma)$) are the set of the atoms it declares.

$$\begin{array}{rcl} \textsc{AtomDecls}(\cdot) & = & \emptyset \\ \textsc{AtomDecls}(\Sigma, \mathbb{a} : A) & = & \{\mathbb{a}\} \cup \textsc{AtomDecls}(\Sigma) \\ \textsc{AtomDecls}(\Sigma, \alpha : A) & = & \textsc{AtomDecls}(\Sigma) \\ \textsc{AtomDecls}(\Sigma, \alpha := t) & = & \textsc{AtomDecls}(\Sigma) \end{array}$$

Definition 2.8 (Constants declared by a signature: $\textsc{decls}(\Sigma)$)**.** The constants declared by a signature Σ (written $\textsc{decls}(\Sigma)$) are the metavariables and atoms it declares ($\textsc{decls}(\Sigma) = \textsc{AtomDecls}(\Sigma) \cup \textsc{support}(\Sigma)$).

Remark 2.9 (Atoms and metavariables are disjoint). For any two signatures Σ_1 and Σ_2, we have $\textsc{decls}(\Sigma_1) = \textsc{decls}(\Sigma_2)$ if and only if $\textsc{AtomDecls}(\Sigma_1) = \textsc{AtomDecls}(\Sigma_2)$ and $\textsc{support}(\Sigma_1) = \textsc{support}(\Sigma_2)$.

Definition 2.10 (Metavariables in a term: $\textsc{metas}(t)$)**.** The set of metavariables occurring in a term t (written $\textsc{metas}(t)$) is the set of metavariables that occur syntactically in t. The full definition is given in Figure 2.2.

Definition 2.11 (Set of atoms in a term: $\textsc{atoms}(t)$)**.** The set of atoms occurring in a term t (written $\textsc{atoms}(t)$) is the set of atoms that occur syntactically in t. The full definition is given in Figure 2.3.

Definition 2.12 (Set of constants of a term: $\textsc{consts}(t)$)**.** The set of constants occurring in a term t (written $\textsc{consts}(t)$) is the set of metavariables and atoms that occur syntactically in t.

$$\textsc{consts}(t) = \textsc{metas}(t) \cup \textsc{atoms}(t)$$

$$
\begin{array}{lcl}
\textsc{metas}(\alpha) & = & \{\alpha\} \\
\textsc{metas}(\mathbb{u}) & = & \emptyset \\
\textsc{metas}(x) & = & \emptyset \\
\textsc{metas}(\text{if}) & = & \emptyset \\
\textsc{metas}(f\,u) & = & \textsc{metas}(f) \cup \textsc{metas}(u) \\
\textsc{metas}(f\,.\pi_1) & = & \textsc{metas}(f) \\
\textsc{metas}(f\,.\pi_2) & = & \textsc{metas}(f) \\
\textsc{metas}(\lambda t) & = & \textsc{metas}(t) \\
\textsc{metas}(\Pi AB) & = & \textsc{metas}(A) \cup \textsc{metas}(B) \\
\textsc{metas}(\Sigma AB) & = & \textsc{metas}(A) \cup \textsc{metas}(B) \\
\textsc{metas}(\text{Bool}) & = & \emptyset \\
\textsc{metas}(\text{Set}) & = & \emptyset \\
\textsc{metas}(c) & = & \emptyset \\
\textsc{metas}(\langle t_1, t_2 \rangle) & = & \textsc{metas}(t_1) \cup \textsc{metas}(t_2)
\end{array}
$$

Figure 2.2: Metavariables occurring in a term

$$
\begin{array}{lcl}
\textsc{atoms}(\mathbb{u}) & = & \{\mathbb{u}\} \\
\textsc{atoms}(\alpha) & = & \emptyset \\
\textsc{atoms}(x) & = & \emptyset \\
\textsc{atoms}(\text{if}) & = & \emptyset \\
\textsc{atoms}(f\,u) & = & \textsc{atoms}(f) \cup \textsc{atoms}(u) \\
\textsc{atoms}(f\,.\pi_1) & = & \textsc{atoms}(f) \\
\textsc{atoms}(f\,.\pi_2) & = & \textsc{atoms}(f) \\
\textsc{atoms}(\lambda t) & = & \textsc{atoms}(t) \\
\textsc{atoms}(\Pi AB) & = & \textsc{atoms}(A) \cup \textsc{atoms}(B) \\
\textsc{atoms}(\Sigma AB) & = & \textsc{atoms}(A) \cup \textsc{atoms}(B) \\
\textsc{atoms}(\text{Bool}) & = & \emptyset \\
\textsc{atoms}(\text{Set}) & = & \emptyset \\
\textsc{atoms}(c) & = & \emptyset \\
\textsc{atoms}(\langle t_1, t_2 \rangle) & = & \textsc{atoms}(t_1) \cup \textsc{atoms}(t_2)
\end{array}
$$

Figure 2.3: Atoms occurring in a term

2.4 Contexts ($\Sigma \vdash \Gamma\,\mathbf{ctx}$)

A context is a list of types:

$$\begin{array}{rcll} \Gamma, \Delta, \Xi & ::= & \cdot & \text{empty context} \\ & | & \Gamma, A & \text{context variable} \end{array}$$

Contexts are read from left to right; that is, a context is well-formed if all its variables are well-typed with respect to the preceding binders.

$$\frac{\Sigma\ \mathbf{sig}}{\Sigma \vdash \cdot\,\mathbf{ctx}}\ \text{CTX-EMPTY}$$

$$\frac{\Sigma \vdash \Gamma\,\mathbf{ctx} \qquad \Sigma; \Gamma \vdash A\ \mathbf{type}}{\Sigma \vdash \Gamma, A\,\mathbf{ctx}}\ \text{CTX-VAR}$$

What it means for a term to be a type ($\Sigma; \Gamma \vdash A\ \mathbf{type}$) is explained in Section 2.5.

Remark 2.13 (Context inversion). Let $\Gamma = \Gamma_1, \Gamma_2$. If $\Sigma \vdash \Gamma_1, \Gamma_2\,\mathbf{ctx}$, then $\Sigma \vdash \Gamma_1\,\mathbf{ctx}$. Also, if $\Sigma \vdash \Gamma, A\,\mathbf{ctx}$, then $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma; \Gamma \vdash A\ \mathbf{type}$.

Definition 2.14 (Support of a context: $|\Gamma|$)**.** The support of a context Γ is the list of variables in a context. Because we use de Bruijn notation, it is solely determined by its length.

- $|\cdot| = 0$
- $|\Gamma, A| = 1 + |\Gamma|$

Notation (Variable names in contexts $\Gamma, x : A$). The syntax for contexts does not include variable names. A variable name in a context indicates the de Bruijn index to which a later-occurring variable name refers. For instance, "$\cdot \vdash x : \mathbb{A}, y : \mathbb{B}\,x, z : \mathbb{C}\,x\,y\,\mathbf{ctx}$" denotes the judgment "$\cdot \vdash \mathbb{A}, \mathbb{B}\,0, \mathbb{C}\,1\,0\,\mathbf{ctx}$".

Notation (Context concatenation: Γ_1, Γ_2). Contexts are syntactically lists of types. The concatenation of two contexts Γ and Δ is written Γ, Δ. This is a purely syntactic operation. It is not implied that Δ is well-formed on its own, even if Γ and Γ, Δ are.

2.5 Types ($\Sigma; \Gamma \vdash A\ \mathbf{type}$)

In a dependent type theory, terms and types share the same syntactic space. However, as we have seen in the well-formedness rules for contexts and signatures, only some terms can actually be used as the types of atoms and variables. Those specific terms we call types.

$$\frac{\Sigma; \Gamma \vdash A : \mathrm{Set}}{\Sigma; \Gamma \vdash A\ \mathbf{type}}\ \text{TYPE}$$

$$\frac{\Sigma; \Gamma \vdash A \equiv B : \mathrm{Set}}{\Sigma; \Gamma \vdash A \equiv B\ \mathbf{type}}\ \text{TYPE-EQ}$$

What it means for a term A to be of type Set ($\Gamma \vdash A : \mathrm{Set}$) is defined in Section 2.11. Correspondingly, what it means for two terms of type Set to be equal ($\Gamma \vdash A \equiv B : \mathrm{Set}$) is defined in Section 2.12.

Remark 2.15 (There is only set). In this system, all terms of type Set are considered types, and the only way for a term to be a type is to be of type Set. Therefore, the judgments $\Sigma; \Gamma \vdash A : \mathrm{Set}$ and $\Sigma; \Gamma \vdash A$ **type** are equivalent for any Σ, Γ and A; as are the judgments $\Sigma; \Gamma \vdash A \equiv B : \mathrm{Set}$ and $\Sigma; \Gamma \vdash A \equiv B$ **type**.

Despite Remark 2.15, our goal is that (with few alterations) this development can be applied to properly stratified theories with a hierarchy of universes (e.g. $\mathrm{Set}_0, \mathrm{Set}_1, \mathrm{Set}_2, \ldots$). In order to facilitate a stratification effort, we keep distinct judgments for "A is a term of type Set" ($\Sigma; \Gamma \vdash A : \mathrm{Set}$) and "$A$ is a type" ($\Sigma; \Gamma \vdash A$ **type**).

2.6 Context equality ($\Sigma \vdash \Gamma \equiv \Gamma'$ **ctx**)

Equality of types extends pointwise to whole contexts.

Definition 2.16 (Equality of contexts)**.** We say that two well-formed contexts $\Sigma \vdash \Gamma_1$ **ctx** and $\Sigma \vdash \Gamma_2$ **ctx** are equal (written $\Sigma \vdash \Gamma_1 \equiv \Gamma_2$ **ctx**) iff they have the same length, and the types of the variables are equal point-wise.

$$\frac{}{\Sigma \vdash \cdot \equiv \cdot}\ \textsc{ctx-empty-eq}$$

$$\frac{\Sigma \vdash \Gamma_1 \equiv \Gamma_2\ \mathbf{ctx} \qquad \Sigma; \Gamma_1 \vdash A_1 \equiv A_2\ \mathbf{type}}{\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2\ \mathbf{ctx}}\ \textsc{ctx-var-eq}$$

Remark 2.17 (Context equality inversion). Let $\Gamma = \Gamma_1, \Gamma_2$, $\Gamma' = \Gamma'_1, \Gamma'_2$.

If $\Sigma \vdash \Gamma_1, \Gamma_2 \equiv \Gamma'_1, \Gamma'_2$ **ctx**, then $\Sigma \vdash \Gamma_1 \equiv \Gamma'_1$ **ctx**. Also, if $\Sigma \vdash \Gamma, A \equiv \Gamma', A'$ **ctx**, then $\Sigma \vdash \Gamma \equiv \Gamma'$ **ctx** and $\Sigma; \Gamma \vdash A \equiv A'$ **type**.

2.7 Binders and variables

As shown in Figure 2.1, the term representation uses de Bruijn indices. Thus, variable names occurring in terms (e.g. x, y, z, ...) stand for natural numbers (e.g. 0, 1, 2, ...). This convention has the benefit of giving the same representation to all α-equivalent terms. For instance, the informally written terms $\lceil \lambda x.\lambda y.x \rfloor$ and $\lceil \lambda z.\lambda x.z \rfloor$ are both represented by the term $\lambda.\lambda.1$.

Notation (Names for de Bruijn indices). For the sake of readability, we will use textual names when describing terms. *Unless otherwise specified*, which binder each textual name refers to is indicated by writing the variable name next to the corresponding binder (λ, Π or Σ). For instance, the syntax $\lambda x.\alpha\, x\, (\lambda x.\lambda y.x)$ denotes the term $\lambda.\alpha\, 0\, (\lambda.1)$. The binders for Π and Σ, when given with a variable name, are written $\Pi(x : A)B$ and $\Sigma(x : A)B$, respectively. Similarly, the expression $\Gamma_1, x : A, \Gamma_2 \vdash x\, (\lambda y.x) : B$ denotes $\Gamma_1, A, \Gamma_2 \vdash x\, (\lambda.x^{(+1)}) : B$, where $x = |\Gamma_2|$ and $x^{(+1)} = 1 + |\Gamma_2|$.

$$
\begin{array}{lcl}
\textsc{fv}(x) & = & \{x\} \\
\\
\textsc{fv}(\alpha) & = & \emptyset \\
\textsc{fv}(\mathbb{0}) & = & \emptyset \\
\textsc{fv}(\text{if}) & = & \emptyset \\
\\
\textsc{fv}(f\ e) & = & \textsc{fv}(f) \cup \textsc{fv}(e) \\
\textsc{fv}(\lambda t) & = & \textsc{fv}(t) - 1 \\
\textsc{fv}(\Pi A B) & = & \textsc{fv}(A) \cup (\textsc{fv}(B) - 1) \\
\textsc{fv}(\Sigma A B) & = & \textsc{fv}(A) \cup (\textsc{fv}(B) - 1) \\
\\
\textsc{fv}(\text{Bool}) & = & \emptyset \\
\textsc{fv}(\text{Set}) & = & \emptyset \\
\textsc{fv}(c) & = & \emptyset \\
\textsc{fv}(\langle t_1, t_2 \rangle) & = & \textsc{fv}(t_1) \cup \textsc{fv}(t_2)
\end{array}
$$

$$\textsc{fv}(.\pi_1) = \textsc{fv}(.\pi_2) = \emptyset$$

Figure 2.4: Free variables in a term

Notation (N-ary binders: $\lambda\vec{x}^n.t$, $\Pi\overrightarrow{(x : A)}^n B$). We may use vector notation to bind several variables at the same time. For instance, $\lambda\vec{x}^n.t$ denotes the term $\lambda x_1.\lambda x_2....\lambda x_n.t$, and $\Pi\overrightarrow{(x : A)}^n B$ denotes the term $\Pi(x_1 : A_1)\Pi(x_2 : A_2)...\Pi(x_n : A_n)B$. If $n = 0$, then there are no binders: $\lambda\vec{x}^0.t \stackrel{\text{def}}{=} t$ and $\Pi\overrightarrow{(x : A)}^0 B \stackrel{\text{def}}{=} B$.

Notation (Arrow notation for Π-types: $(x : A) \to B$, $A \to B$). We may use $(x : A) \to B$ as an alternative syntax to $\Pi(x : A)B$. In cases where the bound variable does not occur in B, we may use the syntax $A \to B$ instead.

Notation (Product notation for Σ-types: $(x : A) \times B$, $A \times B$). We may use $(x : A) \times B$ as an alternative syntax to $\Sigma(x : A)B$. In cases where the bound variable does not occur in B, we may use the syntax $A \times B$ instead.

Notation (Strengthening of a set of variables: $X - 1$, $X - k$). Given $X \subseteq \mathbb{N}$, the notation $X - k$, $k \in \mathbb{N}$ denotes the set $\{n - k \mid n \in X, n \geq k\}$.

Definition 2.18 (Free variables in a term: $\textsc{fv}(t)$)**.** The free variables in a term t (written $\textsc{fv}(t)$) are the set of variables which are not bound by a binder (i.e. λ, Π or Σ). The full definition of $\textsc{fv}(t)$ is given in Figure 2.4.

Definition 2.19 (Free variables of a context: $\textsc{fv}(\Delta)$)**.** Given a (partial) context Δ, the set of free variables of Δ (written $\textsc{fv}(\Delta)$) is defined as follows:

$$
\begin{array}{lcl}
\textsc{fv}(\cdot) & = & \emptyset \\
\textsc{fv}(A, \Delta) & = & \textsc{fv}(A) \cup (\textsc{fv}(\Delta) - 1)
\end{array}
$$

Notation (Membership of names in set of free variables). If t is a term typed in a context, then, in the expression $x \in \textsc{fv}(t)$, x refers to the de Bruijn index of variable x in the context in which t is typed. The expressions $\textsc{fv}(t) \subseteq \{\vec{x}\}$ are interpreted in the same way.

2.8 Renamings

When dealing with a terms, we often need to renumber the variables in them so that a term can be used in a bigger or smaller context. To do this we define a notion of renaming.

Definition 2.20 (Renaming)**.** A renaming ρ is a function $\rho : A \to \mathbb{N}$, where $A \subseteq \mathbb{N}$.

Definition 2.21 (Inline renamings: $[... \mapsto ...]$)**.** We denote renamings by pairs $[x_1, x_2, ... \mapsto y_1, y_2, ...]$ of (possibly infinite) sequences of de Bruijn indices. Each index to the left of the arrow is mapped to the index in the corresponding position to the right of the arrow.

Indices not mentioned in the left list are mapped to themselves.

Definition 2.22 (Weakening: $(+\mathrm{n})$)**.** For $n \in \mathbb{N}$, the renaming $[0... \mapsto n...]$ maps variable x to variable $x + n$:

$$\begin{array}{rlcl} [0... \mapsto n...] : & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & x + n \end{array}$$

We denote this renaming by $(+n)$.

Definition 2.23 (Strengthening: $(-n)$)**.** The renaming $[n... \mapsto 0...]$ (strengthening by n) is the following:

$$\begin{array}{rlcl} [n... \mapsto 0...] : & \mathbb{N}/\{0, ..., n-1\} & \to & \mathbb{N} \\ & x & \mapsto & x - n \end{array}$$

We denote this renaming by $(-n)$. It is not defined for inputs smaller than n.

Definition 2.24 (Weakening of renamings: $(\rho + n)$)**.** Given a renaming $\rho : A \to \mathbb{N}$, and $n \in \mathbb{N}$, the renaming $(\rho + n)$ is defined as follows:

$$\begin{array}{rlcll} (\rho + n) : & 0, 1, 2, ..., n-1 \cup \{x + n \mid x \in A\} & \to \mathbb{N} & & \\ & x & \mapsto x & \textbf{if} & x < n \\ & x & \mapsto \rho(x - n) + n & \textbf{if} & x \geq n \end{array}$$

Example 2.25 (Strengthening by a variable: $((-1) + n)$)**.** The renaming $((-1)+n)$, named strengthening by variable n, maps each number larger than n to its predecessor.

$$\begin{array}{rlcll} ((-1) + n) : & \mathbb{N} - \{n\} & \to & \mathbb{N} & \\ & i & \mapsto & i & \textbf{if } i < n \\ & i & \mapsto & i - 1 & \textbf{if } i > n \end{array}$$

It is not defined for n, so it may only be applied to terms t such that $n \notin \mathrm{FV}(t)$. ◀

Definition 2.26 (Application of a renaming to a term: $t\,\rho$, t^ρ)**.** Let $\rho : A \to \mathbb{N}$ be a renaming, and t a term such that $\mathrm{FV}(t) \subseteq A$. Then $t\,\rho$ is a term where each free variable is renumbered according to ρ. The full definition is stated

$$
\begin{array}{lcl}
x\,\rho & = & \rho(x) \\
 & & \\
\alpha\rho & = & \alpha \\
\mathbb{U}\,\rho & = & \mathbb{U} \\
\mathrm{if}\,\rho & = & \mathrm{if} \\
 & & \\
(f\ e)\,\rho & = & (f\,\rho)\,(e\,\rho) \\
(\lambda t)\rho & = & \lambda(t(\rho+1)) \\
(\Pi AB)\rho & = & \Pi(A\rho)(B(\rho+1)) \\
(\Sigma AB)\rho & = & \Sigma(A\rho)(B(\rho+1)) \\
 & & \\
\mathrm{Bool}\,\rho & = & \mathrm{Bool} \\
\mathrm{Set}\,\rho & = & \mathrm{Set} \\
c\,\rho & = & c \\
\langle t_1, t_2\rangle\rho & = & \langle t_1\,\rho, t_2\,\rho\rangle \\
 & & \\
(.\pi_1)\rho & = & .\pi_1 \\
(.\pi_2)\rho & = & .\pi_2
\end{array}
$$

Figure 2.5: Applying a renaming ρ to a term.

in Figure 2.5.
For conciseness, we may sometimes write renaming applications as t^ρ instead of $t\,\rho$. The two notations have identical meanings.

Definition 2.27 (Renaming of a context: $\Gamma\,\rho$)**.**

$$
\begin{array}{lcl}
(\cdot)\,\rho & = & \cdot \\
(A, \Delta)\,\rho & = & (A\,\rho), (\Delta\,(\rho+1))
\end{array}
$$

Remark 2.28 (Renaming and free variables)*.* Applying a renaming to a term (if defined) commutes with taking the free variables of that term. That is, if $\rho : A \to \mathbb{N}$ and $\mathrm{FV}(t) \subseteq A$, then $\mathrm{FV}(t\,\rho) = \rho(\mathrm{FV}(t))$.

Notation (Composition of renamings: $\rho_1\rho_2$)*.* Let $\rho_1 : A \to \mathbb{N}$, $\rho_2 : B \to \mathbb{N}$ be renamings. Then $\rho_1\rho_2$ denotes the composition of ρ_1 and ρ_2 (i.e. $\rho_1\rho_2 : \rho_1^{-1}(B) \to \mathbb{N}$, with $(\rho_1\rho_2)(x) = \rho_2(\rho_1(x))$).

Remark 2.29 (Composition of renamings)*.* Let t be a term, and $\rho_1 : A \to \mathbb{N}$, $\rho_2 : B \to \mathbb{N}$ be renamings. Then, if $\mathrm{FV}(t) \subseteq \rho_1^{-1}(A)$, we have $(t^{\rho_1})^{\rho_2} = t^{(\rho_1\rho_2)}$.

Remark 2.30 (Properties of renamings)*.* Let ρ be a renaming, and a, b and c be natural numbers. The following hold:

- $(+a)(+b) = (+(a+b))$
- $((\rho + a) + b) = (\rho + (a+b))$
- For any term t, $t(+0) = t$.
- $\rho(+c) = (+c)(\rho + c)$. In particular, $(+1)(\rho+1) = (\rho)(+1)$, $\rho = (\rho + 0)$, and $((+a) + b)(+c) = (+c)((+a) + (b+c))$.

- Let t be a term. If for all $x \in \text{FV}(t)$, $x < a$, then $t^{(\rho+a)} = t$. In particular, if $\text{FV}(t) = \emptyset$, then $t^{\rho} = t$.
- Let t be a term. If for all $x \in \text{FV}(t)$, $x \geq a$, then $t^{(-a)(+b)} = t^{(-a+b)}$.
- Let t be a term. If for all $x \in \text{FV}(t)$, $x \geq a$, then $t^{((+b)+a)} = t^{(+b)}$.

2.9 Hereditary substitution and elimination ($t[u/x]$, t @ e)

In the syntax of terms we consider only a subset of the λ-terms, namely those in β-normal form. Terms in β-normal form are those which do not contain β-redexes. Examples of β-redexes (which are not valid terms according to our syntax) are $\lceil (\lambda.t)\, u \rfloor$, $\lceil \langle t, u \rangle .\pi_1 \rfloor$ and $\lceil \langle t, u \rangle .\pi_2 \rfloor$.

A definition of substitution which simply replaces each variable with its respective term would create β-redexes. Because our syntax only allows for β-normal terms, we need to define substitution in such a way that the result is also in β-normal form: i.e. a *hereditary* substitution [10, 11].

Note that terms such as $\text{if}\,\mathbb{A}\,\text{true}\,\mathfrak{a}\,\mathbb{b}$ and $\text{if}\,\mathbb{A}\,\text{false}\,\mathfrak{a}\,\mathbb{b}$ are not considered β-redexes, but are instead subject to δ-reduction (Section 2.13).

Definition 2.31 (Hereditary substitution: $t[u/x] \Downarrow r$)**.** Hereditary substitution is a relation $t[u/x] \Downarrow r$, defined in Figure 2.6.

Notation ($B[t]$). The syntax $B[t]$ denotes $B[t/0]$.

Notation ($\vec{e}^{\,n}[t/x] \Downarrow \overrightarrow{e'}^{n}$). We write $\vec{e}^{\,n}[t] \Downarrow \overrightarrow{e'}^{n}$ if, for every i, $1 \leq i \leq n$, either:

- $e_i = e'_i = .\pi_1$, or
- $e_i = e'_i = .\pi_2$, or
- There are u, v such that $e_i = u$, $e'_i = v$, and $u[t/x] \Downarrow v$.

When defining hereditary substitution for a case such as $(x\,\vec{e})[u/x]$, one has to apply the eliminators $\vec{e}$ to u, which may introduce β-redexes. Because our syntax is restricted to β-normal terms, we need to ensure that the result of such an application is β-normal. That is, we need to perform a *hereditary* elimination.

Definition 2.32 (Hereditary elimination: t @ $e \Downarrow r$)**.** Hereditary elimination is a relation $t@e \Downarrow r$. The full definition is given Figure 2.6.

Notation (Hereditary substitution as a partial function: $t[u/x]{\Downarrow}$, $t[u/x]$). Hereditary substitution is defined recursively on the syntax of terms, with no overlap among the different cases of the definition. This means that given t, u and x, there exists at most one r such that $t[u/x] \Downarrow r$.

We will use the proposition $t[u/x]{\Downarrow}$ as a shorthand for $\exists r.t[u/x] \Downarrow r$. Furthermore, if in a given proof context, if $\exists r.t[u/x] \Downarrow r$ holds, then we will denote such an r by $t[u/x]$.

$$
\begin{array}{lll}
x[u/x] \Downarrow u & & \\
y[u/x] \Downarrow y & \textbf{if} & x > y \\
y[u/x] \Downarrow (y-1) & \textbf{if} & x < y \\
\\
\alpha[u/x] \Downarrow \alpha & & \\
\square[u/x] \Downarrow \square & & \\
\mathrm{if}[u/x] \Downarrow \mathrm{if} & & \\
\\
(f\,t)[u/x] \Downarrow r & \textbf{if} & f[u/x] \Downarrow r_1 \ \wedge \ t[u/x] \Downarrow r_2 \ \wedge \ r_1 \ @ \ r_2 \Downarrow r \\
(f\,.\pi_1)[u/x] \Downarrow r & \textbf{if} & f[u/x] \Downarrow r_1 \ \wedge \ r_1 \ @ \ .\pi_1 \Downarrow r \\
(f\,.\pi_2)[u/x] \Downarrow r & \textbf{if} & f[u/x] \Downarrow r_1 \ \wedge \ r_1 \ @ \ .\pi_2 \Downarrow r \\
\\
(\lambda.t)[u/x] \Downarrow (\lambda.r) & \textbf{if} & t[u(+1)/x+1] \Downarrow r \\
(\Pi AB)[u/x] \Downarrow (\Pi A'B') & \textbf{if} & A[u/x] \Downarrow A' \ \wedge \ B[u^{(+1)}/x+1] \Downarrow B' \\
(\Sigma AB)[u/x] \Downarrow (\Sigma A'B') & \textbf{if} & A[u/x] \Downarrow A' \ \wedge \ B[u^{(+1)}/x+1] \Downarrow B' \\
\\
\mathrm{Bool}[u/x] \Downarrow \mathrm{Bool} & & \\
\mathrm{Set}[u/x] \Downarrow \mathrm{Set} & & \\
c[u/x] \Downarrow c & & \\
\langle t_1, t_2 \rangle[u/x] \Downarrow \langle t_1', t_2' \rangle & \textbf{if} & t_1[u/x] \Downarrow t_1' \ \wedge \ t_2[u/x] \Downarrow t_2'
\end{array}
$$

(a) Hereditary substitution

$$
\begin{array}{lll}
h\,\vec{e} \ @ \ e' \Downarrow (h\,\vec{e}\,e') & & \\
\langle t_1, t_2 \rangle \ @ \ .\pi_1 \Downarrow t_1 & & \\
\langle t_1, t_2 \rangle \ @ \ .\pi_2 \Downarrow t_2 & & \\
\lambda.t \ @ \ u \Downarrow r & \textbf{if} & t[u/0] \Downarrow r
\end{array}
$$

(b) Hereditary elimination

Figure 2.6: Hereditary substitution and elimination. The syntax $(t(+1))$ denotes the result of weakening t by 1 (see Definition 2.22).

Notation (Hereditary elimination as a partial function: $(t \mathbin{@} e)\Downarrow$, $t \mathbin{@} e$). By the same reasoning, for every term t and eliminator e, there exists at most one r such that $t \mathbin{@} e \Downarrow r$.

We will use $t \mathbin{@} e\Downarrow$ as a shorthand for $\exists r.t \mathbin{@} e \Downarrow r$. Furthermore, if in a given context, $\exists r.t \mathbin{@} e \Downarrow r$ holds, then we will denote such an r by $t \mathbin{@} e$.

Definition 2.33 (Iterated hereditary elimination: $t \mathbin{@} \vec{e} \Downarrow r$, $t \mathbin{@} \vec{e}$)**.** We use $t \mathbin{@} e_1 \mathinner{...} e_n \Downarrow r$ as a shorthand for $\exists t_1, ..., t_{n-1}.(t \mathbin{@} e_1 \Downarrow t_1) \wedge (t_1 \mathbin{@} e_2 \Downarrow t_2) \wedge ... \wedge (t_{n-1} \mathbin{@} e_n \Downarrow r)$. For $n = 0$, $t \mathbin{@} \varepsilon \Downarrow t$, and, for $n = 1$, $t \mathbin{@} \vec{e}^1 \Downarrow r$ if and only if $t \mathbin{@} e_1 \Downarrow r$.

If $t \mathbin{@} e_1 \mathinner{...} e_n \Downarrow r$ holds for some r, we denote such an r by $t \mathbin{@} e_1 \mathinner{...} e_n$.

Definition 2.34 (Iterated hereditary substitution: $t[\vec{u}/\vec{x}] \Downarrow r$)**.** Let $\vec{x}^n = (m+n-1), (m+n-2), ..., (m+1), m$, and let $\vec{u}^n$ be such that $\forall v \in \mathrm{FV}(u_i).v \geq m$. We use $t[\vec{u}/\vec{x}^n] \Downarrow r$ as a shorthand for $\exists t_1, ..., t_{n-1}.(t[u_1^{(+(n-1))}/x_1] \Downarrow t_1) \wedge (t_1[u_2^{(+(n-2))}/x_2] \Downarrow t_2) \wedge ... \wedge (t_{n-1}[u_n^{(+0)}/x_n] \Downarrow r)$. For $n = 0$, $t[\varepsilon/\varepsilon] \Downarrow t$, and, for $n = 1$, $t[\vec{u}^1/\vec{x}^1] \Downarrow r$ if and only if $t[u/x] \Downarrow r$.

If $t[\vec{u}/\vec{x}] \Downarrow r$ holds for some r, we denote such an r by $t[\vec{u}/\vec{x}]$.

We say $t[\vec{u}] \Downarrow r$ if and only if $t[\vec{u}/(n-1), ..., 0] \Downarrow r$.

Remark 2.35 (Iterated application as substitution on body). We have $(\lambda^n.t) \mathbin{@} \vec{u}^n \Downarrow r$ if and only if $t[\vec{u}] \Downarrow r$.

Proof. By induction on n.

- Case 0: Trivial.
- Case $n+1$: We have $(\lambda^{n+1}.t) \mathbin{@} \vec{u}^{n+1} \Downarrow r$ iff $(\lambda^{n+1}.t) \mathbin{@} u \Downarrow t_1$ and $t_1 \mathbin{@} \vec{u}_{2,...,n} \Downarrow r$ for some t_1. By definition, t_1 is of the form $\lambda^n.r_1$, with $t[u/n] \Downarrow r_1$. By the IH, the latter is equivalent to $r_1[\vec{u}_{2,...,n}] \Downarrow r$. By definition, $t[u^{(+n)}/n] \Downarrow r_1$ and $r_1[\vec{u}_{2,...,n}] \Downarrow r$ is equivalent to $t[\vec{u}] \Downarrow r$.

□

Remark 2.36 (Hereditary substitution by a neutral term: $t[f/x]$). Given a term t, a neutral term f and a variable x, we always have $t[f/x] \Downarrow$. Therefore, we can always write $t[f/x]$. Furthermore, if g is a neutral term, then $g[f/x]$ is also a neutral term.

Proof. Proceed by induction on t. In the base cases, $(x)[f/x] \Downarrow f$, which is neutral. For the inductive cases, note that, for every neutral term f, (i) $f^{(+1)}$ is also neutral, and (ii) $f \mathbin{@} \vec{e} \Downarrow f\,\vec{e}$. □

Remark 2.37 (Hereditary elimination of neutral terms: $f \mathbin{@} \vec{e}$). Given a neutral term f and an eliminator e, by Definition 2.32 (hereditary elimination), $f \mathbin{@} e \Downarrow (f\,e)$.

Given $\vec{e}$ and applying the above remark iteratively, we have $f \mathbin{@} \vec{e} \Downarrow (f\,\vec{e})$. Therefore, we can always write $f \mathbin{@} \vec{e}$.

Definition 2.38 (Hereditary substitution for contexts: $\Delta[u/x] \Downarrow \Delta'$)**.** A substitution can be applied to a whole context as follows:

$$\cdot[u/x] \Downarrow \cdot$$
$$(A, \Delta)[u/x] \Downarrow (A', \Delta) \quad \textbf{if} \quad A[u/x] \Downarrow A' \textbf{ and } \Delta'[u(+1)/x+1] \Downarrow \Delta'$$

Notation (Names for de Bruijn indices in hereditary substitution). A variable name in a hereditary substitution denotes the de Bruijn index of that variable in the context in which the term to which the substitution is applied appears. For example, given a term $\Sigma;\Gamma,x:A,\Delta \vdash t : B$, the expressions $\Delta[u/x] \Downarrow r$ and $t[u/x] \Downarrow r$ denote $\Delta[u/0] \Downarrow r$ and $t[u/|\Delta|] \Downarrow r$, respectively.

Lemma 2.39 (Hereditary substitution and application commute with renaming)**.** *Let ρ be a renaming, $\rho : \mathbb{N} \to \mathbb{N}$.*

- *If $\rho = \rho' + x$ and $t[u/x]\Downarrow$, then $t^{(\rho+1)}[u^\rho/x]\Downarrow(t[u/x]^\rho)$.*
- *If $(t \mathbin{@} e) \Downarrow u$, then $(t^\rho \mathbin{@} e^\rho) \Downarrow u^p$.*

Proof. By induction on the derivation for $t[u/x] \Downarrow r$. We enumerate some representative cases:

- $x[u/x] \Downarrow u$: Then $x^{(\rho+1)} = x^{\rho'+(x+1)} = x$, with $x^{\rho+1}[u^\rho/x] \Downarrow u^\rho$.
- $y[u/x] \Downarrow y$, $y < x$: Then $y^{(\rho+1)} = y$ and $y^\rho = y$. Therefore, $y^{(\rho+1)}[u^\rho/x] \Downarrow y^\rho$.
- $y[u/x] \Downarrow (y-1)$, $y > x$: Then $y^{(\rho+1)} = \rho'(y-x-1)+x+1 = y'$ (for some $y' > x$). and $(y-1)^\rho = \rho'(y-1-x)+x = y'-1$. By definition, $y'[u^\rho/x] \Downarrow (y'-1)$. Therefore $y^{(\rho+1)}[u^\rho/x] \Downarrow (y-1)^\rho$.
- $t = \alpha$: $\alpha = \alpha^{(\rho+1)} = \alpha^\rho$. Thus $\alpha[u^\rho/x] \Downarrow \alpha$.
- $(f\,t)[u/x] \Downarrow r$, with $f[u/x] \Downarrow r_1$, $t[u/x] \Downarrow r_2$ and $(r_1 \mathbin{@} r_2) \Downarrow r$: By the second induction hypothesis, $f^{(\rho+1)}[u^\rho/x] \Downarrow r_1^\rho$ and $t^{(\rho+1)}[u^\rho/x] \Downarrow r_2^\rho$. By the first induction hypothesis, $r_1^\rho \mathbin{@} r_2^\rho \Downarrow r^\rho$. Also, $(f\,t)^{(\rho+1)} = f^{(\rho+1)}\,t^{(\rho+1)}$. Therefore $(f\,t)^{(\rho+1)}[u^\rho/x] \Downarrow r^\rho$.
- $(\lambda.t)[u/x] \Downarrow (\lambda.r)$, with $t[u(+1)/x+1] \Downarrow r$: We have $(\lambda.t)^{(\rho+1)} = \lambda.(t^{(\rho+2)})$ and $(\lambda.r)^\rho = \lambda.(r^{(\rho+1)})$. Because $\rho = \rho' + x$, $(\rho+1) = \rho' + (x+1)$. By the induction hypothesis, we have $t^{(\rho+2)}[(u(+1))^{(\rho+1)}/x+1] \Downarrow r^{(\rho+1)}$. By Remark 2.30 (properties of renamings), $u(+1)^{(\rho+1)} = u^\rho(+1)$. Therefore $(\lambda.t)^{\rho+1}[u^\rho/x] \Downarrow (\lambda.r)^\rho$.
- $h\,\vec{e} \mathbin{@} e' \Downarrow (h\,\vec{e}\,e')$: By definition $(h\,\vec{e}\,e')^\rho = (h\,\vec{e})^\rho\,e'^\rho$. Therefore $(h\,\vec{e})^\rho \mathbin{@} e'^\rho \Downarrow (h\,\vec{e}\,e')^\rho$.
- $\lambda.t \mathbin{@} u \Downarrow r$ with $t[u/0] \Downarrow r$: We have $\rho = \rho + 0$. By the induction hypothesis, $t^{(\rho+1)}[u^\rho/0] \Downarrow r^\rho$. By definition, $(\lambda.t)^\rho = \lambda.(t^{(\rho+1)})$. Therefore $(\lambda.t)^\rho \mathbin{@} u^\rho \Downarrow r^\rho$.

□

Lemma 2.40 (Correspondence between renaming and substitution)**.** *We have $V[y/x]\Downarrow V[0,...,x-1,x,x+1,... \mapsto 0,...,x-1,y,x,...]$. In particular, $V[0/0]\Downarrow V[0,... \mapsto 0,0,1,2,...]$.*

Additionally, $V[\vec{x}^n] = V[...,n,(n-1),...,0 \mapsto ...,1,0,\vec{x}]$.

Proof. By induction on the structure of V. □

2.10 Head lookup ($\Sigma; \Gamma \vdash h \Rightarrow A$)

The types of the heads of neutral terms (for instance, variables, metavariables and atoms) are determined by the signature and the context:

$$\frac{\Sigma \vdash \Gamma\,\mathbf{ctx} \qquad \Gamma = \Gamma_1, A, \Gamma_2 \qquad n = |\Gamma_2|}{\Sigma; \Gamma \vdash n \Rightarrow A^{(+(n+1))}}\ \textsc{var}$$

$$\frac{\Sigma \vdash \Gamma\,\mathbf{ctx} \qquad \alpha : A \in \Sigma}{\Sigma; \Gamma \vdash \alpha \Rightarrow A}\ \textsc{meta}_1$$

$$\frac{\Sigma \vdash \Gamma\,\mathbf{ctx} \qquad \alpha := t : A \in \Sigma}{\Sigma; \Gamma \vdash \alpha \Rightarrow A}\ \textsc{meta}_2$$

$$\frac{\Sigma \vdash \Gamma\,\mathbf{ctx} \qquad \mathbb{0} : A \in \Sigma}{\Sigma; \Gamma \vdash \mathbb{0} \Rightarrow A}\ \textsc{atom}$$

$$\frac{\Sigma \vdash \Gamma\,\mathbf{ctx}}{\Sigma; \Gamma \vdash \mathrm{if} \Rightarrow \Pi(\Pi\mathrm{BoolSet})(\Pi\mathrm{Bool}(\Pi(1\,\mathrm{true})(\Pi(2\,\mathrm{false})(3\,2))))}\ \textsc{if}$$

Remark. Using the binder syntax described in Section 2.7, we may write the conclusion of the IF rule as $\Sigma; \Gamma \vdash \mathrm{if} \Rightarrow (X : \mathrm{Bool} \to \mathrm{Set}) \to (y : \mathrm{Bool}) \to X\,\mathrm{true} \to X\,\mathrm{false} \to X\,y$.

Remark. In the conclusion of the rules ATOM, $\textsc{meta}_1$, and $\textsc{meta}_2$, because A is closed, $A^{(+|\Gamma|)} = A$.

2.11 Terms ($\Sigma; \Gamma \vdash t : A$)

The judgement "term t has type A in context Γ under signature Σ" is written $\Sigma; \Gamma \vdash t : A$.

Notation (Implicit signature). In those rules where the signature Σ is omitted, it is understood that all premises and the conclusion share the same signature Σ. The rule then holds for any such Σ. For instance, consider the first typing rule given below, the BOOL rule (left); and its implied full form (right).

$$\frac{\Gamma\,\mathbf{ctx}}{\Gamma \vdash \mathrm{Bool} : \mathrm{Set}}\ \textsc{bool} \qquad\qquad \frac{\Sigma \vdash \Gamma\,\mathbf{ctx}}{\Sigma; \Gamma \vdash \mathrm{Bool} : \mathrm{Set}}\ \textsc{bool}$$

Type constructors

$$\frac{\Gamma\,\mathbf{ctx}}{\Gamma \vdash \mathrm{Bool} : \mathrm{Set}}\ \textsc{bool}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \qquad \Gamma, A \vdash B : \mathrm{Set}}{\Gamma \vdash \Pi AB : \mathrm{Set}}\ \textsc{pi}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \qquad \Gamma, A \vdash B : \mathrm{Set}}{\Gamma \vdash \Sigma AB : \mathrm{Set}}\ \textsc{sigma}$$

$$\frac{\Gamma\,\mathbf{ctx}}{\Gamma \vdash \mathrm{Set} : \mathrm{Set}}\ \textsc{set}$$

Term constructors

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{true} : \mathrm{Bool}}\ \textsc{true}$$

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{false} : \mathrm{Bool}}\ \textsc{false}$$

$$\frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda t : \Pi A B}\ \textsc{abs}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma, A \vdash B\ \mathbf{type} \qquad B[t]{\Downarrow} \qquad \Gamma \vdash u : B[t]}{\Gamma \vdash \langle t, u \rangle : \Sigma A B}\ \textsc{pair}$$

Neutral terms

$$\frac{\Gamma \vdash h \Rightarrow A}{\Gamma \vdash h : A}\ \textsc{head}$$

$$\frac{\Gamma \vdash f : \Sigma A B}{\Gamma \vdash f\,.\pi_1 : A}\ \textsc{proj1}$$

$$\frac{\Gamma \vdash f : \Sigma A B}{\Gamma \vdash f\,.\pi_2 : B[f\,.\pi_1]}\ \textsc{proj2}$$

$$\frac{\Gamma \vdash f : \Pi A B \qquad \Gamma \vdash t : A \qquad B[t]{\Downarrow}}{\Gamma \vdash f\ t : B[t]}\ \textsc{app}$$

Other rules

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B\ \mathbf{type}}{\Gamma \vdash t : B}\ \textsc{conv}$$

The Set : Set **judgment and non-termination**

Terms in a theory where $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$ are not necessarily normalizing, as described first by Girard [22] and then in a more succinct form by Hurkens [30]. Our theoretical description glosses over this fact.

Our intention when using Set : Set is to simplify the exposition. The algorithm is ultimately meant to be used with properly stratified theories where all terms are normalizing. The postulates about types and terms that such a theory needs to satisfy are given in Section 2.14. Note that these postulates may not hold for the unstratified theory defined in this chapter.

2.12 Term equality ($\Sigma; \Gamma \vdash t \equiv u : A$)

The judgmental (or definitional[1]) equality for terms is written $\Gamma \vdash t \equiv u : A$, and is given by the following deduction rules.

If for terms t and u we have $\Gamma \vdash t \equiv u : A$, we say that t and u are judgmentally or definitionally equal. In particular, two types A and B are definitionally equal if $\Gamma \vdash A \equiv B : \mathrm{Set}$.

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{Bool} \equiv \mathrm{Bool} : \mathrm{Set}}\ \textsc{bool-eq}$$

$$\frac{\Gamma \vdash A \equiv A' : \mathrm{Set} \qquad \Gamma, A \vdash B \equiv B' : \mathrm{Set}}{\Gamma \vdash \Pi AB \equiv \Pi A'B' : \mathrm{Set}}\ \textsc{pi-eq}$$

$$\frac{\Gamma \vdash A \equiv A' : \mathrm{Set} \qquad \Gamma, A \vdash B \equiv B' : \mathrm{Set}}{\Gamma \vdash \Sigma AB \equiv \Sigma A'B' : \mathrm{Set}}\ \textsc{sigma-eq}$$

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{Set} \equiv \mathrm{Set} : \mathrm{Set}}\ \textsc{set-eq}$$

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{true} \equiv \mathrm{true} : \mathrm{Bool}}\ \textsc{true-eq}$$

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{false} \equiv \mathrm{false} : \mathrm{Bool}}\ \textsc{false-eq}$$

$$\frac{\Gamma, A \vdash t \equiv u : B}{\Gamma \vdash \lambda.t \equiv \lambda.u : \Pi AB}\ \textsc{abs-eq}$$

$$\frac{\Gamma, A \vdash B\ \mathbf{type} \qquad B[t_1]{\Downarrow} \qquad \begin{array}{c}\Gamma \vdash t_1 \equiv u_1 : A \\ \Gamma \vdash t_2 \equiv u_2 : B[t_1]\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle \equiv \langle u_1, u_2 \rangle : \Sigma AB}\ \textsc{pair-eq}$$

Elimination

$$\frac{\Gamma \vdash h \Rightarrow A}{\Gamma \vdash h \equiv h : A}\ \textsc{head-eq}$$

$$\frac{\Gamma \vdash f \equiv g : \Sigma AB}{\Gamma \vdash f\,.\pi_1 \equiv g\,.\pi_1 : A}\ \textsc{proj1-eq}$$

$$\frac{\Gamma \vdash f \equiv g : \Sigma AB}{\Gamma \vdash f\,.\pi_2 \equiv g\,.\pi_2 : B[f\,.\pi_1]}\ \textsc{proj2-eq}$$

$$\frac{\Gamma \vdash f \equiv g : \Pi AB \qquad \Gamma \vdash t \equiv u : A \qquad B[t]{\Downarrow}}{\Gamma \vdash f\,t \equiv g\,u : B[t]}\ \textsc{app-eq}$$

Remark. In the HEAD-EQ rule, h stands for either (i) a variable "x", (ii) a metavariable "α" (iii) an atom "$\mathbb{a}$", or (iv) the boolean recursor "if".

[1] In an intensional type theory such as this one, the two notions coincide.

η-conversion

$$\frac{\Gamma \vdash f : \Pi A B}{\Gamma \vdash f \equiv \lambda.f^{(+1)}\ 0 : \Pi A B}\ \textsc{eta-abs}$$

$$\frac{\Gamma \vdash f : \Sigma A B}{\Gamma \vdash f \equiv \langle f\ .\pi_1, f\ .\pi_2 \rangle : \Sigma A B}\ \textsc{eta-pair}$$

Remark (η-conversion for general terms). Because all the terms are in β-normal form, neutral terms are the only cases where η-expansion is relevant.

δ-conversion

$$\frac{\Sigma; \Gamma \vdash \alpha\ \vec{e} : T \qquad \Sigma; \Gamma \vdash t' : T \qquad \alpha := t : A \in \Sigma \qquad t\ @\ \vec{e} \Downarrow t'}{\Sigma; \Gamma \vdash \alpha\ \vec{e} \equiv t' : T}\ \textsc{delta-meta}$$

$$\frac{\Gamma \vdash \text{if } A \text{ true } u_{\text{t}}\ u_{\text{f}}\ \vec{e} : T \qquad \Gamma \vdash u' : T \qquad u_{\text{t}}\ @\ \vec{e} \Downarrow u'}{\Gamma \vdash \text{if } A \text{ true } u_{\text{t}}\ u_{\text{f}}\ \vec{e} \equiv u' : T}\ \textsc{delta-if-true}$$

$$\frac{\Gamma \vdash \text{if } A \text{ false } u_{\text{t}}\ u_{\text{f}}\ \vec{e} : T \qquad \Gamma \vdash u' : T \qquad u_{\text{f}}\ @\ \vec{e} \Downarrow u'}{\Gamma \vdash \text{if } A \text{ false } u_{\text{t}}\ u_{\text{f}}\ \vec{e} \equiv u' : T}\ \textsc{delta-if-false}$$

Other rules

$$\frac{\Gamma \vdash t \equiv u : A \qquad \Gamma \vdash A \equiv B\ \mathbf{type}}{\Gamma \vdash t \equiv u : B}\ \textsc{conv-eq}$$

$$\frac{\Gamma \vdash t \equiv u : A \qquad \Gamma \vdash u \equiv v : A}{\Gamma \vdash t \equiv v : A}\ \textsc{trans}$$

$$\frac{\Gamma \vdash t \equiv u : A}{\Gamma \vdash u \equiv t : A}\ \textsc{sym}$$

2.13 Term reduction ($\longrightarrow_{\delta\eta}$, $\longrightarrow^{\star}_{\delta\eta}$)

Definition 2.41 (δη-normalization step: $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} u : T$)**.** Let Σ be a signature, Γ a context and T a type. The relation $\longrightarrow_{\delta\eta}$ is defined in Figure 2.7, with the additional requirement that, whenever $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} u : T$, then $\Sigma; \Gamma \vdash t : T$.

Remark. There is deliberate overlap between the reduction rules. The order in which different subterms are reduced depends ultimately on the unification rules are applied by the unification algorithm (Section 5.1).

Remark. The rule $\textsc{app}_n$ is a family of rules, with one element for each $n \in \mathbb{N}$, $n \geq 1$.

Definition 2.42 (Iterated δη-reduction: $\Sigma; \Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} u : A$)**.** The relation $\Sigma; \Gamma \vdash \_ \longrightarrow^{\star}_{\delta\eta} \_ : A$ is the reflexive and transitive closure of $\Sigma; \Gamma \vdash \_ \longrightarrow_{\delta\eta} \_ : A$.

Remark 2.43 (Free variables of δη-reduct). If $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} u : A$, then $\textsc{fv}(u) \subseteq \textsc{fv}(t)$.

$$
\begin{array}{lll}
(\Pi_1) & \Sigma;\Gamma \vdash \Pi A B \longrightarrow_{\delta\eta} \Pi A' B : T & \textbf{if} \quad \Sigma;\Gamma \vdash A \longrightarrow_{\delta\eta} A' : \mathrm{Set} \\
(\Pi_2) & \Sigma;\Gamma \vdash \Pi A B \longrightarrow_{\delta\eta} \Pi A B' : T & \textbf{if} \quad \Sigma;\Gamma, A \vdash B \longrightarrow_{\delta\eta} B' : \mathrm{Set} \\
(\Sigma_1) & \Sigma;\Gamma \vdash \Sigma A B \longrightarrow_{\delta\eta} \Sigma A' B : T & \textbf{if} \quad \Sigma;\Gamma \vdash A \longrightarrow_{\delta\eta} A' : \mathrm{Set} \\
(\Sigma_2) & \Sigma;\Gamma \vdash \Sigma A B \longrightarrow_{\delta\eta} \Sigma A B' : T & \textbf{if} \quad \Sigma;\Gamma, A \vdash B \longrightarrow_{\delta\eta} B' : \mathrm{Set} \\
(\lambda) & \Sigma;\Gamma \vdash \lambda.t \longrightarrow_{\delta\eta} \lambda.t' : T & \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Pi A B \ \textbf{type} \\
 & & \textbf{and} \ \Sigma;\Gamma, A \vdash t \longrightarrow_{\delta\eta} t' : B \\
(\langle,\rangle_1) & \Sigma;\Gamma \vdash \langle t, u\rangle \longrightarrow_{\delta\eta} \langle t', u\rangle : T & \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Sigma A B \ \textbf{type} \\
 & & \quad \Sigma;\Gamma \vdash t \longrightarrow_{\delta\eta} t' : A \\
(\langle,\rangle_2) & \Sigma;\Gamma \vdash \langle t, u\rangle \longrightarrow_{\delta\eta} \langle t, u'\rangle : T & \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Sigma A B \ \textbf{type} \\
 & & \textbf{and} \ B[t]{\Downarrow} \\
 & & \textbf{and} \ \Sigma;\Gamma \vdash u \longrightarrow_{\delta\eta} u' : B[t] \\
(\eta\text{-}\Pi) & \Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} \lambda.(f^{(+1)})\,0 : T & \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Pi A B \ \textbf{type} \\
(\eta\text{-}\Sigma) & \Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} \langle f\,.\pi_1, f\,.\pi_2\rangle : T & \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Sigma A B \ \textbf{type} \\
(\textsc{app}_n) & \Sigma;\Gamma \vdash h\,\vec{e}^{\,n-1}\,t\,\vec{e}' \longrightarrow_{\delta\eta} h\,\vec{e}\,u\,\vec{e}' : T & \textbf{if} \quad \Sigma;\Gamma \vdash h\,\vec{e} : \Pi U V \\
 & & \textbf{and} \ \Sigma;\Gamma \vdash t \longrightarrow_{\delta\eta} u : U \\
(\textsc{meta}) & \Sigma;\Gamma \vdash \alpha\,\vec{e} \longrightarrow_{\delta\eta} u : T & \textbf{if} \quad \alpha := t : A \in \Sigma \\
 & & \textbf{and} \ (t \ @ \ \vec{e}) \Downarrow u \\
(\textsc{if}_1) & \Sigma;\Gamma \vdash \mathrm{if}\,A\,\mathrm{true}\,t\,u\,\vec{e} \longrightarrow_{\delta\eta} t' : T & \textbf{if} \quad (t \ @ \ \vec{e}) \Downarrow t' \\
(\textsc{if}_2) & \Sigma;\Gamma \vdash \mathrm{if}\,A\,\mathrm{false}\,t\,u\,\vec{e} \longrightarrow_{\delta\eta} u' : T & \textbf{if} \quad (u \ @ \ \vec{e}) \Downarrow u'
\end{array}
$$

Figure 2.7: Cases for Definition 2.41 (δη-normalization step). For each recursive occurrence of the form $\Sigma;\Gamma \vdash t \longrightarrow_{\delta\eta} u : T$, there is an implicit condition that $\Sigma;\Gamma \vdash t : T$.

2.14 Properties

Here are some properties of the dependent type system we have defined in the previous sections. They will be useful when discussing the correctness of a type checking algorithm for the system.

Those properties marked as postulates are assumed to hold without proof. Some of these properties may not hold in the theory as described in this chapter, but we expect they would all hold in a properly stratified version thereof. A complete list of these assumptions may be found on page 216.

2.14.1 Judgments

For the sake of conciseness when stating properties we define a notion of judgment. This allows for a homogeneous treatment of the judgments that have been defined in this chapter.

Definition 2.44 (Judgment: $\Sigma; \Gamma \vdash J$)**.** A judgment J has any of the following forms: $\Delta\,\mathbf{ctx}$, $\Delta \vdash A\ \mathbf{type}$, $\Delta \vdash A \equiv B\ \mathbf{type}$, $\Delta_1 \equiv \Delta_2\,\mathbf{ctx}$, $\Delta \vdash t : A$, $\Delta \vdash t \equiv u : A$, and $J_1 \wedge J_2$.

We write $\Sigma; \Gamma \vdash J$ if any of the following hold:

- $J = \Delta\,\mathbf{ctx}$, and $\Sigma \vdash \Gamma, \Delta\,\mathbf{ctx}$.
- $J = \Delta_1 \equiv \Delta_2\,\mathbf{ctx}$ and $\Sigma \vdash \Gamma, \Delta_1 \equiv \Gamma, \Delta_2\,\mathbf{ctx}$.
- $J = \Delta \vdash A\ \mathbf{type}$ and $\Sigma; \Gamma, \Delta \vdash A\ \mathbf{type}$.
- $J = \Delta \vdash A \equiv B\ \mathbf{type}$ and $\Sigma; \Gamma, \Delta \vdash A \equiv B\ \mathbf{type}$.
- $J = \Delta \vdash t : A$ and $\Sigma; \Gamma, \Delta \vdash t : A$.
- $J = \Delta \vdash t \equiv u : A$ and $\Sigma; \Gamma, \Delta \vdash t \equiv u : A$.
- $J = J_1 \wedge J_2$, with $\Sigma; \Gamma \vdash J_1$ and $\Sigma; \Gamma \vdash J_2$.

Notation (Signature judgment: $\Sigma \vdash J$). The statement $\Sigma \vdash J$ is equivalent to $\Sigma; \cdot \vdash J$.

Judgments can be manipulated in similar ways as terms:

Definition 2.45 (Free variables of a scoped and typed term: $\textsc{fv}(\Delta \vdash t : B)$, $\textsc{fv}(J)$)**.** We can consider the variables free in an entire judgment:

$$
\begin{array}{lcl}
\textsc{fv}(\Delta\,\mathbf{ctx}) & = & \textsc{fv}(\Delta) \\
\textsc{fv}(\Delta_1 \equiv \Delta_2\,\mathbf{ctx}) & = & \textsc{fv}(\Delta_1) \cup \textsc{fv}(\Delta_2) \\
\textsc{fv}(\cdot \vdash A\ \mathbf{type}) & = & \textsc{fv}(A) \\
\textsc{fv}(\cdot \vdash A \equiv B\ \mathbf{type}) & = & \textsc{fv}(A) \cup \textsc{fv}(B) \\
\textsc{fv}(\cdot \vdash t : B) & = & \textsc{fv}(t) \cup \textsc{fv}(B) \\
\textsc{fv}(\cdot \vdash t \equiv u : B) & = & \textsc{fv}(t) \cup \textsc{fv}(u) \cup \textsc{fv}(B) \\
\textsc{fv}(A, J) & = & \textsc{fv}(A) \cup (\textsc{fv}(J) - \{0\}) - 1 \\
\textsc{fv}(J_1 \wedge J_2) & = & \textsc{fv}(J_1) \cup \textsc{fv}(J_2)
\end{array}
$$

Definition 2.46 (Set of constants in a judgment: $\textsc{consts}(J)$)**.** We can consider the set of constants occurring in a judgment:

$$\begin{array}{lcl}
\textsc{consts}(\Delta_1 \equiv \Delta_2\,\mathbf{ctx}) & = & \textsc{consts}(\Delta_1) \cup \textsc{fv}(\Delta_2) \\
\textsc{consts}(\cdot \vdash t \equiv u : B) & = & \textsc{consts}(t) \cup \textsc{consts}(u) \cup \textsc{consts}(B) \\
\textsc{consts}(A, J) & = & \textsc{consts}(A) \cup \textsc{consts}(J) \\
\textsc{consts}(J_1 \wedge J_2) & = & \textsc{consts}(J_1) \cup \textsc{consts}(J_2) \\
\ldots &&
\end{array}$$

The remaining cases follow analogously to Definition 2.45 (free variables of a scoped and typed term).

Definition 2.47 (Renaming of a judgment: $J\,\rho$)**.** A renaming can be applied to an entire judgment:

$$\begin{array}{lcl}
(\Delta\,\mathbf{ctx})\,\rho & = & (\Delta\,\rho)\,\mathbf{ctx} \\
(\Delta_1 \equiv \Delta_2\,\mathbf{ctx})\,\rho & = & \Delta_1\,\rho \equiv \Delta_2\,\rho\,\mathbf{ctx} \\
(\cdot \vdash A\ \mathbf{type})\,\rho & = & \cdot \vdash A\,\rho\ \mathbf{type} \\
(\cdot \vdash A \equiv B\ \mathbf{type})\,\rho & = & \cdot \vdash A\,\rho \equiv B\,\rho\ \mathbf{type} \\
(\cdot \vdash t : B)\,\rho & = & \cdot \vdash t\,\rho : B\,\rho \\
(\cdot \vdash t \equiv u : B)\,\rho & = & \cdot \vdash t\,\rho \equiv u\,\rho : B\,\rho \\
(A, J)\,\rho & = & (A\,\rho), J\,(\rho + 1) \\
(J_1 \wedge J_2)\,\rho & = & (J_1\,\rho) \wedge (J_2\,\rho)
\end{array}$$

Definition 2.48 (Hereditary substitution of judgments: $J[u/x]$)**.** A variable can be substituted hereditarily in an entire judgment. We explicitly define hereditary substitution for judgments of the form $(\Delta \vdash t : B)$; the remaining cases are follow analogously to Definition 2.47.

$$\begin{array}{lll}
(\cdot \vdash t : B)[u/x] \Downarrow (\cdot \vdash t' : B') & \textbf{if} & t[u/x] \Downarrow t'\ \textbf{and}\ B[u/x] \Downarrow B' \\
(A, J)[u/x] \Downarrow (A', J') & \textbf{if} & A[u/x] \Downarrow A'\ \textbf{and}\ J[u(+1)/x + 1] \Downarrow J' \\
\ldots &&
\end{array}$$

2.14.2 Substitution and elimination

The following properties concern the behaviour of hereditary substitution and elimination. As explained in the beginning of this section, we assume without proof that those properties marked as postulates would hold, at least in a properly stratified version of the theory.

Postulate 1 (Typing of hereditary substitution)**.** If $\Gamma, x : B, \Delta \vdash t : A$ and $\Gamma \vdash u : B$, then $\Delta[u/x]\Downarrow$, $t[u^{(+|\Delta|)}/x]\Downarrow$, $A[u^{(+|\Delta|)}/x]\Downarrow$, and $\Gamma, \Delta[u/x] \vdash t[u^{(+|\Delta|)}/x] : A[u^{(+|\Delta|)}/x]$.

Postulate 2 (Typing of hereditary application)**.** If $\Sigma; \Gamma \vdash t : \Pi AB$, and $\Sigma; \Gamma \vdash v : A$, then $(t \mathbin{@} v)\Downarrow$, $B[v]\Downarrow$, and $\Sigma; \Gamma \vdash t \mathbin{@} v : B[v]$.

Postulate 3 (Typing of hereditary projection)**.** If $\Sigma; \Gamma \vdash t : \Sigma AB$, then $(t \mathbin{@} .\pi_1)\Downarrow$, with $\Sigma; \Gamma \vdash t \mathbin{@} .\pi_1 : A$ and $(t \mathbin{@} .\pi_2)\Downarrow$, with $B[t \mathbin{@} .\pi_1]\Downarrow$ and $\Sigma; \Gamma \vdash t \mathbin{@} .\pi_2 : B[t \mathbin{@} .\pi_1]$.

Postulate 4 (Congruence of hereditary substitution)**.** If $\Sigma;\Gamma, x : A, \Delta \vdash t_1 \equiv t_2 : B$ and $\Sigma;\Gamma \vdash u_1 \equiv u_2 : A$, then $\Delta[u_1/x]\Downarrow$, $\Delta[u_2/x]\Downarrow$, $t_1[u_1^{(+|\Delta|)}/x]\Downarrow$, $t_2[u_2^{(+|\Delta|)}/x]\Downarrow$, $B[u_1^{(+|\Delta|)}/x]\Downarrow$, $B[u_2^{(+|\Delta|)}/x]\Downarrow$, $\Sigma \vdash \Gamma, \Delta[u_1/x], B[u_1^{(+|\Delta|)}/x] \equiv \Gamma, \Delta[u_2/x], B[u_2^{(+|\Delta|)}/x]\,\mathbf{ctx}$ and $\Sigma;\Gamma, \Delta[u_1/x] \vdash t_1[u_1^{(+|\Delta|)}/x] \equiv t_2[u_2^{(+|\Delta|)}/x] : B[u_1^{(+|\Delta|)}/x]$.

Remark 2.49 (Strengthening by substitution). If $x \notin \mathrm{FV}(J)$, then $J[u/x] = J^{(-1)+x}$.

Postulate 5 (Hereditary substitution commutes)**.** Let $\Sigma;\Gamma, U, \Delta, V, \Xi \vdash t : A$, $\Sigma;\Gamma, U, \Delta \vdash v : V$, $\Sigma;\Gamma \vdash u : U$, and $|\Delta| \notin \mathrm{FV}(V, \Xi \vdash t : A)$.

Let $(\Delta^{\mathrm{a}}, \Xi^{\mathrm{a}} \vdash t^{\mathrm{a}} : A^{\mathrm{a}}) = (\Delta, (\Xi \vdash t : A)[v])[u]$. Then $\Sigma;\Gamma, \Delta[u], (V, \Xi \vdash t : A)^{(-1)+|\Delta|}$, $\Sigma;\Gamma, \Delta[u] \vdash v[u/|\Delta|] : V^{(-1)+|\Delta|}$.

Also, let $\Delta^{\mathrm{b}} = \Delta[u]$, and $(\Xi^{\mathrm{b}} \vdash t^{\mathrm{b}} : A^{\mathrm{b}}) = (\Xi \vdash t : A)^{(-1)+|\Delta|+1}[v[u/|\Delta|]]$. Then $\Sigma \vdash \Gamma, \Delta^{\mathrm{a}}, \Xi^{\mathrm{a}} \equiv \Gamma, \Delta^{\mathrm{b}}, \Xi^{\mathrm{b}}\,\mathbf{ctx}$, $\Sigma \vdash \Gamma, \Delta^{\mathrm{a}}, \Xi^{\mathrm{a}} \vdash A^{\mathrm{a}} \equiv A^{\mathrm{b}}\,\mathbf{type}$, and $\Sigma;\Gamma, \Delta^{\mathrm{a}}, \Xi^{\mathrm{a}} \vdash t^{\mathrm{a}} \equiv t^{\mathrm{b}} : A^{\mathrm{a}}$.

Postulate 6 (Congruence of hereditary application)**.** If $\Sigma;\Gamma \vdash t \equiv u : \Pi AB$ and $\Sigma;\Gamma \vdash v_1 \equiv v_2 : A$, then $(t \mathbin{@} v_1)\Downarrow$, $(u \mathbin{@} v_2)\Downarrow$, $B[v_1]\Downarrow$, and $\Sigma;\Gamma \vdash t \mathbin{@} v_1 \equiv u \mathbin{@} v_2 : B[v_1]$.

Postulate 7 (Congruence of hereditary projection)**.** If $\Sigma;\Gamma \vdash t \equiv u : \Sigma AB$, then:

(i) $(t \mathbin{@} .\pi_1)\Downarrow$, $(u \mathbin{@} .\pi_1)\Downarrow$ and $\Sigma;\Gamma \vdash t \mathbin{@} .\pi_1 \equiv u \mathbin{@} .\pi_1 : A$;

(ii) and also $(t \mathbin{@} .\pi_2)\Downarrow$, $(u \mathbin{@} .\pi_2)\Downarrow$, $B[t \mathbin{@} .\pi_1]\Downarrow$, and $\Sigma;\Gamma \vdash t \mathbin{@} .\pi_2 \equiv u \mathbin{@} .\pi_2 : B[t \mathbin{@} .\pi_1]$.

Postulate 8 (No infinite chains)**.** If $\Sigma;\Gamma \vdash t : A$, then there is no infinite chain of reductions $\Sigma;\Gamma \vdash \_ \longrightarrow_{\delta\eta} \_ : A$ that starts at t. That is, there does not exist an infinite sequence of terms $u_0, u_1, u_2, \ldots$ with $u_0 = t$ such that, for all $i \in \mathbb{N}$, $\Sigma;\Gamma \vdash u_i \longrightarrow_{\delta\eta} u_{i+1} : A$.

Definition 2.50 (Set of free variables, strengthened: $\mathrm{FV}_x(t)$)**.** The set of free variables of t strengthened by x is denoted by $\mathrm{FV}_x(t)$ and is defined as $\mathrm{FV}_x(t) \stackrel{\text{def}}{=} \{y - 1 \mid y \in \mathrm{FV}(t), y > x\} \cup \{y \mid y \in \mathrm{FV}(t), y < x\}$.

Lemma 2.51 (Free variables in hereditary substitution)**.** *The following hold:*

- *If* $t[u/x] \Downarrow r$*, then* $\mathrm{FV}(r) \subseteq \mathrm{FV}_x(t) \cup \mathrm{FV}(u)$.
- *If* $(t \mathbin{@} e) \Downarrow r$*, then* $\mathrm{FV}(r) \subseteq \mathrm{FV}(t) \cup \mathrm{FV}(e)$.

Proof. By mutual induction on the derivations.

- $x[u/x] \Downarrow u$: $\mathrm{FV}(u) \subseteq \mathrm{FV}_x(x) \cup \mathrm{FV}(u)$.
- $y[u/x] \Downarrow y$, $y < x$: We have $y \in \mathrm{FV}(t)$ and $y < x$; therefore, $y \in \mathrm{FV}_x(y)$. Finally, $\mathrm{FV}(y) \subseteq \mathrm{FV}_x(y) \subseteq \mathrm{FV}_x(y) \cup \mathrm{FV}(u)$.
- $y[u/x] \Downarrow (y - 1)$, $y > x$: We have $y \in \mathrm{FV}(t)$ and $y > x$; therefore, $y - 1 \in \mathrm{FV}_x(y)$. Therefore, $\mathrm{FV}(y - 1) \subseteq \mathrm{FV}_x(y) \subseteq \mathrm{FV}_x(y) \cup \mathrm{FV}(u)$.

- $t = \alpha$, $t = \mathbb{0}$, $t =$ if, $t =$ true, $t =$ false, $t =$ Bool or $t =$ Set: $\text{FV}(t) = \emptyset \subseteq S \cup \text{FV}(u)$.
- $(f\,t)[u/x]\Downarrow r$, with $f[u/x]\Downarrow r_1$ and $t[u/x]\Downarrow r_2$. By the induction hypothesis, $\text{FV}(r_1) \subseteq \text{FV}_x(f) \cup \text{FV}(u)$, $\text{FV}(r_2) \subseteq \text{FV}_x(t) \cup \text{FV}(u)$, and $\text{FV}(r) \subseteq \text{FV}(r_1) \cup \text{FV}(r_2)$. Therefore, $\text{FV}(r) \subseteq \text{FV}_x(f) \cup \text{FV}_x(t) \cup \text{FV}(u)$. By Definition 2.18 (free variables in a term), $\text{FV}(f\,t) = \text{FV}(f) \cup \text{FV}(t)$. Thus, $\text{FV}_x(f\,t) = \text{FV}_x(f) \cup \text{FV}_x(t)$. Therefore, $\text{FV}(r) \subseteq \text{FV}_x(f\,t) \cup \text{FV}(u)$.
- $(f\,.\pi_1)[u/x] \Downarrow r$, with $f[u/x] \Downarrow r_1$ and $(r_1 \;@\; .\pi_1) \Downarrow r$. By the induction hypothesis, $\text{FV}(r) \subseteq \text{FV}(r_1)$, and $\text{FV}(r_1) \subseteq \text{FV}_x(f) \cup \text{FV}(u)$. By Definition 2.18 (free variables in a term), $\text{FV}(f\,.\pi_1) = \text{FV}(f)$; therefore, $\text{FV}_x(f\,.\pi_1) = \text{FV}_x(f)$. Thus, $\text{FV}(r) \subseteq \text{FV}_x(f\,.\pi_1) \cup \text{FV}(u)$.
- $(f\,.\pi_2)[u/x] \Downarrow r$, with $f[u/x] \Downarrow r_1$ and $(r_1 \;@\; .\pi_2) \Downarrow r$: Analogously to the previous case, replacing $.\pi_1$ by $.\pi_2$.
- $(\lambda.t)[u/x] \Downarrow (\lambda.r)$, with $t[u(+1)/x+1] \Downarrow r$: By the induction hypothesis, $\text{FV}(r) \subseteq \text{FV}_{x+1}(t) \cup \text{FV}(u(+1))$. By Definition 2.18 (free variables in a term), $\text{FV}(\lambda.r) = \text{FV}(r) - 1$ and $\text{FV}(\lambda.t) = \text{FV}(t) - 1$. By definition, $\text{FV}_x(\lambda.t) = \text{FV}_{x+1}(t) - 1$. By Remark 2.28, $\text{FV}(u(+1)) - 1 = \text{FV}(u)(+1) - 1 = \text{FV}(u)$. Therefore, $\text{FV}(\lambda.r) = \text{FV}(r) - 1 \subseteq (\text{FV}_{x+1}(t) \cup \text{FV}(u(+1))) - 1 = (\text{FV}_{x+1}(t) - 1) \cup (\text{FV}(u(+1)) - 1) = \text{FV}_x(\lambda.t) \cup \text{FV}(u)$.
- $(\Pi AB)[u/x] \Downarrow (\Pi A'B')$, with $A[u/x] \Downarrow A'$ and $B[u(+1)/x+1] \Downarrow B'$: By the induction hypothesis, $\text{FV}(A') \subseteq \text{FV}_x(A) \cup \text{FV}(u)$, and $\text{FV}(B') \subseteq \text{FV}_{x+1}(B) \cup \text{FV}(u(+1))$. By Definition 2.18 (free variables in a term), $\text{FV}(\Pi A'B') = \text{FV}(A') \cup (\text{FV}(B') - 1) \subseteq \text{FV}_x(A) \cup \text{FV}(u) \cup (\text{FV}_{x+1}(B) - 1) \cup (\text{FV}(u(+1)) - 1) = \text{FV}_x(\Pi AB) \cup \text{FV}(u)$.
- $(\Sigma AB)[u/x] \Downarrow (\Sigma A'B')$, with $A[u/x] \Downarrow A'$ and $B[u(+1)/x+1] \Downarrow B'$: Analogous to the previous case, replacing Π with Σ.
- $\langle t_1, t_2\rangle[u/x] \Downarrow \langle t_1', t_2'\rangle$, with $t_1[u/x] \Downarrow t_1'$ and $t_2[u/x] \Downarrow t_2'$. By the induction hypothesis, $\text{FV}(t_1') \subseteq \text{FV}_x(t_1) \cup \text{FV}(u)$ and $\text{FV}(t_2') \subseteq \text{FV}_x(t_2) \cup \text{FV}(u)$. Therefore, $\text{FV}(\langle t_1', t_2'\rangle) = \text{FV}(t_1') \cup \text{FV}(t_2') \subseteq \text{FV}_x(t_1) \cup \text{FV}_x(t_2) \cup \text{FV}(u) = \text{FV}_x(\langle t_1, t_2\rangle) \cup \text{FV}(u)$.
- $h\,\vec{e} \;@\; e' \Downarrow (h\,\vec{e}\,e')$: By Definition 2.18 (free variables in a term), we have $\text{FV}(h\,\vec{e}\,e') = \text{FV}(h\,\vec{e}) \cup \text{FV}(e')$.
- $\langle t_1, t_2\rangle \;@\; .\pi_1 \Downarrow t_1$: $\text{FV}(t_1) \subseteq \text{FV}(t_1) \cup \text{FV}(t_2) \cup \emptyset = \text{FV}(\langle t_1, t_2\rangle) \cup \text{FV}(.\pi_1)$.
- $\langle t_1, t_2\rangle \;@\; .\pi_2 \Downarrow t_2$: $\text{FV}(t_2) \subseteq \text{FV}(t_1) \cup \text{FV}(t_2) \cup \emptyset = \text{FV}(\langle t_1, t_2\rangle) \cup \text{FV}(.\pi_2)$.
- $\lambda.t \;@\; u \Downarrow r$ with $t[u/0] \Downarrow r$: By the induction hypothesis, $\text{FV}(r) \subseteq \text{FV}_0(t) \cup \text{FV}(u)$. By definition, $\text{FV}_0(t) = \text{FV}(t) - 1 = \text{FV}(\lambda.t)$. Therefore, $\text{FV}(r) \subseteq \text{FV}(\lambda.t) \cup \text{FV}(u)$.

□

Postulate 9 (Commuting of hereditary substitution and application)**.** Assume $\Sigma; \Gamma, V \vdash u : \Pi\vec{A}B$, and $\vec{t}$ such that $\Sigma; \Gamma, V \vdash t_i : A[\vec{t}_{1,\ldots,i-1}]$. Finally, let $\vec{v}$ be such that $\Sigma; \Gamma, V \vdash v : V$. Then $(u \;@\; \vec{t})[v] = (u[v] \;@\; t_1[v] \ldots t_n[v])$.

2.14.3 Typing and equality

Lemma 2.52 (Π inversion)**.** *If* $\Sigma;\Gamma \vdash \Pi AB : T$*, then* $\Sigma;\Gamma \vdash T \equiv \text{Set}$ **type***,* $\Sigma;\Gamma \vdash A : \text{Set}$ *and* $\Sigma;\Gamma, A \vdash B : \text{Set}$*. Also, by Remark 2.15 (there is only set), if* $\Sigma;\Gamma \vdash \Pi AB$ **type***, then* $\Sigma;\Gamma \vdash A$ **type** *and* $\Sigma;\Gamma, A \vdash B$ **type***.*

Proof. By induction on the derivation:

- Case CONV: By the premises of the rule, $\Sigma;\Gamma \vdash \Pi AB : T'$ for some T', and $\Sigma;\Gamma \vdash T' \equiv T$ **type**. By the induction hypothesis, $\Sigma;\Gamma \vdash A : \text{Set}$, $\Sigma;\Gamma, A \vdash B : \text{Set}$ and $\Sigma;\Gamma \vdash T' \equiv \text{Set}$ **type**. By transitivity and symmetry of equality, $\Sigma;\Gamma \vdash T \equiv \text{Set}$ **type**.
- Case PI: $T = \text{Set}$, and by the premises of the rule, $\Sigma;\Gamma \vdash A : \text{Set}$ and $\Sigma;\Gamma, A \vdash B : \text{Set}$.

□

Postulate 10 (Injectivity of Π)**.** If $\Sigma;\Gamma \vdash \Pi AB \equiv \Pi A'B'$ **type**, then $\Sigma;\Gamma \vdash A \equiv A'$ **type** and $\Sigma;\Gamma, A \vdash B \equiv B'$ **type**. Also, by Remark 2.15 (there is only set), if $\Sigma;\Gamma \vdash \Pi AB \equiv \Pi A'B' : \text{Set}$, then $\Sigma;\Gamma \vdash A \equiv A' : \text{Set}$ and $\Sigma;\Gamma, A \vdash B \equiv B' : \text{Set}$.

Lemma 2.53 (Σ inversion)**.** *If* $\Sigma;\Gamma \vdash \Sigma AB : T$*, then* $\Sigma;\Gamma \vdash T \equiv \text{Set}$ **type** $\Sigma;\Gamma \vdash A : \text{Set}$ *and* $\Sigma;\Gamma, A \vdash B : \text{Set}$*. Also, by Remark 2.15 (there is only set), if* $\Sigma;\Gamma \vdash \Sigma AB$ **type***, then* $\Sigma;\Gamma \vdash A$ **type** *and* $\Sigma;\Gamma, A \vdash B$ **type***.*

Proof. Analogous to the proof for Lemma 2.52 (Π inversion). □

Postulate 11 (Injectivity of Σ)**.** If $\Sigma;\Gamma \vdash \Sigma AB \equiv \Sigma A'B'$ **type**, then $\Sigma;\Gamma \vdash A \equiv A'$ **type** and $\Sigma;\Gamma, A \vdash B \equiv B'$ **type**. Also, by Remark 2.15 (there is only set), if $\Sigma;\Gamma \vdash \Sigma AB \equiv \Sigma A'B' : \text{Set}$, then $\Sigma;\Gamma \vdash A \equiv A' : \text{Set}$ and $\Sigma;\Gamma, A \vdash B \equiv B' : \text{Set}$.

Lemma 2.54 (Term equality is an equivalence relation)**.** *Judgmental equality of terms* $(\Sigma;\Gamma \vdash \_ \equiv \_ : A$ *is a reflexive, symmetric and transitive relation.*

- *Reflexivity: If* $\Sigma;\Gamma \vdash t : A$*, then* $\Sigma;\Gamma \vdash t \equiv t : A$*.*
- *Symmetry: If* $\Sigma;\Gamma \vdash t \equiv u : A$*, then* $\Sigma;\Gamma \vdash u \equiv t : A$*.*
- *Transitivity: If* $\Sigma;\Gamma \vdash t \equiv u : A$ *and* $\Sigma;\Gamma \vdash u \equiv v : A$*, then* $\Sigma;\Gamma \vdash t \equiv v : A$*.*

Proof. Reflexivity follows by induction on the typing derivation for t. Each typing rule [X] is replaced by the corresponding equality rule [X]-EQ.

Symmetry and transitivity are rules themselves. □

Remark 2.55 (Type equality is an equivalence relation). Judgmental equality of types ($\Sigma;\Gamma \vdash \_ \equiv \_$ **type** is a reflexive, symmetric and transitive relation:

- Reflexivity: If $\Sigma;\Gamma \vdash A$ **type**, then $\Sigma;\Gamma \vdash A \equiv A$ **type**.
- Symmetry: If $\Sigma;\Gamma \vdash A \equiv B$ **type**, then $\Sigma;\Gamma \vdash B \equiv A$ **type**.

- Transitivity: If $\Sigma;\Gamma \vdash A \equiv B : \mathbf{type}$ and $\Sigma;\Gamma \vdash B \equiv C\ \mathbf{type}$, then $\Sigma;\Gamma \vdash A \equiv C\ \mathbf{type}$.

Proof. By Remark 2.15 (there is only set) and Lemma 2.54 (term equality is an equivalence relation). □

Lemma 2.56 (Neutral inversion)**.**

- *If* $\Sigma;\Gamma \vdash f\,t\,\vec{e} : T$*, then* $\Sigma;\Gamma \vdash f : \Pi AB$ *and* $\Sigma;\Gamma \vdash t : A$ *for some* A *and* B*, with* $B[t]\Downarrow$*.*
- *If* $\Sigma;\Gamma \vdash f.\pi_1\,\vec{e} : T$ *then* $\Sigma;\Gamma \vdash f : \Sigma AB$*.*
- *If* $\Sigma;\Gamma \vdash f.\pi_2\,\vec{e} : T$*, then* $\Sigma;\Gamma \vdash f : \Sigma AB$ *for some* A*,* B*, with* $B[f]\Downarrow$*.*

Proof. We consider the first case; the other two are analogous.

By induction on the derivation, we obtain that $\Sigma;\Gamma \vdash f\,t : T$, with APP being the last rule in the derivation. By the premises of the rule, $\Sigma;\Gamma \vdash f : \Pi AB$ and $\Sigma;\Gamma \vdash t : A$ for some A, B, and $B[t]\Downarrow$. □

Lemma 2.57 (Type of λ-abstraction)**.** *If* $\Sigma;\Gamma \vdash \lambda.t : T$*, then there are* A*,* B *such that* $\Sigma;\Gamma \vdash \Pi AB \equiv T\ \mathbf{type}$ *and* $\Sigma;\Gamma, A \vdash t : B$*.*

Proof. By induction on the derivation (as in the proof for Lemma 2.52), we obtain that $\Sigma;\Gamma \vdash \lambda.t : \Pi AB$, with ABS being the last rule in the derivation, and $\Sigma;\Gamma \vdash \Pi AB \equiv T\ \mathbf{type}$. By the premises of the rule, $\Sigma;\Gamma, A \vdash t : B$. □

Corollary 2.58 (Iterated λ-inversion)**.** *If* $\Sigma;\Gamma \vdash \lambda^n.t : T$*, then* $\Sigma;\Gamma \vdash T \equiv \Pi\vec{A^n}B\ \mathbf{type}$*, with* $\Sigma;\Gamma, \vec{A^n} \vdash t : B$*.*

Proof. By induction on n, using Lemma 2.57. □

Lemma 2.59 (Abstraction equality inversion)**.** *We have* $\Sigma;\Gamma \vdash \lambda^n.t \equiv \lambda^n.u : \Pi\vec{A^n}B$*, if and only if* $\Sigma;\Gamma, \vec{A} \vdash t \equiv u : B$*.*

Proof. $\Rightarrow$ By induction on n, using Postulate 2 (typing of hereditary application).

$\Leftarrow$ By iterated application of the ABS-EQ rule.

□

Lemma 2.60 (Type of a pair)**.** *If* $\Sigma;\Gamma \vdash t : T$*, then there are* A *and* B *such that* $\Sigma;\Gamma \vdash \Sigma AB \equiv T\ \mathbf{type}$*,* $\Sigma;\Gamma \vdash t_1 : A$*.* $B[t_1] \Downarrow$ *and* $\Sigma;\Gamma \vdash t_2 : B[t_1]$*.*

Proof. By induction on the derivation (as in the proof for Lemma 2.52), we obtain that $\Sigma;\Gamma \vdash \langle t_1, t_2\rangle : \Sigma AB$, with PAIR being the last rule in the derivation, and $\Sigma;\Gamma \vdash \Sigma AB \equiv T\ \mathbf{type}$. By the premises of the rule, $\Sigma;\Gamma \vdash t_1 : A$, $B[t_1] \Downarrow$ and $\Sigma;\Gamma \vdash t_2 : B[t_1]$. □

2.14.4 Contexts

Adding new variables to a context does not invalidate existing judgments:

Remark 2.61 (Reflexivity of context equality). Context equality is reflexive (i.e. if $\Sigma \vdash \Gamma\,\mathbf{ctx}$, then $\Sigma \vdash \Gamma \equiv \Gamma\,\mathbf{ctx}$).

Proof. By induction on the derivation of $\Sigma \vdash \Gamma\,\mathbf{ctx}$, using reflexivity of the type equality (Remark 2.55). □

Lemma 2.62 (Context weakening)**.** *Let J be a judgment. If $\Sigma; \Gamma_1 \vdash J$, $\Sigma \vdash \Gamma_1, \Gamma_2\,\mathbf{ctx}$, and $|\Gamma_2| = n$, then $\Sigma; \Gamma_1, \Gamma_2 \vdash J^{(+n)}$.*

Proof. By induction on the derivation of $\Sigma; \Gamma_1 \vdash J$. For most cases, it suffices to apply the induction hypothesis to the premises, and use the same rule to derive the conclusion. We show a representative subset of cases below:

- CTX-EMPTY: Then $\Gamma_1 = \cdot$ and $J = \cdot\,\mathbf{ctx}$. By the assumption, $\Sigma \vdash \Gamma_1, \Gamma_2\,\mathbf{ctx}$. Note that $\cdot^{(+n)} = \cdot$. Therefore, $\Gamma_1, \Gamma_2, \cdot^{(+n)} = \Gamma_1, \Gamma_2$, thus $\Sigma \vdash \Gamma_1, \Gamma_2, \cdot^{(+n)}\,\mathbf{ctx}$.

- CTX-VAR: Then $J = \Delta, A\,\mathbf{ctx}$, $\Sigma \vdash \Gamma_1, \Delta, A\,\mathbf{ctx}$, with $\Sigma; \Gamma_1, \Delta \vdash A\ \mathbf{type}$ and $\Sigma \vdash \Gamma_1, \Delta\,\mathbf{ctx}$. Let $m = |\Delta|$. By the induction hypothesis, we have $\Sigma; \Gamma_1, \Gamma_2, (\Delta \vdash A\ \mathbf{type})^{(+n)}$. This gives $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash A^{((+n)+m)}\ \mathbf{type}$ and $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)}\,\mathbf{ctx}$. By the CTX-VAR rule, we have $\Sigma \vdash \Gamma_1, \Gamma_2, \Delta^{(+n)}, A^{((+n)+m)}\,\mathbf{ctx}$; that is, $\Sigma; \Gamma_1, \Gamma_2 \vdash (\Delta, A\,\mathbf{ctx})^{(+n)}$

- TYPE, TYPE-EQ: Apply the induction hypothesis to the premises, then use the same derivation rule.

- CTX-EMPTY-EQ: Analogous to CTX-EMPTY, use Remark 2.61 (reflexivity of context equality) to show $\Sigma \vdash \Gamma_2 \equiv \Gamma_2\,\mathbf{ctx}$.

- CTX-VAR-EQ: Analogous to CTX-VAR.

- PAIR: $(\Sigma; \Gamma_1, \Delta \vdash \langle t, u\rangle : \Sigma AB)$, with $B[t]\Downarrow$. Let $m = |\Delta|$. By the induction hypothesis, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash t^{(+n)+m} : A^{(+n)+m}$, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)}, A^{((+n)+m)} \vdash B^{((+n)+m)}\ \mathbf{type}$, and $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash u^{((+n)+m)} : B[t]^{((+n)+m)}$. By Lemma 2.39 (hereditary substitution and application commute with renaming), $B[t]^{((+n)+m)} = B^{((+n)+m+1)}[t^{((+n)+m)}]$. By the PAIR rule and Definition 2.26 (application of a renaming to a term), $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash \langle t, u\rangle^{((+n)+m)} : (\Sigma AB)^{((+n)+m)}$.

- HEAD, then VAR: $\Sigma; \Gamma_1, \Delta \vdash x : A(+(x+1))$, with $\Sigma; \Gamma_1, \Delta \vdash x \Rightarrow A(+(x+1))$ for some A. We want to show $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash x^{((+n)+m)} : A(+(x+1))^{((+n)+m)}$. $\Sigma \vdash \Gamma_1, \Delta\,\mathbf{ctx}$, so, by the induction hypothesis, $\Sigma; \Gamma_1, \Gamma_2 \vdash \Delta(+n)\,\mathbf{ctx}$. Let $m = |\Delta|$.

  - If $x < |\Delta|$: Then we have $\Sigma; \Gamma_1, \Delta', A, \Delta'' \vdash x \Rightarrow A(+(x+1))$, where $x = |\Delta''|$. By the VAR rule, $\Sigma; \Gamma_1, \Gamma_2, \Delta'^{(+n)}, A^{((+n)+|\Delta'|)}$, $\Delta''^{((+n)+|\Delta'|+1)} \vdash x \Rightarrow A^{((+n)+|\Delta'|)(+(x+1))}$. Note that $|\Delta'| + x + 1 = m$. Therefore, this is the same as $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash x^{((+n)+m)} \Rightarrow A^{(+(x+1))((+n)+m)}$.

  - If $x \geq |\Delta|$: Then $\Sigma; \Gamma_1', A, \Gamma_1'', \Delta \vdash x \Rightarrow A^{(+(x+1))}$, where $x = |\Gamma_1'', \Delta|$. Also, by the VAR rule, $\Sigma; \Gamma_1', A, \Gamma_1'', \Gamma_2, \Delta^{(+n)} \vdash (x+n) \Rightarrow A^{(+(x+n+1))}$, which is the same as $\Sigma; \Gamma_1', A, \Gamma_1'', \Gamma_2, \Delta^{(+n)} \vdash x^{((+n)+m)} \Rightarrow A^{(+(x+1))((+n)+m)}$.

  By the HEAD rule, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash x^{((+n)+m)} : A^{(+(x+1))((+n)+m)}$.

- DELTA-META: $\Sigma; \Gamma_1, \Delta \vdash \alpha\, \vec{e} \equiv t' : T$, with $\alpha := t : A \in \Sigma$, $\Sigma; \Gamma_1, \Delta \vdash \alpha\, \vec{e} : T$, $t \mathbin{@} \vec{e} \Downarrow t'$ and $\Sigma; \Gamma_1, \Delta \vdash t' : T$. By the induction hypothesis, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash (\alpha\, \vec{e})^{((+n)+m)} : T^{((+n)+m)}$, that is, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash \alpha\, \vec{e}^{((+n)+m)} : T^{((+n)+m)}$. Also by the induction hypothesis, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash t'^{((+n)+m)} : T^{((+n)+m)}$.

  By Lemma 2.39 (hereditary substitution and application commute with renaming), $t^{((+n)+m)} \mathbin{@} \vec{e}^{((+n)+m)} \Downarrow t'^{((+n)+m)}$, that is, $t \mathbin{@} \vec{e}^{((+n)+m)} \Downarrow t'^{((+n)+m)}$.

  By the DELTA-META rule, $\Sigma; \Gamma_1, \Gamma_2, \Delta^{(+n)} \vdash \alpha\, \vec{e}^{((+n)+m)} \equiv t'^{((+n)+m)} : T^{((+n)+m)}$.

□

Lemma 2.63 (Preservation of judgments by type conversion)**.** *Let J be a judgment such that $\Sigma; \Gamma \vdash J$.*

- *If $\Sigma \vdash \Gamma \equiv \Gamma'$ **ctx**, then $\Sigma; \Gamma' \vdash J$.*
- *Furthermore, if $J = (t : A)$ (or $J = (t \equiv u : A)$), and $\Sigma; \Gamma \vdash A \equiv A'$ **type** (that is, $\Sigma \vdash \Gamma, A \equiv \Gamma', A'$ **ctx**), then $\Sigma; \Gamma' \vdash t : A'$ (respectively, $\Sigma; \Gamma' \vdash t \equiv u : A'$).*

Proof. We prove the simpler case where the type of only one variable changes (that is, $\Gamma = \Gamma_1, A, \Gamma_2$, $\Gamma' = \Gamma_1, A', \Gamma_2$, and $\Sigma; \Gamma_1 \vdash A \equiv A'$ **type**). For the general case, we apply the lemma to each of the variables whose types differ between Γ and Γ'.

We proceed by induction on the derivation of J, applying the rules as is. The change of the type of a variable is only relevant in two cases:

- Rule VAR: This rule is always followed by either HEAD or HEAD-EQ. We apply Lemma 2.62 (context weakening) $\Sigma; \Gamma_1 \vdash A \equiv A'$ **type**, and then use either the CONV or the CONV-EQ rules to obtain a judgment in the new context.
- Rule CTX-VAR-EQ: By reflexivity, $\Sigma; \Gamma_1 \vdash A' \equiv A'$ **type**.

$$\frac{\Sigma \vdash \Gamma_1 \equiv \Gamma_1 \textbf{ ctx} \qquad \Sigma; \Gamma_1 \vdash A' \equiv A' \textbf{ type}}{\Sigma \vdash \Gamma_1, A' \equiv \Gamma_1, A' \textbf{ ctx}}\ \text{CTX-VAR-EQ}$$

For the second statement, we apply the CONV or CONV-EQ rules to the whole judgment. □

Lemma 2.64 (Equality of contexts is an equivalence relation)**.** *Context equality is a reflexive, transitive and symmetric relation.*

Proof. Reflexivity follows from Remark 2.61 (reflexivity of context equality).

Symmetry and transitivity follow by induction on the corresponding derivations, by applying Lemma 2.63 (preservation of judgments by type conversion), and symmetry and transitivity of type equality (Remark 2.55). □

Lemma 2.65 (No extraneous variables in term)**.** *If* $\Sigma; \Gamma \vdash t : A$*, then* $\text{FV}(t) \subseteq \{0, ..., |\Gamma| - 1\}$.

Proof. By induction on the derivation of $\Sigma; \Gamma \vdash t : A$, for each variable x freely occurring in t there will be an instance of the VAR rule of the following form:

$$\frac{\Sigma \vdash \Gamma, \Delta\ \mathbf{ctx} \qquad \Gamma, \Delta = \Gamma_1, A, \Gamma_2, \Delta \qquad n = |\Gamma_2, \Delta|}{\Sigma; \Gamma \vdash (x + |\Delta|) \Rightarrow A^{(+(n+1))}}\ \text{VAR}$$

Because $n = x + |\Delta| = |\Gamma_2| + |\Delta|$, we have $x = |\Gamma_2| \in \{0, ..., |\Gamma| - 1\}$. □

Corollary 2.66 (The signature is closed)**.** *Let* Σ *be a well-formed signature (*Σ **sig***). If* $\alpha : A \in \Sigma$ *or* $\mathbb{a} : A \in \Sigma$*, then* $\text{FV}(A) = \emptyset$*. Also, if* $\alpha := t : A \in \Sigma$*, then* $\text{FV}(t) = \text{FV}(A) = \emptyset$*.*

2.14.5 Signatures

Definition 2.67 (Signature subsumption: $\Sigma \subseteq \Sigma'$)**.** If Σ **sig**, Σ' **sig**, and all the declarations in Σ are present in Σ', then we say $\Sigma \subseteq \Sigma'$.

That is, we have $\Sigma \subseteq \Sigma'$ if, for every Σ_1, Σ_2:

(i) if $\Sigma = \Sigma_1, \mathbb{a} : A, \Sigma_2$, there are Σ'_1 and Σ'_2 such that $\Sigma' = \Sigma'_1, \mathbb{a} : A, \Sigma'_2$.

(ii) and, if $\Sigma = \Sigma_1, \alpha : A, \Sigma_2$, then there are Σ'_1 and Σ'_2 such that $\Sigma' = \Sigma'_1, \alpha : A, \Sigma'_2$.

(iii) and, if $\Sigma = \Sigma_1, \alpha := t : A, \Sigma_2$, then there are Σ'_1 and Σ'_2 such that $\Sigma' = \Sigma'_1, \alpha := t : A, \Sigma'_2$.

Definition 2.68 (Well-formed reordering)**.** We say that Σ is a well-formed reordering of Σ' if $\Sigma \subseteq \Sigma'$ and $\Sigma' \subseteq \Sigma$.

Lemma 2.69 (Signature weakening)**.** *Let* Σ*,* Σ' *be signatures such that* $\Sigma \subseteq \Sigma'$*, and* J *a judgment. If* $\Sigma \vdash J$*, then* $\Sigma' \vdash J$*.*

Proof. By induction on the derivation. The constructed derivation for $\Sigma' \vdash J$ consists of the same rules as the derivation for $\Sigma \vdash J$. □

Lemma 2.70 (Piecewise well-formedness of typing judgments)**.** *If a typing or well-formedness judgment holds (i.e. has a derivation), then each of its elements are themselves well-formed or well-typed.*

More specifically:

(i) If $\Sigma \vdash \Gamma\ \mathbf{ctx}$*, then* Σ **sig**.

(ii) If $\Sigma; \Gamma \vdash h \Rightarrow A$*, then* $\Sigma; \Gamma \vdash A$ **type**.

(iii) If $\Sigma; \Gamma \vdash A$ **type***, then* $\Sigma \vdash \Gamma\ \mathbf{ctx}$.

(iv) *If* $\Sigma; \Gamma \vdash A \equiv B$ **type**, *then* $\Sigma; \Gamma \vdash A$ **type** *and* $\Sigma; \Gamma \vdash B$ **type**.

(v) *If* $\Sigma; \Gamma \vdash t : A$, *then* $\Sigma; \Gamma \vdash A$ **type**.

(vi) *If* $\Sigma; \Gamma \vdash t \equiv u : A$, *then* $\Sigma; \Gamma \vdash t : A$ *and* $\Sigma; \Gamma \vdash u : A$.

Proof. The first two statements are proved individually by structural induction on the derivation.

(i) *If* $\Sigma \vdash \Gamma$ **ctx**, *then* Σ **sig**.

- CTX-EMPTY: By the premise, Σ **sig**.
- CTX-VAR: Then $\Gamma = \Gamma', A$. By induction on $\Sigma \vdash \Gamma'$ **ctx**, we have Σ **sig**.

(ii) *If* $\Sigma; \Gamma \vdash h \Rightarrow A$, *then* $\Sigma; \Gamma \vdash A$ **type** *and* $\Sigma \vdash \Gamma$ **ctx**.
In all five cases (VAR, META_1, META_2, ATOM and IF), by the premises of the rule, $\Sigma \vdash \Gamma$ **ctx**. It remains to show $\Sigma; \Gamma \vdash A$ **type**.

- VAR: By Remark 2.13 (context inversion), $\Sigma; \Gamma_1 \vdash A$ **type**. By Lemma 2.62 (context weakening), $\Sigma; \Gamma \vdash A$ **type**.
- META_1, META_2, ATOM: By Remark 2.5 (signature inversion), $\Sigma = \Sigma_1, \Sigma_2$ with $\Sigma_1; \cdot \vdash A$ **type**. By Lemma 2.69 (signature weakening), $\Sigma; \cdot \vdash A$ **type**. By Lemma 2.62, $\Sigma; \Gamma \vdash A$ **type**.
- IF: By the PI, BOOL, SET, VAR, HEAD, TRUE, FALSE and APP rules, and then the TYPE rule, $\Sigma; \Gamma \vdash (X : \text{Bool} \to \text{Set}) \to (y : \text{Bool}) \to X\,\text{true} \to X\,\text{false} \to X\,y$ **type**.

For the rest, we prove a stronger version using mutual induction.

iii) *If* $\Sigma; \Gamma \vdash A$ **type**, *then* $\Sigma \vdash \Gamma$ **ctx**.

By mutual induction on the depth of the derivation.

- TYPE: Follows by induction on $\Sigma; \Gamma \vdash A : \text{Set}$.

iv) *If* $\Sigma; \Gamma \vdash A \equiv B$ **type**, *then* $\Sigma; \Gamma \vdash A$ **type**, $\Sigma; \Gamma \vdash B$ **type**, *and* $\Sigma \vdash \Gamma$ **ctx**.

By mutual induction on the depth of the derivation.

- TYPE-EQ: By induction on $\Sigma; \Gamma \vdash A \equiv B : \text{Set}$, we have $\Sigma; \Gamma$ **ctx**, $\Sigma; \Gamma \vdash A : \text{Set}$ and $\Sigma; \Gamma \vdash B : \text{Set}$. By the TYPE rule, $\Sigma; \Gamma \vdash A$ **type** and $\Sigma; \Gamma \vdash B$ **type**.

v) *If* $\Sigma; \Gamma \vdash t : A$, *then* $\Sigma; \Gamma \vdash A$ **type** *and* $\Sigma \vdash \Gamma$ **ctx**.

- BOOL, SET: By the premises, $\Sigma \vdash \Gamma$ **ctx**. By the SET and TYPE rules, $\Sigma; \Gamma \vdash \text{Set}$ **type**.
- PI, SIGMA: By induction on $\Sigma; \Gamma \vdash A : \text{Set}$.
- TRUE, FALSE: By the premises, $\Sigma \vdash \Gamma$ **ctx**. By the BOOL and TYPE rules, $\Sigma; \Gamma \vdash \text{Set}$ **type**.

- ABS: By induction, $\Sigma \vdash \Gamma, A\,\mathbf{ctx}$ and $\Sigma; \Gamma, A \vdash B\,\mathbf{type}$. By Remark 2.13 (context inversion), $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma; \Gamma \vdash A\,\mathbf{type}$. By Remark 2.15, $\Sigma; \Gamma \vdash A : \mathrm{Set}$ and $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$. By the PI and TYPE rules, $\Sigma; \Gamma \vdash \Pi AB\,\mathbf{type}$.
- PAIR: By induction, $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma; \Gamma \vdash A\,\mathbf{type}$. By Remark 2.15, $\Sigma; \Gamma \vdash A : \mathrm{Set}$. By the premises, $\Sigma; \Gamma \vdash B : \mathrm{Set}$. By the SIGMA and TYPE rules, $\Sigma; \Gamma \vdash \Sigma AB\,\mathbf{type}$.
- HEAD: Follows by the premise $\Sigma; \Gamma \vdash h \Rightarrow A$ and (ii).
- PROJ1: By induction, $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma; \Gamma \vdash \Sigma AB\,\mathbf{type}$. By Lemma 2.53 ($\Sigma$ inversion) and Remark 2.15, $\Sigma; \Gamma \vdash A\,\mathbf{type}$.
- PROJ2: By induction and lemma 2.53 (Σ inversion), $\Sigma \vdash \Gamma\,\mathbf{ctx}$, and $\Sigma; \Gamma, A \vdash B\,\mathbf{type}$. By Remark 2.15, $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$. By the PROJ1 rule, $\Sigma; \Gamma \vdash f.\pi_1 : A$. By Postulate 1 and the TYPE rule, $\Sigma; \Gamma \vdash B[f.\pi_1]\,\mathbf{type}$.
- APP: By induction and lemma 2.52 (Π inversion), $\Sigma \vdash \Gamma\,\mathbf{ctx}$, and $\Sigma; \Gamma, A \vdash B\,\mathbf{type}$. By Remark 2.15, $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$. By the premise, $\Sigma; \Gamma \vdash t : A$. By Postulate 1 and the TYPE rule, $\Sigma; \Gamma \vdash B[f.\pi_1]\,\mathbf{type}$.
- CONV: By induction on $\Sigma; \Gamma \vdash A \equiv B\,\mathbf{type}$, we have $\Sigma \vdash \Gamma\,\mathbf{ctx}$. $\Sigma; \Gamma \vdash B\,\mathbf{type}$.

vi) *If* $\Sigma; \Gamma \vdash t \equiv u : A$, *then* $\Sigma; \Gamma \vdash t : A$, $\Sigma; \Gamma \vdash u : A$, $\Sigma; \Gamma \vdash A\,\mathbf{type}$ *and* $\Sigma \vdash \Gamma\,\mathbf{ctx}$.

- BOOL-EQ (resp. SET-EQ): By the BOOL (resp. SET) rule, $\Sigma; \Gamma \vdash \mathrm{Bool} : \mathrm{Set}$ (resp. $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$). By the premise, $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By the SET and TYPE rules, $\Sigma; \Gamma \vdash \mathrm{Set}\,\mathbf{type}$.
- PI-EQ, SIGMA-EQ: By the induction hypothesis, $\Sigma; \Gamma \vdash A : \mathrm{Set}$ and $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$. By the PI (resp. SIGMA) rule, $\Sigma; \Gamma \vdash \Pi AB : \mathrm{Set}$ (resp. $\Sigma; \Gamma \vdash \Sigma AB : \mathrm{Set}$). Analogously, $\Sigma; \Gamma \vdash \Pi A'B' : \mathrm{Set}$ (resp. $\Sigma; \Gamma \vdash \Sigma A'B' : \mathrm{Set}$). By induction, $\Sigma; \Gamma \vdash \mathrm{Set}\,\mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$.
- TRUE-EQ (resp. FALSE-EQ): By the TRUE (resp. FALSE) rule, $\Sigma; \Gamma \vdash \mathrm{true} : \mathrm{Bool}$ (resp. $\Sigma; \Gamma \vdash \mathrm{false} : \mathrm{Bool}$). By the premise, $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By the BOOL and TYPE rules, $\Sigma; \Gamma \vdash \mathrm{Bool} :\ \mathbf{type}$.
- ABS-EQ: By the induction hypothesis, $\Sigma; \Gamma \vdash t : B$, $\Sigma; \Gamma, A \vdash B\,\mathbf{type}$ and $\Sigma \vdash \Gamma, A\,\mathbf{ctx}$. By the ABS rule, $\Sigma; \Gamma \vdash \lambda.t : \Pi AB$. Analogously, $\Sigma; \Gamma \vdash \lambda.u : \Pi AB$.

  By Remark 2.13 (context inversion), $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma; \Gamma \vdash A\,\mathbf{type}$. By Remark 2.15 and the PI rule, $\Sigma; \Gamma \vdash \Pi AB\,\mathbf{type}$.
- PAIR-EQ: By the induction hypothesis, $\Sigma; \Gamma \vdash t_1 : A$ and $\Sigma; \Gamma \vdash t_2 : B[t_1]$. By rule PAIR, $\Sigma; \Gamma \vdash \langle t_1, t_2 \rangle : \Sigma AB$.

  By the premise, $\Sigma; \Gamma, A \vdash B\,\mathbf{type}$. By Remark 2.15, By reflexivity, $\Sigma; \Gamma, A \vdash B \equiv B : \mathrm{Set}$. By the premise, $\Sigma; \Gamma \vdash t_1 \equiv u_1 : A$. By Postulate 4, $\Sigma; \Gamma \vdash B[t_1] \equiv B[u_1] : \mathrm{Set}$, which, by the TYPE rule gives $\Sigma; \Gamma \vdash B[t_1] \equiv B[u_1] : \mathrm{Set}$.

By the induction hypothesis, $\Sigma;\Gamma \vdash u_1 : A$ and $\Sigma;\Gamma \vdash u_2 : B[t_1]$. By the CONV rule, $\Sigma;\Gamma \vdash u_2 : B[u_1]$. By the PAIR rule, $\Sigma;\Gamma \vdash \langle u_1, u_2\rangle : \Sigma AB$.

Also by the induction hypothesis, $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma;\Gamma \vdash A\ \mathbf{type}$. By Remark 2.15 and the SIGMA rule, $\Sigma;\Gamma \vdash \Sigma AB\ \mathbf{type}$.

- HEAD-EQ: By the premise, $\Sigma;\Gamma \vdash h \Rightarrow A$. By item (ii), $\Sigma;\Gamma \vdash A\ \mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By the HEAD rule, $\Sigma;\Gamma \vdash h : A$.
- APP-EQ ($\Sigma;\Gamma \vdash f\,t \equiv g\,u : B[t]$): By the induction hypothesis, $\Sigma;\Gamma \vdash f : \Pi AB$ and $\Sigma;\Gamma \vdash t : A$. By the APP rule, $\Sigma;\Gamma \vdash f\,t : B[t]$. Analogously, $\Sigma;\Gamma \vdash g\,u : B[u]$.

  By the induction hypothesis, $\Sigma;\Gamma \vdash \Pi AB\ \mathbf{type}$ and $\Sigma;\Gamma\,\mathbf{ctx}$. By Remark 2.15, $\Sigma;\Gamma \vdash \Pi AB : \mathrm{Set}$. By Lemma 2.52 ($\Pi$ inversion), $\Sigma;\Gamma, A \vdash B : \mathrm{Set}$. By reflexivity, $\Sigma;\Gamma, A \vdash B \equiv B : \mathrm{Set}$. By Postulate 1 (typing of hereditary substitution), $\Sigma;\Gamma \vdash B[t] : \mathrm{Set}$. By Postulate 4 (congruence of hereditary substitution), $\Sigma;\Gamma \vdash B[t] \equiv B[u] : \mathrm{Set}$. By the SYM and CONV rules, $\Sigma;\Gamma \vdash g\,u : B[u]$.
- PROJ1-EQ ($\Gamma \vdash f\,.\pi_1 \equiv g\,.\pi_1 : A$): By the induction hypothesis, $\Sigma;\Gamma \vdash f : \Sigma AB$ with $\Sigma;\Gamma \vdash \Sigma AB\ \mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By the PROJ1 rule, $\Sigma;\Gamma \vdash f\,.\pi_1 : A$. Analogously, $\Sigma;\Gamma \vdash g\,.\pi_1 : A$.

  By Lemma 2.53 (Σ inversion) and Remark 2.15, $\Sigma;\Gamma \vdash A\ \mathbf{type}$.
- PROJ2-EQ ($\Gamma \vdash f\,.\pi_2 \equiv g\,.\pi_2 : B[f\,.\pi_1]$): By the induction hypothesis, $\Sigma;\Gamma \vdash f : \Sigma AB$ with $\Sigma;\Gamma \vdash \Sigma AB\ \mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By the PROJ2 rule, $\Sigma;\Gamma \vdash f\,.\pi_2 : B[f\,.\pi_1]$.

  By Remark 2.15 and Lemma 2.53 (Σ inversion), $\Sigma;\Gamma, A \vdash B : \mathrm{Set}$.

  By the PROJ1 rule, $\Sigma;\Gamma \vdash f\,.\pi_1 : A$. By Postulate 1 (typing of hereditary substitution), $\Sigma;\Gamma \vdash B[f\,.\pi_1] : \mathrm{Set}$. By Remark 2.15, $\Sigma;\Gamma \vdash B[f\,.\pi_1] : \mathbf{type}$.

  Because $\Sigma;\Gamma \vdash f \equiv g : \Sigma AB$, by the PROJ1-EQ rule, $\Sigma;\Gamma \vdash f\,.\pi_1 \equiv g\,.\pi_1 : A$. By reflexivity, $\Sigma;\Gamma, A \vdash B \equiv B : \mathrm{Set}$. By Postulate 4 (congruence of hereditary substitution), $\Sigma;\Gamma \vdash B[f\,.\pi_1] \equiv B[g\,.\pi_1] : \mathrm{Set}$. Analogously to $\Sigma;\Gamma \vdash f\,.\pi_2 : B[f\,.\pi_1]$, $\Sigma;\Gamma \vdash g\,.\pi_2 : B[g\,.\pi_1]$. By the SYM and CONV rules, $\Sigma;\Gamma \vdash g\,.\pi_2 : B[f\,.\pi_1]$.
- ETA-ABS ($\Sigma;\Gamma \vdash f \equiv \lambda.(f^{(+1)})\,0 : \Pi AB$): From the rule premise, $\Sigma;\Gamma \vdash f : \Pi AB$. By the induction hypothesis, $\Sigma;\Gamma \vdash \Pi AB\ \mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By Lemma 2.52 ($\Pi$ inversion) and Remark 2.15, $\Sigma;\Gamma \vdash A\ \mathbf{type}$, which means $\Sigma \vdash \Gamma, A\,\mathbf{ctx}$.

  By the rule premise and Lemma 2.62 (context weakening), $\Sigma;\Gamma, A \vdash f^{(+1)} : \Pi AB$. By the VAR and HEAD rules, $\Sigma;\Gamma, A \vdash 0 : A$. By the APP rule, $\Sigma;\Gamma, A \vdash f\,0 : B$. By the ABS rule, $\Sigma;\Gamma \vdash \lambda.(f^{(+1)})\,0 : \Pi AB$.
- ETA-PAIR ($\Sigma;\Gamma \vdash f \equiv \langle f\,.\pi_1, f\,.\pi_2\rangle : \Sigma AB$): From the rule premise, $\Sigma;\Gamma \vdash f : \Sigma AB$. By the induction hypothesis, $\Sigma;\Gamma \vdash \Sigma AB\ \mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$.

  By Remark 2.15 and Lemma 2.53 (Σ inversion), $\Sigma;\Gamma, A \vdash B\ \mathbf{type}$.

  By the PROJ1 and PROJ2 rules, $\Sigma;\Gamma \vdash f\,.\pi_1 : A$ and $\Sigma;\Gamma \vdash f\,.\pi_2 : B[f\,.\pi_1]$.

By Remark 2.36 (hereditary substitution by a neutral term), $B[f\,.\pi_1]\Downarrow$.

By the PAIR rule, $\Sigma;\Gamma \vdash \langle f\,.\pi_1, f\,.\pi_2\rangle : \Sigma AB$.

- DELTA-META, DELTA-IF-TRUE, DELTA-IF-FALSE ($\Sigma;\Gamma \vdash t \equiv u : A$): By the premises, $\Sigma;\Gamma \vdash t : A$ and $\Sigma;\Gamma \vdash u : A$. By induction on the premises, $\Sigma \vdash \Gamma\,\mathbf{ctx}$ and $\Sigma;\Gamma \vdash A\,\mathbf{type}$.
- CONV-EQ ($\Sigma;\Gamma \vdash t \equiv u : B$): By the induction hypothesis, $\Sigma;\Gamma \vdash t : A$ and $\Sigma;\Gamma \vdash u : A$. By the CONV rule, $\Sigma;\Gamma \vdash t : B$ and $\Sigma;\Gamma \vdash u : B$. By item iv), $\Sigma;\Gamma \vdash B\,\mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$.
- TRANS ($\Sigma;\Gamma \vdash t \equiv v : A$): By induction on $\Sigma;\Gamma \vdash t \equiv u : A$, we have $\Sigma;\Gamma \vdash t : A$, $\Sigma;\Gamma \vdash A\,\mathbf{type}$ and $\Sigma \vdash \Gamma\,\mathbf{ctx}$. By induction on $\Sigma;\Gamma \vdash u \equiv v : A$, we have $\Sigma;\Gamma \vdash v : A$.
- SYM ($\Sigma;\Gamma \vdash u \equiv t : A$): Follows by induction on $\Sigma;\Gamma \vdash t \equiv u$.

□

Lemma 2.69 (signature weakening) shows that judgments may hold in larger signatures. We postulate that there is a converse property in the other direction; namely, that judgments also hold in smaller signatures as long as they only use constants present in the smaller signature.

Postulate 12 (Signature strengthening)**.** Assume $\Sigma \subseteq \Sigma'$, and let J be a judgment. If $\Sigma' \vdash J$ and $\textsc{consts}(J) \subseteq \textsc{decls}(\Sigma)$, then $\Sigma \vdash J$.

We also postulate that an analogous property holds for variables in a context:

Postulate 13 (Context strengthening)**.** If $\Sigma;\Gamma, x : A \vdash J$ and $x \notin \textsc{fv}(J)$, then $\Sigma;\Gamma \vdash J(-1)$.

Lemma 2.71 (Variables of irrelevant type)**.** *Let B be such that $\Sigma;\Gamma \vdash B\,\mathbf{type}$. If $\Sigma;\Gamma, x : A \vdash J$, and $x \notin \textsc{fv}(J)$, then $\Sigma;\Gamma, x : B \vdash J$.*

Proof. By Postulate 13, Lemma 2.62 (context weakening), and the fact that, if $0 \notin \textsc{fv}(J)$, then $J(-1)(+1) = J$. □

Lemma 2.72 (No extraneous constants)**.** *If $\Sigma \vdash J$, then $\textsc{consts}(J) \subseteq \textsc{decls}(\Sigma)$.*

Proof. By induction on the typing derivation, if $\Sigma;\Gamma \vdash t : A$ then $\textsc{consts}(t) \subseteq \textsc{decls}(\Sigma)$.

Given a judgment J, we use Lemma 2.70 (piecewise well-formedness of typing judgments), induction on the corresponding derivations and the above result to show $\textsc{consts}(J) \subseteq \textsc{decls}(\Sigma)$. □

Remark 2.73 (Signature piecewise well-formed). By Corollary 2.66, all the terms involved in a well-formed signature are closed terms. By Remark 2.5 (signature inversion), Lemma 2.69 (signature weakening) and Lemma 2.62 (context weakening), for any context Γ with $\Sigma \vdash \Gamma\,\mathbf{ctx}$, we have:

- If $\mathbb{c} : A \in \Sigma$, then $\Sigma;\Gamma \vdash A\,\mathbf{type}$.

- If $\alpha : A \in \Sigma$, then $\Sigma; \Gamma \vdash A$ **type**.
- If $\alpha := t : A \in \Sigma$, then $\Sigma; \Gamma \vdash A$ **type** and $\Sigma; \Gamma \vdash t : A$.

Remark 2.74 (Simplified DELTA-META rule: DELTA-META$_0$). The following rule is admissible:

$$\frac{\Sigma \vdash \Gamma\ \mathbf{ctx} \qquad \alpha := t : A \in \Sigma}{\Sigma; \Gamma \vdash \alpha \equiv t : A}\ \text{DELTA-META}_0$$

Proof. By Lemma 2.70 (piecewise well-formedness of typing judgments), Σ **sig**. By the META$_1$ and HEAD rules, $\Sigma; \Gamma \vdash \alpha : A$. By Remark 2.73 (signature piecewise well-formed), $\Sigma; \Gamma \vdash t : A$. By Definition 2.33 (iterated hereditary elimination), $t \mathbin{@} \varepsilon \Downarrow t$. By the DELTA-META rule, $\Sigma; \Gamma \vdash \alpha \equiv t : A$. □

Lemma 2.75 (Uniqueness of typing for neutrals)**.** *Let f be a neutral term such that $\Sigma; \Gamma \vdash f : A$ and $\Sigma; \Gamma \vdash f : B$. Then $\Sigma; \Gamma \vdash A \equiv B$* **type***.*

Proof. By induction on the structure of derivations for $\Sigma; \Gamma \vdash f : A$ and $\Sigma; \Gamma \vdash f : B$.

We proceed by case analysis on the derivations:

- One of the derivations ends in a CONV rule. Then, either:
  - $\Sigma; \Gamma \vdash f : A'$, with $\Sigma; \Gamma \vdash A' \equiv A$ **type** By the induction hypothesis, $\Sigma; \Gamma \vdash A' \equiv B$ **type**. By transitivity and symmetry, $\Sigma; \Gamma \vdash A \equiv B$ **type**.
  - $\Sigma; \Gamma \vdash f : B'$, with $\Sigma; \Gamma \vdash B' \equiv B$ **type** By the induction hypothesis, $\Sigma; \Gamma \vdash A \equiv B'$ **type**. By transitivity, $\Sigma; \Gamma \vdash A \equiv B$ **type**.
- Both derivations end in a HEAD rule: Necessarily, $A = B$. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma; \Gamma \vdash A$ **type**. By reflexivity, $\Sigma; \Gamma \vdash A \equiv B$ **type**.
- Both derivations end in an APP rule: Then $f = f'\, t$, with $\Sigma; \Gamma \vdash f' : \Pi U_1 V_1$, $V_1[t] \Downarrow A$, and $\Sigma; \Gamma \vdash f' : \Pi U_2 V_2$ with $V_2[t] \Downarrow B$. Also, $\Sigma; \Gamma \vdash t : U_1$. By the induction hypothesis, $\Sigma; \Gamma \vdash \Pi U_1 V_1 \equiv \Pi U_2 V_2$ **type**. By Postulate 10 (injectivity of Π), $\Sigma; \Gamma, U_1 \vdash V_1 \equiv V_2$ **type**. By Postulate 4 (congruence of hereditary substitution) and reflexivity, $\Sigma; \Gamma \vdash V_1[t] \equiv V_2[t]$ **type**, that is, $\Sigma; \Gamma \vdash A \equiv B$ **type**.
- Both derivations end in a PROJ1 or PROJ2 rule: Analogous to the previous case, using Postulate 11 (injectivity of Σ) instead.

□

Corollary 2.76 (Uniqueness of typing for equality of neutrals)**.** *Suppose that $\Sigma; \Gamma \vdash h\, \overrightarrow{e_1} \equiv h\, \overrightarrow{e_2} : B$, and either $\Sigma; \Gamma \vdash h\, \overrightarrow{e_1} : B'$ or $\Sigma; \Gamma \vdash h\, \overrightarrow{e_2} : B'$. Then $\Sigma; \Gamma \vdash h\, \overrightarrow{e_1} \equiv h\, \overrightarrow{e_2} : B'$.*

Corollary 2.77 (Uniqueness of typing for heads)**.** *Assume that $\Sigma; \Gamma \vdash h : B$. Then $\Sigma; \Gamma \vdash h \Rightarrow A$, and $\Sigma; \Gamma \vdash A \equiv B$* **type***.*

Proof. By induction on the typing derivation for $\Sigma; \Gamma \vdash h : B$, we obtain $\Sigma; \Gamma \vdash h \Rightarrow A$ (there are two possible cases, CONV and HEAD).

By the HEAD rule, $\Sigma; \Gamma \vdash h : A$. Because h is a neutral term, by Lemma 2.75, $\Sigma; \Gamma \vdash A \equiv B$ **type**. □

Lemma 2.78 (Variable types say everything)**.** *Suppose* $\Sigma; \Gamma', y : A, \Gamma'', x : A' \vdash J$ *holds, with* $\Sigma; \Gamma', y : A, \Gamma'', x : A' \vdash x : A^{(+|A,\Gamma'',A'|)}$. *Then* $\Sigma; \Gamma', y : A, \Gamma'', x : A' \vdash J[x \mapsto y]$. *(Note that, by Definition 2.21, the renaming* $[x \mapsto y]$ *is such that it leaves all variables except* x *unchanged.)*

Proof. By Lemma 2.75 (uniqueness of typing for neutrals), $\Sigma; \Gamma', y : A, \Gamma'', x : A' \vdash A'^{(+1)} \equiv A^{(+|A,\Gamma'',A'|)}$ **type**. By Postulate 13 (context strengthening) and Remark 2.30, $\Sigma; \Gamma', y : A, \Gamma'' \vdash A' \equiv A^{(+(1+|\Gamma''|))}$ **type**.

Let $\Gamma = \Gamma', y : A, \Gamma'', x : A^{(+(1+|\Gamma''|))}$. By Lemma 2.63 (preservation of judgments by type conversion), $\Sigma; \Gamma \vdash J$.

$\Sigma; \Gamma \vdash J[x \mapsto y]$ follows by induction on the derivation for $\Sigma; \Gamma \vdash J$. We detail the case for the HEAD rule followed by the VAR rule:

$$\dfrac{\dfrac{\Sigma; \Gamma, \Delta\, \mathbf{ctx}}{\Sigma; \Gamma, \Delta \vdash x \Rightarrow A^{(+(1+|\Gamma''|))(+(1+|\Delta|))}}\,\text{VAR}}{\Sigma; \Gamma, \Delta \vdash x : A^{(+(1+|\Gamma''|))(+(1+|\Delta|))}}\,\text{HEAD}$$

We need to show $\Sigma; \Gamma, \Delta[x \mapsto y] \vdash x[x \mapsto y] : A^{(+(1+|\Gamma''|))(+(1+|\Delta|))}[x \mapsto y]$. By Remark 2.28 (renaming and free variables) and Remark 2.30 (properties of renamings) this is equivalent to $\Sigma; \Gamma, \Delta[x \mapsto y] \vdash y : A^{(+(2+|\Gamma''|+|\Delta|))}[x \mapsto y]$.

By the induction hypothesis, $\Sigma; \Gamma \vdash \Delta[x \mapsto y]\, \mathbf{ctx}$; that is, $\Sigma; \Gamma \vdash \Delta[x \mapsto y]\, \mathbf{ctx}$. Therefore, by the VAR and HEAD rules, we have:

$$\dfrac{\dfrac{\Sigma \vdash \Gamma, \Delta[x \mapsto y]\, \mathbf{ctx}}{\Sigma; \Gamma, \Delta \vdash y \Rightarrow A^{(+(2+|\Gamma''|+|\Delta|))}}\,\text{VAR}}{\Sigma; \Gamma, \Delta \vdash y : A^{(+(2+|\Gamma''|+|\Delta|))}}\,\text{HEAD}$$

Finally, from $\Sigma; \Gamma \vdash J[x \mapsto y]$, by Lemma 2.63 (preservation of judgments by type conversion), we have $\Sigma; \Gamma', y : A, \Gamma'', x : A' \vdash J[x \mapsto y]$.

□

Lemma 2.79 (Typing and congruence of elimination)**.** *Assume* $\Sigma; \Gamma \vdash f\, \vec{e}^{\,n} : T$, *with* $\Sigma; \Gamma \vdash f : A$.

Then, for every t, u *such that* $\Sigma; \Gamma \vdash t \equiv u : A$, *there exist* t' *and* u' *such that* $t \mathbin{@} \vec{e} \Downarrow t'$, $u \mathbin{@} \vec{e} \Downarrow u'$, *and* $\Sigma; \Gamma \vdash t' \equiv u' : T$.

In particular, by reflexivity, for every t *such that* $\Sigma; \Gamma \vdash t : A$, *we have* $t \mathbin{@} \vec{e} \Downarrow t'$ *and* $\Sigma; \Gamma \vdash t' : T$.

Proof. By induction on the length of $\vec{e}^{\,n}$.

- 0: By Definition 2.33 (iterated hereditary elimination), $t' = t$ and $u' = u$. By Lemma 2.75 (uniqueness of typing for neutrals), $\Sigma; \Gamma \vdash T \equiv A$ **type**. By the CONV-EQ rule, $\Sigma; \Gamma \vdash t \equiv u : T$; that is, $\Sigma; \Gamma \vdash t' \equiv u' : T$.

- $n+1$: We do the case where $\vec{e} = \vec{e}'\, v$. The cases for $\vec{e} = \vec{e}'\, .\pi_1$ and $\vec{e} = \vec{e}'\, .\pi_2$ are similar, but use Postulate 7 (congruence of hereditary projection) instead.

  By Lemma 2.56 (neutral inversion), $\Sigma; \Gamma \vdash f\, \vec{e}' : \Pi UV$, $\Sigma; \Gamma \vdash v : U$ and $V[v] \Downarrow$. By the APP rule, $\Sigma; \Gamma \vdash f\, \vec{e}'\, v : V[v]$. By Lemma 2.75, $\Sigma; \Gamma \vdash V[v] \equiv T$ **type**.

  By the induction hypothesis, $(t \mathbin{@} \vec{e}')\Downarrow$, $(u \mathbin{@} \vec{e}')\Downarrow$, and $\Sigma; \Gamma \vdash t \mathbin{@} \vec{e}' \equiv u \mathbin{@} \vec{e}' : \Pi UV$. By Postulate 6 (congruence of hereditary application), there exist t' and u' such that $(t \mathbin{@} \vec{e}')\, v \Downarrow t'$, $(u \mathbin{@} \vec{e}')\, v \Downarrow u'$, and $\Sigma; \Gamma \vdash t' \equiv u' : V[v]$. By the CONV-EQ rule, $\Sigma; \Gamma \vdash t' \equiv u' : T$.

□

2.14.6 Admissible rules

Lemma 2.80 (Simplified APP, APP-EQ: APP_0, APP-EQ_0)**.** *The following rules are admissible:*

$$\frac{\Gamma \vdash f : \Pi AB \qquad \Gamma \vdash t : A}{\Gamma \vdash f\, t : B[t]}\ \text{APP}_0$$

$$\frac{\Gamma \vdash f \equiv g : \Pi AB \qquad \Gamma \vdash t \equiv u : A}{\Gamma \vdash f\, t \equiv g\, u : B[t]}\ \text{APP-EQ}_0$$

Proof. Use Lemma 2.52 (Π inversion), Postulate 1 (typing of hereditary substitution), and Lemma 2.70 (piecewise well-formedness of typing judgments) to derive $B[t]\Downarrow$. The consequent follows by APP and APP-EQ, respectively. □

Remark 2.81 (Cancellation of weakening with substitution). For every term t, $t((+1) + 1 + x)[x/x]\Downarrow t$. (Note that, by Remark 2.36, for any term u and variable x, $u[x/x]\Downarrow$.)

Proof. By induction on the structure of t. □

Lemma 2.82 (λ inversion)**.** *If* $\Sigma; \Gamma \vdash \lambda.t : \Pi AB$*, then* $\Sigma; \Gamma, A \vdash t : B$*.*

Proof. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma \vdash \Gamma$ **ctx** and $\Sigma; \Gamma \vdash \Pi AB$ **type**. By Lemma 2.52 (Π inversion), $\Sigma; \Gamma \vdash A$ **type** and $\Sigma; \Gamma, A \vdash B$ **type**. By the CTX-VAR rule, $\Sigma \vdash \Gamma, A$ **ctx**.

By Lemma 2.62 (context weakening), we have $\Sigma; \Gamma, A \vdash \lambda.(t^{((+1)+1)}) : \Pi A^{(+1)} B^{((+1)+1)}$. By the HEAD and VAR rules, $\Sigma; \Gamma, A \vdash 0 : A^{(+1)}$.

By Remark 2.81 (cancellation of weakening with substitution), $t^{((+1)+1)}[0/0] \Downarrow t$ and $B^{((+1)+1)}[0/0] \Downarrow B$. Thus, by Definition 2.32 (hereditary elimination), $(\lambda.(t^{((+1)+1)})) \mathbin{@} 0 \Downarrow t$. By Postulate 2 (typing of hereditary application), $\Sigma; \Gamma, A \vdash t : B$. □

Lemma 2.83 (Injectivity of λ)**.** *If* $\Sigma; \Gamma \vdash \lambda.t \equiv \lambda.u : \Pi AB$*, then* $\Sigma; \Gamma, A \vdash t \equiv u : B$*.*

Proof. As in the proof for Lemma 2.82 (λ inversion), $\Sigma \vdash \Gamma, A\,\mathbf{ctx}$. By Lemma 2.62 (context weakening), $\Sigma; \Gamma, A \vdash \lambda.(t^{((+1)+1)}) \equiv \lambda.(u^{((+1)+1)}) : \Pi A^{(+1)} B^{((+1)+1)}$. By the VAR and the HEAD-EQ rules, $\Sigma; \Gamma, x : A \vdash 0 \equiv 0 : A^{(+1)}$.

As in the proof of Lemma 2.82, $(\lambda.(t^{((+1)+1)}) \mathbin{@} 0) \Downarrow t$, $(\lambda.(u^{((+1)+1)}) \mathbin{@} 0) \Downarrow u$, and $B^{((+1)+1)}[0/0] \Downarrow B$. By Postulate 6 (congruence of hereditary application), $\Sigma; \Gamma, A \vdash t \equiv u : B$. □

Lemma 2.84 (⟨,⟩-inversion)**.** *If* $\Sigma; \Gamma \vdash \langle t, u\rangle : \Sigma AB$*, then* $\Sigma; \Gamma \vdash t : A$*,* $B[t]\Downarrow$*, and* $\Sigma; \Gamma \vdash u : B[t]$*.*

Proof. By Definition 2.32 (hereditary elimination), $\langle t, u\rangle \mathbin{@} .\pi_1 \Downarrow t$ and $\langle t, u\rangle \mathbin{@} .\pi_2 \Downarrow u$.

By Postulate 3 (typing of hereditary projection), $\Sigma; \Gamma \vdash t : A$, $B[t]\Downarrow$ and $\Sigma; \Gamma \vdash u : B[t]$. □

Lemma 2.85 (Injectivity of ⟨,⟩)**.** *If* $\Sigma; \Gamma \vdash \langle t_1, t_2\rangle \equiv \langle u_1, u_2\rangle : \Sigma AB$*, then* $\Sigma; \Gamma \vdash t_1 \equiv u_1 : A$*,* $B[t_1]\Downarrow$ *and* $\Sigma; \Gamma \vdash t_2 \equiv u_2 : B[t_1]$*.*

Proof. By Definition 2.32 (hereditary elimination), $\langle t_1, t_2\rangle \mathbin{@} .\pi_1 \Downarrow t_1$, $\langle u_1, u_2\rangle \mathbin{@} .\pi_1 \Downarrow t_1$, $\langle t_1, t_2\rangle \mathbin{@} .\pi_2 \Downarrow t_2$, $\langle u_1, u_2\rangle \mathbin{@} .\pi_2 \Downarrow u_2$.

By Postulate 7 (congruence of hereditary projection), $\Sigma; \Gamma \vdash t_1 \equiv u_1 : A$, $B[t_1]\Downarrow$ and $\Sigma; \Gamma \vdash t_2 \equiv u_2 : B[t_1]$. □

2.14.7 Term reduction

Lemma 2.86 (Equality of δη-reduct)**.** *If* $\Sigma; \Gamma \vdash t : A$ *and* $\Sigma; \Gamma \vdash t \longrightarrow^\star_{\delta\eta} u : A$*, then* $\Sigma; \Gamma \vdash u : A$ *and* $\Sigma; \Gamma \vdash t \equiv u : A$*.*

Proof. By structural induction on the derivation of $\Sigma; \Gamma \vdash t \longrightarrow^\star_{\delta\eta} u : A$. □

We postulate that δη-reduction characterizes term equality: that is, two terms are equal if and only if they can be reduced to a common form.

Postulate 14 (Existence of a common reduct)**.** Given $\Sigma; \Gamma \vdash t \equiv u : A$, there exists v such that $\Sigma; \Gamma \vdash t \longrightarrow^\star_{\delta\eta} v : A$ and $\Sigma; \Gamma \vdash u \longrightarrow^\star_{\delta\eta} v : A$.

Definition 2.87 (Full normal form: $\Sigma; \Gamma \vdash t \not\longrightarrow_{\delta\eta} : A$)**.** We say that a term t is in full normal form (written $\Sigma; \Gamma \vdash t \not\longrightarrow_{\delta\eta} : A$), if $\Sigma; \Gamma \vdash t : A$ and there is no v such that $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} v : A$.

Postulate 15 (Existence of a unique full normal form)**.** If $\Sigma; \Gamma \vdash t : A$, then there exists v such that $\Sigma; \Gamma \vdash t \longrightarrow^\star_{\delta\eta} v : A$, and $\Sigma; \Gamma \vdash v \not\longrightarrow_{\delta\eta} : A$.

Remark 2.88 (Existence of a common normal form). Given $\Sigma; \Gamma \vdash t \equiv u : A$, there exists v such that $\Sigma; \Gamma \vdash t \longrightarrow^\star_{\delta\eta} v : A$, $\Sigma; \Gamma \vdash u \longrightarrow^\star_{\delta\eta} v : A$, and $\Sigma; \Gamma \vdash v \not\longrightarrow_{\delta\eta} : A$.

Proof. By Postulate 15 and Postulate 14. □

Remark (Uniqueness of full normal form). Let v_1, v_2 be terms such that $\Sigma; \Gamma \vdash v_1 \not\longrightarrow_{\delta\eta} : A$ and $\Sigma; \Gamma \vdash v_2 \not\longrightarrow_{\delta\eta} : A$.

(i) If $\Sigma; \Gamma \vdash v_1 \equiv v_2 : A$, then $v_1 = v_2$.

(ii) If there is t such that $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} v_1 : A$ and $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} v_2 : A$, then $v_1 = v_2$.

Proof. Statement (i) follows from Postulate 14. Statement (ii) follows from Lemma 2.86, symmetry and transitivity of judgmental equality, and (i). □

Remark 2.89 (Disjointness of primitive types). For any T, it is not possible to have more than one of $\Sigma; \Gamma \vdash T \equiv \mathrm{Set} : \mathrm{Set}$ $\Sigma; \Gamma \vdash T \equiv \Pi A_1 B_1 : \mathrm{Set}$ and $\Sigma; \Gamma \vdash T \equiv \Sigma A_2 B_2 : \mathrm{Set}$ for any A_1, A_2, B_1 and B_2.

Proof. Proceed by contradiction. If more than one of the equalities hold (in particular, two of them) then, by Lemma 2.54 (term equality is an equivalence relation) and Postulate 14 (existence of a common reduct), this means there exists r such that two of the following hold $\Sigma; \Gamma \vdash \mathrm{Set} \longrightarrow^{\star}_{\delta\eta} r : \mathrm{Set}$, $\Sigma; \Gamma \vdash \mathrm{Bool} \longrightarrow^{\star}_{\delta\eta} r : \mathrm{Set}$, $\Sigma; \Gamma \vdash \Pi A_1 B_1 \longrightarrow^{\star}_{\delta\eta} r : \mathrm{Set}$, and $\Sigma; \Gamma \vdash \Sigma A_2 B_2 \longrightarrow^{\star}_{\delta\eta} r : \mathrm{Set}$.

For the first and second cases, none of the rules in Definition 2.41 (δη-normalization step) apply, so $r = \mathrm{Set}$ and $r = \mathrm{Bool}$ respectively. In the third case, only the rules Π_1 and Π_2 apply, which means that $r = \Pi A_1' B_1'$ for some A_1' and B_1'. Similarly, in the fourth case, only the rules Σ_1 and Σ_2 apply, therefore $r = \Sigma A_2' B_2'$ for some A_2' and B_2'. All of these alternatives are incompatible; therefore at most one of the equalities holds. □

Lemma 2.90 (Reduction under equal context)**.** *If* $\Sigma; \Gamma \vdash t \longrightarrow^{n}_{\delta\eta} u : T$ *and* $\Sigma \vdash \Gamma, T \equiv \Gamma', T'$ **ctx**, *then* $\Sigma; \Gamma' \vdash t \longrightarrow^{n}_{\delta\eta} u : T'$.

Proof. The case with one step follows by induction on the structure of the derivation. The general case follows by induction on the number of steps. □

Remark 2.91 (Inversion of reduction under λ). If $\Sigma; \Gamma \vdash \lambda.f \longrightarrow^{n}_{\delta\eta} \lambda.g : T$, then there are A, B such that $\Sigma; \Gamma, A \vdash f \longrightarrow^{n}_{\delta\eta} g : B$ and $\Sigma; \Gamma \vdash T \equiv \Pi A B$ **type**.

In fact, for any A', B' such that $\Sigma; \Gamma \vdash T \equiv \Pi A' B'$ **type**, $\Sigma; \Gamma, A' \vdash f \longrightarrow^{n}_{\delta\eta} g : B'$.

Proof. We proceed by induction on n.

- 0: By Lemma 2.57 (type of λ-abstraction), $\Sigma; \Gamma \vdash T \equiv \Pi A B$ **type** for some A, B. By definition, $\Sigma; \Gamma, A \vdash f \longrightarrow^{0}_{\delta\eta} f : B$.

- $n+1$: We have $\Sigma; \Gamma \vdash \lambda.f \longrightarrow_{\delta\eta} \lambda.f_0 : T$ and $\Sigma; \Gamma \vdash \lambda.f_0 \longrightarrow^{n}_{\delta\eta} \lambda.g : T$

  By case analysis, the only possible rule for $\Sigma; \Gamma \vdash \lambda.f \longrightarrow_{\delta\eta} \lambda.f_0 : T$ is λ. Therefore, there are A, B such that $\Sigma; \Gamma, A \vdash f \longrightarrow_{\delta\eta} f_0 : B$ and $\Sigma; \Gamma \vdash T \equiv \Pi A B$ **type**.

  By the induction hypothesis, $\Sigma; \Gamma, A' \vdash f_0 \longrightarrow^{n}_{\delta\eta} g : B'$ for some A' and B', with $\Sigma; \Gamma \vdash T \equiv \Pi A' B'$ **type**. By Postulate 10 (injectivity of Π), $\Sigma \vdash \Gamma, A, B \equiv \Gamma, A', B'$ **ctx**. By Lemma 2.90 (reduction under equal context), $\Sigma; \Gamma, A \vdash f_0 \longrightarrow^{n}_{\delta\eta} g : B$. Therefore, $\Sigma; \Gamma, A \vdash f \longrightarrow^{n+1}_{\delta\eta} g : B$.

By Postulate 10 and Lemma 2.90, the property holds for any A', B' such that $\Sigma; \Gamma \vdash T \equiv \Pi A' B'$ **type**. □

Remark 2.92 (Inversion of reduction under $\langle,\rangle$). If $\Sigma;\Gamma \vdash \langle f_1, f_2\rangle \longrightarrow^{n}_{\delta\eta} \langle g_1, g_2\rangle : T$, then there are A, B such that $\Sigma;\Gamma \vdash T \equiv \Sigma AB$ **type**, $\Sigma;\Gamma \vdash f_1 \longrightarrow^{m_1}_{\delta\eta} g_1 : A$ and $\Sigma;\Gamma \vdash f_2 \longrightarrow^{m_2}_{\delta\eta} g_2 : B[f_1]$, with $m_1 + m_2 = n$.

In fact, for any A', B' such that $\Sigma;\Gamma \vdash T \equiv \Sigma A'B'$ **type**, there exist m_1' and m_2' such that $\Sigma;\Gamma \vdash f_1 \longrightarrow^{m_1'}_{\delta\eta} g_1 : A'$ and $\Sigma;\Gamma \vdash f_2 \longrightarrow^{m_2'}_{\delta\eta} g_2 : B'[f_2]$, with $m_1' + m_2' = n$.

Proof. The proof is analogous to Remark 2.91 (inversion of reduction under λ). □

Remark 2.93 (Strengthening of hereditary substitution and elimination). For all x, t, u, with $x \notin \text{FV}(t)$, $x \notin \text{FV}(u)$, $x \geq y$, if $t[u/y] \Downarrow v$ for some v, then $t^{(-1)+x}[u^{(-1)+x}/y] \Downarrow v^{(-1)+x}$.

For all x, t and $\vec{e}$, with $x \notin \text{FV}(t)$ and $x \notin \text{FV}(\vec{e})$, if $t \mathbin{@} \vec{e} \Downarrow u$ for some u, then $t^{(-1)+x} \mathbin{@} \vec{e}^{(-1)+x} \Downarrow u^{(-1)+x}$.

Proof. By mutual induction on the derivations (see Definition 2.31 (hereditary substitution) and Definition 2.32 (hereditary elimination)). □

Remark 2.94 (Strengthening of reduction). If $\Sigma;\Gamma, A, \Delta \vdash t \longrightarrow^{m}_{\delta\eta} t' : B$, $|\Delta| \notin \text{FV}(t)$ and $|\Delta| \notin \text{FV}(B)$, then $\Sigma;\Gamma, \Delta^{(-1)} \vdash t^{(-1)+|\Delta|} \longrightarrow^{m}_{\delta\eta} t'^{(-1)+|\Delta|} : B^{(-1)+|\Delta|}$.

Proof. By Remark 2.43 (free variables of δη-reduct), $|\Delta| \notin \text{FV}(t') \subseteq \text{FV}(t)$.

By induction on m, then by induction on the derivation (see Definition 2.41 (δη-normalization step)), using Postulate 13 (context strengthening) and Remark 2.93. □

2.15 Weak head normalization ($\searrow$)

In this section we define a special case of δη-reduction, which is i) fully deterministic, and ii) can be performed with knowledge of just the signature in which the term is typed, but not necessarily the context or the type.

WHNF is used for defining type application (Definition 2.104), and for determining which unification rule to apply in the unification algorithm (Section 5.1).

Definition 2.95 (Weak head normal form: $\Sigma \vdash t \searrow u$)**.** Given a signature Σ, the relation $\Sigma \vdash t \searrow u$ is inductively defined as follows (the signature Σ is implicit):

$$
\begin{array}{rcll}
\mathrm{Bool} & \searrow & \mathrm{Bool} & \\
\Sigma AB & \searrow & \Sigma AB & \\
\Pi AB & \searrow & \Pi AB & \\
\mathrm{Set} & \searrow & \mathrm{Set} & \\
c & \searrow & c & \\
\lambda.t & \searrow & \lambda.t & \\
\langle t, u\rangle & \searrow & \langle t, u\rangle & \\
x\,\vec{e} & \searrow & x\,\vec{e} & \\
\mathbb{o}\,\vec{e} & \searrow & \mathbb{o}\,\vec{e} & \\
\alpha\,\vec{e} & \searrow & u' & \textbf{if} \quad \alpha := t : A \in \Sigma \textbf{ and } (t \mathbin{@} \vec{e}) \Downarrow u \\
 & & & \quad \textbf{and } u \searrow u' \\
\alpha\,\vec{e} & \searrow & \alpha\,\vec{e} & \textbf{if} \quad \alpha \text{ is uninstantiated in } \Sigma \\
\mathrm{if}\,\vec{e}^{\,n} & \searrow & \mathrm{if}\,\vec{e} & \textbf{if} \quad n \leq 3 \\
\mathrm{if}\,A\,b\,t\,u\,\vec{e} & \searrow & t'' & \textbf{if} \quad b \searrow \mathrm{true} \textbf{ and } (t \mathbin{@} \vec{e}) \Downarrow t' \textbf{ and } t' \searrow t'' \\
\mathrm{if}\,A\,b\,t\,u\,\vec{e} & \searrow & u'' & \textbf{if} \quad b \searrow \mathrm{false} \textbf{ and } (u \mathbin{@} \vec{e}) \Downarrow u' \\
 & & & \quad \textbf{and } u' \searrow u'' \\
\mathrm{if}\,A\,b\,t\,u\,\vec{e} & \searrow & \mathrm{if}\,A\,b'\,t\,u\,\vec{e} & \textbf{if} \quad b \searrow b' \textbf{ and } b' \neq \mathrm{true} \textbf{ and } b' \neq \mathrm{false}
\end{array}
$$

Remark 2.96 (WHNF reduction is deterministic). If $\Sigma \vdash t \searrow u_1$, and $\Sigma \vdash t \searrow u_2$, then $u_1 = u_2$.

Proof. By induction on the derivations, noting that given a signature Σ and a term t (typed or untyped), there is always at most one case in Definition 2.95 (weak head normal form) which applies to t. □

Remark 2.97 (WHNF reduction is δη-reduction). If $\Sigma; \Gamma \vdash t : A$ and $\Sigma \vdash t \searrow u$, then $\Sigma; \Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} u : A$

Proof. By induction on the derivation for $\Sigma \vdash t \searrow u$. □

Lemma 2.98 (Equality of WHNF)**.** *If $\Sigma; \Gamma \vdash t : A$ then there is u such that $\Sigma \vdash t \searrow u$, with $\Sigma; \Gamma \vdash u : A$ and $\Sigma; \Gamma \vdash t \equiv u : A$.*

Proof. We first show the following property: If $\Sigma; \Gamma \vdash t : A$ and $\nexists v.\Sigma \vdash t \searrow v$, then there exists u such that $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} u : A$, $\Sigma; \Gamma \vdash u : A$ and $\nexists v.\Sigma \vdash u \searrow v$.

We proceed by induction on t, and the fact that $\nexists v.\Sigma \vdash t \searrow v$, we are in either one of these four cases:

- Case $t = \alpha\,\vec{e}$, with $\alpha := t : T \in \Sigma$. By postulates Postulate 2 (typing of hereditary application), Postulate 3 (typing of hereditary projection), $(t \mathbin{@} \vec{e}) \Downarrow u$. By META, $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} u : T$, which by Lemma 2.86 (equality of δη-reduct) means $\Sigma; \Gamma \vdash u : T$. Because $\nexists v.\Sigma \vdash t \searrow v$, by Definition 2.95 (weak head normal form), then $\nexists v.\Sigma \vdash u \searrow v$.

- Case $t = \mathrm{if}\,A\,b\,t_0\,u_0\,\vec{e}$, with $\nexists b'.\Sigma \vdash b \searrow b'$: By Lemma 2.56 (neutral inversion), $\Sigma; \Gamma \vdash b : \mathrm{Bool}$. By induction, there exists b' such that $\Sigma; \Gamma \vdash b \longrightarrow_{\delta\eta} b' : \mathrm{Bool}$, $\Sigma; \Gamma \vdash b' : \mathrm{Bool}$, and $\nexists b''.\Sigma \vdash b' \searrow b''$. Let $u = \mathrm{if}\,A\,b'\,t_0\,u_0\,\vec{e}$. By APP_n, we have $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} u : A$. And, because $\nexists b''.\Sigma \vdash b' \searrow b''$ and the form of u, $\nexists v.\Sigma \vdash u \searrow v$.

- Case $t = \mathrm{if}\, A\, b\, t_0\, u_0\, \vec{e}$, with $\Sigma \vdash b \searrow \mathrm{true}$, $t = \mathrm{if}\, A\, b\, t_0\, u_0\, \vec{e}$, with $\Sigma \vdash b \searrow \mathrm{false}$: They are analogous to the case for $t = \alpha\, \vec{e}$, with $\alpha := t : T \in \Sigma$.

Note that the u in the property fulfills the same conditions as t. Therefore, given $\Sigma; \Gamma \vdash t : A$ and $\nexists v. \Sigma \vdash t \searrow v$, we can construct an infinite chain of reductions starting from t. This contradicts Postulate 8 (no infinite chains). Therefore, if $\Sigma; \Gamma \vdash t : A$, then there is u such that $\Sigma \vdash t \searrow u$.

By Remark 2.97 (WHNF reduction is $\delta\eta$-reduction), $\Sigma \vdash t \searrow u$ implies $\Sigma; \Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} u : A$; and by Lemma 2.86 (equality of $\delta\eta$-reduct), this means that $\Sigma; \Gamma \vdash t \equiv u : A$.

□

Lemma 2.99 (Term in WHNF)**.** *Assume that $\Sigma; \Gamma \vdash t : A$, and $\Sigma \vdash t \searrow u$. Then, either:*

- $u = \mathrm{Bool}$.
- $u = \Sigma A B$ *for some* A, B.
- $u = \Pi A B$ *for some* A, B.
- $u = \mathrm{Set}$.
- $u = c$.
- $u = \lambda.u'$ *for some term* u'.
- $u = \langle u_1, u_2 \rangle$ *for some term* u_1 *and* u_2.
- $u = x\, \vec{e}$ *for some variable* x, *and vector of eliminators* $\vec{e}$.
- $u = \mathbb{c}\, \vec{e}$ *for some atom* $\mathbb{c}$, *and vector of eliminators* $\vec{e}$.
- $u = \alpha\, \vec{e}$ *for some metavariable* α *such that* α *is not instantiated in* Σ.
- $u = \mathrm{if}\, \vec{e}^{\,n}$, *where* $n \leq 3$.
- $u = \mathrm{if}\, A\, b\, t\, u\, \vec{e}$, *where neither* $\Sigma; \Gamma \vdash b \equiv \mathrm{true} : \mathrm{Bool}$ *nor* $\Sigma; \Gamma \vdash b \equiv \mathrm{false} : \mathrm{Bool}$.

Proof. By induction on the derivation for $\Sigma \vdash t \searrow u$, and the typing rules. □

Definition 2.100 (Head of a term: Set, Σ, Π, Bool, λ, h, c, $\langle \_, \_ \rangle$)**.** When discussing terms, we will often refer to the "head" of a term. The head of a term or type is its "top-most" syntactic element. More specifically, the head of a term can be any of Set, Σ, Π, Bool, λ, h, c, or the pair constructor $\langle \_, \_ \rangle$.

The weak-head normal form determines the head of term. In particular:

Lemma 2.101 (Nose of weak-head normal form)**.** *Let T be a term such that $\Sigma; \Gamma \vdash T : \mathrm{Set}$.*

(i) If $\Sigma; \Gamma \vdash T \equiv \Pi A B : \mathrm{Set}$, then there are A', B' such that $\Sigma \vdash T \searrow \Pi A' B'$.

(ii) If $\Sigma; \Gamma \vdash T \equiv \Sigma A B : \mathrm{Set}$, then there are A', B' such that $\Sigma \vdash T \searrow \Sigma A' B'$.

Proof.

(i) By Lemma 2.98 (equality of WHNF), there is U such that $\Sigma \vdash T \searrow U$ and $\Sigma; \Gamma \vdash T \equiv U : \mathrm{Set}$. By transitivity, this means $\Sigma; \Gamma \vdash U \equiv \Pi AB : \mathrm{Set}$.

By Postulate 14 (existence of a common reduct), there is V such that $\Sigma; \Gamma \vdash U \longrightarrow^\star_{\delta\eta} V : \mathrm{Set}$ and $\Sigma; \Gamma \vdash \Pi AB \longrightarrow^\star_{\delta\eta} V : \mathrm{Set}$. By Definition 2.41 ($\delta\eta$-normalization step), V is necessarily of the form $\Pi A'' B''$.

By Lemma 2.99 (term in WHNF), and the fact that $\Sigma; \Gamma \vdash U \longrightarrow^\star_{\delta\eta} \Pi A'' B'' : \mathrm{Set}$, we have that U is necessarily of the form $\Pi A' B'$ for some A' and B', as it is the only alternative listed in Lemma 2.99 (term in WHNF) which can reduce to a Π-headed term.

(ii) Same as the proof for (i), replacing Π with Σ.

□

Remark 2.102 (Preservation of free variables by WHNF). If $\Sigma \vdash t \searrow u$, then $\mathrm{FV}(u) \subseteq \mathrm{FV}(t)$.

Proof. By induction on the derivation. □

2.16 Type elimination ($\hat{@}$)

The type of a neutral term is fully determined by the head and its type. We now give a deterministic procedure to obtain this type given a signature and a context.

Definition 2.103 (Type elimination: $\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow U$)**.** In a signature Σ and context Γ, the elimination of the type of a head h by a spine $\vec{e}$, resulting in a type U (written $\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow U$) is defined as follows:

$$
\begin{array}{lll}
\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \varepsilon \Downarrow T & \textbf{if} & \Sigma; \Gamma \vdash h \Rightarrow T \\
\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e}\, u \Downarrow U & \textbf{if} & \Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow T \textbf{ and} \\
 & & \Sigma \vdash T \searrow \Pi AB \textbf{ and} \\
 & & \Sigma; \Gamma \vdash u : A \textbf{ and} \\
 & & B[u] \Downarrow U \\
\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e}\,.\pi_1 \Downarrow A & \textbf{if} & \Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow T \textbf{ and} \\
 & & \Sigma \vdash T \searrow \Sigma AB \\
\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e}\,.\pi_2 \Downarrow B[h\,\vec{e}] & \textbf{if} & \Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow T \textbf{ and} \\
 & & \Sigma \vdash T \searrow \Sigma AB
\end{array}
$$

If the elimination spine $\vec{e}$ consists only of terms (i.e. $\vec{e} = \vec{t}$), then $\Sigma; \Gamma \vdash (h :) \mathbin{\hat{@}} \vec{t}$ only depends on the type T such that $\Sigma; \Gamma \vdash h \Rightarrow T$. Therefore:

Definition 2.104 (Type application: $\Sigma; \Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow U$)**.** In a signature Σ and context Γ, elimination of a type T by a spine $\vec{t}$, resulting in a type U (written $T \mathbin{\hat{@}} \vec{t} \Downarrow U$) is defined as follows:

$$\begin{array}{lll} \Sigma;\Gamma \vdash T \mathbin{\hat{@}} \varepsilon \Downarrow T & & \\ \Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t}\, u \Downarrow U & \textbf{if} & \Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow T' \textbf{ and} \\ & & \Sigma \vdash T' \searrow \Pi A B \textbf{ and} \\ & & \Sigma;\Gamma \vdash u : A \textbf{ and} \\ & & B[u] \Downarrow U \end{array}$$

Remark 2.105 (Type elimination without projections). If $\Sigma;\Gamma \vdash h \Rightarrow T$, then $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{t} \Downarrow T'$ if and only if $\Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow T'$.

The relation $\mathbin{\hat{@}}$ is consistent with the typing rules.

Lemma 2.106 (Type elimination)**.** *If* $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow B$*, then* $\Sigma;\Gamma \vdash h\,\vec{e} : B$.

Proof. By induction on the derivation of $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow B$. Use Lemma 2.98 (equality of WHNF) and the CONV rule. □

Lemma 2.107 (Type elimination inversion)**.** *Assume that* $\Sigma;\Gamma \vdash h\,\vec{e} : B$. *Then* $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow B'$ *with* $\Sigma;\Gamma \vdash B' \equiv B$ **type**.

Proof. By induction on the derivation for $\Sigma;\Gamma \vdash h\,\vec{e} : B$.

- HEAD (base case): Then $\vec{e} = \varepsilon$, and, by the rule's premises, $\Sigma;\Gamma \vdash h \Rightarrow B$. By Definition 2.103 (type elimination), $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \varepsilon \Downarrow B$.
- CONV: By the premises, $\Sigma;\Gamma \vdash h\,\vec{e} : B''$ for some B'' with $\Sigma;\Gamma \vdash B'' \equiv B$ **type**.

  By the induction hypothesis, $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow B'$ for some B', with $\Sigma;\Gamma \vdash B'' \equiv B'$ **type**.

  By symmetry and transitivity, $\Sigma;\Gamma \vdash B' \equiv B$ **type**.
- APP: Then $\vec{e} = \vec{e}'\, u$, and we have $\Sigma;\Gamma \vdash h\,\vec{e} : \Pi A' B'$ for some A' and B', with $\Sigma;\Gamma \vdash u : A$ and $B = B'[u]$.

  By the induction hypothesis, $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow U$, with $\Sigma;\Gamma \vdash U \equiv \Pi A' B'$ **type**.

  By Lemma 2.101 (nose of weak-head normal form), $\Sigma \vdash U \searrow \Pi A_0 B_0$ for some A_0 and B_0. By Lemma 2.98 (equality of WHNF) and transitivity, $\Sigma;\Gamma \vdash \Pi A' B' \equiv \Pi A_0 B_0$ **type**.

  By Postulate 10 (injectivity of Π), $\Sigma \vdash \Gamma, A', B' \equiv \Gamma, A_0, B_0$ **ctx**. By the CONV rule and Postulate 1 (typing of hereditary substitution), $B_0[u]\Downarrow$.

  Thus, by Definition 2.103 (type elimination), $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e}'\, u \Downarrow B_0[u]$.

  Finally, by Postulate 4 (congruence of hereditary substitution), $\Sigma;\Gamma \vdash B_0[u] \equiv B'[u]$ **type**; that is, $\Sigma;\Gamma \vdash B_0[u] \equiv B$ **type**.
- PROJ1, PROJ2: Analogous to the APP case.

□

Remark 2.108 (Uniqueness of head type lookup). If $\Sigma;\Gamma \vdash h \Rightarrow A$ and $\Sigma;\Gamma \vdash h \Rightarrow A'$, then $A = A'$.

Proof. By case analysis on the derivations of $\Sigma;\Gamma \vdash h \Rightarrow A$ and $\Sigma;\Gamma \vdash h \Rightarrow A'$. □

Lemma 2.109 (Type application inversion)**.** *Assume that* $\Sigma;\Gamma \vdash h\,\vec{u} : B$. *Then there is a unique* A *and a* B' *such that* $\Sigma;\Gamma \vdash h \Rightarrow A$, $\Sigma;\Gamma \vdash A \mathbin{\hat{@}} \vec{u} \Downarrow B'$, *and* $\Sigma;\Gamma \vdash B' \equiv B : \mathrm{Set}$.

Proof. Analogous to the proof of Lemma 2.107 (type elimination inversion). Uniqueness follows from Remark 2.108 (uniqueness of head type lookup). □

Lemma 2.110 (Type of hereditary application)**.** *Assume* $\Sigma;\Gamma \vdash t : A$, *and* $\Sigma;\Gamma \vdash A \mathbin{\hat{@}} \vec{u} \Downarrow A'$. *Then* $t \mathbin{@} \vec{u} \Downarrow t'$, *and* $\Sigma;\Gamma \vdash t' : A'$ *Additionally, if* $\Sigma;\Gamma \vdash t_1 \equiv t_2 : A$, *then* $t_1 \mathbin{@} \vec{u} \Downarrow t_1'$, $t_2 \mathbin{@} \vec{u} \Downarrow t_2'$, *and* $\Sigma;\Gamma \vdash t_1' \equiv t_2' : A'$.

Proof. By induction on the length of $\vec{u}$, using Postulate 2 (typing of hereditary application) and case analysis on $\Sigma;\Gamma \vdash A \mathbin{\hat{@}} \vec{u} \Downarrow A'$ to build the typing derivation. For the second part, by induction on the length of u and using Postulate 6 (congruence of hereditary application). □

Lemma 2.111 (Application inversion)**.** *If* $\Sigma;\Gamma \vdash h\,\vec{e}\,u\,\vec{e}' : T$, *then there are* A, U *and* V *such that* $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow A$, $\Sigma \vdash A \searrow \Pi UV$ *and* $\Sigma;\Gamma \vdash u : U$. *Also,* $\Sigma;\Gamma \vdash A \equiv \Pi UV$ **type** *and* $\Sigma;\Gamma \vdash h\,\vec{e} : \Pi UV$. *Furthermore, if* $\Sigma;\Gamma \vdash h\,\vec{e} : \Pi U'V'$, *then* $\Sigma;\Gamma \vdash U \equiv U'$ **type**, *then* $\Sigma;\Gamma, U \vdash V \equiv V'$ **type**, *and* $\Sigma;\Gamma \vdash u : U'$ **type**.

Proof. By Lemma 2.107 (type elimination inversion), $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e}\,u\,\vec{e}' \Downarrow T_0$ for some T_0.

By induction on the derivation of $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e}\,u\,\vec{e}'\Downarrow$, we have $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow C$ and $\Sigma \vdash C \searrow \Pi UV$. By Lemma 2.98 (equality of WHNF), $\Sigma;\Gamma \vdash C \equiv \Pi UV$ **type**. By Lemma 2.106 (type elimination) and the CONV rule, $\Sigma;\Gamma \vdash h\,\vec{e} : \Pi UV$.

By induction on the derivation of $\Sigma;\Gamma \vdash h\,\vec{e}\,u\,\vec{e}' : T$, we have $\Sigma;\Gamma \vdash h\,\vec{e} : \Pi AB$ and $\Sigma;\Gamma \vdash u : A$.

By Lemma 2.75 (uniqueness of typing for neutrals), $\Sigma;\Gamma \vdash \Pi AB \equiv \Pi UV$ **type**. By Postulate 10 (injectivity of Π), $\Sigma;\Gamma \vdash A \equiv U$ **type**. By the CONV rule, $\Sigma;\Gamma \vdash u : U$.

By Lemma 2.98 (equality of WHNF), $\Sigma;\Gamma \vdash A \equiv \Pi UV$ **type**; and by Lemma 2.106 (type elimination), $\Sigma;\Gamma \vdash h\,\vec{e} : \Pi UV$.

By Lemma 2.75 (uniqueness of typing for neutrals), and Postulate 10 (injectivity of Π), $\Sigma;\Gamma \vdash U \equiv U'$ **type** and $\Sigma;\Gamma, U \vdash V \equiv V'$ **type**. By the CONV rule, $\Sigma;\Gamma \vdash u : U'$ **type**. □

Lemma 2.112 (Iterated application inversion)**.** *Assume* $\Sigma;\Gamma \vdash h\,\vec{u}^n : T$. *Then there exist* $\vec{A}$ *and* B *such that* $\Sigma;\Gamma \vdash T \equiv \Pi\vec{A}B$ **type** *and* $\Sigma;\Gamma \vdash h\,\vec{u}^n : \Pi\vec{A}B$.

Proof. By induction on n and Lemma 2.111 (application inversion). □

Lemma 2.113 (Projection inversion)**.** *If* $\Sigma;\Gamma \vdash h\,\vec{e}\,.\pi_1\,\vec{e}' : T$ *or* $\Sigma;\Gamma \vdash h\,\vec{e}\,.\pi_2\,\vec{e}' : T$, *then there are* A, U *and* V *such that* $\Sigma;\Gamma \vdash (h :) \mathbin{\hat{@}} \vec{e} \Downarrow A$ *and* $\Sigma \vdash A \searrow \Sigma UV$. *In particular,* $\Sigma;\Gamma \vdash A \equiv \Sigma UV$ **type**, *and* $\Sigma;\Gamma \vdash h\,\vec{e} : \Sigma UV$.

Proof. Analogous to the proof of Lemma 2.111 (application inversion). □

Definition 2.114 (Type application, reversed: $\Sigma; \Gamma \vdash T\ \hat{@}^{\mathrm{R}}\ \vec{t} \Downarrow U$)**.** Given a signature Σ and context Γ, reverse elimination of a type T by a spine $\vec{t}$, resulting in a type U (written $T\ \hat{@}\ \vec{t} \Downarrow U$) is defined as follows:

$$\begin{array}{lll} \Sigma; \Gamma \vdash T\ \hat{@}^{\mathrm{R}}\ \varepsilon \Downarrow T & & \\ \Sigma; \Gamma \vdash T\ \hat{@}^{\mathrm{R}}\ u\,\vec{t} \Downarrow T' & \textbf{if} & \Sigma \vdash T \searrow \Pi AB \textbf{ and} \\ & & \Sigma; \Gamma \vdash u : A \textbf{ and} \\ & & B[u] \Downarrow B' \textbf{ and} \\ & & \Sigma; \Gamma \vdash B'\ \hat{@}^{\mathrm{R}}\ \vec{t} \Downarrow T' \end{array}$$

Lemma 2.115 (Type application, reversed)**.** *We have* $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow U$ *if and only if* $\Sigma; \Gamma \vdash T\ \hat{@}^{R}\ \vec{t} \Downarrow U$.

Proof. We show the following, stronger property: "For all V, we have $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow V$ and $\Sigma; \Gamma \vdash V\ \hat{@}^{\mathrm{R}}\ \vec{u} \Downarrow U$ if and only if $\Sigma; \Gamma \vdash T\ \hat{@}^{\mathrm{R}}\ \vec{t}\,\vec{u} \Downarrow U$."

By induction on the derivation of $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow V$:

- Case $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{\varepsilon} \Downarrow V$: Then by Definition 2.104 (type application), necessarily $T = V$, and the property holds trivially.
- Case $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t}\,v \Downarrow V$: By Definition 2.104 (type application), $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t}\,v \Downarrow V$ and $\Sigma; \Gamma \vdash V\ \hat{@}^{\mathrm{R}}\ \vec{u} \Downarrow U$ are equivalent to having $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow V'$, $\Sigma \vdash V' \searrow \Pi AB$, $\Sigma \vdash v : A$, $B[u] \Downarrow B'$ (and $\Sigma; \Gamma \vdash V\ \hat{@}^{\mathrm{R}}\ \vec{u} \Downarrow U$), for some A, B. By Definition 2.114 (type application, reversed), this is equivalent to ($\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow V'$ and) $\Sigma; \Gamma \vdash V'\ \hat{@}^{\mathrm{R}}\ v\,\vec{u} \Downarrow U$. By the induction hypothesis, this is equivalent to $\Sigma; \Gamma \vdash T\ \hat{@}^{\mathrm{R}}\ \vec{t}\,v\,\vec{u} \Downarrow U$.

The lemma follows from the property by taking $\vec{u} := \varepsilon$ and $V := T$. □

Lemma 2.116 (Free variables in type application)**.** *If* $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow A$*, or* $\Sigma; \Gamma \vdash T\ \hat{@}^{R}\ \vec{t} \Downarrow A$ *then* $\mathrm{FV}(T) \cup \mathrm{FV}(t) \supseteq A$.

Proof. By Lemma 2.115 (type application, reversed), it suffices to show the property for the case $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow A$. We proceed by induction on the derivation of $\Sigma; \Gamma \vdash T\ \hat{@}\ \vec{t} \Downarrow A$, using Remark 2.102 (preservation of free variables by WHNF) and Lemma 2.51 (free variables in hereditary substitution) for the inductive step. □

Lemma 2.117 (Commuting of renamings with hereditary substitution and elimination)**.** *The following hold:*

(i) If $t[u/x] \Downarrow r$*, then* $t^{(\rho+x+1)}[u^{(\rho+x)}/x] \Downarrow r^{(\rho+x)}$.

(ii) If $(t\ @\ \vec{e}) \Downarrow u$*, then* $(t^{\rho}\ @\ \vec{e}^{\rho}) \Downarrow u^{\rho}$.

Proof. By mutual induction on the derivations. □

Lemma 2.118 (Commuting of renamings with WHNF)**.** *Let* ρ *be a renaming, if* $\Sigma \vdash t \searrow u$*, then* $\Sigma \vdash t^{\rho} \searrow u^{\rho}$.

Proof. By induction on the derivation, using Lemma 2.117 (commuting of renamings with hereditary substitution and elimination). □

Lemma 2.119 (Commuting of renaming with reversed type application)**.** *The following hold:*

(i) Assume $\Sigma; \Gamma' \vdash A$ **type**. *If* $\Sigma; \Gamma', \Gamma'' \vdash T \mathbin{\hat{@}^R} \vec{u} \Downarrow U$, *then* $\Sigma; \Gamma', A, \Gamma''^{(+1)} \vdash T^{(+1)+|\Gamma''|} \mathbin{\hat{@}^R} \vec{u}^{(+1)+|\Gamma''|} \Downarrow U^{(+1)+|\Gamma''|}$.

(ii) If $\Sigma; \Gamma', y : A, \Gamma'', x : A', \Gamma''' \vdash T \mathbin{\hat{@}^R} \vec{u} \Downarrow U$, *with* $\Sigma; \Gamma', y : A, \Gamma'', x : A' \vdash x : A^{(+|A,\Gamma'',A'|)}$, *then* $\Sigma; \Gamma', y : A, \Gamma'', x : A', \Gamma'''[x \mapsto y] \vdash T[x \mapsto y] \mathbin{\hat{@}^R} \vec{u}[x \mapsto y] \Downarrow U[x \mapsto y]$.

Proof. By induction on the derivations, using Lemma 2.98 (equality of WHNF) Lemma 2.118 (commuting of renamings with WHNF), Lemma 2.78 (variable types say everything) and Lemma 2.62 (context weakening). □

The following lemma is key for ensuring that metavariable solutions are well-typed:

Lemma 2.120 (Typing of metavariable bodies)**.** *Assume we have* $\alpha : A \in \Sigma$, *with* $\Sigma; \Gamma \vdash \alpha\, \vec{x}^n : B$, *where all the variables in the vector* $\vec{x}$ *are pairwise distinct. Let* t *be a term such that* $\Sigma; \Gamma \vdash t : B$, *and* $\mathrm{FV}(t) \subseteq \{\vec{x}\}$. *Then,* $\Sigma; \cdot \vdash \lambda \vec{y}^n.t[\vec{x} \mapsto \vec{y}] : A$ *and* $(\lambda \vec{y}^n.t[\vec{x} \mapsto \vec{y}]) \mathbin{@} \vec{x} \Downarrow t$.

Proof. We show the following (stronger) property:

For all $\Delta' = \overrightarrow{T'_{\mathrm{y}}}$ with $\overrightarrow{y'} = (|\Delta'| - 1), ..., 0$; $\Gamma = \overrightarrow{T_{\mathrm{z}}}$ with $\vec{z} = (|\Gamma| - 1), ..., 0$, $\{\vec{x}\} \subseteq \{\vec{z}\}$ with all variables in $\vec{x}$ pairwise distinct, A', B' and t, suppose:

(i) $\Sigma; \Delta' \vdash A'$ **type**,

(ii) $\Sigma; \Delta', \Gamma \vdash A'^{(+|\Gamma|)} \mathbin{\hat{@}^{\mathrm{R}}} \vec{x}^n \Downarrow B'$,

(iii) $\Sigma; \Delta', \Gamma \vdash t : B'$,

(iv) $\mathrm{FV}(A') \subseteq \{\overrightarrow{y'}\}$ and

(v) $\mathrm{FV}(t) \cup \mathrm{FV}(B') \subseteq \{\vec{x}\} \cup \{\overrightarrow{y'}^{(+|\Gamma|)}\}$.

Then there exists $\Delta = \overrightarrow{T_{\mathrm{y}}}^n$ with $\vec{y} = (|\Delta| - 1), ..., 0$ such that $\Sigma; \Delta' \vdash \Pi\Delta(B'[\overrightarrow{y'}^{(+|\Gamma|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]) \equiv A'$ **type**, and $\Sigma; \Delta', \Delta \vdash t[\overrightarrow{y'}^{(+|\Gamma|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}] : B'[\overrightarrow{y'}^{(+|\Gamma|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]$.

The proof proceeds by induction on $\vec{x}$.

In all cases, note that the variables in $\vec{x}$ must be pairwise distinct for the renaming "$[\overrightarrow{y'}^{(+|\Gamma|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]$" to be well-formed.

- ε: Take $\Delta = \cdot$. $\Sigma; \Delta', \Gamma \vdash A'^{(+|\Gamma|)} \mathbin{\hat{@}^{\mathrm{R}}} \varepsilon \Downarrow B'$ implies $B' = A'^{(+|\Gamma|)}$. Also, $B'[\overrightarrow{y'}^{(+|\Gamma|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}] = B'[\overrightarrow{y'}^{(+|\Gamma|)} \mapsto \overrightarrow{y'}] = B'^{(-|\Gamma|)} = A'^{(+|\Gamma|)(-|\Gamma|)} = A'$. Thus, by reflexivity, $\Sigma; \Delta' \vdash B'[\overrightarrow{y'}^{(+|\Gamma|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}] \equiv A'$ **type**.

  By the assumptions $\mathrm{FV}(A') \subseteq \{\overrightarrow{y'}\}$ and $\mathrm{FV}(t) \cup \mathrm{FV}(B') \subseteq \varepsilon \cup \{\overrightarrow{y'}^{(+|\Gamma|)}\}$.

By Postulate 13 (context strengthening), $\Sigma;\Delta' \vdash t^{(-\Gamma)} : B'^{(-\Gamma)}$. By construction, $t[\overrightarrow{y'}^{(+|\Gamma|)} \mapsto \overrightarrow{y'}^{(+|\Delta|)}] = t[\overrightarrow{y'}^{(+|\Gamma|)} \mapsto \overrightarrow{y'}] = t^{(-\Gamma)}$, and, as shown earlier, $B'[\overrightarrow{y'}^{(+|\Gamma|)} \mapsto \overrightarrow{y'}^{(+|\Delta|)}] = B'^{(-\Gamma)}$. Therefore, $\Sigma;\Delta' \vdash t[\overrightarrow{y'}^{(+|\Gamma|)} \mapsto \overrightarrow{y'}^{(+|\Delta|)}] : B'[\overrightarrow{y'}^{(+|\Gamma|)} \mapsto \overrightarrow{y'}^{(+|\Delta|)}]$.

- $x'\ \vec{x}^n$: By Definition 2.114 (type application, reversed), we have (i) $\Sigma \vdash A'^{(+|\Gamma|)} \searrow \Pi UV$, (ii) $\Sigma;\Delta',\Gamma \vdash x' : U$, (iii) $V[x']\Downarrow$ and (iv) $\Sigma;\Delta',\Gamma \vdash V[x']\ \hat{@}^{\mathrm{R}}\ \vec{x}^n \Downarrow B'$; where $\Gamma = \Gamma', U', \Gamma''$ for some U', and $x' = |\Gamma''|$. Let $x_0 \stackrel{\text{def}}{=} 0$; thus $x' = x_0^{(+|\Gamma''|)}$.

  Because $\mathrm{FV}(A') \subseteq \{\overrightarrow{y'}^{(+|\Gamma|)}\}$, by Remark 2.102 (preservation of free variables by WHNF) then also $\mathrm{FV}(U) \subseteq \{\overrightarrow{y'}^{(+|\Gamma|)}\}$. Therefore, $U^{(-|\Gamma|)}$ is well-defined, and, by Postulate 13 (context strengthening), $\Sigma;\Delta' \vdash U^{(-|\Gamma|)}$ **type**.

  By Lemma 2.62 (context weakening) $\Sigma;\Delta', y_0 : U^{(-|\Gamma|)},\Gamma^{(+1)} \vdash t^{(+1)+|\Gamma|} : B'^{(+1)+|\Gamma|}$, that is, $\Sigma;\Delta', y_0 : U^{(-|\Gamma|)},\Gamma'^{(+1)}, U'^{((+1)+|\Gamma'|)}, (\Gamma''^{((+1)+|\Gamma',U'|)}) \vdash t^{(+1)+|\Gamma|} : B'^{(+1)+|\Gamma|}$, Let $y_0 \stackrel{\text{def}}{=} 0$.

  From $\Sigma;\Delta',\Gamma \vdash x' : U$ and by Postulate 13 (context strengthening), $\Sigma;\Delta',\Gamma', x_0 : U' \vdash x_0 : U^{(-|\Gamma''|)}$. By Remark 2.30 (properties of renamings), $\Sigma;\Delta',\Gamma', x_0 : U' \vdash x_0 : U^{(-|\Gamma|)(+|\Gamma',U'|)}$. By Lemma 2.62 (context weakening), $\Sigma;\Delta', y_0 : U^{(-|\Gamma|)},\Gamma'^{(+1)}, x_0 : U'^{(+1)+|\Gamma'|} \vdash x_0 : U^{(-|\Gamma|)(+|\Gamma',U'|)((+1)+|\Gamma',U'|)}$. By Remark 2.30, $\Sigma;\Delta', y_0 : U^{(-|\Gamma|)},\Gamma'^{(+1)}, x_0 : U'^{(+1)+|\Gamma'|} \vdash x_0 : U^{(-|\Gamma|)(+|U,\Gamma',U'|)}$.

  By Lemma 2.78 (variable types say everything) $\Sigma;\Delta', y_0 : U^{(-|\Gamma|)},\Gamma_0 \vdash t_0 : B_0$, where $\Gamma_0 = \Gamma'^{(+1)}, U'^{((+1)+|\Gamma'|)}, (\Gamma''^{((+1)+|\Gamma',U'|)})[x_0 \mapsto y_0^{(+|\Gamma',U'|)}]$, $t_0 = t^{((+1)+|\Gamma|)}[x_0^{(+|\Gamma''|)} \mapsto y_0^{(+|\Gamma',U',\Gamma''|)}]$, and $B_0 = B'^{((+1)+|\Gamma|)}[x_0^{(+|\Gamma''|)} \mapsto y_0^{(+|\Gamma',U',\Gamma''|)}]$.

  Note that $|\Gamma', U', \Gamma''| = |\Gamma| = |\Gamma_0|$. By Remark 2.102 (preservation of free variables by WHNF), $\mathrm{FV}(V) \subseteq \{0\} \cup (\mathrm{FV}(A'^{(+|\Gamma|)}) + 1) \subseteq \{0\} \cup \{\overrightarrow{y'}^{(+(1+|\Gamma|))}\}$.

  By Postulate 13 (context strengthening), $\Sigma;\Delta',\Gamma', x' : U' \vdash x' : U^{(-|\Gamma''|)}$. By Lemma 2.62 (context weakening), $\Sigma;\Delta', U^{(-|\Gamma|)},\Gamma'^{(+1)}, U'^{((+1)+|\Gamma'|)} \vdash x_0^{((+1)+|\Gamma',U'|)} : U^{(-|\Gamma''|)((+1)+|\Gamma',U'|)}$. Note that $\mathrm{FV}(U) \subseteq \{|\Delta',\Gamma| - 1,...,|\Gamma|\}$, i.e. $\mathrm{FV}(U^{(-|\Gamma''|)}) \subseteq \{|\Delta',\Gamma',U'| - 1,...,|\Gamma',U'|\}$. Therefore, $U^{(-|\Gamma''|)((+1)+|\Gamma',U'|)} = U^{(-|\Gamma''|)((+1))}$. By Remark 2.30 (properties of renamings), $U^{(-|\Gamma''|)((+1))} = U^{(-|\Gamma|)((+|U,\Gamma',U'|))}$. Therefore, $\Sigma;\Delta', U^{(-|\Gamma|)},\Gamma'^{(+1)}, U'^{((+1)+|\Gamma'|)} \vdash x_0 : U^{(-|\Gamma|)(+|U,\Gamma',U'|)}$.

  Thus, by Lemma 2.119 (Commuting of renaming with reversed type application, (i) and then (ii)), $\Sigma;\Delta', U^{(-|\Gamma|)},\Gamma_0 \vdash V_0\ \hat{@}^{\mathrm{R}}\ \vec{x}^n \Downarrow B_0$, where $V_0 = V[x_0^{(+|\Gamma''|)}]^{((+1)+|\Gamma|)}[x_0^{(+|\Gamma''|)} \mapsto y_0^{(+|\Gamma|)}]$.

  Similarly, by Lemma 2.62 (context weakening) and then Lemma 2.78 (variable types say everything), $\Sigma;\Delta', U^{(-|\Gamma|)},\Gamma_0 \vdash V_0$ **type**. Note that

$\forall x \in \text{FV}(V_0).x \geq |\Gamma_0|$. By Postulate 13 (context strengthening) (and noting $|\Gamma| = |\Gamma_0|$), $\Sigma; \Delta', U^{(-|\Gamma_0|)} \vdash V_0^{(-|\Gamma_0|)}$ **type**.

By Lemma 2.40 (correspondence between renaming and substitution) and Remark 2.29 (composition of renamings), $V_0 = V[\overrightarrow{y'}^{(+|\Gamma,U'|)}, \vec{z}^{(+1)}, 0 \mapsto \overrightarrow{y'}^{(+|U,\Gamma|)}, \vec{z}, y_0^{(+\Gamma)}]$. Note that $\{\vec{z}^{(+1)}\} \cap \text{FV}(V) = 0$. By Remark 2.29, Definition 2.23 (strengthening), Definition 2.22 (weakening) and Definition 2.24 (weakening of renamings), $V_0 = V^{((-|\Gamma|)+1)(+|\Gamma|)}$.

By Remark 2.28 (renaming and free variables) we have $\text{FV}(V_0) \subseteq (\text{FV}(V))^{((-|\Gamma|)+1)(+|\Gamma|)} = (\{0\} \cup \{\overrightarrow{y'}^{(+(1+|\Gamma|))}\})^{((-|\Gamma|)+1)(+|\Gamma|)} = \{(\overrightarrow{y'}^{(+1)}, y_0)^{(+|\Gamma|)}\}$. By Remark 2.28 and Remark 2.30 (properties of renamings), $\text{FV}(V_0^{(-|\Gamma_0|)}) \subseteq \{(\overrightarrow{y'}^{(+1)}, y_0)\}$.

Similarly, $\text{FV}(t_0) \cup \text{FV}(B_0) \subseteq \{\vec{x}\} \cup \{(\overrightarrow{y'}^{(+1)}, y_0)^{(+|\Gamma|)}\}$.

By the induction hypothesis (instantiating $\Delta' := (\Delta', U^{(-|\Gamma_0|)})$, $\Gamma := \Gamma_0$, $A' := V_0^{(-|\Gamma_0|)}$, $B' := B_0$ and $t := t_0$, thus $\overrightarrow{y'} = (\overrightarrow{y'}^{(+1)}, y_0)$), there exists Δ_1 with $|\Delta_1| = n$ such that $\Sigma; \Delta', U^{(-|\Gamma|)} \vdash \Pi\Delta_1 B_1 \equiv V_1$ **type** and $\Sigma; \Delta', U^{(-|\Gamma|)}, \Delta_1 \vdash t_1 : B_1$ where $\overrightarrow{y_{(1)}} = (|\Delta_1| - 1), ..., 0$, $B_1 = B_0[(\overrightarrow{y'}^{(+1)}, y_0)^{(+|\Gamma_0|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \overrightarrow{y_{(1)}}]$, $t_1 = t_0[(\overrightarrow{y'}^{(+1)}, y_0)^{(+|\Gamma_0|)}, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \overrightarrow{y_{(1)}}]$ and $V_1 = V_0^{(-|\Gamma|)}$.

Note that $|\Gamma_0| = |\Gamma|$. Take $\Delta \stackrel{\text{def}}{\equiv} (U^{(-|\Gamma|)}, \Delta_1)$ and $\vec{y} \stackrel{\text{def}}{\equiv} (y_0^{(+|\Delta_1|)}, \overrightarrow{y_{(1)}}) = ((|\Delta| - 1), ..., 0)$.

By Remark 2.29 (composition of renamings), $B_1 = B'[\overrightarrow{y'}^{(+|\Gamma|)}, x', \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]$, $t_1 = t[\overrightarrow{y'}^{(+|\Gamma|)}, x', \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]$. (Note that all the variables in $(x', \vec{x})$ are distinct and smaller than $|\Gamma|$, so the renaming is indeed well-formed).

This gives $\Sigma; \Delta', \Delta \vdash t[\overrightarrow{y'}^{(+|\Gamma|)}, x', \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}] : B'[\overrightarrow{y'}^{(+|\Gamma|)}, x', \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]$.

As shown above, $\Sigma; \Delta', U^{(-|\Gamma|)} \vdash \Pi\Delta_1 B_1 \equiv V_1$ **type**. By Remark 2.30 (properties of renamings), $V_1 = V^{(-|\Gamma|)+1}$. By reflexivity and the PI-EQ rule, $\Sigma; \Delta' \vdash \Pi\Delta B_1 \equiv \Pi U^{(-\Gamma)}(V^{(-|\Gamma|)+1})$ **type**.

By Lemma 2.62 (context weakening), $\Sigma; \Delta' \vdash A'^{(+|\Gamma|)}$ **type**. Because $\Sigma \vdash A'^{(+|\Gamma|)} \searrow \Pi UV$, by Lemma 2.98 (equality of WHNF), $\Sigma; \Delta' \vdash A'^{(+|\Gamma|)} \equiv \Pi UV$ **type**. By Postulate 13 (context strengthening), $\Sigma; \Delta' \vdash A' \equiv (\Pi UV)^{(-|\Gamma|)}$ **type**.

By Definition 2.20 (renaming), transitivity and symmetry of equality, $\Sigma; \Delta' \vdash \Pi\Delta(B'[\overrightarrow{y'}^{(+|\Gamma|)}, x_0, \vec{x} \mapsto \overrightarrow{y'}^{(+|\Delta|)}, \vec{y}]) \equiv A'$ **type**.

We now use the property to prove the main theorem.

By the META_1 rule, $\Sigma; \Gamma \vdash \alpha \Rightarrow A^{(+|\Gamma|)}$. By Lemma 2.109 (type application inversion) and Lemma 2.115 (type application, reversed), $\Sigma; \Gamma \vdash A^{(+|\Gamma|)}\ \widehat{@}^{\text{R}}\ \vec{x} \Downarrow B'$ for some B', and $\Sigma; \Gamma \vdash B \equiv B'$ **type**. By the CONV rule, $\Sigma; \Gamma \vdash t :$

B'. Because Σ **sig**, we have $\Sigma_1; \cdot \vdash A$ **type** for some Σ_1. By Lemma 2.65 (no extraneous variables in term), this means $\text{FV}(A) = \emptyset$. By Remark 2.28 (renaming and free variables), $\text{FV}(A^{(+|\Gamma|)}) = \emptyset$. By Lemma 2.116 (free variables in type application), $\text{FV}(B') \subseteq \{\vec{x}\} \cup \text{FV}(A^{(+|\Gamma|)}) \subseteq \{\vec{x}\} \cup \emptyset$.

By applying the above-proven property with $\Delta' := \varepsilon$, $\overrightarrow{y'} := \varepsilon$, $\Gamma := \Gamma$, $B' := B'$, $A' := A$, $t := t$, $\overrightarrow{x'} := \overrightarrow{x'}$, we have $\Sigma; \cdot \vdash \Pi\Delta(B'[\vec{x} \mapsto \vec{y}]) \equiv A$ **type**, and $\Sigma; \Delta \vdash t[\vec{x} \mapsto \vec{y}] : B'[\vec{x} \mapsto \vec{y}]$, where $\Delta = \overrightarrow{T_y}^n$ and $\vec{y} = (|\Delta| - 1), ..., 0$. By the ABS rule, $\Sigma; \cdot \vdash \lambda\vec{y}^n.t[\vec{x} \mapsto \vec{y}] : \Pi\Delta(B'[\vec{x} \mapsto \vec{y}])$, which by the CONV-EQ rule gives $\Sigma; \cdot \vdash \lambda\vec{y}^n.t[\vec{x} \mapsto \vec{y}] : A$.

Finally, by induction on n, using Definition 2.33 (iterated hereditary elimination), Definition 2.32 (hereditary elimination), Lemma 2.40 (correspondence between renaming and substitution) and Remark 2.29 (composition of renamings), $\lambda\vec{y}^n.(t[\vec{x} \mapsto \vec{y}]) \mathbin{@} \vec{x} \Downarrow t$.

□

2.17 Metasubstitutions (Θ)

Metasubstitutions are signatures which instantiate all the metavariables with terms containing no metavariables.

Definition 2.121. Metasubstitution: (Θ)

Θ	::=	$\cdot$	empty metasubstitution
	$\mid$	$\Theta, \mathbb{a} : A$	atom
	$\mid$	$\Theta, \alpha := t : A$	metavariable instantiation

Definition 2.122 (Well-formed metasubstitution: Θ **wf**)**.** A metasubstitution Θ is well-formed (written Θ **wf**) if it is well-formed as a signature, and none of the types and terms in it contain any metavariables:

$$\frac{}{\cdot \mathbf{wf}}\ \text{EMPTY}$$

$$\frac{\Theta \mathbf{wf} \qquad \mathbb{a} \text{ is fresh in } \Theta \qquad \Theta; \cdot \vdash A \textbf{ type} \qquad \text{METAS}(A) = \emptyset}{\Theta, \mathbb{a} : A \mathbf{wf}}\ \text{SUBST-AXIOM}$$

$$\frac{\Theta \mathbf{wf} \qquad \alpha \text{ is fresh in } \Theta \qquad \Theta; \cdot \vdash t : A \qquad \text{METAS}(t) = \text{METAS}(A) = \emptyset}{\Theta, \alpha := t : A \mathbf{wf}}\ \text{SUBST-META}$$

Remark 2.123 (Metasubstitutions are signatures). Given a metasubstitution Θ, if Θ **wf** then Θ **sig**.

Furthermore, if Θ is a metasubstitution such that, for all $\mathbb{a} : A \in \Theta$, $\text{METAS}(A) = \emptyset$; and, for all $\alpha := t : A \in \Theta$, $\text{METAS}(t) \cup \text{METAS}(A) = \emptyset$; and Θ **sig**, then Θ **wf**.

(Note that, by Lemma 2.70 (piecewise well-formedness of typing judgments), if $\Theta; \cdot \vdash t : A$ then $\Theta; \cdot \vdash A$ **type**).

Definition 2.124 (Metasubstitution subsumption: $\Theta \subseteq \Theta'$)**.** We say $\Theta \subseteq \Theta'$ if Θ **wf**, Θ' **wf**, and, when taking Θ and Θ' as signatures, $\Theta \subseteq \Theta'$.

Definition 2.125 (Compatible metasubstitution: $\Theta \vDash \Sigma$)**.** We say that Θ is compatible with Σ (written $\Theta \vDash \Sigma$) if $\Theta\,\mathbf{wf}$, $\Sigma\,\mathbf{sig}$, $\textsc{decls}(\Theta) = \textsc{decls}(\Sigma)$, and, for every judgment J, if $\Sigma \vdash J$, then $\Theta \vdash J$.

Definition 2.126 (Declaration: (D))**.** Each of the elements of a signature is a *declaration*. If D is a declaration, then either $D = \mathbb{o} : A$, $D = \alpha : A$ or $D = \alpha := t : A$.

Definition 2.127 (Compatibility of a metasubstitution with a declaration: Θ compatible with D)**.** We say that a metasubstitution Θ is compatible with a declaration D if any of the following hold:

- $D = \mathbb{o} : A$, and $\Theta; \cdot \vdash \mathbb{o} : A$.
- $D = \alpha : A$, and $\Theta; \cdot \vdash \alpha : A$.
- $D = \alpha := u : A$, and $\Theta; \cdot \vdash \alpha \equiv u : A$.

Remark 2.128 (Compatibility with a declaration as a judgment: $J = D$)*.* Given a declaration D, there is a judgment J such that, for any metasubstitution Θ, Θ is compatible with D if and only if $\Theta \vdash J$.

Proof.

- If $D = \mathbb{o} : A$, then $J = \cdot \vdash \mathbb{o} : A$.
- If $D = \alpha : A$, then $J = \cdot \vdash \alpha : A$.
- If $D = \alpha := u : A$, then $J = \cdot \vdash \alpha \equiv u : A$.

□

Remark 2.129 (Alternative characterization of compatibility of a metasubstitution with a declaration). A well-formed metasubstitution $\Theta\,\mathbf{wf}$ is compatible with a declaration D iff any of the following hold:

i) $D = \mathbb{o} : A$, and there is $\mathbb{o} : B \in \Theta$ such that $\Theta; \cdot \vdash B \equiv A\ \mathbf{type}$.

ii) $D = \alpha : A$, and there is $\alpha := t : B \in \Theta$ and $\Theta; \cdot \vdash B \equiv A\ \mathbf{type}$.

iii) $D = \alpha := t : A$, and there is $\alpha := u : B \in \Theta$ such that $\Theta; \cdot \vdash B \equiv A\ \mathbf{type}$ and $\Theta; \cdot \vdash t \equiv u : B$.

Proof.

$\Leftarrow$. Follows from the typing rules.

$\Rightarrow$. Follows by induction on the derivation of $\Theta\,\mathbf{wf}$.

□

Lemma 2.130 (Alternative characterization of a compatible metasubstitution)**.** *Let Θ be a well-formed metasubstitution, and Σ be a well-formed signature. We have $\Theta \vDash \Sigma$ if and only if $\textsc{decls}(\Theta) = \textsc{decls}(\Sigma)$, and, for each declaration $D \in \Sigma$, D is compatible with Θ.*

Proof.

$\Rightarrow$. Assume $\Theta \vDash \Sigma$.

- $\mathbb{c} : A \in \Sigma$. By the rules ATOM and HEAD, $\Sigma; \cdot \vdash \mathbb{c} : A$. Because $\Theta \vDash \Sigma$, we have $\Theta; \cdot \vdash \mathbb{c} : A$.
- $\alpha : A \in \Sigma$. By the rules META_1 and HEAD, $\Sigma; \cdot \vdash \alpha : A$. Because $\Theta \vDash \Sigma$, we have $\Theta; \cdot \vdash \alpha : A$.
- $\alpha := t : A \in \Sigma$. By the rule DELTA-META_0, $\Sigma; \cdot \vdash \alpha \equiv t : A$. By the assumption, $\Theta; \cdot \vdash \alpha \equiv t : A$.

$\Leftarrow$. We need to show that, if $\Sigma \vdash J$, then $\Theta \vdash J$. We do induction on the structure of J (in case J is a conjunction of judgments), and, for the base cases, induction on the derivation of the corresponding judgment.

We do analysis on the lowest rule of the derivation tree. For those rules which do not involve the signature, we apply the induction hypothesis to the premise and use the same rule.

There are three rules that involve the signature: HEAD, HEAD-EQ and DELTA-META.

a) HEAD:

$$\frac{\Sigma; \Gamma \vdash h \Rightarrow A}{\Sigma; \Gamma \vdash h : A}\ \text{HEAD}$$

- If $h = \alpha$, then necessarily $\alpha : A \in \Sigma$ or $\alpha := t : A \in \Sigma$. By the assumption, Θ is compatible with $\alpha : A$; therefore $\Theta; \cdot \vdash \alpha : A$. By Lemma 2.62 (context weakening), $\Theta; \Gamma \vdash \alpha : A$.
- If $h = \mathbb{c}$, then, analogously to the previous case, $\Theta; \Gamma \vdash \mathbb{c} : A$.
- If $h = x$ or $h = \text{if}$, and $\Sigma; \Gamma \vdash h \Rightarrow A$, then also $\Theta; \Gamma \vdash h \Rightarrow A$. By the HEAD rule, $\Theta; \Gamma \vdash h : A$.

b) HEAD-EQ:

$$\frac{\Sigma; \Gamma \vdash h \Rightarrow A}{\Sigma; \Gamma \vdash h \equiv h : A}\ \text{HEAD-EQ}$$

By the same reasoning as above, $\Theta; \Gamma \vdash h : A$. By reflexivity of equality, $\Theta; \Gamma \vdash h \equiv h : A$.

c) DELTA-META:

$$\frac{\Sigma; \Gamma \vdash \alpha\, \vec{e} : T \qquad \Sigma; \Gamma \vdash t' : T \qquad \alpha := t : A \in \Sigma \quad t \mathbin{@} \vec{e} \Downarrow t'}{\Sigma; \Gamma \vdash \alpha\, \vec{e} \equiv t' : T}\ \text{DELTA-META}$$

By the assumption, Θ is compatible with $\alpha := t : A$; therefore, $\Theta; \cdot \vdash \alpha \equiv t : A$. By Lemma 2.62 (context weakening), $\Theta; \Gamma \vdash \alpha \equiv t : A$. By definition, $(\alpha \mathbin{@} \vec{e}) \Downarrow \alpha\, \vec{e}$. By the assumption, $(t \mathbin{@} \vec{e}) \Downarrow t'$. By Lemma 2.79 (typing and congruence of elimination), $\Theta; \Gamma \vdash t' : T$ and $\Theta; \Gamma \vdash \alpha\, \vec{e} \equiv t' : T$.

□

Remark 2.131 (Compatibility of extended metasubstitutions with declarations). Let Σ **sig** be a well-formed signature, and Θ **wf** a well-formed metasubstitution such that $\Theta \vDash \Sigma$. Let Θ' be a metasubstitution such that Θ' **wf**, and $\Theta \subseteq \Theta'$. Then, for every $D \in \Sigma$, Θ' is compatible with D.

Proof. By the assumption, $\Theta \vDash \Sigma$. By Lemma 2.130, therefore, Θ is compatible with D. Because $\Theta \subseteq \Theta'$, by Remark 2.128 (compatibility with a declaration as a judgment), and Lemma 2.69 (signature weakening), Θ' is compatible with D. □

Definition 2.132 (Restriction of a metasubstitution to a set of metavariables)**.** The restriction of Θ to a set S (written Θ_S) is a metasubstitution which assigns the same metavariable values as Θ, but only to those metavariables in S.

$$
\begin{array}{lcll}
(\cdot)_S & = & \cdot & \\
(\Theta, \mathbb{a} : A)_S & = & \Theta_S, \mathbb{a} : A & \\
(\Theta, \alpha := t : A)_S & = & \Theta_S, \alpha := t : A & \textbf{if } \alpha \in S \\
(\Theta, \alpha := t : A)_S & = & \Theta_S & \textbf{otherwise}
\end{array}
$$

We overload the notation so that, when restricting metasubstitutions, signatures stand for the set of metavariables they declare ($\Theta_\Sigma = \Theta_{\text{SUPPORT}(\Sigma)}$), and terms stand for the set of metavariables they contain ($\Theta_t = \Theta_{\text{METAS}(t)}$). The union of signatures, terms and sets stands for the union of the corresponding sets (e.g. $\Theta_{\Sigma \cup t} = \Theta_{\text{SUPPORT}(\Sigma) \cup \text{METAS}(t)}$).

Remark 2.133 (Restriction to a compatible signature). Whenever $\text{SUPPORT}(\Theta) = \text{SUPPORT}(\Sigma)$ (for instance, because $\Theta \vDash \Sigma$), then we have $\Theta_\Sigma = \Theta$.

Proof. If $\Theta \vDash \Sigma$, by Definition 2.125 (compatible metasubstitution), $\text{DECLS}(\Theta) = \text{DECLS}(\Sigma)$, which by Remark 2.9 (atoms and metavariables are disjoint), gives $\text{SUPPORT}(\Theta) = \text{SUPPORT}(\Sigma)$.

The property follows by induction on Θ, using Definition 2.6 (support of a signature) and Definition 2.132 (restriction of a metasubstitution to a set of metavariables). □

Remark 2.134 (Subsumption of restriction). For any well-formed metasubstitution Θ **wf** and set of metavariables S, Θ_S **wf** and $\Theta_S \subseteq \Theta$. In particular, for any signature Σ, Θ_Σ **wf** and $\Theta_\Sigma \subseteq \Theta$.

Proof. By Postulate 12 (signature strengthening), and the fact that, by Definition 2.122 (well-formed metasubstitution), terms in a well-formed metasubstitution do not contain metavariables. □

Remark 2.135 (Declarations in a metasubstitution restriction). Given a metasubstitution Θ and a set S, $\text{ATOMDECLS}(\Theta) = \text{ATOMDECLS}(\Theta_S)$ and $\text{SUPPORT}(\Theta_S) = \text{SUPPORT}(\Theta) \cap S$. In particular, given a signature Σ, $\text{ATOMDECLS}(\Theta) = \text{ATOMDECLS}(\Theta_\Sigma)$ and $\text{SUPPORT}(\Theta_\Sigma) = \text{SUPPORT}(\Theta) \cap \text{SUPPORT}(\Sigma)$.

Remark 2.136 (Nested metasubstitution restriction). Let Θ be a metasubstitution, and S and S' sets of metavariables such that $S \subseteq S'$. Then $(\Theta_{S'})_S = \Theta_S$.

Remark 2.137 (Metasubstitution weakening). Let Θ, Θ' be metasubstitutions such that $\Theta \subseteq \Theta'$, and J a judgment. (For instance, if $\Theta = \Theta'_\Sigma$.) If $\Theta \vdash J$, then $\Theta' \vdash J$.

Proof. By Remark 2.123 (metasubstitutions are signatures) and Lemma 2.69 (signature weakening). □

Remark 2.138 (Metasubstitution strengthening). Assume $\Theta\,\mathbf{wf}$, $\Theta \subseteq \Theta'$.
Let J be a judgment. If $\Theta' \vdash J$ and $\textsc{consts}(J) \subseteq \textsc{decls}(\Theta)$, then $\Theta \vdash J$.

Proof. By Remark 2.123 (metasubstitutions are signatures) and Postulate 12 (signature strengthening). □

2.18 Closing metasubstitution ($\textsc{close}(\Sigma)$)

Definition 2.139 (Closed signature)**.** Let Σ be a signature. We say that Σ is closed if it assigns a term to every metavariable it declares. In other words, there are no α and A such that $\alpha : A \in \Sigma$.

Definition 2.140 (Normalization to meta-free terms: $\Sigma \vdash t \mathrel{\hat{\searrow}} u$)**.** Given a closed signature Σ and a term $\Sigma; \Gamma \vdash t : A$, we say that u is the metavariable-free normal form of term t (written $\Sigma \vdash t \mathrel{\hat{\searrow}} u$) if u is the result of replacing all metavariables occurring in t by their bodies given in Σ. (See Figure 2.8.)

Lemma 2.141 (Existence of meta-free normal form)**.** *Given a closed signature Σ, and a term $\Sigma; \Gamma \vdash t : A$, there exists a unique term u such that $\Sigma \vdash t \mathrel{\hat{\searrow}} u$. For this u, we have $\textsc{metas}(u) = \emptyset$, $\textsc{consts}(u) \subseteq \textsc{consts}(t)$, and $\Sigma; \Gamma \vdash t \equiv u : A$.*

Proof. By induction on the structure of t and Definition 2.140 (normalization to meta-free terms), we have $\Sigma \vdash t \mathrel{\hat{\searrow}} u$ with $\textsc{consts}(u) \subseteq \textsc{consts}(t)$ and $\textsc{metas}(u) = \emptyset$. The induction is well-founded because the cases of $\mathrel{\hat{\searrow}}$ correspond to reduction rules, and Postulate 8 prevents infinite chains of reductions. The relation is deterministic because all the cases are disjoint, and each case only depends (inductively) on deterministic relations. Note that, if $\Sigma \vdash t \mathrel{\hat{\searrow}} u$, then $\Sigma; \Gamma \vdash t \longrightarrow^\star_{\delta\eta} u : A$. Therefore, $\Sigma; \Gamma \vdash t \equiv u : A$. □

Remark 2.142 (Metavariable-free term). Let $\Sigma\ \mathbf{sig}$ be a closed signature. By Lemma 2.141 (existence of meta-free normal form), if $\Sigma; \cdot \vdash t : A$, then there are unique t' and A' such that $\Sigma \vdash t \mathrel{\hat{\searrow}} t'$, $\Sigma \vdash A \mathrel{\hat{\searrow}} A'$, $\textsc{metas}(A') = \textsc{metas}(t') = \emptyset$, $\Sigma; \cdot \vdash A' \equiv A\ \mathbf{type}$, $\Sigma; \cdot \vdash t' \equiv t : A'$. Also, by Remark 2.15 (there is only set), if $\Sigma; \cdot \vdash A\ \mathbf{type}$, there is A' such that $\Sigma \vdash A \mathrel{\hat{\searrow}} A'$, $\Sigma; \cdot \vdash A\ \mathbf{type}$, and $\Sigma; \cdot \vdash A' \equiv A\ \mathbf{type}$.

Remark. If $\textsc{metas}(t) = \emptyset$, then for any signature Σ, $\Sigma \vdash t \mathrel{\hat{\searrow}} t$.

Definition 2.143 (Closing metasubstitution: $\textsc{close}(\Sigma) \Downarrow \Theta$)**.** Given a closed signature Σ, whether a metasubstitution Θ is a closing metasubstitution for Σ (written $\textsc{close}(\Sigma) \Downarrow \Theta$) is inductively defined as follows:

$$
\begin{array}{rcll}
\Sigma \vdash \mathrm{Bool} & \hat{\searrow} & \mathrm{Bool} & \\
\Sigma \vdash \Pi A B & \hat{\searrow} & \Pi A' B' & \textbf{if} \quad \Sigma \vdash A \hat{\searrow} A' \\
 & & & \textbf{and} \quad \Sigma \vdash B \hat{\searrow} B' \\
 & & & \\
\Sigma \vdash \Sigma A B & \hat{\searrow} & \Sigma A' B' & \textbf{if} \quad \Sigma \vdash A \hat{\searrow} A' \\
 & & & \textbf{and} \quad \Sigma \vdash B \hat{\searrow} B' \\
 & & & \\
\Sigma \vdash \mathrm{Set} & \hat{\searrow} & \mathrm{Set} & \\
\Sigma \vdash c & \hat{\searrow} & c & \\
\Sigma \vdash \lambda.t & \hat{\searrow} & \lambda.t' & \textbf{if} \quad \Sigma \vdash t \hat{\searrow} t' \\
 & & & \\
\Sigma \vdash \langle t, u \rangle & \hat{\searrow} & \langle t', u' \rangle & \textbf{if} \quad \Sigma \vdash t \hat{\searrow} t' \\
 & & & \textbf{and} \quad \Sigma \vdash u \hat{\searrow} u' \\
 & & & \\
\Sigma \vdash \alpha\, \vec{e}^{\,n} & \hat{\searrow} & v & \textbf{if} \quad \alpha := t : A \in \Sigma \\
 & & & \textbf{and} \quad t \mathbin{@} \overrightarrow{e'} \Downarrow v' \\
 & & & \textbf{and} \quad \Sigma \vdash v' \hat{\searrow} v \\
 & & & \\
\Sigma \vdash h\, \vec{e}^{\,n} & \hat{\searrow} & h\, \overrightarrow{e'} & \textbf{if} \quad (h = x \ \textbf{or} \ h = \mathbb{0} \ \textbf{or} \ h = \mathrm{if}) \\
 & & & \textbf{and} \quad \forall i \in \{1, ..., n\}. \\
 & & & \qquad e'_i := e_i = .\pi_1 \\
 & & & \qquad \textbf{or} \ e'_i := e_i = .\pi_2 \\
 & & & \qquad \textbf{or} \ (e_i = t_i \\
 & & & \qquad\quad \textbf{and} \ \Sigma \vdash t_i \hat{\searrow} u_i \\
 & & & \qquad\quad \textbf{and} \ e'_i := u_i)
\end{array}
$$

Figure 2.8: Inductive definition of the meta-free normal form of a term

$$
\begin{array}{lll}
\textsc{close}(\cdot) \Downarrow \cdot & & \\
\textsc{close}(\Sigma, \mathbb{c} : A) \Downarrow (\Theta, \mathbb{c} : A') & \textbf{if} & \textsc{close}(\Sigma) \Downarrow \Theta \\
 & \textbf{and} & \Sigma \vdash A \mathrel{\hat{\searrow}} A' \\
\textsc{close}(\Sigma, \alpha := t : A) \Downarrow (\Theta, \alpha := t' : A') & \textbf{if} & \textsc{close}(\Sigma) \Downarrow \Theta \\
 & \textbf{and} & \Sigma \vdash A \mathrel{\hat{\searrow}} A' \\
 & \textbf{and} & \Sigma \vdash t \mathrel{\hat{\searrow}} t'
\end{array}
$$

Lemma 2.144 (Compatibility of closing metasubstitution)**.** *If* Σ **sig** *is closed, then there is a unique metasubstitution* Θ *such that* $\textsc{close}(\Sigma) \Downarrow \Theta$, Θ **wf** *and* $\Theta \vDash \Sigma$.

Proof. By induction on Σ. In all cases, we use Lemma 2.130 (alternative characterization of a compatible metasubstitution) to show $\Theta \vDash \Sigma$.

i) $\Sigma = \cdot$: Take $\Theta = \cdot$. Then $\textsc{close}(\cdot) \Downarrow \cdot$, Θ **wf** and $\Theta \vDash \Sigma$.

ii) $\Sigma', \mathbb{c} : A$: By the induction hypothesis, there is Θ' such that $\textsc{close}(\Sigma') \Downarrow \Theta'$, Θ' **wf** and $\Theta' \vDash \Sigma'$.

Because Σ **sig**, we have $\Sigma' \vdash A$ **type**. By Remark 2.142 (metavariable-free term), there exists a unique A' such that $\Sigma' \vdash A \mathrel{\hat{\searrow}} A'$, $\Sigma'; \cdot \vdash A'$ **type**, $\Sigma'; \cdot \vdash A' \equiv A$ **type**.

Take $\Theta = \Theta, \mathbb{c} : A'$ Because $\Theta' \vDash \Sigma'$, $\Theta'; \cdot \vdash A'$ **type**. Therefore Θ **wf**.

Because $\Theta' \vDash \Sigma'$, for any declaration $D \in \Sigma'$, Θ' is compatible with D. By Remark 2.128 (compatibility with a declaration as a judgment) and Lemma 2.69 (signature weakening), Θ is compatible with D. Because $\Theta' \vDash \Sigma'$ and $\Sigma'; \cdot \vdash A' \equiv A$ **type**, we have $\Theta'; \cdot \vdash A' \equiv A$ **type**. By Lemma 2.69, $\Theta; \cdot \vdash A' \equiv A$ **type**. By $\textsc{meta}_2$, HEAD and CONV rules, $\Theta; \cdot \vdash \mathbb{c} : A$.

iii) $\Sigma', \alpha := t : A$: By the induction hypothesis, there is Θ' such that $\textsc{close}(\Sigma') \Downarrow \Theta'$, Θ' **wf** and $\Theta' \vDash \Sigma'$.

Because Σ **sig**, we have $\Sigma' \vdash t : A$. By Remark 2.142 (metavariable-free term), there exist unique t' and A' such that $\Sigma' \vdash t \mathrel{\hat{\searrow}} t'$, $\Sigma' \vdash A \mathrel{\hat{\searrow}} A'$, $\Sigma'; \cdot \vdash t' \equiv t : A'$, $\Sigma'; \cdot \vdash A' \equiv A$ **type**. Let $\Theta = \Theta', \alpha : t' : A'$. It suffices to show that $\Theta; \cdot \vdash \alpha \equiv t : A$, which follows analogously to the previous case.

□

2.19 Equality of metasubstitutions ($\Theta_1 \equiv \Theta_2$)

Definition 2.145 (Equality of metasubstitutions: $\Theta \equiv \Theta'$)**.** We say that metasubstitutions Θ and Θ' are equal (written $\Theta \equiv \Theta'$) if Θ **wf**, Θ' **wf**, $\textsc{decls}(\Theta) = \textsc{decls}(\Theta')$, and each declaration in Θ is judgmentally equal to a corresponding declaration in Θ' (and vice versa).

More precisely, it is the transitive closure of the following relation:

$$
\begin{array}{rcll}
\Theta_1, \mathbb{u} : A, \Theta_2 & \equiv & \Theta_1, \mathbb{u} : A', \Theta_2 & \textbf{if} \quad \Theta_1; \cdot \vdash A \equiv A' \textbf{ type} \\
\Theta_1, \alpha := t : A, \Theta_2 & \equiv & \Theta_1, \alpha := t : A', \Theta_2 & \textbf{if} \quad \Theta_1; \cdot \vdash A \equiv A' \textbf{ type} \\
\Theta_1, \alpha := t : A, \Theta_2 & \equiv & \Theta_1, \alpha := t' : A, \Theta_2 & \textbf{if} \quad \Theta_1; \cdot \vdash t \equiv t' : A \\
\Theta & \equiv & \Theta' & \textbf{if } \Theta' \text{ is a well-formed reordering of } \Theta
\end{array}
$$

Lemma 2.146 (Metasubstitution equality is an equivalence relation)**.**

Proof. Transitivity and reflexivity follow by definition. Symmetry follows by Lemma 2.54 (term equality is an equivalence relation). □

Lemma 2.147 (Compatibility respects equality)**.** *If* $\Theta_1 \equiv \Theta_2$ *and* $\Theta_1 \vDash \Sigma$*, then* $\Theta_2 \vDash \Sigma$*.*

Proof. By induction on the proof for $\Theta_1 \equiv \Theta_2$, using Lemma 2.69 (signature weakening) and Lemma 2.130 (alternative characterization of a compatible metasubstitution). □

Lemma 2.148 (Uniqueness of closing metasubstitution)**.** *Let* Σ *be a closed signature, and* Θ_1*,* Θ_2 *metasubstitutions such that* $\Theta_1 \vDash \Sigma$*,* $\Theta_2 \vDash \Sigma$*. Then* $\Theta_1 \equiv \Theta_2$*.*

Proof. We show the following stronger property:

Given metasubstitutions Θ^0, Θ^1, Θ such that (Θ^0, Θ^1) **wf**, (Θ^0, Θ) **wf**, $\Theta^0, \Theta^1 \vDash \Sigma$ and $\textsc{close}(\Sigma) \Downarrow (\Theta^0, \Theta)$, we have $\Theta^0, \Theta^1 \equiv \Theta^0, \Theta$.

Note that, if $\textsc{close}(\Sigma) \Downarrow (\Theta^0, \Theta)$, then we have $\Sigma = \Sigma^0, \Sigma'$ and $\textsc{close}(\Sigma_0) \Downarrow (\Theta^0)$.

We proceed by induction on the length of Θ.

- Case $\Theta = \cdot$: Because $\textsc{decls}(\Theta^0, \Theta^1) = \textsc{decls}(\Sigma) = \textsc{decls}(\Theta^0, \Theta)$, then $\Theta^1 = \cdot$. By reflexivity, $\Theta^0 \equiv \Theta^0$.

- Case $\Theta = \alpha := t^0 : A^0, \Theta'$:

  Because $\textsc{close}(\Sigma) \Downarrow (\Theta^0, \Theta)$, we have $\Sigma = \Sigma^0, \alpha := t : A, \Sigma'$ and $\textsc{close}(\Sigma^0) \Downarrow \Theta^0$.

  By Lemma 2.130 (alternative characterization of a compatible metasubstitution), Θ^0, Θ is compatible with $\alpha := t : A \in \Sigma$. Therefore, $\Theta^0, \Theta; \cdot \vdash A^0 \equiv A$ **type**, and $\Theta^0, \Theta; \cdot \vdash t^0 \equiv t : A^0$.

  Because Θ^0, Θ is well-formed, by Lemma 2.72 (no extraneous constants) $\textsc{consts}(A^0) \subseteq \textsc{decls}(\Theta^0)$ and $\textsc{consts}(t^0) \subseteq \textsc{decls}(\Theta^0)$. Because $\Sigma^0, \alpha := t : A$ **sig**, by Lemma 2.72, we also have $\textsc{consts}(A) \subseteq \textsc{decls}(\Theta^0)$ and $\textsc{consts}(t) \subseteq \textsc{decls}(t^0)$. By Postulate 12 (signature strengthening), $\Theta^0; \cdot \vdash A^0 \equiv A$ **type** and $\Theta^0; \cdot \vdash t^0 \equiv t : A^0$.

  Because $\Theta^0, \Theta^1 \vDash \Sigma$, we have $\alpha := t^1 : A^1 \in \Theta^0, \Theta^1$. In fact, from Θ^0, Θ **wf**, we know that α is fresh in Θ^0, so $\alpha := t^1 : A^1 \in \Theta^1$. Because $\Theta^0, \Theta^1 \vDash \Sigma$, we have that $\Theta^0, \Theta^1; \cdot \vdash A^1 \equiv A$ **type**, and $\Theta^0, \Theta^1; \cdot \vdash t^1 \equiv t : A^1$. Let Θ_1^1 and Θ_2^1 be such that $\Theta^1 = \Theta_1^1, \alpha := t^1 : A^1, \Theta_2^1$. Because Θ^0, Θ^1 **wf**, we have $\textsc{consts}(A^1) \subseteq \textsc{decls}(\Theta^0, \Theta_1^1)$ and $\textsc{consts}(t^1) \subseteq \textsc{decls}(\Theta^0, \Theta_1^1)$. As shown above, $\textsc{consts}(A) \subseteq \textsc{decls}(\Theta^0) \subseteq \textsc{decls}(\Theta^0, \Theta_1^1)$. and

$\textsc{consts}(t) \subseteq \textsc{decls}(\Theta^0) \subseteq \textsc{decls}(\Theta^0, \Theta_1^1)$ By Postulate 12 (signature strengthening), $\Theta^0, \Theta_1^1; \cdot \vdash A^1 \equiv A$ **type** and $\Theta^0, \Theta_1^1; \cdot \vdash t^1 \equiv t : A^1$.

By Lemma 2.69 (signature weakening), $\Theta^0, \Theta_1^1; \cdot \vdash A^0 \equiv A$ **type**, and $\Theta^0, \Theta_1^1; \cdot \vdash t^0 \equiv t : A^0$. By Lemma 2.146 (metasubstitution equality is an equivalence relation) $\Theta^0, \Theta_1^1; \cdot \vdash A^1 \equiv A^0$ **type**. This means that $\Theta^0, \Theta^1 = \Theta^0, \Theta_1^1, \alpha := t^1 : A^1, \Theta_2^1 \equiv \Theta^0, \Theta_1^1, \alpha := t^1 : A^0, \Theta_2^1$. By the CONV-EQ rule, $\Theta^0, \Theta_1^1; \cdot \vdash t^1 \equiv t : A^0$. By transitivity and symmetry of equality, $\Theta^0, \Theta_1^1; \cdot \vdash t^1 \equiv t^0 : A^0$. Therefore, $\Theta^0, \Theta_1^1, \alpha := t^1 : A^0, \Theta_2^1 \equiv \Theta^0, \Theta_1^1, \alpha := t^0 : A^0, \Theta_2^1$.

Because $\textsc{consts}(\mathrm{A}^0) \subseteq \textsc{decls}(\Theta^0)$, and $\textsc{consts}(t^0) \subseteq \textsc{decls}(\Theta^0)$, $\Theta^0, \alpha := t^0 : A^0, \Theta_1^1, \Theta_2^1$ is a well-formed reordering of $\Theta^0, \Theta_1^1, \alpha := t^0 : A^0, \Theta_2^1$. Therefore, $(\Theta^0, \Theta_1^1, \alpha := t^0 : A^0, \Theta_2^1) \equiv (\Theta^0, \alpha := t^0 : A^0, \Theta_1^1, \Theta_2^1)$.

Because $\Theta^0, \alpha := t^0 : A^0$ **wf**, one can show by induction on the derivation for $\Theta^0, \Theta_1^1, \alpha := t^0 : A^0, \Theta_2^1$ **wf** that $\Theta^0, \Theta_1^1, \alpha := t^0 : A^0, \Theta_2^1$ **wf** (use Lemma 2.69 (signature weakening) for the declarations in Θ_1^1, and the fact that typing judgments do not rely on the order of declarations in the signature, only on their well-formedness).

By transitivity $\Theta^0, \Theta^1 \equiv \Theta^0, \alpha := t^0 : A^0, \Theta_1^1, \Theta_2^1$. By Lemma 2.147 (compatibility respects equality), $\Theta^0, \alpha := t^0 : A^0, \Theta_1^1, \Theta_2^1 \vDash \Sigma$.

By applying the induction hypothesis to $\Theta^0, \alpha := t^0 : A^0$, Θ_1^1, Θ_2^1 and Θ' we have $((\Theta^0, \alpha := t^0 : A^0), \Theta_1^1, \Theta_2^1) \equiv ((\Theta^0, \alpha := t^0 : A^0), \Theta') = \Theta^0, \Theta$.

By Lemma 2.146 (metasubstitution equality is an equivalence relation), $\Theta^0, \Theta^1 \equiv \Theta^0, \Theta$.

- Case $\Theta = \mathbb{c} : A^0, \Theta'$:

  Because $\textsc{close}(\Sigma) \Downarrow (\Theta^0, \Theta)$, we have $\Sigma = \Sigma^0, \mathbb{c} : A, \Sigma'$, $\textsc{close}(\Sigma^0) \Downarrow \Theta^0$.

  By Lemma 2.144 (compatibility of closing metasubstitution), $\Theta^0, \Theta \vDash \Sigma$. By Lemma 2.130 (alternative characterization of a compatible metasubstitution), Θ^0, Θ is compatible with $\mathbb{c} : A \in \Sigma$. Therefore, $\Theta^0, \Theta; \cdot \vdash A^0 \equiv A$ **type**, and $\Theta^0, \Theta; \cdot \vdash t^0 \equiv t : A^0$.

  Because $\Theta^0, \mathbb{c} : A^0$ **wf** is well-formed, by Lemma 2.72 (no extraneous constants), $\textsc{consts}(A^0) \subseteq \textsc{decls}(\Theta^0)$. By the definition of $\textsc{close}(\Sigma)$, we have that $\Sigma_0; \cdot \vdash A$ **type**. By Lemma 2.72 (no extraneous constants), we also have $\textsc{consts}(A) \subseteq \textsc{decls}(\Sigma^0) = \textsc{decls}(\Theta^0)$. By Postulate 12 (signature strengthening), $\Theta^0; \cdot \vdash A^0 \equiv A$ **type**.

  Because $\Theta^0, \Theta^1 \vDash \Sigma$, we have $\mathbb{c} : A^1 \in \Theta^0, \Theta^1$. In fact, from Θ^0, Θ **wf**, we know that $\mathbb{c}$ is fresh in Θ^0, so $\mathbb{c} : A^1 \in \Theta^1$. Because $\Theta^0, \Theta^1 \vDash \Sigma$, we have that $\Theta^0, \Theta^1; \cdot \vdash A^1 \equiv A$ **type**. More specifically, $\Theta^1 = \Theta_1^1, \mathbb{c} : A^1, \Theta_2^1$. Because Θ^0, Θ^1 **wf**, we have $\textsc{consts}(A^1) \subseteq \textsc{decls}(\Theta^0, \Theta_1^1)$. As we explained above, $\textsc{consts}(A) \subseteq \textsc{decls}(\Theta^0) \subseteq \textsc{decls}(\Theta^0, \Theta_1^1)$. By Postulate 12 (signature strengthening), $\Theta^0, \Theta_1^1; \cdot \vdash A^1 \equiv A$ **type**.

  By Lemma 2.69 (signature weakening), $\Theta^0, \Theta_1^1; \cdot \vdash A^0 \equiv A$ **type**. By transitivity and symmetry, $\Theta^0, \Theta_1^1; \cdot \vdash A^1 \equiv A^0$ **type**. This means that $\Theta^0, \Theta^1 = \Theta^0, \Theta_1^1, \mathbb{c} : A^1, \Theta_2^1 \equiv \Theta^0, \Theta_1^1, \mathbb{c} : A^0, \Theta_2^1$.

Because $\textsc{consts}(\mathrm{A}^0) \subseteq \textsc{decls}(\Theta^0)$, we have that $\Theta^0, \mathbb{a} : A^0, \Theta_1^1, \Theta_2^1$ is a well-formed reordering of $\Theta^0, \Theta_1^1, \mathbb{a} : A^0, \Theta_2^1$. Therefore, $(\Theta^0, \Theta_1^1, \mathbb{a} : A^0, \Theta_2^1) \equiv (\Theta^0, \mathbb{a} : A^0, \Theta_1^1, \Theta_2^1)$.

By applying the induction hypothesis to $\Theta^0, \mathbb{a} : A^0$, Θ_1^1, Θ_2^1 and Θ' we have $(\Theta^0, \mathbb{a} : A^0), \Theta_1^1, \Theta_2^1) \equiv (\Theta^0, \mathbb{a} : A^0), \Theta') = \Theta$ (the required well-formedness and compatibility conditions are shown analogously to the previous case).

By Lemma 2.146 (metasubstitution equality is an equivalence relation), $\Theta^0, \Theta^1 \equiv \Theta^0, \Theta$.

Because Σ **sig** and Σ is closed, by Lemma 2.144 (compatibility of closing metasubstitution), there is Θ such that $\textsc{close}(\Sigma) \Downarrow \Theta$. By taking $\Theta^0 := \cdot$ and $\Theta^1 := \Theta_1$, we obtain $\Theta^1 \equiv \Theta$. By taking $\Theta^0 := \cdot$ and $\Theta^1 := \Theta_2$, we obtain $\Theta_2 \equiv \Theta$. By Lemma 2.146 (metasubstitution equality is an equivalence relation), $\Theta_1 \equiv \Theta_2$. □

Corollary 2.149 (Solution to closed signature)**.** *Let Σ be a closed signature. Then $\textsc{close}(\Sigma) \Downarrow \Theta$, $\Theta \vDash \Sigma$, and, for any other Θ' such that $\Theta' \vDash \Sigma$, we have $\Theta \equiv \Theta'$.*

Proof. By Lemma 2.144 (compatibility of closing metasubstitution) and Lemma 2.148 (uniqueness of closing metasubstitution). □

Lemma 2.150 (Equality of restricted metasubstitutions)**.** *If $\Theta^1 \equiv \Theta^2$, then $(\Theta^1)_\Sigma$* **wf***, $(\Theta^2)_\Sigma$* **wf** *and $(\Theta^1)_\Sigma \equiv (\Theta^2)_\Sigma$.*

Proof. By induction on the derivations of Θ^1 **wf**, Θ^2 **wf**, $\Theta^1 \equiv \Theta^2$, using Postulate 12 (signature strengthening). □

2.20 Signature extensions ($\Sigma \sqsubseteq \Sigma'$)

During type checking, the initial signature may be extended with new metavariables, and existing metavariables may be instantiated. The end goal is to obtain an *extension* of the original signature in which all metavariables are instantiated (i.e. a closed signature) and in which the terms provided by the user are well-typed.

We say that Σ' is an extension of Σ (written $\Sigma \sqsubseteq \Sigma'$) if Σ' contains all the declarations in Σ. The signature Σ' may instantiate some metavariables that are not already instantiated in Σ, and/or replace some types and terms in Σ by equal ones; but Σ' must not declare any new atoms (i.e. $\textsc{AtomDecls}(\Sigma) = \textsc{AtomDecls}(\Sigma')$). More formally:

Definition 2.151 (Signature extension: $\Sigma \sqsubseteq \Sigma'$)**.** Consider the signatures Σ and Σ'. We say that Σ' extends Σ (written $\Sigma' \sqsupseteq \Sigma$ or $\Sigma \sqsubseteq \Sigma'$), if Σ **sig**, Σ' **sig**, and it does so inductively in any of the following cases:

Declarations

$$\Sigma_1, \Sigma_2 \sqsubseteq \Sigma_1, \alpha : A, \Sigma_2$$
$$\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := t : A, \Sigma_2$$

Composition

$$\Sigma_1 \sqsubseteq \Sigma_3 \quad \textbf{if} \quad \Sigma_1 \sqsubseteq \Sigma_2 \textbf{ and } \Sigma_2 \sqsubseteq \Sigma_3$$

Normalization

$$\Sigma_1, \mathbb{a} : A, \Sigma_2 \sqsubseteq \Sigma_1, \mathbb{a} : A', \Sigma_2 \quad \textbf{if} \quad \Sigma_1; \cdot \vdash A \equiv A' \textbf{ type}$$
$$\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha : A', \Sigma_2 \quad \textbf{if} \quad \Sigma_1; \cdot \vdash A \equiv A' \textbf{ type}$$
$$\Sigma_1, \alpha := t : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := t : A', \Sigma_2 \quad \textbf{if} \quad \Sigma_1; \cdot \vdash A \equiv A' \textbf{ type}$$
$$\Sigma_1, \alpha := t : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := t' : A, \Sigma_2 \quad \textbf{if} \quad \Sigma_1; \cdot \vdash t \equiv t' : A$$

Permutation

$$\Sigma \sqsubseteq \Sigma' \quad \textbf{if } \Sigma' \text{ is a well-formed reordering of } \Sigma$$

Remark 2.152 (Signature extension is reflexive and transitive). The relation $- \sqsubseteq -$ is reflexive and transitive.

Remark 2.153 (Signature extension declarations). If $\Sigma \sqsubseteq \Sigma'$, then $\textsc{AtomDecls}(\Sigma) = \textsc{AtomDecls}(\Sigma')$ and $\textsc{support}(\Sigma) \subseteq \textsc{support}(\Sigma')$.

Remark 2.154 (Metasubstitution restriction to extension). If $\Sigma \sqsubseteq \Sigma'$, then by Remark 2.153 (signature extension declarations), $\textsc{support}(\Sigma) \subseteq \textsc{support}(\Sigma')$. Therefore, by Remark 2.136 $(\Theta_{\Sigma'})_\Sigma = \Theta_\Sigma$.

The key insight is that extending the signature preserves all relevant properties about contexts, terms, types and constraints.

Lemma 2.155 (Preservation of judgments under signature extensions)**.** *Let $\Sigma \sqsubseteq \Sigma'$, and J be a judgment. If $\Sigma \vdash J$, then $\Sigma' \vdash J$.*

Proof. By induction on the derivation of $\Sigma \sqsubseteq \Sigma'$.

- Rules for *declarations* are proven by induction on J, using the same rules as in the original derivation of J.
- The rule for *composition* is handled inductively.
- For the *normalization* rules, we proceed by induction on the derivation of the judgment, using CONV, CONV-EQ and laws of equality as appropriate, similarly to the proof of Lemma 2.130 (alternative characterization of a compatible metasubstitution).
- The rule for *permutation* is a special case of Lemma 2.69 (signature weakening).

□

Corollary 2.156 (Horizontal composition of extensions)**.** *Let $\Sigma = \Sigma_1, \Sigma_2$, Σ* **sig***, and Σ_1' such that $\Sigma_1 \sqsubseteq \Sigma_1'$ (in particular, Σ_1'* **sig***). Also, $\textsc{decls}(\Sigma_1') \cap \textsc{decls}(\Sigma_2) = \emptyset$. (that is, all the new declarations in Σ_1' are fresh in Σ_2). Then $\Sigma_1, \Sigma_2 \sqsubseteq \Sigma_1', \Sigma_2$ (in particular, Σ_1', Σ_2* **sig***).*

Proof. By induction on the length of Σ_2. Use Lemma 2.155 (preservation of judgments under signature extensions) to prove the well-formedness of each declaration in Σ_2. □

Lemma 2.157 (Restriction of a metasubstitution to an extended signature)**.** *Let Θ be a metasubstitution, and Σ and Σ' signatures such that $\Sigma \sqsubseteq \Sigma'$ and $\Theta \vDash \Sigma'$. Then Θ_Σ **wf** and $\Theta_\Sigma \vDash \Sigma$.*

Proof. By Remark 2.134 (subsumption of restriction), Θ_Σ **wf**. By Definition 2.125 (compatible metasubstitution), it suffices to show that $\textsc{decls}(\Theta_\Sigma) = \textsc{decls}(\Sigma)$, and that for every signature judgment J, if $\Sigma \vdash J$ then $\Theta_\Sigma \vdash J$.

$$
\begin{array}{rcl}
\textsc{decls}(\Theta_\Sigma) & = & \text{Definition 2.8} \\
\textsc{AtomDecls}(\Theta_\Sigma) \cup \textsc{support}(\Theta_\Sigma) & = & \text{Remark 2.135} \\
\textsc{AtomDecls}(\Theta) \cup (\textsc{support}(\Theta) \cap \textsc{support}(\Sigma)) & = & \Theta \vDash \Sigma', \text{ Remark 2.9} \\
\textsc{AtomDecls}(\Sigma') \cup (\textsc{support}(\Sigma') \cap \textsc{support}(\Sigma)) & = & \text{Remark 2.153} \\
\textsc{AtomDecls}(\Sigma) \cup \textsc{support}(\Sigma) & = & \text{Definition 2.8} \\
\textsc{decls}(\Sigma) & &
\end{array}
$$

Let J be a judgment such that $\Sigma \vdash J$. By Lemma 2.69 (signature weakening), for each judgment J such that $\Sigma \vdash J$, we have $\Sigma' \vdash J$. Because $\Theta \vDash \Sigma'$, we also have $\Theta \vdash J$.

By Lemma 2.72 (no extraneous constants), $\textsc{consts}(J) \subseteq \textsc{decls}(\Sigma) = \textsc{decls}(\Theta_\Sigma)$. By construction, $\Theta_\Sigma \subseteq \Theta$. By Postulate 12 (signature strengthening), $\Theta_\Sigma \vdash J$. □

2.21 Non-reducible terms

A key step in our unification algorithm is simplifying constraints into new constraints involving smaller terms. For instance, one may reduce equating two neutral terms $h\,\vec{e}$ and $h\,\vec{e}'$ to pointwise equating each of the eliminators (e.g. Rule schema 14). In order to ensure that we do not lose solutions, we need to restrict which terms such a transformation may be applied to.

Definition 2.158 (Strongly neutral term)**.** A strongly neutral term is a neutral term of one of the following forms:

- $x\,\vec{e}$.
- $\mathbb{0}\,\vec{e}$.
- if $\vec{e}^{\,n}$, where either $n < 2$, or e_2 is a strongly neutral term.

Remark 2.159 (Prefixes of strongly neutral terms)*.* Prefixes of strongly neutral terms are strongly neutral. That is, if $h\,\vec{e_1}\,\vec{e_2}$ is strongly neutral, then $h\,\vec{e_1}$ is also strongly neutral.

Remark 2.160 (Closure of strongly neutral terms)*.* If f is a strongly neutral term, then:

(i) If ρ is a renaming, then f^ρ is strongly neutral.

(i) If t is a strongly neutral term, then then $f\,t$ is strongly neutral.

(i) If $e = .\pi_1$ or $e = .\pi_2$, and $\Sigma;\Gamma \vdash f\,e : T$ for some type T, then $f\,e$ is strongly neutral.

Proof. From Definition 2.158 (strongly neutral term). For 2.160, observe that, by Lemma 2.56 (neutral inversion) and Lemma 2.75 (uniqueness of typing for neutrals), the terms if $A\,.\pi_1$ and if $A\,.\pi_2$ can never be well typed. □

Remark 2.161 (Intermediate steps of reduction of strong neutrals). Assume $\Sigma;\Gamma \vdash f_0 \longrightarrow_{\delta\eta} t : T$, and $\Sigma;\Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} f_1 : T$, where f_0 is a strongly neutral term. Then t is a neutral term.

Proof. By Definition 2.41 (δη-normalization step), there are only three possible cases for $\Sigma;\Gamma \vdash f_0 \longrightarrow_{\delta\eta} t : T$:

$$(\eta\text{-}\Pi) \qquad \Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} \lambda.f^{(+1)}\,0 : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Pi AB\ \textbf{type}$$

$$(\eta\text{-}\Sigma) \qquad \Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} \langle f\,.\pi_1, f\,.\pi_2\rangle : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash T \equiv \Sigma AB\ \textbf{type}$$

$$(\text{APP}_n) \qquad \Sigma;\Gamma \vdash h\,\vec{e}^{\,n-1}\,u\,\vec{e}' \longrightarrow_{\delta\eta} h\,\vec{e}\,v\,\vec{e}' : T \qquad \begin{array}{ll}\textbf{if} & \Sigma;\Gamma \vdash h\,\vec{e} : \Pi UV \\ \textbf{and} & \Sigma;\Gamma \vdash u \longrightarrow_{\delta\eta} v : U\end{array}$$

However, if t was of the form $\lambda.t_0$ or $\langle t_1, t_2\rangle$, and $\Sigma;\Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} u : T$, then necessarily u is of the form $\lambda.u_0$ or $\langle u_1, u_2\rangle$. But $\Sigma;\Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} f_1 : T$, where f_1 is a neutral term. Therefore, we must be in case APP_n, where t is also a neutral term. □

Remark 2.162 (Reduction preserves strongly neutral terms). If f is strongly neutral, and $\Sigma;\Gamma \vdash f \longrightarrow^{\star}_{\delta\eta} f' : T$, where f' is a neutral term, then f' is also strongly neutral.

Proof. Consider the case with a single step, $\Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} f' : T$. We proceed by induction on the structure of the derivation.

Observe that the only rule that reduces a strongly neutral term to another neutral term is APP_i for some i. If $f = h\,\vec{e}$, then $f' = h\,\overrightarrow{e'}$.

- If $h = x$ or $h = \mathbb{o}$, then f' is strongly neutral.
- If $h =$ if and $i \neq 2$, then f' is strongly neutral.
- If $i = 2$, we have $\Sigma;\Gamma \vdash e_2 \longrightarrow_{\delta\eta} e_2' : U$ for some type U. By the induction hypothesis, e_2' is strongly neutral; therefore, f' is also strongly neutral.

By Remark 2.161, the intermediate steps in the reduction are necessarily neutral terms. Therefore, the general case follows from induction on the number of steps. □

Lemma 2.163 (Injectivity of elimination for strongly neutral terms)**.** *Assume that f and g are strongly neutral terms, with $f = h^1\,\overrightarrow{e^1}^n$ and $g = h^2\,\overrightarrow{e^2}^n$, and $\Sigma;\Gamma \vdash f \equiv g : T$. Then, $h^1 = h^2$, and for each $i \in 1, ..., n$:*

- *If* $e_i^1 = t$ *and* $e_i^2 = u$*, then there are* U *and* V *such that* $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : \Pi UV$ *and* $\Sigma;\Gamma \vdash t \equiv u : U$*.*

- *If* $e_i^1 = e_i^2 = .\pi_1$ *or* $e_i^1 = e_i^2 = .\pi_2$*, then there are* U *and* V *such that* $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : \Sigma UV$*.*

Note that, by Lemma 2.75 (uniqueness of typing for neutrals) and Postulate 10 (injectivity of Π*), the above hold for any* U*,* V *such that* $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : \Pi UV$ *(first case) or* $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^1}_{1,\dots,i-1} : \Sigma UV$ *(second case).*

Proof. By Postulate 14 (existence of a common reduct), there exists v such that $\Sigma;\Gamma \vdash f \longrightarrow^M_{\delta\eta} v : T$ and $\Sigma;\Gamma \vdash g \longrightarrow^N_{\delta\eta} v : T$. We show by induction on the sum of M and N that for all M, N, and for all f, g such that $\Sigma;\Gamma \vdash f \longrightarrow^M_{\delta\eta} v : T$ and $\Sigma;\Gamma \vdash g \longrightarrow^N_{\delta\eta} v : T$, the consequences of the theorem hold.

- $M = N = 0$: Then $f = g$. The result follows by Lemma 2.56 (neutral inversion), and reflexivity.

- $M > 0$ or $N > 0$: We proceed by case analysis on the derivations. Note that, for any f which is strongly neutral, only the reduction rules η-Π, η-Σ, and $\textsc{app}_n$ for some n can be applied to it.

  - $\textsc{app}_j$ for f: $\Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} f' \longrightarrow^{M-1}_{\delta\eta} v : T$, where $f' = h^1\,\overrightarrow{e'^1}$, such that, for some j, $\overrightarrow{e'^1} = \overrightarrow{e^1}_{1,\dots,j-1}\,e'^1_j\,\overrightarrow{e^1}_{j+1,\dots,n}$, with $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,j-1} : \Pi UV$ and $\Sigma;\Gamma \vdash e^1_j \longrightarrow_{\delta\eta} e'^1_j : U$.
    By Remark 2.162 (reduction preserves strongly neutral terms), f' is strongly neutral. There are two possible cases:

    1. $i \neq j$: Then $e^1_i = e'^1_i$. By Lemma 2.86 (equality of $\delta\eta$-reduct) and transitivity of equality, $\Sigma;\Gamma \vdash f' \equiv g : T$, and $\Sigma;\Gamma \vdash f' \longrightarrow^{M-1}_{\delta\eta} v : T$.
       By the induction hypothesis, we have that $h^1 = h^2$, and:
       * If $e^1_i = e'^1_i = t$ and $e^2_i = u$, then there are U, V, T'' such that $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e'^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : T''$, $T'' = \Pi UV$, and $\Sigma;\Gamma \vdash t \equiv u : U$.
       * If $e^1_i = e'^1_i = e^2_i = .\pi_1$ or $e^1_i = e'^1_i = e^2_i = .\pi_2$, then there are U, V, T'' such that $T'' = \Sigma UV$ and $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e'^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : T''$.

       If $i \neq j$, then either $i < j$, or $i > j$:
       * If $i < j$, then $\overrightarrow{e^1}_{1,\dots,i-1} = \overrightarrow{e'^1}_{1,\dots,i-1}$, and the lemma is proven.
       * If $j < i$, then, by Lemma 2.56 (neutral inversion) and the $\textsc{app}_j$ rule, we have $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \longrightarrow_{\delta\eta} h^1\,\overrightarrow{e'^1}_{1,\dots,i-1} : T'$. By Lemma 2.86, this gives $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^1\,\overrightarrow{e'^1}_{1,\dots,i-1} : T'$. By Lemma 2.75, $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^1\,\overrightarrow{e'^1}_{1,\dots,i-1} : T''$, and, by transitivity

$\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : T''$. Then the lemma is proven.

2. $i = j$: Then $\Sigma;\Gamma \vdash t \longrightarrow_{\delta\eta} e_j'^1 : U$, which, by Lemma 2.86 (equality of $\delta\eta$-reduct), gives $\Sigma;\Gamma \vdash t \equiv e_j'^1 : U$.

   By the induction hypothesis, $h^1 = h^2$, $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : \Pi UV$ and $\Sigma;\Gamma \vdash e_j'^1 \equiv u : U$. By transitivity, $\Sigma;\Gamma \vdash t \equiv u : U$.

– APP$_j$ for g: Symmetric to the previous case.

– None of the above apply, the first reduction for f is η-Π:

  By Remark 2.89 (disjointness of primitive types), we cannot have both $\Sigma;\Gamma \vdash T \equiv \Pi AB : \mathrm{Set}$ and $\Sigma;\Gamma \vdash T \equiv \Sigma A'B' : \mathrm{Set}$ for some A, A', B and B'. Then, necessarily, the first reduction for g is also η-Π, $\Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} \lambda.(f(+1))\,0 \longrightarrow_{\delta\eta}^{M-1} \lambda.v' : T$, and $\Sigma;\Gamma \vdash g \longrightarrow_{\delta\eta} \lambda.(g(+1))\,0 \longrightarrow_{\delta\eta}^{N-1} \lambda.v' : T$, where $\Sigma;\Gamma \vdash T \equiv \Pi AB$ **type** for some A and B.

  By Remark 2.91 (inversion of reduction under λ), $\Sigma;\Gamma, A \vdash g^{(+1)}\,0 \longrightarrow_{\delta\eta}^{N-1} v' : B$, and $\Sigma;\Gamma, A \vdash f^{(+1)}\,0 \longrightarrow_{\delta\eta}^{M-1} v' : B$. By Remark 2.160 (closure of strongly neutral terms), if f and g are strongly neutral, then so are $f^{(+1)}\,0$ and $g^{(+1)}\,0$.

  1. $e_i^1 = t$ and $e_i^2 = u$:
     By the induction hypothesis, $(h^1)^{(+1)} = (h^2)^{(+1)}$, (i.e. $h^1 = h^2$), $\Sigma;\Gamma, A \vdash (h^1\,\overrightarrow{e^1}_{1,\dots,i-1})^{(+1)} \equiv (h^2\,\overrightarrow{e^2}_{1,\dots,i-1})^{(+1)} : \Pi UV$, and $\Sigma;\Gamma, A \vdash t^{(+1)} \equiv u^{(+1)} : U$.

     By Lemma 2.56 (neutral inversion), $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} : \Pi U'V'$. By Lemma 2.62 (context weakening), $\Sigma;\Gamma, A \vdash (h^1\,\overrightarrow{e^1}_{1,\dots,i-1})^{(+1)} : (\Pi U'V')^{(+1)}$. By Lemma 2.75 (uniqueness of typing for neutrals), $\Sigma;\Gamma, A \vdash \Pi UV \equiv (\Pi U'V')^{(+1)}$ **type**. By Postulate 10 (injectivity of Π), $\Sigma;\Gamma, A \vdash U \equiv U'^{(+1)}$ **type**. By the CONV-EQ rule, we have $\Sigma;\Gamma, A \vdash (h^1\,\overrightarrow{e^1}_{1,\dots,i-1})^{(+1)} \equiv (h^2\,\overrightarrow{e^2}_{1,\dots,i-1})^{(+1)} : (\Pi U'V')^{(+1)}$, and $\Sigma;\Gamma, A \vdash t^{(+1)} \equiv u^{(+1)} : U'^{(+1)}$.

     By Postulate 13 (context strengthening), we have $\Sigma;\Gamma \vdash (h^1\,\overrightarrow{e^1}_{1,\dots,i-1})^{(+1)(-1)} \equiv (h^2\,\overrightarrow{e^2}_{1,\dots,i-1})^{(+1)(-1)} : (\Pi U'V')^{(+1)(-1)}$, and $\Sigma;\Gamma \vdash t^{(+1)(-1)} \equiv u^{(+1)(-1)} : U'^{(+1)(-1)}$, that is, $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,\dots,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,\dots,i-1} : \Pi U'V'$, and $\Sigma;\Gamma \vdash t \equiv u : U'$.
  2. $e_i^1 = e_i^2 = .\pi_1$ or $e_i^1 = e_i^2 = .\pi_2$. Analogous to the previous case.

– None of the above apply, and the first reduction for f is η-Σ:

  As in the previous case, necessarily, the first reduction for g is η-Σ, $\Sigma;\Gamma \vdash f \longrightarrow_{\delta\eta} \langle f\,.\pi_1, f\,.\pi_2\rangle \longrightarrow_{\delta\eta}^{M-1} \langle v_1', v_2'\rangle : T$, and $\Sigma;\Gamma \vdash g \longrightarrow_{\delta\eta} \langle f\,.\pi_1, f\,.\pi_2\rangle \longrightarrow_{\delta\eta}^{N-1} \langle v_1', v_2'\rangle : T$.

  The proof is similar to the above case, using Remark 2.92 (inversion of reduction under $\langle,\rangle$).

□

Definition 2.164 (Irreducible terms)**.** A strongly neutral term is irreducible if it is in one of the following forms:

- $x\,\vec{e}$.
- $\mathbb{0}\,\vec{e}$.
- if $e_1\,e_2\,\vec{e}$, where e_2 is a strongly neutral term.

Remark 2.165 (Extensions of irreducible terms)*.* If f is an irreducible term, then so is $f\,\vec{e}$.

Lemma 2.166 (Reduction at Π-type)**.** *Assume* $\Sigma;\Gamma \vdash f : \Pi\vec{U}^n V$ *for some* $\vec{U}$*,* V*. Then there is* v *such that* $\Sigma;\Gamma \vdash \lambda\vec{z}^n.f\,\vec{z}^n \longrightarrow^\star_{\delta\eta} \lambda\vec{z}^n.v : \Pi\overrightarrow{U}^n V$*, and* $\Sigma;\Gamma \vdash \lambda\vec{z}^n.v\not\longrightarrow_{\delta\eta} : \Pi\overrightarrow{U}^n V$*.*

Proof. By the ETA-ABS rule, we have $\Sigma;\Gamma \vdash f \equiv \lambda\vec{z}^n.f\,\vec{z}^n : \Pi\overrightarrow{z : U}^n V$. By Postulate 15 (existence of a unique full normal form), there is r such that $\Sigma;\Gamma \vdash \lambda\vec{z}^n.f\,\vec{z}^n \longrightarrow^\star_{\delta\eta} r : \Pi\overrightarrow{U}^n V$, and $\Sigma;\Gamma \vdash r\not\longrightarrow_{\delta\eta} : \Pi\overrightarrow{U} V$. Note that, by Definition 2.41 (δη-normalization step), for any t, if $\Sigma;\Gamma \vdash \lambda\vec{z}^n.t \longrightarrow_{\delta\eta} r_0 : \Pi\overrightarrow{U}^n V$, then only λ applies (recursively, perhaps), so necessarily $r_0 = \lambda\vec{z}^n.t'$ for some t'.

By induction on the derivation, $r = \lambda\vec{z}^n.v$ for some v, and therefore there is v such that $\Sigma;\Gamma \vdash \lambda\vec{z}^n.f\,\vec{z}^n \longrightarrow^\star_{\delta\eta} \lambda\vec{z}^n.v : \Pi\overrightarrow{U}^n V$, and $\Sigma;\Gamma \vdash \lambda\vec{z}^n.v\not\longrightarrow_{\delta\eta} : \Pi\overrightarrow{U} V$. □

Lemma 2.167 (Characterization of normal forms)**.** *Suppose that* $\Theta;\Gamma \vdash v : V$ *and* $\Theta;\Gamma \vdash v\not\longrightarrow_{\delta\eta} : V$*. Then* v *is of the form* v_{nf} *for some* v_{nf} *generated by the following grammar:*

$$
\begin{array}{rcll}
t^{\text{nf}}, u^{\text{nf}}, v^{\text{nf}}, A^{\text{nf}}, B^{\text{nf}} & ::= & \text{Set} & \\
& | & \text{Bool} & \\
& | & \Pi A^{\text{nf}} B^{\text{nf}} & \\
& | & \Sigma A^{\text{nf}} B^{\text{nf}} & \\
& | & c & \\
& | & \lambda.t^{\text{nf}} & \\
& | & \langle t^{\text{nf}}, u^{\text{nf}} \rangle & \\
& | & h\,\overrightarrow{e^{\text{nf}}} & \textbf{\textit{if}}\ h\,\overrightarrow{e^{\text{nf}}}\ \textit{is strongly neutral}
\end{array}
$$

$$
e^{\text{nf}} \quad ::= \quad t^{\text{nf}} \quad | \quad .\pi_1 \quad | \quad .\pi_2
$$

Note that the converse does not hold; for instance:

$$
\mathbb{A} : \text{Set}; x : \mathbb{A} \to \mathbb{A} \vdash x \longrightarrow_{\delta\eta} \lambda y.x\,y : \mathbb{A} \to \mathbb{A}
$$

Proof. Assume that v is not of the form v^{nf}. Then, by induction, show that it can be subjected to at least one reduction step. □

2.22 Rigidly occurring terms ($t[\![u]\!]$)

A subterm occurs rigidly in a term if the occurrence cannot be made to "disappear" from the term by normalizing it (c.f. Definition 2.41, δη-normalization step).

Definition 2.168 (Rigid occurrence)**.** Let t and u be terms. Whether u occurs rigidly in t under n binders (written $t[\![u]\!]^n$) is defined recursively on t as follows:

$$
\begin{array}{rll}
(\text{R-ID}) & u[\![u]\!]^0 & \\
(\text{R-}\Pi_1) & (\Pi AB)[\![u]\!]^n & \textbf{if} \quad A[\![u]\!]^n \\
(\text{R-}\Pi_2) & (\Pi AB)[\![u]\!]^{1+n} & \textbf{if} \quad B[\![u]\!]^n \\
(\text{R-}\Sigma_1) & (\Sigma AB)[\![u]\!]^n & \textbf{if} \quad A[\![u]\!]^n \\
(\text{R-}\Sigma_2) & (\Sigma AB)[\![u]\!]^{1+n} & \textbf{if} \quad B[\![u]\!]^n \\
(\text{R-}\lambda) & (\lambda.t)[\![u]\!]^{1+n} & \textbf{if} \quad t[\![u]\!]^n \\
(\text{R-}\langle,\rangle_1) & (\langle t_1, t_2\rangle)[\![u]\!]^n & \textbf{if} \quad t_1[\![u]\!]^n \\
(\text{R-}\langle,\rangle_2) & (\langle t_1, t_2\rangle)[\![u]\!]^n & \textbf{if} \quad t_2[\![u]\!]^n \\
(\text{R-IRRED}) & (f\,\vec{e})[\![f]\!]^0 & \textbf{if} \quad f \text{ is an irreducible term} \\
(\text{R-STRONG}) & (h\,\vec{e})[\![u]\!]^n & \textbf{if} \quad h\,\vec{e} \text{ is strongly neutral} \\
 & & \textbf{and} \text{ there is } t \in \vec{e} \text{ such that } t[\![u]\!]^n
\end{array}
$$

Note that the term u is not weakened when the definition goes under a binder. Instead, the superindex n keeps track of the number of binders above u.

Definition 2.169 (Typed rigid occurrence)**.** Let t and u be terms. We say that u occurs rigidly with type U and context Δ (written $\Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : T$) if $\Sigma;\Gamma \vdash t : T$, $\Sigma;\Gamma,\Delta \vdash u : U$, and $\Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!]' : T$, where the latter is defined as follows:

$$
\begin{array}{rll}
(\text{TR-ID}) & \Sigma;\Gamma \vdash u[\![\cdot \vdash u : U]\!]' : T & \textbf{if} \quad \Sigma;\Gamma \vdash U \equiv T \textbf{ type} \\
(\text{TR-}\Pi_1) & \Sigma;\Gamma \vdash (\Pi AB)[\![\Delta \vdash u : U]\!]' : T & \textbf{if} \quad \Sigma;\Gamma \vdash A[\![\Delta \vdash u : U]\!] : \text{Set} \\
(\text{TR-}\Pi_2) & \Sigma;\Gamma \vdash (\Pi AB)[\![A',\Delta \vdash u : U]\!]' : T & \textbf{if} \quad \Sigma;\Gamma \vdash A' \equiv A \textbf{ type} \\
 & & \textbf{and} \; \Sigma;\Gamma, A \vdash B[\![\Delta \vdash u : U]\!] : \text{Set} \\
(\text{TR-}\Sigma_1) & \Sigma;\Gamma \vdash (\Sigma AB)[\![\Delta \vdash u : U]\!]' : T & \textbf{if} \quad \Sigma;\Gamma \vdash A[\![\Delta \vdash u : U]\!] : \text{Set} \\
(\text{TR-}\Sigma_2) & \Sigma;\Gamma \vdash (\Sigma AB)[\![A',\Delta \vdash u : U]\!]' : T & \textbf{if} \quad \Sigma;\Gamma \vdash A' \equiv A \textbf{ type} \\
 & & \textbf{and} \; \Sigma;\Gamma, A \vdash B[\![\Delta \vdash u : U]\!] : \text{Set} \\
(\text{TR-}\lambda) & \Sigma;\Gamma \vdash (\lambda.t)[\![A,\Delta \vdash u : U]\!]' : T & \textbf{if} \quad \Sigma;\Gamma \vdash \Pi AB \equiv T \textbf{ type} \\
 & & \textbf{and} \; \Sigma;\Gamma, A \vdash t[\![\Delta \vdash u : U]\!] : B
\end{array}
$$

$$(\text{TR-}\langle,\rangle_1)\quad \Sigma;\Gamma \vdash (\langle t_1, t_2\rangle)[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if}\quad \Sigma;\Gamma \vdash \Sigma AB \equiv T\ \textbf{type} \quad \textbf{and}\ \Sigma;\Gamma \vdash t_1[\![\Delta \vdash u : U]\!] : A$$

$$(\text{TR-}\langle,\rangle_2)\quad \Sigma;\Gamma \vdash (\langle t_1, t_2\rangle)[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if}\quad \Sigma;\Gamma \vdash \Sigma AB \equiv T\ \textbf{type} \quad \textbf{and}\ \Sigma;\Gamma \vdash t_2[\![\Delta \vdash u : U]\!] : B[t_1]$$

$$(\text{TR-IRRED})\ \Sigma;\Gamma \vdash (f\,\vec{e})[\![\cdot \vdash f : U]\!]' : T \qquad \textbf{if}\quad f \text{ is an irreducible term}$$

$$(\text{TR-STRONG})\Sigma;\Gamma \vdash (h\,\overrightarrow{e_1}\,t\,\overrightarrow{e_2})[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if}\quad \Sigma;\Gamma \vdash h\,\overrightarrow{e_1} : \Pi AB \quad \textbf{and}\ \Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : A \quad \textbf{and}\ h\,\overrightarrow{e_1}\,t\,\overrightarrow{e_2} \text{ is strongly neutral}$$

Lemma 2.170 (Typing of rigid occurrences)**.** *Suppose* $\Sigma;\Gamma \vdash t : T$ *and* $t[\![u]\!]^n$*. Then there exist* Δ *and* U*,* $|\Delta| = n$*, such that* $\Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : T$*.*

Proof. By induction on the derivation of $t[\![u]\!]$, and using the corresponding inversion lemmas (i.e. Lemma 2.53 (Σ inversion), Lemma 2.52 (Π inversion), Lemma 2.57 (type of λ-abstraction) and Lemma 2.82 (λ inversion), Lemma 2.60 (type of a pair) and Lemma 2.84 ($\langle,\rangle$-inversion), or Lemma 2.111 (application inversion), respectively). □

Remark 2.171 (Free variables of rigid occurrence). If $\Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : T$ then $\text{FV}(u) - |\Delta| \subseteq \text{FV}(t)$.

Lemma 2.172 (Free variables in reduction of rigid occurrences)**.** *Let* t *and* u *be terms such that* $\Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : T$*. If there is* r *such that* $\Sigma;\Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} r : T$*, then there is* v *such that* $\Sigma;\Gamma,\Delta \vdash u \equiv v : U$*, and* $\text{FV}(v) - |\Delta| \subseteq \text{FV}(r)$*.*

Proof. First, we show the following lemmas by mutual induction on n. For all Γ, t, u, T, A, B, u_1, u_2, f, $\vec{e}$, h, $\overrightarrow{e_1}$, $\overrightarrow{e_2}$ and t':

(i) If $\Sigma;\Gamma \vdash \Pi AB \longrightarrow^n_{\delta\eta} t : T$ then there is A' and $m \le n$ such that $\Sigma;\Gamma \vdash A \longrightarrow^m_{\delta\eta} A' : \text{Set}$ and $\text{FV}(A') \subseteq \text{FV}(t)$.

(ii) If $\Sigma;\Gamma \vdash \Pi AB \longrightarrow^n_{\delta\eta} t : T$ then there is B' and $m \le n$ such that $\Sigma;\Gamma, A \vdash B \longrightarrow^m_{\delta\eta} B' : \text{Set}$ and $\text{FV}(B') - 1 \subseteq \text{FV}(t)$.

(iii) If $\Sigma;\Gamma \vdash \Sigma AB \longrightarrow^n_{\delta\eta} t : T$ then there is A' and $m \le n$ such that $\Sigma;\Gamma \vdash A \longrightarrow^m_{\delta\eta} A' : \text{Set}$ and $\text{FV}(A') \subseteq \text{FV}(t)$.

(iv) If $\Sigma;\Gamma \vdash \Sigma AB \longrightarrow^n_{\delta\eta} t : T$ then there is B' and $m \le n$ such that $\Sigma;\Gamma, A \vdash B \longrightarrow^m_{\delta\eta} B' : \text{Set}$ and $\text{FV}(B') - 1 \subseteq \text{FV}(t)$.

(v) If $\Sigma;\Gamma \vdash \lambda.u \longrightarrow^n_{\delta\eta} t : T$ and $\Sigma;\Gamma \vdash T \equiv \Pi AB$ **type** then there is u' and $m \le n$ such that $\Sigma;\Gamma, A \vdash u \longrightarrow^m_{\delta\eta} u' : B$ and $\text{FV}(u') - 1 \subseteq \text{FV}(t)$.

(vi) If $\Sigma;\Gamma \vdash \langle u_1, u_2\rangle \longrightarrow^n_{\delta\eta} t : T$ and $\Sigma;\Gamma \vdash T \equiv \Sigma AB$ **type** then there is u_1' and $m \le n$ such that $\Sigma;\Gamma \vdash u_1 \longrightarrow^m_{\delta\eta} u_1' : A$, and $\mathrm{FV}(u_1') \subseteq \mathrm{FV}(t)$.

(vii) If $\Sigma;\Gamma \vdash \langle u_1, u_2\rangle \longrightarrow^n_{\delta\eta} t : T$ and $\Sigma;\Gamma \vdash T \equiv \Sigma AB$ **type** then there is u_2' and $m \le n$ such that $\Sigma;\Gamma \vdash u_2 \longrightarrow^m_{\delta\eta} u_2' : B[u_1]$ and $\mathrm{FV}(u_2') \subseteq \mathrm{FV}(t)$.

(viii) If $\Sigma;\Gamma \vdash f\,\vec{e} \longrightarrow^n_{\delta\eta} t : T$, with f irreducible, and $\Sigma;\Gamma \vdash f : B'$ then there is f' and $m \le n$ such that $\Sigma;\Gamma \vdash f \longrightarrow^m_{\delta\eta} f' : B'$ and $\mathrm{FV}(f') \subseteq \mathrm{FV}(t)$.

(ix) If $\Sigma;\Gamma \vdash h\,\overrightarrow{e_1}\,u\,\overrightarrow{e_2} \longrightarrow^n_{\delta\eta} t : B$, with $h\,\overrightarrow{e_1}\,u\,\overrightarrow{e_2}$ strongly neutral, and $\Sigma;\Gamma \vdash h\,\overrightarrow{e_1} : \Pi UV$ then there is u' such that $\Sigma;\Gamma \vdash u \longrightarrow^m_{\delta\eta} u' : U$, $m \le n$ and $\mathrm{FV}(u') \subseteq \mathrm{FV}(t)$.

Here we do one case of (viii):

- Case where $\Sigma;\Gamma \vdash f\,\vec{e} \longrightarrow^{n+1}_{\delta\eta} t : T$ and the first step is η-Π:

  In this case, $\Sigma;\Gamma \vdash f\,\vec{e} \longrightarrow_{\delta\eta} \lambda.(f\,\vec{e})^{(+1)}\,0 : T$, and $\Sigma;\Gamma \vdash \lambda.(f\,\vec{e})^{(+1)}\,0 \longrightarrow^n_{\delta\eta} t : T$.

  By Lemma 2.56 (neutral inversion), there is B' such that $\Sigma;\Gamma \vdash f : B'$.

  Because $\Sigma;\Gamma \vdash \lambda.(f\,\vec{e})^{(+1)}\,0 : T$, by Lemma 2.57 (type of λ-abstraction), then $\Sigma;\Gamma \vdash \Pi AB \equiv T$ **type**.

  By (v), there is t' and m' such that $\Sigma;\Gamma, A \vdash (f\,\vec{e})^{(+1)}\,0 \longrightarrow^{m'}_{\delta\eta} t' : B$, with $\mathrm{FV}(t') - 1 \subseteq \mathrm{FV}(t)$ and $m' \le n$. If f is irreducible, then $f^{(+1)}$ is also irreducible.

  By Lemma 2.62 (context weakening), $\Sigma;\Gamma, A \vdash f^{(+1)} : B'^{(+1)}$. By (viii), there is f' such that $\Sigma;\Gamma, A \vdash f^{(+1)} \longrightarrow^m_{\delta\eta} f' : B'^{(+1)}$, $m \le m' \le n \le n+1$ and $\mathrm{FV}(f') \subseteq \mathrm{FV}(t')$. By Remark 2.94 (strengthening of reduction), $\Sigma;\Gamma \vdash f \longrightarrow^m_{\delta\eta} f'^{(-1)} : B'$.

  As shown earlier, $m \le n+1$ and $\mathrm{FV}(f') \subseteq \mathrm{FV}(t')$. The latter gives $\mathrm{FV}(f'^{(-1)}) \subseteq \mathrm{FV}(t') - 1$. Note that $\mathrm{FV}(t') - 1 \subseteq \mathrm{FV}(t)$; therefore $\mathrm{FV}(f'^{(-1)}) \subseteq \mathrm{FV}(t)$.

Then, proceed by induction on $\Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : T$. We do a few representative cases:

- Case (TR-ID): Then $\Sigma;\Gamma \vdash u[\![\cdot \vdash u : U]\!] : T$, $\Sigma;\Gamma \vdash u \longrightarrow^\star_{\delta\eta} r : T$, and $\Sigma;\Gamma \vdash U \equiv T$ **type**. Take $v = r$. By Lemma 2.86 (equality of $\delta\eta$-reduct) and the CONV-EQ rule, $\Sigma;\Gamma \vdash u \equiv v : U$. By construction, $\mathrm{FV}(v) - 0 \subseteq \mathrm{FV}(v) = \mathrm{FV}(r)$.

- Case (TR-Π_2): Then $\Delta = A', \Delta'$ and $\Sigma;\Gamma \vdash (\Pi AB)[\![A', \Delta' \vdash u : U]\!] : T$, with $\Sigma;\Gamma, A \vdash B[\![\Delta' \vdash u : U]\!] : \mathrm{Set}$ and $\Sigma;\Gamma \vdash \Pi AB \longrightarrow^\star_{\delta\eta} r : T$, By (ii), there is B' such that $\Sigma;\Gamma, A \vdash B \longrightarrow^\star_{\delta\eta} B' : T$, and $\mathrm{FV}(B') - 1 \subseteq \mathrm{FV}(r)$. By the induction hypothesis, there is v such that $\Sigma;\Gamma, A, \Delta' \vdash u \equiv v : U$ and $\mathrm{FV}(v) - |\Delta'| \subseteq \mathrm{FV}(B')$, that is, $\mathrm{FV}(v) - |\Delta| \subseteq \mathrm{FV}(B') - 1 \subseteq \mathrm{FV}(r)$.

□

Corollary 2.173 (Preservation of head variable)**.** *If* $\Sigma;\Gamma \vdash x\,\vec{e} \equiv t : U$*, then* $x \in \mathrm{FV}(t)$*.*

Proof. By Postulate 14 (existence of a common reduct) and Remark 2.43 (free variables of δη-reduct), there is r such that $\Sigma;\Gamma \vdash x\,\vec{e} \longrightarrow_{\delta\eta} r : U$ and $\Sigma;\Gamma \vdash t \longrightarrow_{\delta\eta} r : U$, with $\mathrm{FV}(r) \subseteq \mathrm{FV}(t)$.

By property (viii) in the proof of Lemma 2.172 (free variables in reduction of rigid occurrences), there exists f' such that $\Sigma;\Gamma \vdash x\,\vec{e} \longrightarrow^{m}_{\delta\eta} f' : U$ for some m, and $\mathrm{FV}(f') \subseteq \mathrm{FV}(t)$.

By Definition 2.41 (δη-normalization step), at each of the m reduction steps, only the APP_j rule may be applied (for perhaps a different j at each step) and thus $f' = x\,\overrightarrow{e'}$ for some $\overrightarrow{e'}$. Therefore, $x \in \mathrm{FV}(f')$, and thus, $x \in \mathrm{FV}(t)$. □

Lemma 2.174 (Rigidity of substitution by neutral terms in normal forms)**.** *Given a vector* $\vec{f}$ *of* irreducible *neutral terms, a normal form term* v^{nf}*, and a vector* $\vec{x}$ *of variables fulfilling the hypothesis of Definition 2.34 (iterated hereditary substitution). Then we have* $v^{\mathrm{nf}}[\vec{f}/\vec{x}] \Downarrow u$ *for some* u*, and, for all* $i \in \{1,...,n\}$*, if* $x_i \in \mathrm{FV}(v^{\mathrm{nf}})$*, then there is* m *such that* $u[\![f_i^{(+m)}]\!]^m$*.*

Proof. By Remark 2.36 (hereditary substitution by a neutral term), u exists. By induction on v^{nf} and Definition 2.31 (hereditary substitution), $u[\![f^{(+m)}]\!]^m$. □

Lemma 2.175 (Preservation of irreducibles by normal forms)**.** *Suppose that* $\Theta;\Gamma,\overrightarrow{x : U}^{n} \vdash v : V$*, and* $\Theta;\Gamma,\vec{U} \vdash v \not\longrightarrow_{\delta\eta} : V$*.*

Let $\vec{f}^{n}$ *be a vector of irreducible terms, such that, for all* $i = 1,...,n$*,* $f_i = h_i\,\overrightarrow{e_i}$*, and, for some* h*,* $\Theta;\Gamma \vdash h : \Pi\vec{U}^{n}B$*, and* $\Theta;\Gamma \vdash h\,\vec{f} : B[\vec{f}]$*.*

Take $i \in \{1,...,n\}$ *such that* $x_i \in \mathrm{FV}(v)$*. If* $\Theta;\Gamma \vdash v[\vec{f}] \equiv u : T$ *for some term* u *and type* T*, and* $h_i = y$*, then* $y \in \mathrm{FV}(u)$*.*

Proof. By Lemma 2.167 (characterization of normal forms), v is of the form v^{nf}.

By Lemma 2.174 (rigidity of substitution by neutral terms in normal forms), we have that $v[\vec{f}][\![f_i^{(+m)}]\!]^m$ for some m, which, by Lemma 2.170 (typing of rigid occurrences), gives $\Theta;\Gamma \vdash v^{\mathrm{nf}}[\vec{f}][\![\Delta \vdash f_i^{(+|\Delta|)} : U]\!] : T$ for some Δ, U, with $|\Delta| = m$.

Because $\Theta;\Gamma \vdash v[\vec{f}] \equiv u : T$, by Postulate 14 (existence of a common reduct), there is a term r such that $\Theta;\Gamma \vdash v[\vec{f}] \longrightarrow^{\star}_{\delta\eta} r : T$, and $\Theta;\Gamma \vdash u \longrightarrow^{\star}_{\delta\eta} r : T$.

Because $\Theta;\Gamma \vdash v^{\mathrm{nf}}[\vec{f}][\![\Delta \vdash f_i^{(+|\Delta|)} : U]\!] : T$ and $\Theta;\Gamma \vdash v[\vec{f}] \longrightarrow^{\star}_{\delta\eta} r : T$, by Lemma 2.172 (free variables in reduction of rigid occurrences) and Remark 2.28 (renaming and free variables), there is $\tilde{u}$ such that $\Theta;\Gamma,\Delta \vdash f_i^{(+|\Delta|)} \equiv \tilde{u} : U$ and $\mathrm{FV}(\tilde{u}) - |\Delta| \subseteq \mathrm{FV}(r)$.

By Corollary 2.173 (preservation of head variable), $h_i^{(+|\Delta|)} = y^{(+|\Delta|)} \in \mathrm{FV}(\tilde{u})$, therefore $h_i = y \in \mathrm{FV}(\tilde{u}) - |\Delta| \subseteq \mathrm{FV}(r)$.

Because $\Theta;\Gamma \vdash u \longrightarrow^{\star}_{\delta\eta} r : T$, by Remark 2.43 (free variables of δη-reduct), $h_i = y \in \mathrm{FV}(u)$. □

Lemma 2.176 (Injectivity of normal forms with respect to irreducibles)**.** *Suppose that* $\Theta;\Gamma,\overrightarrow{x:U}^n \vdash v : V$*, and* $\Theta;\Gamma,\vec{U} \vdash v \not\to_{\delta\eta} : V$*.*

Let $\vec{f}^n$ *and* $\overrightarrow{f'}^n$ *be two vectors of irreducible terms, such that, for all* $i = 1,...,n$*,* $f_i = h_i\,\overrightarrow{e_i}$*,* $f'_i = h'_i\,\overrightarrow{e'_i}$*, and, for some* h*,* $\Theta;\Gamma \vdash h : \Pi\vec{U}B$*,* $\Theta;\Gamma \vdash h\,\vec{f} : B[\vec{f}]$*,* $\Theta;\Gamma \vdash h\,\overrightarrow{f'} : B[\overrightarrow{f'}]$*.*

Let $i \in \{1,...,n\}$ *such that* $x_i \in \mathrm{FV}(v)$*. If* $\Theta;\Gamma \vdash v[\vec{f}] \equiv v[\overrightarrow{f'}] : V[\vec{f}]$*, then* $h_i = h'_i$*.*

Proof. Because $\Theta;\Gamma,\vec{U} \vdash v \not\to_{\delta\eta} : V$, by Lemma 2.167 (characterization of normal forms), v is of the form v_{nf} for some v_{nf}.

Let $\vec{x}^n = n-1,...,0$. It suffices to show the following, stronger property, and taking $\Delta = \cdot$:
For all Δ *and* u^{nf} *if* $x_i^{(+|\Delta|)} \in \mathrm{FV}(u^{\mathrm{nf}})$ *and for some* T*,* $\Theta;\Gamma,\Delta \vdash u^{\mathrm{nf}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] \equiv u^{\mathrm{nf}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] : T$*, then* $h_i^{(+|\Delta|)} = {h'}_i^{(+|\Delta|)}$*.*

By induction on u_{nf}:

(a) Case $u^{\mathrm{nf}} = x_i^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}$: By Definition 2.34 (iterated hereditary substitution) and Remark 2.36 (hereditary substitution by a neutral term), we have $u^{\mathrm{nf}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] = f_i^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$, $u^{\mathrm{nf}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] = {f'}_i^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$. By the assumption, f_i and f'_i are irreducible. By construction, $\overrightarrow{f_i}^{(+|\Delta|)}$ and $\overrightarrow{f'_i}^{(+|\Delta|)}$ are also irreducible. By Remark 2.165 (extensions of irreducible terms), because $\overrightarrow{f_i}^{(+|\Delta|)}$ and $\overrightarrow{f'_i}^{(+|\Delta|)}$ are irreducible, $f_i^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ and ${f'}_i^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ are irreducible, and therefore strongly neutral. By Lemma 2.163 (injectivity of elimination for strongly neutral terms), $h_i^{(+|\Delta|)} = {h'}_i^{(+|\Delta|)}$.

(b) Case $u^{\mathrm{nf}} = x_j\,\overrightarrow{e^{\mathrm{nf}}}$ ($x_j \in \vec{x}$, $j \neq i$): By Remark 2.165 (extensions of irreducible terms), $f_j^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ and ${f'}_j^{(+|\Delta|)}\,\overrightarrow{e^{\mathrm{nf}}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ are irreducible, and therefore strongly neutral. The proof follows as above.

(c) Case $u^{\mathrm{nf}} = h\,\overrightarrow{e^{\mathrm{nf}}}$, $h = y$, $y \notin \vec{x}^{(+|\Delta|)}$: Then $u^{\mathrm{nf}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] = y'\,\overrightarrow{e^{\mathrm{nf}}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ and $u^{\mathrm{nf}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] = y'\,\overrightarrow{e^{\mathrm{nf}}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ for some y'.

If $x_i^{(+|\Delta|)} \in \mathrm{FV}(u^{\mathrm{nf}})$ and $h \neq x_i^{(+|\Delta|)}$, then $x_i^{(+|\Delta|)} \in \mathrm{FV}(e_j^{\mathrm{nf}})$ for some j. By the assumption and Lemma 2.163 (injectivity of elimination for strongly neutral terms), we have $\Theta;\Gamma,\Delta \vdash e_j^{\mathrm{nf}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] \equiv e_j^{\mathrm{nf}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] : U$ for some U. By the induction hypothesis, $h_i^{(+|\Delta|)} = {h'}_i^{(+|\Delta|)}$.

(d) Case $u^{\mathrm{nf}} = h\,\overrightarrow{e^{\mathrm{nf}}}$, $h = \mathbb{o}$: Then $u^{\mathrm{nf}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] = h\,\overrightarrow{e^{\mathrm{nf}}}[\vec{f}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$ $u^{\mathrm{nf}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}] = h\,\overrightarrow{e^{\mathrm{nf}}}[\overrightarrow{f'}^{(+|\Delta|)}/\vec{x}^{(+|\Delta|)}]$, The proof follows as in case (c).

(e) Case $u^{\mathrm{nf}} = h\,\overrightarrow{e^{\mathrm{nf}}}$, $h = \mathrm{if}$, u^{nf} strongly neutral: The proof follows as in case (d).

(f) Case $u^{\mathrm{nf}} = c$: Then $\mathrm{FV}(c) = \emptyset \not\ni x_i$. This case holds vacuously.

(g) Case $u^{\mathrm{nf}} = \lambda.t^{\mathrm{nf}}$: Follows by Lemma 2.83 (injectivity of λ) and the induction hypothesis.

(h) Case $u^{\mathrm{nf}} = \langle t_1^{\mathrm{nf}}, t_2^{\mathrm{nf}} \rangle$: Follows by Lemma 2.85 (injectivity of $\langle,\rangle$) and the induction hypothesis.

(i) Case $u^{\mathrm{nf}} = \Pi A^{\mathrm{nf}} B^{\mathrm{nf}}$: . Follows by Postulate 10 (injectivity of Π) and the induction hypothesis.

(j) Case $u^{\mathrm{nf}} = \Sigma A^{\mathrm{nf}} B^{\mathrm{nf}}$: Analogous to case (i).

(k) Case $u^{\mathrm{nf}} = \mathrm{Bool}$ or $u^{\mathrm{nf}} = \mathrm{Set}$: Analogous to case (f).

□

2.23 Out of scope features

The theoretical description is a subset of the actual implementation. Our focus is on the metavariable solving aspect of dependent type checking. Our goal is to provide a unification algorithm which will only produce well-typed terms throughout its operation. We have thus side-lined some equally important, but mostly orthogonal aspects of dependent type checking.

2.23.1 Inductive definitions and inductive families

Recursive definitions or pattern matching are not described in the theory. We do this to avoid cluttering the exposition with redundant details.

In the implementation, expansion of definitions is performed analogously to metavariable expansion, and pattern matching for inductive families is a generalization of the recursor for booleans (if).

2.23.2 Identity types

The implementation also includes a built-in identity type (i), the corresponding constructor (ii), the J axiom (iii) and its computation rule (iv).

$$\frac{\Sigma \vdash \Gamma\ \mathbf{ctx}}{\Sigma; \Gamma \vdash \mathbb{Id} : (X : \mathrm{Set}) \to X \to X \to \mathrm{Set}}\ (\mathrm{i})$$

$$\frac{\Sigma; \Gamma \vdash t : T}{\Sigma; \Gamma \vdash \mathbb{refl} : \mathbb{Id}\, T\, t\, t}\ (\mathrm{ii})$$

$$\frac{\Sigma \vdash \Gamma\ \mathbf{ctx}}{\begin{array}{l}\Sigma; \Gamma \vdash \mathbb{J} : (X : \mathrm{Set}) \to (x_1 : X) \to (x_2 : X) \to \\ \qquad (Y : (x_1 : X) \to (x_2 : X) \to \mathbb{Id}\, X\, x_1\, x_2 \to \mathrm{Set}) \\ \qquad ((x : X) \to Y\, x\, x\, \mathbb{refl}) \to \\ \qquad (z : \mathbb{Id}\, X\, x_1\, x_2) \to Y\, x_1\, x_2\, z\end{array}}\ (\mathrm{iii})$$

$$\Sigma; \Gamma \vdash \mathbb{J}\, T\, t\, t\, U\, v\, \mathbb{refl} \longrightarrow_{\delta\eta} v\, t : U\, t\, t\, \mathbb{refl} \tag{iv}$$

2.23.3 Generalized records with η

We include a dependent sum type (Σ) with η-equality in the theoretical presentation. The implementation includes record types with an arbitrary number of fields, and η-equality.

- Records with more than two fields can be modelled in the theoretical system as nested Σ-types.
- Record types with no fields (i.e. the *unit type with* η*-equality*) can however not be modelled easily in the theoretical system.

  In fact, η-expansion for a record type with no fields can be particularly challenging to handle rigorously. In the implementation we adopt a pragmatic approach, where we reduce the power of our unification rules in order to preserve completeness (Section 4.7.1).

Chapter 3

Unification for type checking

We are interested in the problem of dependent type checking with metavariables, or, more succinctly, the type checking problem.

When describing a type checking problem, we use metavariables to represent those subterms omitted by the user. Specifically, each omitted term is replaced by a fresh metavariable applied to all the variables that are in scope at that particular point in the term. The result is a *term with holes*.

Definition 3.1 (Term with holes)**.** Consider a signature Σ and context Γ, such that $\Sigma \vdash \Gamma\,\mathbf{ctx}$.

We say that t is a term with holes in signature Σ and context Γ if, for each metavariable α occurring in t ($\alpha \in \textsc{metas}(t)$), either:

(a) $\alpha \in \textsc{decls}(\Sigma)$, or

(b) α occurs only once in t, and it occurs applied to all variables in the scope of its occurrence.

More precisely, if we define the relation "$\textsc{holes}(\_,\_,\_) \Downarrow \_$" as in Figure 3.1, we say that a term t is a term with holes in signature Σ and context Γ if $(\textsc{holes}(\Sigma, |\Gamma|, t) \Downarrow H)$ and $\textsc{metas}(t) \subseteq \textsc{support}(\Sigma) \cup H$.

Definition 3.2 (Well-formed type checking problem: $\Sigma; \Gamma \vdash^? t : A$)**.** Consider a signature Σ and context Γ, such that $\Sigma \vdash \Gamma\,\mathbf{ctx}$, and a type A such that $\Sigma; \Gamma \vdash A\ \mathbf{type}$. Let t be a term with holes in signature Σ and context Γ. Then $\Sigma; \Gamma \vdash^? t : A$ is a well-formed type checking problem.

Definition 3.3 (Solution to a type checking problem: $\Theta \vDash \Sigma; \Gamma \vdash^? t : A$)**.** We say that a metasubstitution Θ is a solution to the type checking problem $\Sigma; \Gamma \vdash^? t : A$ (written $\Theta \vDash \Sigma; \Gamma \vdash^? t : A$), if $\Theta\,\mathbf{wf}$, $\Theta_\Sigma \vDash \Sigma$, $\textsc{decls}(\Theta) = \textsc{decls}(\Sigma) \cup \textsc{holes}(\Sigma, |\Gamma|, t)$, and $\Theta; \Gamma \vdash t : A$.

Definition 3.4 (Unique solution to a type checking problem)**.** In our development we are interested in finding a unique solution; namely, a metasubstitution Θ such that i) $\Theta \vDash \Sigma; \Gamma \vdash^? t : A$, and ii) for any other metasubstitution Θ' such that $\Theta' \vDash \Sigma; \Gamma \vdash^? t : A$, we have $\Theta \equiv \Theta'$.

$$
\begin{array}{lll}
\textsc{holes}(\Sigma, n, \text{Set}) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, \text{Bool}) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, \Pi A B) \Downarrow (H_1 \cup H_2) & \textbf{if} & \textsc{holes}(\Sigma, n, A) \Downarrow H_1 \\
 & \textbf{and} & \textsc{holes}(\Sigma, n+1, B) \Downarrow H_2 \\
 & \textbf{and} & H_1 \cap H_2 = \emptyset \\
\textsc{holes}(\Sigma, n, \Sigma A B) \Downarrow (H_1 \cup H_2) & \textbf{if} & \textsc{holes}(\Sigma, n, A) \Downarrow H_1 \\
 & \textbf{and} & \textsc{holes}(\Sigma, n+1, B) \Downarrow H_2 \\
 & \textbf{and} & H_1 \cap H_2 = \emptyset \\
\textsc{holes}(\Sigma, n, c) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, \lambda.t) \Downarrow H & \textbf{if} & \textsc{holes}(\Sigma, n+1, t) \Downarrow H \\
\textsc{holes}(\Sigma, n, \langle t_1, t_2 \rangle) \Downarrow (H_1 \cup H_2) & \textbf{if} & \textsc{holes}(\Sigma, n, t_1) \Downarrow H_1 \\
 & \textbf{and} & \textsc{holes}(\Sigma, n, t_2) \Downarrow H_2 \\
 & \textbf{and} & H_1 \cap H_2 = \emptyset \\
\textsc{holes}(\Sigma, n, \alpha\ (n-1)\ (n-2)\ ...\ 0) \Downarrow \{\alpha\} & \textbf{if} & \alpha \notin \textsc{decls}(\Sigma) \\
\textsc{holes}(\Sigma, n, h\ \vec{e}^m) \Downarrow (\bigcup_{i=1}^{m} H_i) & \multicolumn{2}{l}{\textbf{if } \forall i \in \{1, ..., m\}.\textsc{holes}(\Sigma, n, e_i) \Downarrow H_i} \\
 & \multicolumn{2}{l}{\textbf{and } \forall i, j.i \neq j.H_i \cap H_j = \emptyset} \\
 & \multicolumn{2}{l}{\textbf{and } (h = \alpha, \alpha \in \textsc{decls}(\Sigma)} \\
 & \multicolumn{2}{l}{\quad \textbf{or } h = x \textbf{ or } h = \mathbb{U} \textbf{ or } h = \text{if})} \\
 & & \\
\textsc{holes}(\Sigma, n, .\pi_1) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, .\pi_2) \Downarrow \emptyset & &
\end{array}
$$

Figure 3.1: Recursive definition of the set of holes in a term

This is one simple example of a type checking problem:

Example 3.5 (Dependent type checking with metavariables, unique solution)**.** Consider $\Sigma = \mathbb{A} : \mathrm{Set}, \mathbb{c} : (A : \mathrm{Set}) \to A \to A$. We have Σ **sig**, $\Sigma \vdash \cdot$ **ctx**, and $\Sigma; \cdot \vdash \mathbb{A} \to \mathbb{A}$ **type**.

Take $t = \lambda x.\mathbb{c}\,(\alpha\,x)\,x$. The metavariable α does not occur in the signature Σ, and is applied to all the variables in scope (here, only x, because Γ is empty). By Definition 3.1 (term with holes), t is a term with holes in signature Σ and context Γ, and the following is a well-formed type checking problem:

$$\Sigma; \cdot \vdash^{?} \lambda x.\mathbb{c}\,(\alpha\,x)\,x : \mathbb{A} \to \mathbb{A}$$

If we take $\Theta = \mathbb{A} : \mathrm{Set}, \mathbb{c} : (A : \mathrm{Set}) \to A \to A, \alpha := \lambda x.\mathbb{A} : \mathbb{A} \to \mathrm{Set}$, then $\Theta_\Sigma \vDash \Sigma$, and $\Theta; \cdot \vdash t : \mathbb{A} \to \mathbb{A}$. Therefore, Θ is a solution to the type checking problem $\Sigma; \cdot \vdash^{?} t : \mathbb{A} \to \mathbb{A}$.

Let Θ' be another solution to the given problem.

(i) $\Theta' \vDash \Sigma$, therefore $\Theta'; \cdot \vdash \mathbb{A} : \mathrm{Set}$ and $\mathbb{c} : (A : \mathrm{Set}) \to A \to A$.

(ii) By Lemma 2.82 (λ inversion), Lemma 2.56 (neutral inversion), and Lemma 2.52 (Π inversion), $\Theta'; \cdot \vdash \alpha : \mathbb{A} \to \mathrm{Set}$ and $\Theta'; \cdot \vdash \alpha\,x \equiv \mathbb{A} : \mathrm{Set}$. By Lemma 4.36 (Miller's pattern condition), $\Theta'; \cdot \vdash \alpha \equiv \lambda x.\mathbb{A} : \mathbb{A} \to \mathrm{Set}$.

By Lemma 2.130 (alternative characterization of a compatible metasubstitution), $\Theta' \vDash \Theta$. By Lemma 2.148 (uniqueness of closing metasubstitution), $\Theta' \equiv \Theta$. Therefore, Θ is a unique solution to the given type checking problem. ◀

Example 3.6 (Dependent type checking with metavariables, no unique solution)**.** Take the same type checking problem as in Example 3.5:

$$\Sigma; \cdot \vdash^{?} \lambda x.\mathbb{c}\,(\alpha\,\mathbb{A})\,x : \mathbb{A} \to \mathbb{A}$$

Both Θ_1 and Θ_2 (below) are solutions:

$$\Theta_1 = \mathbb{A} : \mathrm{Set}, \mathbb{c} : (A : \mathrm{Set}) \to A \to A, \alpha := \lambda x.\mathbb{A} : \mathbb{A} \to \mathrm{Set}$$
$$\Theta_2 = \mathbb{A} : \mathrm{Set}, \mathbb{c} : (A : \mathrm{Set}) \to A \to A, \alpha := \lambda x.x : \mathbb{A} \to \mathrm{Set}$$

However, by postulate Postulate 14 (existence of a common reduct), $\lceil \Theta_1; \cdot \vdash \alpha \not\equiv \lambda x.x : \mathrm{Set} \rfloor$ thus $\lceil \Theta_1 \not\vDash \Theta_2 \rfloor$. By Lemma 2.147 (compatibility respects equality) this means $\lceil \Theta_1 \not\equiv \Theta_2 \rfloor$. Therefore neither Θ_1 nor Θ_2 are unique solutions. ◀

3.1 From type checking to unification

In this section we give an account of an elaboration algorithm proposed by Mazzoli and Abel [36], which reduces a type checking problem to a higher-order unification problem.

Definition 3.7 (Basic constraint)**.** A basic constraint is a well-typed equation between two terms and their types.

$$\Gamma \vdash t : A \cong u : B$$

A basic constraint is well-formed in a signature Σ (written $\Sigma; \Gamma \vdash t : A \cong u : B\,\mathbf{wf}$) when each of the two sides is well-typed.

$$\frac{\Sigma; \Gamma \vdash t : A \qquad \Sigma; \Gamma \vdash u : B}{\Sigma; \Gamma \vdash t : A \cong u : B\,\mathbf{wf}}$$

Basic constraints thus defined are *heterogeneous*: each side may have a different type.

Definition 3.8 (Solution to a basic constraint: $\Theta \vDash \Sigma; \Gamma \vdash t : A \cong u : B$)**.** A metasubstitution Θ is a solution to a well-formed basic constraint $\Sigma; \Gamma \vdash t : A \cong u : B\,\mathbf{wf}$ (written $\Theta \vDash \Sigma; \Gamma \vdash t : A \cong u : B$) if $\Theta \vDash \Sigma$, $\Theta; \Gamma \vdash A \equiv B : \mathrm{Set}$ and $\Theta; \Gamma \vdash t \equiv u : A$.

Given a set of basic constraints in a common signature, we can formulate a dependent unification problem.

Problem 3.9 (Unification of dependently-typed terms)**.** Given a signature Σ and a set of basic constraints of the form $\Sigma; \Gamma_i \vdash t_i : A_i \cong u_i : B_i$ ($i \in \{1, ..., n\}$), is there a metasubstitution Θ such that $\Theta \vDash \Sigma$, and for each $i \in \{1, ..., n\}$, Θ is a solution to $\Sigma; \Gamma_i \vdash t_i : A_i \cong u_i : B_i$?

An elaboration algorithm reduces a type checking problem to a unification problem:

Definition 3.10 (Elaboration algorithm)**.** An elaboration algorithm takes as input a type-checking problem $\Sigma; \Gamma \vdash^? t : A$ (Definition 3.2) and produces a signature Σ', a term u, and a set of basic constraints $\vec{\mathcal{C}}$.

Definition 3.11 (Well-formedness of an elaboration algorithm)**.** We say that the elaboration algorithm is well-formed if, for any well-formed type checking problem $\Sigma; \Gamma \vdash^? t : A$, the algorithm produces a signature Σ', a term u and a set of basic constraints $\vec{\mathcal{C}}$, such that $\Sigma \subseteq \Sigma'$, $\textsc{AtomDecls}(\Sigma) = \textsc{AtomDecls}(\Sigma')$, $\textsc{support}(\Sigma') \supseteq \textsc{metas}(t)$, $\Sigma'\ \mathbf{sig}$, $\Sigma'; \Gamma \vdash u : A$, and each of the basic constraints $\mathcal{C} \in \vec{\mathcal{C}}$ is well-formed.

For an elaboration algorithm to be correct, the solutions to the constraints must be in correspondence with the solutions to the original type checking problem:

Definition 3.12 (Correctness of an elaboration algorithm)**.** We say that an elaboration algorithm is correct if it is well-formed, sound and complete. That is, given a well-formed type checking problem $\Sigma; \Gamma \vdash^? t : A$, if Σ' is the signature produced by the elaboration algorithm, u the term, and $\vec{\mathcal{C}}$ the basic constraints, then the following hold:

Soundness Let Θ be such that $\Theta \vDash \Sigma'$ and $\Theta \vDash \vec{\mathcal{C}}$. Then $\Theta_{\Sigma \cup t}\,\mathbf{wf}$, $\Theta_{\Sigma \cup t} \vDash \Sigma; \Gamma \vdash^? t : A$, and $\Theta; \Gamma \vdash t \equiv u : A$.

Completeness Let Θ be a metasubstitution such that $\Theta \vDash \Sigma; \Gamma \vdash^{?} t : A$. Then there is a metasubstitution $\tilde{\Theta}$ such that $\tilde{\Theta}_{\Sigma \cup t} = \Theta$, $\tilde{\Theta} \vDash \Sigma'$, and $\tilde{\Theta} \vDash \vec{\mathcal{C}}$.

The algorithm proposed by Mazzoli and Abel [36] thus elaborates a type checking problem (Definition 3.2) to Problem 3.9 (unification of dependently-typed terms). Both those subterms omitted by the user, and those subterms which cannot be immediately type-checked are replaced by metavariables of the appropriate type. The signature Σ is thus extended into a signature Σ', which adds declarations for any metavariables used in the term t and the type A. A term u of type A, and a set of basic constraints $\vec{\mathcal{C}}$ is returned. These are such that, when the constraints are such that, when the constraints $\vec{\mathcal{C}}$ are solved, the term u becomes judgmentally equal to the term t.

We describe some of the cases of their algorithm in Algorithm 1; namely the cases for holes, λ-abstraction, application and atoms.

Example 3.13 (Elaboration of a type checking problem with metavariables)**.** Consider again the type-checking problem of Example 3.5 (dependent type checking with metavariables, unique solution):

$$
\begin{array}{l}
\Sigma; \cdot \vdash^{?} t : \mathbb{A} \to \mathbb{A} \\
\quad \textbf{where} \\
\quad \Sigma \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{c} : (X : \mathrm{Set}) \to X \to X \\
\quad t \stackrel{\text{def}}{=} \lambda x. \mathbb{c}\ (\alpha\ x)\ x
\end{array}
$$

Solving the type-checking problem involves i) finding a type and a body for α, such that ii) the term t has type $\mathbb{A} \to \mathbb{A}$ in the empty context $(\cdot)$.

If we apply Algorithm 1 to the problem $\Sigma; \cdot \vdash^{?} t : \mathbb{A} \to \mathbb{A}$, we obtain the constraints below, and the elaborated term $u = \alpha_0$. Note how each output constraint corresponds to one level in the syntax of the term t.

$$
\begin{array}{l}
\Sigma, \\
\alpha_0 : \mathbb{A} \to \mathbb{A}, \gamma_1 : \mathrm{Set}, \gamma_2 : \gamma_1 \to \mathrm{Set} \\
\alpha_1 : (x : \gamma_1) \to \gamma_2\ x, \\
\gamma_3 : \gamma_1 \to \mathrm{Set}, \\
\gamma_4 : (x : \gamma_1) \to (z : \gamma_3\ x) \to \mathrm{Set} \\
\alpha_2 : \gamma_1 \to \gamma_3 x \\
\alpha_3 : (x : \gamma_1) \to (z : \gamma_3\ x) \to \gamma_4\ x\ z, \\
\gamma_5 : \gamma_1 \to \mathrm{Set}, \\
\gamma_6 : (x : \gamma_1) \to (z : \gamma_5\ x) \to \mathrm{Set} \\
\alpha : (x : \gamma_1) \to \gamma_5 x \\
\alpha_4 : (x : \gamma_1) \to (z : \gamma_5\ x) \to \gamma_6\ x\ z \\
; \\
\cdot \vdash \alpha_0 : \mathbb{A} \to \mathbb{A} \cong \lambda x. (\alpha_1\ x) : (x : \gamma_1) \to (\gamma_2\ x) \\
\quad \wedge\ x : \gamma_1 \vdash \alpha_1\ x : \gamma_2\ x \cong (\alpha_3 x)\ (\alpha_2 x) : \gamma_4\ x\ (\alpha_2\ x) \\
\quad \wedge\ x : \gamma_1 \vdash \alpha_2\ x : \gamma_3\ x \cong x : \gamma_1 \\
\quad \wedge\ x : \gamma_1 \vdash \alpha_3\ x : (z : \gamma_3\ x) \to \gamma_4\ x\ z \cong (\alpha_4\ x)\ (\alpha\ x) : \gamma_6\ x\ (\alpha\ x) \\
\quad \wedge\ x : \gamma_1 \vdash \alpha_4\ x : (z : \gamma_5\ x) \to \gamma_6\ x\ z \cong \mathbb{c} : (y : \mathrm{Set}) \to y \to y
\end{array}
$$

◀

Remark 3.14. The elaboration process generates many metavariables and constraints. In particular, given a term of the form $h\,\vec{e}^{\,n}$, n metavariables with corresponding constraints will be generated, which will then need to be solved. In Section 5.5.2, we explain the shortcuts that we take to reduce the number of constraints when elaborating neutral terms.

We consider a proof of the correctness of the elaboration out of the scope of this work, as the unification approach described in Chapter 4 (unifying without order) does not depend on the particular details of the elaboration.

Once we have elaborated the type checking problem into an extended signature and a set of basic constraints, finding a solution to all the constraints will give us values for each metavariable; in particular, to those corresponding to the holes in the original type checking problem.

The solution to the unification problem in Example 3.13 is:

$$
\begin{array}{lcll}
\Theta \stackrel{\text{def}}{=} \Sigma, & & & \\
\quad \alpha_0 & := & \lambda x.\mathtt{c}\,\mathbb{A}\,x & : \mathbb{A} \to \mathbb{A}, \\
\quad \gamma_1 & := & \mathbb{A} & : \mathrm{Set}, \\
\quad \gamma_2 & := & \lambda x.\mathbb{A} & : \mathbb{A} \to \mathrm{Set} \\
\quad \alpha_1 & := & \lambda x.\mathtt{c}\,\mathbb{A}\,x & : \mathbb{A} \to \mathbb{A} \\
\quad \gamma_3 & := & \lambda x.\mathbb{A} & : \mathbb{A} \to \mathrm{Set}, \\
\quad \gamma_4 & := & \lambda x.\lambda y.\mathbb{A} & : \mathbb{A} \to \mathbb{A} \to \mathrm{Set} \\
\quad \alpha_2 & := & \lambda x.x & : \mathbb{A} \to \mathbb{A} \\
\quad \alpha_3 & := & \lambda x.\mathtt{c}\,\mathbb{A} & : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \\
\quad \gamma_5 & := & \lambda x.\mathrm{Set} & : \mathbb{A} \to \mathrm{Set}, \\
\quad \gamma_6 & := & \lambda x.\lambda z.(z \to z) & : \mathbb{A} \to (z : \mathrm{Set}) \to \mathrm{Set} \\
\quad \alpha & := & \lambda x.\mathbb{A} & : \mathbb{A} \to \mathrm{Set} \\
\quad \alpha_4 & := & \lambda x.\mathtt{c} & : \mathbb{A} \to (z : \mathrm{Set}) \to z \to z
\end{array}
$$

Note how, if we substitute α for $\lambda x.\mathbb{A}$ in the original term, we obtain a well-typed term: $\Sigma; \cdot \vdash \lambda x.\mathtt{c}\,\mathbb{A}\,x : \mathbb{A} \to \mathbb{A}$. The term that the user omitted is $\mathbb{A}$.

Finding a solution to the above constraints is an instance of higher-order unification. A review of how this problem has been approached both historically and in the context of dependent types follows. The specific approach we adopt is explained in Chapter 4.

Note that we are interested in solutions to the unification problem only when they are unique. What it means for two solutions to be equal is described in Definition 2.145 (equality of metasubstitutions).

Algorithm 1: ELABORATE

input : Type-checking problem $\Sigma; \Gamma \vdash^? t : A$
output : Signature Σ'
Elaborated term u
Constraints $\vec{\mathcal{C}}$

Let $\Sigma' := \Sigma$;
Let $\overrightarrow{z : T} = \Gamma$;
if $t = \alpha\, \vec{z}$ *for some* α, $\alpha \notin \text{SUPPORT}(\Sigma)$ **then**
  $\Sigma' := \Sigma', \alpha : \Pi \vec{T} A$;
  $u := \alpha\, \vec{z}$;
  $\vec{\mathcal{C}} := \Box$;
  return
Let α_0 be fresh in Σ;
$\Sigma' := \Sigma, \alpha_0 : \Pi \vec{T} A$;
$u := \alpha_0\, \vec{z}$;
switch t **do**
  case $\lambda.t_1$ *for some* t_1 **do**
    Let γ_1, γ_2 be fresh in Σ';
    $\Sigma' := \Sigma', \gamma_1 : \overrightarrow{T} \to \text{Set}, \gamma_2 : \overrightarrow{z : T} \to \gamma_1\, \vec{z} \to \text{Set}$;
    $\Sigma', u', \overrightarrow{\mathcal{C}'} := \text{ELABORATE}(\Sigma'; \Gamma, x : \gamma_1\, \vec{z} \vdash^? t_1 : \gamma_2\, \vec{z}\, x)$;
    $\vec{\mathcal{C}} := \Gamma \vdash u : A \cong \lambda.u' : (x : \gamma_1\, \vec{z}) \to \gamma_2\, \vec{z}\, x \ \wedge\ \overrightarrow{\mathcal{C}'}$;
    return
  case $f\, t_1$ *for some* f *and* t_1 **do**
    Let γ_1, γ_2 be fresh in Σ;
    $\Sigma' := \Sigma', \gamma_1 : \overrightarrow{T} \to \text{Set}, \gamma_2 : \overrightarrow{z : T} \to \gamma_1\, \vec{z} \to \text{Set}$;
    $\Sigma', u_1, \overrightarrow{\mathcal{C}'} := \text{ELABORATE}(\Sigma'; \Gamma \vdash^? t_1 : \gamma_1\, \vec{z})$;
    $\Sigma', u_2, \overrightarrow{\mathcal{C}''} := \text{ELABORATE}(\Sigma'; \Gamma \vdash^? f : (x : \gamma_1\, \vec{z}) \to \gamma_2\, \vec{z}\, x)$;
    $\vec{\mathcal{C}} := \Gamma \vdash u : A \cong (u_2\ @\ u_1) : \gamma_2\, \vec{z}\, u_1 \ \wedge\ \overrightarrow{\mathcal{C}'} \ \wedge\ \overrightarrow{\mathcal{C}''}$;
    return
  case $\mathbb{b}$ *for some* $\mathbb{b}$ *with* $\mathbb{b} : B \in \Sigma$ **do**
    $\vec{\mathcal{C}} := \Gamma \vdash u : A \cong \mathbb{b} : B$;
    return

3.2 Higher-order unification

The problems of first-order and higher-order unification were initially of interest because of the immediate application to theorem proving over first-order (respectively higher-order) logic.

Unification can be stated for any language with a notion of equality between terms. For instance, unification can be considered in the context of simply-typed λ-terms, where equality is given by the η and β-rules.

Notation (Terms and constraints). In the rest of this chapter we consider unification constraints of the form $\Gamma \vdash t \approx u : T$, which are satisfied in a signature Σ when $\Sigma; \Gamma \vdash t \equiv u : T$,

Unification is first-order when the types of metavariables are all first-order (e.g. $\alpha : \mathbb{A}_1 \to ... \to \mathbb{A}_n$, with all $\mathbb{A}_i$ atomic types). First-order unification was independently shown to be decidable by Guard [24] and Robinson [53].

Example 3.15 (First-order problem)**.** Consider the following first-order problem:

$$\alpha : \mathrm{Set}, \mathbb{F} : \mathrm{Set} \to \mathrm{Set}, \mathbb{a} : \mathrm{Set}; \cdot \vdash \mathbb{F}\ \alpha \approx \mathbb{F}\ \mathbb{a} : \mathrm{Set}$$

The problem has the solution $\Theta \stackrel{\text{def}}{=} \mathbb{F} : \mathrm{Set} \to \mathrm{Set}, \mathbb{a} : \mathrm{Set}, \alpha := \mathbb{a} : \mathrm{Set}$. Indeed, replacing all occurrences of α by $\mathbb{a}$ in $\mathbb{F}\ \alpha$ gives $\mathbb{F}\ \mathbb{a}$. ◀

Unification is higher-order when the types of metavariables are higher-order, that is, the arguments of a metavariable may in turn be of function type. This more general version of unification is the one we need to elaborate our dependently-typed language, where metavariables of function type may appear in types; and thus the unifier must be aware of the workings of β-reduction.

Example 3.16 (Higher-order problem)**.** Consider the following higher-order problem:

$$\alpha : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}, \mathbb{F} : \mathrm{Set} \to \mathrm{Set}; x : \mathrm{Set}, y : \mathrm{Set} \vdash \alpha\ x\ y \approx \mathbb{F}\ x : \mathrm{Set}$$

Note that α is of function type, and in fact occurs at the head of a term $(\alpha\ x\ y)$. The problem has the solution $\Theta \stackrel{\text{def}}{=} \mathbb{F} : \mathrm{Set} \to \mathrm{Set}, \alpha := \lambda x.\lambda y.\mathbb{F}\ x : \mathrm{Set}$. Finding this solution requires accounting for the fact that substituting $\lambda x.\lambda y.\mathbb{F}\ x$ for α is not a purely syntactic substitution, which would give the (ungrammatical) term $\lceil(\lambda x.\lambda y.\mathbb{F}\ x)\ x\ y\rfloor$; but instead also involves a computation step (resulting in $\mathbb{F}\ x$). ◀

3.3 (Un)decidability of higher order unification

Deciding whether a solution to a higher-order unification problem exists is undecidable, as shown by Huet [28]. Huet shows the undecidability of higher-order unification by encoding the (undecidable) Post correspondence problem [50] as a higher-order unification problem. In this section we sketch Huet's argument, adapting the notation to our setting, and generalizing it to show

the undecidability of not only the existence of a solution, but also of whether a given solution is unique.

Given a Post correspondence problem for words $\{a_i, b_i\}_{i=1}^n$ over the alphabet $\{\mathsf{x}, \mathsf{y}\}$, consider the corresponding higher-order unification problem:

$$\begin{aligned}
&\mathbb{A} : \mathrm{Set},\\
&\alpha : (\mathbb{A} \to \mathbb{A}) \to \ldots_n \to (\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A}),\\
&\beta : (\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A})\,;\\
&x, y : \mathbb{A} \to \mathbb{A} \vdash \alpha\, \widetilde{a_1} \ldots \widetilde{a_n} \approx \alpha\, \widetilde{b_1} \ldots \widetilde{b_n} : \mathbb{A}\\
&x : \mathbb{A} \to \mathbb{A} \vdash \alpha\, x \ldots_n x \approx \lambda z.x\,(\beta\, x\, z) : \mathbb{A} \to \mathbb{A}
\end{aligned}$$

The tilde $\widetilde{\ldots}$ over a word denotes its encoding as a list of variables. For example, the encoding of the word xyx is $\widetilde{\mathsf{xyx}} = \lambda z.(x\,(y\,(x\,z)))$.

By a combinatorial argument, because of the type of metavariable α, in all well-typed solutions to the problem, the body of α, when fully η-expanded, is of the form $\lambda w_1 \ldots \lambda w_n.\lambda z.\, w_{i_1}(\ldots(w_{i_p}\, z))$. The indices $i_1, \ldots, i_p$ encode one candidate solution to the Post correspondence problem. Given a solution, the first constraint ensures that concatenating the words $a_{i_1}, \ldots, a_{i_p}$ will yield the same result as concatenating $b_{i_1}, \ldots, b_{i_p}$. The second constraint of the problem guarantees that $p > 0$.

Conversely, any solution to the Post correspondence problem can be translated into a solution to the unification problem. Therefore, deciding whether there is a solution to an instance of the Post correspondence problem corresponds to deciding whether the corresponding unification problem has a solution.

Note that whether a unique solution exists is also undecidable. To show this, observe that, if we drop the metavariable β and the second equation, the problem always has at least one solution, namely, $\Theta \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \alpha := \lambda w_1 \ldots \lambda w_n \lambda z.\, z : (\mathbb{A} \to \mathbb{A}) \to \ldots_n \to (\mathbb{A} \to \mathbb{A}) \to \mathbb{A} \to \mathbb{A}$. This solution is unique if and only if the matching Post correspondence problem has no solution.

3.4 Miller pattern unification

Even if higher-order unification is in general not decidable, some particular instances of this problem can be solved.

Example 3.17 (Solvable higher-order unification problem)**.** Consider the following higher-order unification problem:

$$\begin{aligned}
&\mathbb{A} : \mathrm{Set},\\
&\alpha : (\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A}) \to \mathbb{A} \to \mathbb{A};\\
&u : \mathbb{A} \to \mathbb{A}, v : \mathbb{A} \to \mathbb{A} \vdash \alpha\, u\, v \approx \lambda z.\, v\,(v\,(u\,z))
\end{aligned}$$

This problem has the following solution:

$$\Theta = \mathbb{A} : \mathrm{Set}, \alpha := (\lambda u.\lambda v.\lambda s.\, v\,(v\,(u\,s))) : (\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A}) \to \mathbb{A} \to \mathbb{A}$$

◀

Note that all the metavariables in Example 3.17 are applied only to distinct variables. This means that the unification problem is in the pattern fragment as described by Miller [41]. Problems in this fragment always have a unique, most-general solution (see Lemma 4.36, Miller's pattern condition). Paraphrasing Gundry and McBride [25], the behaviour of a metavariable is fully characterized by its application to distinct variables.

3.5 Dynamic pattern unification

It may be the case that, in a problem, some but not all constraints are in the pattern fragment. For example, the following problem is not entirely in the pattern fragment, because the first argument to α, $(\beta\, x\, y)$, is not a variable.

$$\begin{aligned}&\mathbb{A} : \mathrm{Set}, \alpha : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \beta : \mathbb{A} \to \mathbb{A} \to \mathbb{A};\\ &x, y : \mathbb{A} \vdash \alpha\, (\beta\, x\, y)\, y \approx y : \mathbb{A}\\ &x, y : \mathbb{A} \vdash \beta\, x\, y \approx x : \mathbb{A}\end{aligned}$$

However, the second constraint is in the pattern fragment. This means that, in any solution to the problem, β is necessarily instantiated to the term $\lambda x.\lambda y.x$ (or a term judgmentally equal to said term). Following this assignment, the first constraint becomes $x : \mathbb{A}, y : \mathbb{A} \vdash \alpha\, x\, y \approx y : \mathbb{A}$, which is in the pattern fragment, and necessitates $\lceil x : \mathbb{A}, y : \mathbb{A} \vdash \alpha\, x\, y \equiv y : \mathbb{A} \rfloor$. This means that the unique solution is $\mathbb{A} : \mathrm{Set}, \alpha := \lambda x.\, \lambda y.\, x : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \beta := \lambda x.\lambda y.y : \mathbb{A} \to \mathbb{A} \to \mathbb{A}$.

As Michaylov and Pfenning [40] observed, by postponing certain constraints, we can solve problems which are not strictly in the pattern fragment.

Furthermore, by using *pruning* (see Section 4.5.9), it is possible to use information in one constraint to remove certain arguments from a metavariable in another constraint, thus bringing more constraints into the pattern fragment.

The use of constraint postponement and pruning constitutes dynamic pattern unification and is treated rigorously by Reed [52] in the context of dependent types.

3.6 Extension to product types

So far we have discussed unification for the simply typed λ-calculus. The pattern fragment can be extended to accommodate terms with product types ($\times$), including pairs $\langle t, u\rangle$, projections ($.\pi_1$, $.\pi_2$) and the corresponding η-equality.

For example, the following problem is not in the pattern fragment, because the arguments are projected variables:

$$\begin{aligned}&\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set},\\ &\alpha : \mathbb{A} \to \mathbb{B} \to \mathbb{B};\\ &x : A \times B \vdash \alpha\, (x\, .\pi_1)\, (x\, .\pi_2) \approx (x\, .\pi_2) : \mathbb{B}\end{aligned}$$

Duggan [19] observes that this is not a problem, as long as the projections applied to a given variable are distinct.

In fact, we can obtain an equivalent problem which is in the pattern fragment by "currying" the context variable $x : \mathbb{A} \times \mathbb{B}$ into two separate variables $x_1 : \mathbb{A}$ and $x_2 : \mathbb{B}$, with $x = \langle x_1, x_2 \rangle$.

$$\begin{aligned}&\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set},\\ &\alpha : \mathbb{A} \to \mathbb{B} \to \mathbb{B};\\ &x_1 : \mathbb{A}, x_2 : \mathbb{B} \vdash \alpha\, x_1\, x_2 \approx x_2 : \mathbb{B}\end{aligned}$$

Similarly, the following problem is also not in the pattern fragment, because the argument is not a single variable, but a pair:

$$\begin{aligned}&\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set},\\ &\alpha : \mathbb{A} \times \mathbb{B} \to \mathbb{B};\\ &x : \mathbb{A}, y : \mathbb{B} \vdash \alpha\, \langle x, y \rangle \approx y : \mathbb{B}\end{aligned}$$

However, because all the components of the pair *are* distinct variables, this does not preclude a unique solution either. In this case, we can curry the first argument of α. The problem then becomes the following, which is in the pattern fragment:

$$\begin{aligned}&\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set},\\ &\alpha' : \mathbb{A} \to \mathbb{B} \to \mathbb{B},\\ &\alpha := \lambda x.\alpha'\ (x\ .\pi_1)\ (x\ .\pi_2) : \mathbb{A} \times \mathbb{B} \to \mathbb{B};\\ &x : \mathbb{A}, y : \mathbb{B} \vdash \alpha'\, x\, y \approx y : \mathbb{B}\end{aligned}$$

Because of η-equality, a term of record type is determined by its projections, therefore the set of possible solutions for α stays unchanged after this transformation. The resulting constraint implies that any solution must fulfill $x : \mathbb{A}, y : \mathbb{B} \vdash \alpha'\, x\, y \equiv y : \mathbb{B}$. Because x and y are distinct, then, for any solution Θ, $\Theta; \cdot \vdash \alpha' \equiv \lambda x.\lambda y.y : \mathbb{A} \to \mathbb{B} \to \mathbb{B}$ (see Lemma 4.36, Miller's pattern condition). In fact, the new problem has the following unique solution:

$$\Theta \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set}, \alpha := \lambda x.(x\ .\pi_2) : \mathbb{A} \times \mathbb{B} \to \mathbb{B}, \alpha' := \lambda x.\lambda y.y : \mathbb{A} \to \mathbb{B} \to \mathbb{B}$$

Therefore the original problem has the unique solution $\Theta_{\{\alpha\}} = (\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set}, \alpha := \lambda x.\ x.\pi_2 : \mathbb{A} \times \mathbb{B} \to \mathbb{B})$.

Abel and Pientka [3] extensively elaborate on this insight to extend dynamic pattern unification for a theory containing both dependent function (Π) and record (Σ) types.

3.7 Interleaving type checking with unification

In dependent type checking with metavariables, the type of all terms is not known in the beginning, as it may depend on uninstantiated metavariables.

At the same time, some unification problems require awareness of the types of terms in order to be solved. For example, see Section 3.6 (extension to product types).

Approaches for interleaving type checking with unification must deal with the fact that some terms might not be well-typed until some constraints are solved.

Reed [52] and Abel and Pientka [3] use a formulation of typing modulo constraints. This formulation relies on the fact that in their case, unsolvable constraints do not jeopardize the correctness of normalization. More specifically, in their system, metavariables in types can only appear as parameters to atomic type families.

In section 3.4 of Norell's thesis [44], an example is given where a non-recursive, yet non-terminating term can be typed. This failure to prevent non-normalizing terms leads to non-termination in the type checker. [1]

For Agda, where metavariables may appear anywhere in a type, Norell and Coquand [44] designed the system in such a way that certain subterms are blocked from being reduced until the constraints ensuring their well-typedness are solved. This restriction is quite robust in practice, but one can still create ill-typed terms under certain circumstances [33].

In the elaboration algorithm described in Section 3.1 (from type checking to unification), subterms which cannot be immediately type-checked are replaced by metavariables of the appropriate type. These metavariables take a role similar to Norell and Coquand's guarded constants [44]: in the same way that a guarded constant prevents a term from normalizing until a constraint is solved, a metavariable effectively prevents a term from normalizing until the metavariable is instantiated.

3.8 The Π problem

A unification problem can sometimes be solved by breaking down the constraints into smaller ones, until they can either be dismissed as trivial, or solved by instantiating a metavariable.

A difficulty with solving heterogeneous constraints appears when attempting to simplify a constraint involving binders, like Π, Σ, or λ.

For example, consider a basic constraint which unifies two Π-types: $\Pi(x : A)B$ and $\Pi(x : A')B'$. In order to obtain constraints that we can be solved, we may want to simplify ($\rightsquigarrow$) the given constraint into two new constraints, one that unifies A and A', and another which unifies B and B':

$$\begin{aligned}\Sigma;\Gamma \vdash \Pi(x : A)B : \mathrm{Set} \cong \Pi(x : A')B' : \mathrm{Set} \;\rightsquigarrow\\ \Sigma;\Gamma \vdash A : \mathrm{Set} \cong A' : \mathrm{Set} \;\wedge\\ \Gamma, x : \lceil ? \rfloor \vdash B \cong B' : \mathrm{Set}\end{aligned}$$

The question is, in the second constraint, what the type $\lceil ? \rfloor$ of the new variable x should be. If the type is A, then the right side of the constraint may not be well-formed; mutatis mutandis for A'.

[1] In Agda and Coq, non-terminating recursive definitions are disallowed by a termination checker.

3.9 Strictly ordered, homogeneous constraints

The approach suggested by Mazzoli and Abel [36] sidesteps the Π problem by having both sides of the constraints have the same type. Such constraints are called *homogeneous*.

Definition 3.18 (Homogeneous constraint)**.** A homogeneous constraint is of the form $\Sigma;\Gamma \vdash t \approx u : A$. Such a constraint is well-formed if and only if $\Sigma;\Gamma \vdash t : A$ and $\Sigma;\Gamma \vdash u : A$.

In their implementation, each well-formed basic constraint $\Sigma;\Gamma \vdash t : A \cong u : B$ is translated into two homogeneous internal constraints; namely $\Sigma;\Gamma \vdash A \approx B : \mathrm{Set}$ and $\Sigma;\Gamma \vdash t \approx u : A$. Note that the second constraint is not necessarily well-formed, because it might not be the case that $\Sigma;\Gamma \vdash u : A$. Solving the second constraint before the first constraint is solved could lead to inconsistencies [4].

However, once an extension $\Sigma' \sqsupseteq \Sigma$ is found such that $\Sigma';\Gamma \vdash A \equiv B : \mathrm{Set}$, then the second constraint will be well-formed in this extended signature: $\Sigma';\Gamma \vdash t \approx u : A$, and can be solved.

Unifying types and then terms sequentially help ensure well-formedness of constraints throughout the algorithm, but, at the same time, prevents using information which could help unify types.

Example 3.19 (Limitations of sequential solving)**.** Given metavariables $\alpha : \mathrm{Set} \to \mathrm{Set}$ and $\beta : \alpha\,\mathrm{Bool}$, consider the constraint $x : \mathrm{Set} \vdash \langle \alpha\, x, \mathrm{true}\rangle : \mathrm{Set} \times \mathrm{Bool} \approx \langle \mathrm{Bool}, \beta\rangle : \mathrm{Set} \times \alpha\,\mathrm{Bool}$. This constraint has the unique solution $\alpha := \lambda x.\mathrm{Bool} : \mathrm{Set}, \beta := \mathrm{true} : \mathrm{Bool}$.

A strictly ordered approached based on homogeneous constraints would first solve $x : \mathrm{Set} \vdash \mathrm{Set} \times \mathrm{Bool} \approx \mathrm{Set} \times \alpha\,\mathrm{Bool} : \mathrm{Set}$, and once it is solved (and only then), attempt $x : \mathrm{Set} \vdash \langle \alpha\, x, \mathrm{true}\rangle \approx \langle \mathrm{Bool}, \beta\rangle : \mathrm{Set} \times \mathrm{Bool}$. However, the first constraint has two possible, incompatible solutions: $\alpha := \lambda x.\mathrm{Bool} : \mathrm{Set}$ and $\alpha := \lambda x.x : \mathrm{Set}$.

In order to determine that the first alternative is the right one, we need information from the second constraint. However, this information is inaccessible until the first constraint is solved. ◀

3.10 Heterogeneous constraints using twin variables

Gundry and McBride [25] propose considering, as internal constraints, constraints with a twin context: that is, a context where each variable has two possible types.

$$
\begin{array}{lcll}
\Gamma & ::= & \cdot & \text{empty twin context} \\
 & | & \Gamma, x : A & \text{variable of simple type} \\
 & | & \Gamma, \hat{x} : A_1 \ddagger A_2 & \text{variable of twin type}
\end{array}
$$

Each occurrence of the variable in the rest of the constraint is annotated with either an acute (´) or a grave (`) accent, depending on whether that variable should have the left or the right type, respectively.

In our particular type system, this would correspond to extending the syntax of terms in this way:

$$\begin{array}{rcll} h & ::= & \ldots & \textit{neutral heads} \\ & | & \acute{x} & \text{left twin variable} \\ & | & \grave{x} & \text{right twin variable} \end{array}$$

The rule VAR is replace by the following two rules:

$$\frac{}{\Gamma, x : A \ddagger A', \Delta \vdash \acute{x} \Rightarrow A}\ \textsc{var-left}$$

$$\frac{}{\Gamma, x : A \ddagger A', \Delta \vdash \grave{x} \Rightarrow A'}\ \textsc{var-right}$$

In this new setting, the basic constraint from Section 3.8 is simplified in this manner, where the type $B[x \mapsto \acute{x}]$ (respectively $B'[x \mapsto \grave{x}]$) is the result of syntactically replacing each occurrence of x in B by $\acute{x}$ (respectively, each occurrence of x in B' by $\grave{x}$):

$$\begin{aligned} \Sigma; \Gamma \vdash \Pi(x : A)B : \mathrm{Set} \cong \Pi(x : A')B' : \mathrm{Set} \ &\rightsquigarrow \\ \Sigma; \Gamma \vdash A : \mathrm{Set} \cong u : A' : \mathrm{Set} \ &\wedge \\ \Sigma; \Gamma, x : A \ddagger A' \vdash B[x \mapsto \acute{x}] : \mathrm{Set} \cong B'[x \mapsto \grave{x}] : \mathrm{Set} & \end{aligned}$$

In Chapter 4 we build on this approach to implement a unification algorithm.

Chapter 4

Unifying without order

In this chapter we present an unordered, heterogeneous approach to unification.

Our goal is to specify an algorithm that can be easily implemented for an existing dependent type checker, such as Agda. Our approach builds on Gundry and McBride's [25]. We both simplify and extend their unification algorithm to make it easier for us to implement side-by-side with Mazzoli et al.'s prototype type checker [37].

There are three main differences with respect to Gundry and McBride's [25] approach:

- In Gundry and McBride's [25] approach, variables on any side of the constraint may refer to either side of the context, depending on an annotation which is added to the variable. See Section 4.1 (two-sided internal constraints) for more details.

  These twin variable annotations would eventually need to be removed in case of successful unification, impacting performance. In case of error, they would need to be displayed to the user, possibly resulting in confusion. In our approach, variables on the left or right side of the constraint may only refer to the left or right side of the context, respectively; thus rendering twin variable annotations superfluous.

- A constraint can be deemed solved even before the both sides of the context or the types are equal, thanks to a more general notion of equality (see Definition 4.12, heterogeneous equality).

  This allows for a syntactic equality check (see Rule schema 1, syntactic equality) which only checks the terms. This way constraints where the terms on both sides are syntactically equal can be solved as directly as in a homogeneous setting. This syntactic equality check can lead to improved performance in some cases, as shown in Section 5.6.1.

- All rules can be applied to terms which are not in δ-normal form. Excessive normalization may affect both readability [2] and performance [12]. Being able to handle partially-normalized terms may be a useful tool in order to obtain a well-performing type checker.

4.1 Two-sided internal constraints

The constraints in the problem are all basic constraints (Definition 3.7). However, in order to solve the Π problem (Section 3.8) without enforcing a strict ordering of constraints (Section 3.9), we use Gundry and McBride's [25] notion of contexts, where each variable may have two different types: one for each side of the constraint.

Definition 4.1 (Twin contexts)**.** A twin context $\Gamma_1 ‡ \Gamma_2$ is a pair of contexts Γ_1 and Γ_2, such that $|\Gamma_1| = |\Gamma_2|$.

A twin context is well-formed if each of the sides Γ_1 and Γ_2 are well-formed. The precondition $|\Gamma_1| = |\Gamma_2|$ follows from the use of the syntax $\Gamma_1 ‡ \Gamma_2$. For the sake of clarity, we reiterate it in the derivation rule.

$$\frac{\Sigma \vdash \Gamma_1\,\mathbf{ctx} \qquad \Sigma \vdash \Gamma_2\,\mathbf{ctx} \qquad (|\Gamma_1| = |\Gamma_2|)}{\Sigma \vdash \Gamma_1 ‡ \Gamma_2\,\mathbf{wf}}$$

Notation (Twin context). Given a well-formed twin context $\Gamma_1 ‡ \Gamma_2$, it can also be viewed as a context where each variable has two types:

$$\begin{array}{rcll} \Gamma_1 ‡ \Gamma_2 & ::= & \cdot & \text{empty twin context} \\ & | & \Gamma_1 ‡ \Gamma_2, A_1 ‡ A_2 & \text{variable of twin type} \end{array}$$

Twin contexts can be concatenated by concatenating each of their sides.

Notation (Twin context concatenation). The concatenation of two twin-contexts $\Gamma_1 ‡ \Gamma_2$ and $\Delta_1 ‡ \Delta_2$ is written $(\Gamma_1 ‡ \Gamma_2), (\Delta_1 ‡ \Delta_2)$, and corresponds to the twin context $(\Gamma_1, \Delta_1) ‡ (\Gamma_2, \Delta_2)$. Writing $(\Gamma_1 ‡ \Gamma_2), (\Delta_1 ‡ \Delta_2)$ instead of $(\Gamma_1, \Delta_1) ‡ (\Gamma_2, \Delta_2)$ indicates that $|\Gamma_1| = |\Gamma_2|$ and $|\Delta_1| = |\Delta_2|$.

Internal constraints extend the notion of basic constraint by replacing the context with a twin context. In contrast to Gundry and McBride [25], we do not extend the syntax of terms with twin variables (see Section 3.10). Instead, the variables on the left (or right) side of the constraint only reference those on the left (respectively right) side of the context.

Definition 4.2 (Well-formed internal constraint)**.** Given a twin context $\Gamma_1 ‡ \Gamma_2$, two terms t and u, and two types A and B, an internal constraint is a 5-tuple of the form $\Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A ‡ B$.

An internal constraint $\mathcal{C}$ is well-formed in a signature Σ (written $\Sigma; \mathcal{C}\,\mathbf{wf}$) if and only if each side of the constraint is well-typed in the corresponding side of the context.

$$\frac{\Sigma \vdash \Gamma_1 ‡ \Gamma_2\,\mathbf{wf} \qquad \Sigma; \Gamma_1 \vdash t : A \qquad \Sigma; \Gamma_2 \vdash u : B}{\Sigma; \Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A ‡ B\,\mathbf{wf}}$$

A unification problem is a set of constraints sharing the same signature:

Definition 4.3 (Unification problem)**.** Given a signature Σ and a sequence of constraints $\vec{\mathcal{C}}$, a unification problem is a pair of the form $\Sigma; \vec{\mathcal{C}}$, where $\vec{\mathcal{C}}$ is a vector of internal constraints.

$$\mathcal{C}, \mathcal{D}, \mathcal{E} \quad ::= \quad \Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A ‡ B \quad \text{internal constraint}$$

Notation. The vector $\vec{\mathcal{C}}$ may be understood as a conjunction of constraints; therefore we use $\wedge$ as the element separator:

$$\vec{\mathcal{C}}^n = \mathcal{C}_1 \wedge ... \wedge \mathcal{C}_n$$

An empty vector of constraints is denoted by "$\square$":

$$\vec{\mathcal{C}}^0 = \square$$

Definition 4.4 (Set of constants in a constraint or a vector of constraints: $\textsc{consts}(\mathcal{C}), \textsc{consts}(\vec{\mathcal{C}})$)**.**

$$\begin{array}{rcl} \textsc{consts}(\Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B) & = & \textsc{consts}(\Gamma_1) \cup \textsc{consts}(\Gamma_2) \\ & & \cup\, \textsc{consts}(t) \cup \textsc{consts}(u) \\ & & \cup\, \textsc{consts}(A) \cup \textsc{consts}(B) \\ \textsc{consts}(\vec{\mathcal{C}}^n) & = & \bigcup_{i=1}^{n} \textsc{consts}(\mathcal{C}_i) \end{array}$$

Definition 4.5 (Well-formed unification problem)**.** A unification problem $\Sigma; \vec{\mathcal{C}}$ is well-formed if Σ is well-formed, and each of the constraints in $\vec{\mathcal{C}}$ is well-formed in Σ.

$$\frac{\Sigma\ \mathbf{sig} \qquad \forall \mathcal{C} \in \vec{\mathcal{C}}, \Sigma; \mathcal{C}\ \mathbf{wf}}{\Sigma; \vec{\mathcal{C}}\ \mathbf{wf}}$$

Remark 4.6 (No extraneous constants in constraint). If $\Sigma; \mathcal{C}\,\mathbf{wf}$, then $\textsc{consts}(\mathcal{C}) \subseteq \textsc{decls}(\Sigma)$. Also, if $\Sigma; \vec{\mathcal{C}}\,\mathbf{wf}$, then $\textsc{consts}(\vec{\mathcal{C}}) \subseteq \textsc{decls}(\Sigma)$.

Proof. By definition Definition 4.2 (well-formed internal constraint), Definition 4.5 (well-formed unification problem), and Lemma 2.72 (no extraneous constants). $\square$

Remark 4.7 (Well-formed unification constraint is a judgment: $J = \mathcal{C}$). Given an internal constraint $\mathcal{C}$, there is a judgment J such that, for any well-formed signature Σ, $\Sigma; \mathcal{C}\,\mathbf{wf}$ if and only if $\Sigma \vdash J$; and $\textsc{consts}(J) = \textsc{consts}(\mathcal{C})$. Namely, $J = (\Gamma_1 \vdash t : A) \wedge (\Gamma_2 \vdash u : B)$.

Remark 4.8 (Well-formed unification problem is a judgment: $J = \vec{\mathcal{C}}$). Given a vector of internal constraints $\vec{\mathcal{C}}$, there is a judgment such that, for any well-formed signature Σ, we have $\Sigma; \vec{\mathcal{C}}\,\mathbf{wf}$ if and only if $\Sigma \vdash J$, and $\textsc{consts}(J)$ includes only those constants mentioned in $\vec{\mathcal{C}}$. Namely, the judgment J is the conjunction of the judgments given by Remark 4.7 (well-formed unification constraint is a judgment). If $\vec{\mathcal{C}} = \square$, then let J be a judgment that holds in any signature, such as $J \stackrel{\text{def}}{=} \cdot\,\mathbf{ctx}$.

Definition 4.9 (Solution to a constraint: $\Theta \vDash \mathcal{C}$, $\Theta \vDash \vec{\mathcal{C}}$)**.** Let Θ be a meta-substitution, and $\mathcal{C} = \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B$ be an internal constraint.

We say that Θ is a solution to $\mathcal{C}$ (written $\Theta \vDash \mathcal{C}$) iff $\Theta; \mathcal{C}\,\mathbf{wf}$, $\Theta \vdash \Gamma_1, A \equiv \Gamma_2, B\,\mathbf{ctx}$ and $\Theta; \Gamma_1 \vdash t \equiv u : A$ (or, equivalently, $\Theta \vdash \Gamma_1 \equiv \Gamma_2\,\mathbf{ctx}$, $\Theta; \Gamma_1 \vdash A \equiv B\ \mathbf{type}$ and $\Theta; \Gamma_1 \vdash t \equiv u : A$).

If $\vec{\mathcal{C}}$ is a vector of constraints, we say that $\Theta \vDash \vec{\mathcal{C}}$ if, for each $\mathcal{C} \in \vec{\mathcal{C}}$, $\Theta \vDash \mathcal{C}$.

Remark 4.10 (Solution to a constraint as a judgment). Let $\mathcal{C}$ be a constraint, $\mathcal{C} = \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B$. Then there is a judgment J, $J = (\Gamma_1, A \equiv \Gamma_2, B\,\mathbf{ctx}) \wedge (\Gamma_1 \vdash t \equiv u : A)$ such that $\textsc{consts}(J) = \textsc{consts}(\mathcal{C})$ and, for any well-formed metasubstitution $\Theta\,\mathbf{wf}$, $\Theta \vDash \mathcal{C}$ if and only if $\Theta \vdash J$.

A solution to a problem is a metasubstitution which is compatible with the problem signature, such that each of the problem constraints is satisfied in the metasubstitution.

Definition 4.11 (Solution to a unification problem: $\Theta \vDash \Sigma; \vec{\mathcal{C}}$)**.** Let $\Sigma; \vec{\mathcal{C}}$ be a well-formed unification problem, and Θ a well-formed metasubstitution. We say that Θ is a solution to $\Sigma; \vec{\mathcal{C}}$ (written $\Theta \vDash \Sigma; \vec{\mathcal{C}}$) if we have $\Theta \vDash \Sigma$ and $\Theta \vDash \vec{\mathcal{C}}$.

4.2 Heterogeneous equality

A key point in the flexibility of Gundry and McBride's approach [25] is the possibility of partially solving a constraint before the types of both sides have been deemed equal. For instance, one can unify the first projections of two pairs as long as the types of the first projections are equal, and then use this information to unify the types of the second projections.

We take this idea a step further, and define an equality for terms which may have distinct types. This equality allows us to implement the type-agnostic Rule schema 1 (syntactic equality), which results in a significant impact in performance for some examples (Section 5.6.1).

Definition 4.12 (Heterogeneous equality: $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$)**.** Two terms $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Delta \vdash u : B$ are heterogeneously equal (written $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$), iff there exists a term v such that (1a) $\Sigma; \Gamma \vdash t \equiv v : A$, (1b) $\Sigma; \Delta \vdash u \equiv v : B$, and (2) $\textsc{fv}(v) \subseteq \textsc{fv}(t) \cap \textsc{fv}(u)$.

Given such a witness v, we can write $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv\{v\}\equiv u : A \ddagger B$.

Condition (2) ensures that the witness v only uses those variables that are used by both t and u. This is helpful when extending the heterogeneous equality to whole contexts, as done in Definition 4.37 (heterogeneously equal contexts modulo variables) and Lemma 4.38 (typing in heterogeneously equal contexts).

The notion of an equality with intermediate witness is inspired by the ternary equality relation due to Gundry and McBride [25], but different in two key aspects: the types A and B are not necessarily equal, and the witness v is not necessarily a fully-normalized term.

The heterogeneous notion of equality generalizes the judgmental equality: In other words, if the contexts and types on both sides are equal, then the two notions are equivalent:

Lemma 4.13 (Homogenization)**.** *Assume that* $\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2\,\mathbf{ctx}$. *Then, we have* $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \equiv u : A_1 \ddagger A_2$ *iff* $\Sigma; \Gamma_1 \vdash t \equiv u : A_1$.

Proof. $\Rightarrow$ By assumption, $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \equiv\{v\}\equiv u : A_1 \ddagger A_2$

By Lemma 2.63 (preservation of judgments by type conversion), $\Sigma; \Gamma_1 \ddagger \Gamma_1 \vdash t \equiv\{v\}\equiv u : A_1 \ddagger A_1$. By transitivity and symmetry of homogeneous equality, $\Sigma; \Gamma_1 \vdash t \equiv u : A_1$.

⇐ By Postulate 14 (existence of a common reduct), we have v such that $\Sigma; \Gamma_1 \vdash t \longrightarrow^\star_{\delta\eta} v : A_1$, $\Sigma; \Gamma_1 \vdash u \longrightarrow^\star_{\delta\eta} v : A_1$. By Remark 2.43 (free variables of δη-reduct), $\text{FV}(v) \subseteq \text{FV}(t) \cap \text{FV}(u)$. By Lemma 2.86 (equality of δη-reduct), $\Sigma; \Gamma_1 \vdash t \equiv v : A_1$ and $\Sigma; \Gamma_1 \vdash u \equiv v : A_1$.

Because $\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2$ **ctx**, by Lemma 2.63 (preservation of judgments by type conversion), $\Sigma; \Gamma_2 \vdash u \equiv v : A_2$.

By Definition 4.12 (heterogeneous equality), $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \equiv\{v\}\equiv u : A_1 \ddagger A_2$.

□

Example 4.14 shows that the heterogeneous equality is strictly stronger than the judgmental equality of the underlying theory:

Example 4.14 (Heterogeneous equality)**.**

$$\begin{array}{l} \mathbb{A} : \text{Set}, \mathbb{o} : \mathbb{A}, \alpha : \mathbb{A} \to \text{Set}; \\ x : \mathbb{A} \to \mathbb{A} \ddagger (\alpha\, \mathbb{o}), z : (\alpha\, \mathbb{o}) \ddagger \mathbb{A} \to \mathbb{A} \\ \vdash \\ \langle x, \lambda y.z\, y \rangle \equiv\{\langle x, z \rangle\}\equiv \langle \lambda y.x\, y, z \rangle : \\ (\alpha\, \mathbb{o} \times (\mathbb{A} \to \mathbb{A})) \ddagger ((\mathbb{A} \to \mathbb{A}) \times \alpha\, \mathbb{o}) \end{array}$$

Note that each side of the heterogeneous equality is equal to the witness ($\langle x, z \rangle$) but both sides are not judgmentally equal to each other at either of their respective types. ◀

Like the judgmental equality, the heterogeneous equality is reflexive and symmetric relation:

Remark 4.15 (Reflexivity of heterogeneous equality). Heterogeneous equality is reflexive. That is, given $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Delta \vdash t : B$, we have $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv\{t\}\equiv t : A \ddagger B$ (even if $\Gamma \neq \Delta$).

Remark 4.16 (Symmetry of heterogeneous equality). Heterogeneous equality is symmetric. That is, given $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv\{v\}\equiv u : A \ddagger B$, we also have $\Sigma; \Delta \ddagger \Gamma \vdash u \equiv\{v\}\equiv t : B \ddagger A$.

For our development, we are not concerned with whether the heterogeneous equality is transitive.

The heterogeneous equality is used to define when a constraint is *satisfied* in a given signature.

Definition 4.17 (Constraint satisfaction: $\Sigma \models\!\!\approx \mathcal{C}$, $\Sigma \models\!\!\approx \vec{\mathcal{C}}$)**.** Let Σ be a signature, and $\mathcal{C}$ an internal constraint, $\mathcal{C} = \Gamma \ddagger \Delta \vdash t \approx u : A \ddagger B$. We say that $\mathcal{C}$ is satisfied in Σ (written $\Sigma \models\!\!\approx \mathcal{C}$), if $\Sigma; \mathcal{C}$ **wf** and the two sides of the constraint are heterogeneously equal. That is, $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$.

We say that the constraints $\vec{\mathcal{C}}$ are satisfied in signature Σ (written $\Sigma \models\!\!\approx \vec{\mathcal{C}}$), if, Σ is well-formed and for each $\mathcal{C} \in \vec{\mathcal{C}}$, $\Sigma \models\!\!\approx \mathcal{C}$.

Remark (Relationship between constraint solution and constraint satisfaction). Constraint satisfaction is a weaker notion than Definition 4.9 (solution to a constraint). If $\Theta \vDash \vec{\mathcal{C}}$, then, in particular, $\Theta \models\!\!\approx \vec{\mathcal{C}}$.

When we can go in the other direction (that is, $\Theta \models\!\!\approx \vec{\mathcal{C}}$ implies $\Theta \vDash \vec{\mathcal{C}}$), we say that $\vec{\mathcal{C}}$ is an essentially homogeneous set of constraints. That is, even if each constraint is not necessarily homogeneous (e.g. $\Gamma_1 \ddagger \Gamma_2 \vdash t \equiv u : A_1 \ddagger A_2 \in \vec{\mathcal{C}}$ with $\Gamma_1 \neq \Gamma_2$ and/or $A_1 \neq A_2$) both sides of the context and of the type are equal in any solution Θ to the problem (i.e. $\Theta \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2\ \mathbf{ctx}$):

Definition 4.18 (Essentially homogeneous set of constraints)**.** Let $\vec{\mathcal{C}}$ be a vector of constraints. We say that $\vec{\mathcal{C}}$ is an essentially homogeneous set of constraints iff for every metasubstitution Θ, such that $\Theta \models\!\!\approx \vec{\mathcal{C}}$ we have $\Theta \vDash \vec{\mathcal{C}}$.

Definition 4.19 (Essentially homogeneous problem)**.** A problem $\Sigma; \vec{\mathcal{C}}$ is essentially homogeneous iff $\vec{\mathcal{C}}$ is an essentially homogeneous set of constraints.

As we show in Lemma 4.23 (well-formedness of elaboration into internal constraints), all problems resulting from type checking will be essentially homogeneous.

Solving constraints is done by extending the signature. It is thus critical that extending a signature does not invalidate previously solved constraints:

Lemma 4.20 (Constraint satisfaction in extended signature)**.** *Assume* $\Sigma \sqsubseteq \Sigma'$*, and* $\Sigma \models\!\!\approx \vec{\mathcal{C}}$*. Then* $\Sigma' \models\!\!\approx \vec{\mathcal{C}}$*.*

Proof. By Definition 4.17 (constraint satisfaction), it suffices to show that, for any $\Gamma \ddagger \Delta \vdash t \approx u : A \ddagger B \in \vec{\mathcal{C}}$, if $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$, then $\Sigma'; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$.

Let $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$. By Definition 4.12 (heterogeneous equality), there exists a term v such that $\Sigma; \Gamma \vdash t \equiv v : A$, $\Sigma; \Delta \vdash u \equiv v : B$, and $\mathrm{FV}(v) \subseteq \mathrm{FV}(t) \cap \mathrm{FV}(u)$.

By Lemma 2.69 (signature weakening), $\Sigma'; \Gamma \vdash t \equiv v : A$ and $\Sigma'; \Delta \vdash u \equiv v : B$. Therefore, $\Sigma'; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$. □

Lemma 4.21 (Constraint satisfaction by compatible metasubstitution)**.** *Assume* $\Theta \vDash \Sigma$ *and* $\Sigma \models\!\!\approx \vec{\mathcal{C}}$*. Then* $\Theta \models\!\!\approx \vec{\mathcal{C}}$*.*

Proof. By Definition 4.17 (constraint satisfaction), it suffices to show that, for any $\Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$, if $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$, then $\Theta; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$.

Let $\Sigma; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$. By Definition 4.12 (heterogeneous equality), there exists a term v such that $\Sigma; \Gamma \vdash t \equiv v : A$, $\Sigma; \Delta \vdash u \equiv v : B$, and $\mathrm{FV}(v) \subseteq \mathrm{FV}(t) \cap \mathrm{FV}(u)$.

By Definition 2.125 (compatible metasubstitution), $\Theta; \Gamma \vdash t \equiv v : A$ and $\Theta; \Delta \vdash u \equiv v : B$. Therefore, $\Theta; \Gamma \ddagger \Delta \vdash t \equiv u : A \ddagger B$. □

In general terms, the unification algorithm (Algorithm 2) works by extending the signature step by step until the resulting signature satisfies all the constraints in the original problem. The solution to the original problem is obtained as a restriction of the closing metasubstitution of the resulting signature (see Theorem 4.31, correctness of unification).

4.3 From type checking to internal constraints

With the notions at hand we can give a precise description of how a type checking problem is reduced to a set of internal constraints.

Definition 4.22 (Elaboration into internal constraints)**.** Let $\Sigma; \Gamma \vdash^? t : A$ be a type checking problem, to which an elaboration algorithm is applied (Definition 3.10). Let $\Sigma'; \vec{\mathcal{C}}$ be a unification problem, where Σ' is a signature produced by the elaboration algorithm and $\vec{\mathcal{C}}$ contains, for each basic constraint of the form $\Gamma \vdash u : A \cong v : B$ produced by the elaboration algorithm, the internal constraints $\Gamma \ddagger \Gamma \vdash A \approx B : \mathrm{Set} \ddagger \mathrm{Set}$ and $\Gamma \ddagger \Gamma \vdash u \approx v : A \ddagger B$. Then we say that $\Sigma'; \vec{\mathcal{C}}$ is the elaboration of $\Sigma; \Gamma \vdash^? t : A$ into internal constraints by said elaboration algorithm.

Lemma 4.23 (Well-formedness of elaboration into internal constraints)**.** *Let $\Sigma; \Gamma \vdash^? t : A$ be a type checking problem, and $\Sigma'; \vec{\mathcal{C}}$ its elaboration into internal constraints by a well-formed elaboration algorithm.*
Then, the following hold:

Well-formedness *The problem $\Sigma'; \vec{\mathcal{C}}$ is well-formed, $\Sigma \subseteq \Sigma'$ and* $\textsc{decls}(\Sigma') \supseteq \textsc{decls}(\Sigma) \cup \textsc{metas}(t)$.

Essential homogeneity *The set of constraints $\vec{\mathcal{C}}$ is essentially homogeneous.*

Proof.

Well-formedness By construction.

Essential homogeneity Assume $\Theta \models\!\approx \vec{\mathcal{C}}$,

Let $\mathcal{C} \in \vec{\mathcal{C}}$, Let $\mathcal{C} = \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B$.

There are two possible cases:

i) $\mathcal{C} = \Gamma \ddagger \Gamma \vdash A \approx B : \mathrm{Set} \ddagger \mathrm{Set}$: Assume $\Theta; \Gamma \ddagger \Gamma \vdash A \equiv \{V\} \equiv B : \mathrm{Set} \ddagger \mathrm{Set}$. This implies $\Theta; \Gamma \vdash A \equiv V : \mathrm{Set}$ and $\Theta; \Gamma \vdash B \equiv V : \mathrm{Set}$. By Lemma 2.70 (piecewise well-formedness of typing judgments) we have $\Theta \vdash \Gamma\,\mathbf{ctx}$. Also, by the SET rule and Remark 2.15 (there is only set), $\Theta; \Gamma \vdash \mathrm{Set}\ \mathbf{type}$, which gives $\Theta \vdash \Gamma, \mathrm{Set}\,\mathbf{ctx}$. By reflexivity of context equality $\Theta \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set}\,\mathbf{ctx}$. Because $\Theta \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set}\,\mathbf{ctx}$ and $\Theta; \Gamma \ddagger \Gamma \vdash A \equiv B : \mathrm{Set} \ddagger \mathrm{Set}$, by Lemma 4.13 (homogenization), $\Theta \models \mathcal{C}$.

ii) $\mathcal{C} = \Gamma \ddagger \Gamma \vdash t \approx u : A \ddagger B$: By Definition 4.22 (elaboration into internal constraints), the elaboration algorithm produced a basic constraint $\mathcal{C} = \Gamma \vdash t : A \cong u : B$, which means there is a constraint $\mathcal{C}' \in \vec{\mathcal{C}}$, $\mathcal{C}' = \Gamma \ddagger \Gamma \vdash A \approx B : \mathrm{Set} \ddagger \mathrm{Set}$. Because $\Theta \models \vec{\mathcal{C}}$, in particular, $\Theta \models \vec{\mathcal{C}}'$. This means $\Theta; \Gamma \vdash A \equiv A_0 : \mathrm{Set}$ and $\Theta; \Gamma \vdash B \equiv B_0 : \mathrm{Set}$. By transitivity of equality, $\Theta; \Gamma \vdash A \equiv B : \mathrm{Set}$. By Remark 2.15, $\Theta; \Gamma \vdash A \equiv B\ \mathbf{type}$. By reflexivity of context equality, $\Theta \vdash \Gamma \equiv \Gamma\,\mathbf{ctx}$. By Definition 2.16, $\Theta \vdash \Gamma, A \equiv \Gamma, B\,\mathbf{ctx}$. By Lemma 4.13 (homogenization), $\Theta \models \mathcal{C}$.

Therefore, for all $\mathcal{C} \in \vec{\mathcal{C}}$, $\Theta \vDash \mathcal{C}$. Thus, $\vec{\mathcal{C}}$ is an essentially homogeneous set of constraints. □

If the elaboration algorithm is correct, the solutions to the type checking problem will coincide to the solutions with the resulting internal constraints:

Lemma 4.24 (Correctness of elaboration into internal constraints)**.** *Let $\Sigma; \Gamma \vdash^? t : A$ be a type checking problem, and $\Sigma'; \vec{\mathcal{C}}$ its elaboration into internal constraints by a well formed and correct unification algorithm. Then the following hold:*

Soundness *For each Θ such that $\Theta \vDash \Sigma'; \vec{\mathcal{C}}$, we have $\Theta_{\Sigma \cup t}$ **wf** and $\Theta_{\Sigma \cup t} \vDash \Sigma; \Gamma \vdash^? t : A$.*

Completeness *If there is Θ with $\Theta \vDash \Sigma; \Gamma \vdash^? t : A$, then there is $\tilde{\Theta}$ with $\tilde{\Theta}_{\Sigma \cup t} = \Theta$ and $\tilde{\Theta} \vDash \Sigma'; \vec{\mathcal{C}}$.*

Proof.

Soundness Assume $\Theta \vDash \Sigma'; \vec{\mathcal{C}}$. By Definition 4.11 (solution to a unification problem), we have $\Theta \vDash \Sigma'$.

Let $\Gamma \vdash t : A \approx u : B$ be a basic constraint produced by the elaboration algorithm. By Definition 4.22 (elaboration into internal constraints), there is a corresponding constraint $\mathcal{C} = \Gamma \ddagger \Gamma \vdash t \cong u : A \ddagger B \in \vec{\mathcal{C}}$. Again by Definition 4.11 (solution to a unification problem), this means $\Theta \vdash \Gamma, A \equiv \Gamma, B \, \mathbf{ctx}$ and $\Theta; \Gamma \vdash t \equiv u : A$.

By Definition 3.12 (correctness of an elaboration algorithm), $\Theta_{\Sigma \cup t} \vDash \Gamma \vdash^? t : A$.

Completeness Assume there is a metasubstitution Θ such that $\Theta \vDash \Sigma; \Gamma \vdash^? t : A$. By Definition 3.12 (correctness of an elaboration algorithm), there is $\tilde{\Theta}$ such that $\tilde{\Theta}_{\Sigma \cup t} = \Theta$ and $\tilde{\Theta} \vDash \Sigma'$.

Let $\mathcal{C} \in \vec{\mathcal{C}}$. We proceed by case analysis:

- $\mathcal{C} = \Gamma \ddagger \Gamma \vdash A \approx B : \mathrm{Set} \ddagger \mathrm{Set}$, and the elaboration algorithm produced a basic constraint $\mathcal{C} = \Gamma \vdash t : A \cong u : B$.

  By Definition 3.11 (well-formedness of an elaboration algorithm), $\Sigma'; \Gamma \vdash t : A$. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma' \vdash \Gamma \, \mathbf{ctx}$. As in the proof of Lemma 4.23 (well-formedness of elaboration into internal constraints), $\Sigma' \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set} \, \mathbf{ctx}$. Because $\tilde{\Theta} \vDash \Sigma'$, we have $\tilde{\Theta} \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set} \, \mathbf{ctx}$.

  Also, by Definition 3.12 (correctness of an elaboration algorithm), $\tilde{\Theta}; \Gamma \vdash A \equiv B \, \mathbf{type}$. By Remark 2.15 (there is only set), $\tilde{\Theta}; \Gamma \vdash A \equiv B : \mathrm{Set}$; that is, $\tilde{\Theta} \vDash \mathcal{C}$.

- $\mathcal{C} = \Gamma \ddagger \Gamma \vdash t \approx u : A \ddagger B$, and the elaboration algorithm produced a basic constraint $\mathcal{C} = \Gamma \vdash t : A \cong u : B$. By Definition 3.12 (correctness of an elaboration algorithm), $\tilde{\Theta}; \Gamma \vdash A \equiv B \, \mathbf{type}$ and $\tilde{\Theta}; \Gamma \vdash t \equiv u : A$.

By Lemma 2.70 (piecewise well-formedness of typing judgments), we have $\widetilde{\Theta} \vdash \Gamma\,\mathbf{ctx}$. By reflexivity, this means $\widetilde{\Theta} \vdash \Gamma \equiv \Gamma\,\mathbf{ctx}$. By Definition 2.16 (equality of contexts), $\widetilde{\Theta} \vdash \Gamma, A \equiv \Gamma, B\,\mathbf{ctx}$; that is, $\widetilde{\Theta} \vDash \mathcal{C}$.

Therefore, $\widetilde{\Theta} \vDash \Sigma'; \vec{\mathcal{C}}$. □

4.4 A unification relation

Our approach makes use of reduction rules to simplify unification problems. A reduction rule may create new constraints and/or extend the signature.

Definition 4.25 (Reduction rule)**.** A rule is a four tuple of the form $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \vec{\mathcal{D}}$, where Σ and Σ' are signatures, and $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$ are vectors of internal constraints.
A rule $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \vec{\mathcal{D}}$ states that, under signature Σ, the constraints $\vec{\mathcal{C}}$ reduce to a list of constraints $\vec{\mathcal{D}}$, extending the signature to Σ'.

Correct rules are those which preserve the set of possible solutions. In Section 4.5 we give a collection of rule schemas, and show that all the resulting rules are correct.

Definition 4.26 (Rule correctness)**.** A rule $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \vec{\mathcal{D}}$ is correct if, assuming that $\Sigma;\vec{\mathcal{C}}$ is well-formed, we have:

Well-formedness The problem $\Sigma';\vec{\mathcal{D}}$ is well-formed, and $\Sigma \sqsubseteq \Sigma'$ (in particular, Σ' **sig**).

Soundness For every Σ'' with $\Sigma'' \sqsupseteq \Sigma'$, if $\Sigma'' \mid\!\approx \vec{\mathcal{D}}$ then $\Sigma'' \mid\!\approx \vec{\mathcal{C}}$.

Completeness For each metasubstitution Θ such that $\Theta \vDash \Sigma;\vec{\mathcal{C}}$, there is a metasubstitution Θ' such that $\Theta = \Theta'_\Sigma$ and $\Theta' \vDash \Sigma';\vec{\mathcal{D}}$.

Remark. The soundness property is stated in terms of constraint satisfaction ($\mid\!\approx$), and the completeness in terms of constraint solutions ($\vDash$).
We make this distinction to make the correctness proofs of the individual rules more succinct, as for soundness it suffices to show satisfaction ($\Sigma'' \mid\!\approx \vec{\mathcal{C}}$). However, for proving completeness, we use the stronger premise ($\Theta \vDash \vec{\mathcal{C}}$). Theorem 4.31 (correctness of unification) only discusses constraint solutions.

We can define a reduction relation on problems by applying correct rules to an individual constraints, while preserving the other constraints as they are.

Definition 4.27 (One-step problem reduction: $\Sigma; \vec{\mathcal{E}} \rightsquigarrow \Sigma'; \vec{\mathcal{E}'}$)**.** We say that the problem $\Sigma;\vec{\mathcal{E}}$ reduces to $\Sigma';\vec{\mathcal{E}'}$ in one step (written $\Sigma; \vec{\mathcal{E}} \rightsquigarrow \Sigma'; \vec{\mathcal{E}'}$), if, $\vec{\mathcal{E}} = \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{C}} \wedge \overrightarrow{\mathcal{E}_2}$, $\vec{\mathcal{E}'} = \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{D}} \wedge \overrightarrow{\mathcal{E}_2}$, and $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \vec{\mathcal{D}}$ is a correct rule.

Definition 4.28 (Problem reduction: $\Sigma'; \vec{\mathcal{E}} \rightsquigarrow^\star \Sigma'; \vec{\mathcal{E}'}$)**.** We say that the problem $\Sigma;\vec{\mathcal{E}}$ reduces to $\Sigma';\vec{\mathcal{E}'}$ if $\Sigma;\vec{\mathcal{E}} \rightsquigarrow^\star \Sigma';\vec{\mathcal{E}'}$, where $\rightsquigarrow^\star$ is the reflexive, transitive closure of $\rightsquigarrow$.

Lemma 4.29 (Correctness of problem reduction)**.** *Given a well-formed problem $\Sigma;\vec{\mathcal{E}}$, such that $\Sigma;\vec{\mathcal{E}} \rightsquigarrow^\star \Sigma';\vec{\mathcal{E}}'$, then $\Sigma;\vec{\mathcal{E}} \rightsquigarrow \Sigma';\vec{\mathcal{E}}'$ is a correct rule. That is, the following hold:*

Well-formedness *The problem $\Sigma';\vec{\mathcal{E}}'$ is well-formed, and $\Sigma \sqsubseteq \Sigma'$.*

Soundness *For every Σ'' with $\Sigma'' \sqsupseteq \Sigma'$, if $\Sigma'' \models\!\!\approx \vec{\mathcal{E}}'$ then $\Sigma'' \models\!\!\approx \vec{\mathcal{E}}$.*

Completeness *For each metasubstitution Θ such that $\Theta \vDash \Sigma;\vec{\mathcal{E}}$, there is a metasubstitution Θ' such that $\Theta = \Theta'_\Sigma$ and $\Theta' \vDash \Sigma';\vec{\mathcal{E}}'$.*

Proof. By induction on the length of the derivation of $\Sigma;\vec{\mathcal{E}} \rightsquigarrow^\star \Sigma';\vec{\mathcal{E}}''$.

- Base case ($\Sigma;\vec{\mathcal{E}} = \Sigma'';\vec{\mathcal{E}}''$):

  Well-formedness By assumption, $\Sigma'';\vec{\mathcal{E}}'' = \Sigma;\vec{\mathcal{E}}$ is well-formed, and $\Sigma \sqsubseteq \Sigma''$.

  Soundness Take $\Sigma''' \sqsupseteq \Sigma''$ such that $\Sigma''' \models\!\!\approx \vec{\mathcal{E}}''$. Because $\vec{\mathcal{E}}'' = \vec{\mathcal{E}}$, Then $\Sigma''' \models\!\!\approx \vec{\mathcal{E}}$.

  Completeness Assume $\Theta \vDash \Sigma;\vec{\mathcal{E}}$. Then we have $\Theta' = \Theta$, with $\Theta = \Theta_\Sigma$.

- Inductive step ($\Sigma;\vec{\mathcal{E}} \rightsquigarrow \Sigma';\vec{\mathcal{E}}'$ and $\Sigma';\vec{\mathcal{E}}' \rightsquigarrow^\star \Sigma'';\vec{\mathcal{E}}''$):

  By Definition 4.27 (one-step problem reduction), we have $\vec{\mathcal{E}} = \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{C}} \wedge \overrightarrow{\mathcal{E}_2}$, $\vec{\mathcal{E}}' = \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{D}} \wedge \overrightarrow{\mathcal{E}_2}$, and $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$ is a correct rule.

  Well-formedness $\Sigma;\vec{\mathcal{E}}$ is well-formed. By Definition 4.5 (well-formed unification problem), this means that, for each constraint $\mathcal{E} \in \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{C}} \wedge \overrightarrow{\mathcal{E}_2}$, we have $\Sigma;\mathcal{E}$ **wf**. In particular:

  1. $\Sigma;\vec{\mathcal{C}}$ **wf**. By well-formedness of the rule, $\Sigma';\vec{\mathcal{D}}$ **wf**, which implies Σ' **sig**.
  2. By the well-formedness of the rule, $\Sigma' \sqsupseteq \Sigma$ By Lemma 2.155 (preservation of judgments under signature extensions), for every $\mathcal{E} \in \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{E}_2}$, we have $\Sigma';\mathcal{E}$ **wf**.

  Therefore, for every $\mathcal{E} \in \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{D}} \wedge \overrightarrow{\mathcal{E}_2}$, $\Sigma';\mathcal{E}$ **wf**; thus $\Sigma';\vec{\mathcal{E}}'$ **wf**. By the induction hypothesis, $\Sigma'';\vec{\mathcal{E}}''$ **wf** and $\Sigma \sqsubseteq \Sigma''$.

  Soundness Take $\Sigma''' \sqsupseteq \Sigma''$ such that $\Sigma''' \models\!\!\approx \vec{\mathcal{E}}''$. By the induction hypothesis, $\Sigma''' \models\!\!\approx \vec{\mathcal{E}}'$; that is, $\Sigma''' \models\!\!\approx \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{D}} \wedge \overrightarrow{\mathcal{E}_2}$. In particular:

  1. $\Sigma''' \models\!\!\approx \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{E}_2}$.
  2. $\Sigma''' \models\!\!\approx \overrightarrow{\mathcal{D}}$. By Definition 4.26 (rule correctness), $\Sigma'' \sqsupseteq \Sigma'$. Because $\Sigma''' \sqsupseteq \Sigma''$, by Remark 2.152 (signature extension is reflexive and transitive) we have $\Sigma''' \sqsupseteq \Sigma'$. Because $\Sigma''' \models\!\!\approx \overrightarrow{\mathcal{D}}$, by the soundness of the rule, $\Sigma''' \models\!\!\approx \vec{\mathcal{C}}$.

  Therefore, $\Sigma''' \models\!\!\approx \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{C}} \wedge \overrightarrow{\mathcal{E}_2}$; that is, $\Sigma''' \models\!\!\approx \vec{\mathcal{E}}$.

Completeness Assume $\Theta \vDash \Sigma; \vec{\mathcal{E}}$. In particular, $\Theta \vDash \Sigma$, $\Theta \vDash \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{E}_2}$ and $\Theta \vDash \vec{\mathcal{C}}$.

Because $\Theta \vDash \Sigma$ and $\Theta \vDash \vec{\mathcal{C}}$, we have $\Theta \vDash \Sigma; \vec{\mathcal{C}}$. By Definition 4.26 (rule correctness), there is Θ' such that $\Theta = \Theta'_\Sigma$ and $\Theta' \vDash \Sigma'; \vec{\mathcal{D}}$.

By Remark 2.137 (metasubstitution weakening), $\Theta' \vDash \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{E}_2}$.

Because $\Theta' \vDash \Sigma'$ and $\Theta' \vDash \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{D}} \wedge \overrightarrow{\mathcal{E}_2}$, we have $\Theta' \vDash \Sigma'; \overrightarrow{\mathcal{E}'}$. By the induction hypothesis, there is Θ'' such that $\Theta'' \vDash \Sigma'', \vec{\mathcal{E}''}$ and $\Theta''_{\Sigma'} = \Theta'$.

By Remark 2.154 (metasubstitution restriction to extension), because $\Theta'_\Sigma = \Theta$ and $\Theta''_{\Sigma'} = \Theta'$, we have $\Theta''_\Sigma = (\Theta''_{\Sigma'})_\Sigma = \Theta'_\Sigma = \Theta$.

□

Our unification algorithm (Algorithm 2) will start from a problem $\Sigma; \vec{\mathcal{C}}$ and apply rules iteratively, stopping if it produces a signature Σ' such that $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^\star \Sigma'; \square$.

If Σ' is closed (that is, instantiates all metavariables) we can obtain a solution to the original problem $\Sigma; \vec{\mathcal{C}}$ by constructing the closing metasubstitution of Σ'.

Definition 4.30 (Solved problem)**.** Let $\Sigma; \vec{\mathcal{D}}$ be a problem. We say that $\Sigma; \vec{\mathcal{D}}$ is a solved problem if $\Sigma; \vec{\mathcal{D}}$ **wf**, Σ is a closed signature, and $\vec{\mathcal{D}} = \square$.

Theorem 4.31 (Correctness of unification)**.** *Let $\Sigma; \vec{\mathcal{C}}$ be an essentially homogeneous, well-formed problem such that: $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^\star \Sigma'; \square$, where $\Sigma'; \square$ is a solved problem (i.e. Σ' is closed).*

Then the following hold:

1. *The signature Σ' is well-formed.*
2. *There is Θ such that $\textsc{close}(\Sigma') \Downarrow \Theta$ and $\Theta_\Sigma \vDash \Sigma; \vec{\mathcal{C}}$.*
3. *For every $\tilde{\Theta}$ such that $\tilde{\Theta} \vDash \Sigma; \vec{\mathcal{C}}$, we have $\Theta_\Sigma \equiv \tilde{\Theta}$.*

Proof.

1. By Lemma 4.29 (correctness of problem reduction), we have that $\Sigma';\square$ is well-formed.

   By Definition 4.5 (well-formed unification problem), Σ' **sig** and $\Sigma \sqsubseteq \Sigma'$.

2. By Corollary 2.149 (solution to closed signature), $\textsc{close}(\Sigma') \Downarrow \Theta$, and $\Theta \vDash \Sigma'$. By Lemma 2.157 (restriction of a metasubstitution to an extended signature), $\Theta_\Sigma \vDash \Sigma$. By the soundness statement in Lemma 4.29, $\Sigma' \mid\approx \vec{\mathcal{C}}$. Because $\Theta \vDash \Sigma'$, by Lemma 4.21 (constraint satisfaction by compatible metasubstitution) we have $\Theta \mid\approx \vec{\mathcal{C}}$. Because the set of constraints $\vec{\mathcal{C}}$ is essentially homogeneous, we have $\Theta \vDash \vec{\mathcal{C}}$.

   By Remark 2.134, $\Theta_\Sigma \subseteq \Theta$.

   Because $\Sigma; \vec{\mathcal{C}}$ **wf**, by Remark 4.8 (well-formed unification problem is a judgment) and Remark 4.6 (no extraneous constants in constraint), $\textsc{consts}(\vec{\mathcal{C}}) \subseteq \textsc{decls}(\Sigma)$.

By Definition 2.8 (constants declared by a signature), $\textsc{decls}(\Sigma) = \textsc{AtomDecls}(\Sigma) \cup \textsc{support}(\Sigma)$. By Remark 2.153 (signature extension declarations), $\textsc{AtomDecls}(\Sigma) = \textsc{AtomDecls}(\Sigma')$. and $\textsc{support}(\Sigma) \subseteq \textsc{support}(\Sigma')$, which means that $\textsc{support}(\Sigma) = \textsc{support}(\Sigma') \cap \textsc{support}(\Sigma)$. Therefore, $\textsc{decls}(\Sigma) = \textsc{AtomDecls}(\Sigma') \cup (\textsc{AtomDecls}(\Sigma') \cap \textsc{support}(\Sigma))$.

By Lemma 2.130 (alternative characterization of a compatible metasubstitution) and Remark 2.9 (atoms and metavariables are disjoint), $\textsc{AtomDecls}(\Theta) = \textsc{AtomDecls}(\Sigma')$ and $\textsc{support}(\Theta) = \textsc{support}(\Sigma')$. Therefore, $\textsc{decls}(\Sigma) = \textsc{AtomDecls}(\Theta) \cup (\textsc{AtomDecls}(\Theta) \cap \textsc{support}(\Sigma))$.

By Remark 2.135 (declarations in a metasubstitution restriction), $\textsc{AtomDecls}(\Theta) = \textsc{AtomDecls}(\Theta_\Sigma)$ and $\textsc{support}(\Theta_\Sigma) = \textsc{support}(\Theta) \cap \textsc{support}(\Sigma)$. Therefore, $\textsc{decls}(\Sigma) = \textsc{AtomDecls}(\Theta_\Sigma) \cup \textsc{AtomDecls}(\Theta_\Sigma) = \textsc{decls}(\Theta_\Sigma)$, and thus $\textsc{consts}(\vec{\mathcal{C}}) \subseteq \textsc{decls}(\Theta_\Sigma)$.

By Remark 2.138 (metasubstitution strengthening) and Remark 4.10 (solution to a constraint as a judgment) $\Theta_\Sigma \vDash \vec{\mathcal{C}}$.

Because $\Theta_\Sigma \vDash \Sigma$ and $\Theta_\Sigma \vDash \vec{\mathcal{C}}$, we have $\Theta_\Sigma \vDash \Sigma; \vec{\mathcal{C}}$.

3. Let $\tilde{\Theta}$ be a metasubstitution such that $\tilde{\Theta} \vDash \Sigma; \vec{\mathcal{C}}$. By the completeness statement in Lemma 4.29, there is a metasubstitution $\tilde{\Theta}'$, such that $\tilde{\Theta}'_\Sigma = \tilde{\Theta}$ and $\tilde{\Theta}' \vDash \Sigma'; \square$.

   By Corollary 2.149 (solution to closed signature), this gives $\Theta \equiv \tilde{\Theta}'$. By Lemma 2.150 (equality of restricted metasubstitutions), this gives $\Theta_\Sigma \equiv \tilde{\Theta}'_\Sigma$, or, equivalently, $\Theta_\Sigma \equiv \tilde{\Theta}$.

□

Corollary 4.32 (Correctness of type checking by unification)**.** *Let $\Sigma; \Gamma \vdash^? t : A$ be a well-formed type checking problem, and $\Sigma'; \vec{\mathcal{C}}$ the output produced by a well-formed and correct elaboration algorithm.*

If $\Sigma'; \vec{\mathcal{C}} \rightsquigarrow^\star \Sigma''; \square$, and Σ'' is closed, then there is Θ such that $\textsc{close}(\Sigma'') \Downarrow \Theta$, and $\Theta_{\Sigma \cup t}$ is a unique solution to the type checking problem $\Sigma; \Gamma \vdash^? t : A$ (up to equality of metasubstitutions).

Proof. By Theorem 4.31 (correctness of unification), there is Θ such that $\textsc{close}(\Sigma'') \Downarrow \Theta$, $\Theta_{\Sigma'}$ **wf**, $\Theta_{\Sigma'} \vDash \Sigma'$, $\Theta_{\Sigma'} \vDash \Sigma'; \vec{\mathcal{C}}$.

By Lemma 4.23 (well-formedness of elaboration into internal constraints), $\textsc{decls}(\Sigma') \supseteq \textsc{decls}(\Sigma) \cup \textsc{metas}(t)$. By Remark 2.136 (nested metasubstitution restriction), $(\Theta_{\Sigma'})_{\Sigma \cup t} = \Theta_{\Sigma \cup t}$. By Lemma 4.24 (correctness of elaboration into internal constraints), $\Theta_{\Sigma \cup t} \vDash \Sigma; \Gamma \vdash^? t : A$.

To show the uniqueness of Θ, we assume there is $\tilde{\Theta}$ such that $\tilde{\Theta} \vDash \Sigma; \Gamma \vdash^? t : A$. By Lemma 4.24 (correctness of elaboration into internal constraints), there is $\tilde{\tilde{\Theta}}$ such that $\tilde{\tilde{\Theta}}_{\Sigma \cup t} = \tilde{\Theta}$ and $\tilde{\tilde{\Theta}} \vDash \Sigma'; \vec{\mathcal{C}}$.

By Theorem 4.31 (correctness of unification), because $\Theta_{\Sigma'} \vDash \Sigma'; \vec{\mathcal{C}}$ and $\tilde{\tilde{\Theta}} \vDash \Sigma'; \vec{\mathcal{C}}$, we have $\tilde{\tilde{\Theta}} \equiv \Theta_{\Sigma'}$. By Lemma 2.150 (equality of restricted metasubstitutions), $\tilde{\tilde{\Theta}}_{\Sigma \cup t} \equiv (\Theta_{\Sigma'})_{\Sigma \cup t}$; that is, $\tilde{\Theta} \equiv \Theta_{\Sigma \cup t}$. □

4.5 A reduction rule toolkit

Below, we describe a set reduction rules (or more precisely, reduction rule schemas), and show their correctness according to Definition 4.26.

These rules can then be used to define a correct unification algorithm. The way the unification algorithm is built from the individual rules is discussed in Section 5.1 (unification algorithm). According to Theorem 4.31 (correctness of unification), in order to show the correctness of the algorithm it suffices to show the correctness of each individual rule.

4.5.1 Syntactic equality check

The heterogeneous equality is reflexive (Remark 4.15). We can exploit this remark to discharge those constraints whose two sides are syntactically identical.

Thanks to how the heterogeneous equality is defined (Definition 4.12), this rule applies even if the type and/or the context on each side of the constraint are distinct.

Rule-Schema 1 (Syntactic equality)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx t : A \ddagger A' \rightsquigarrow \Sigma; \square$$

Proof of correctness. By Definition 4.26 (rule correctness), it suffices to show:

Well-formedness Assume that $\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx t : A \ddagger A'$ is well-formed. Then, Σ **sig**. If Σ is well-formed, then the problem $\Sigma;\square$ is also trivially well-formed. By Definition 2.151 (signature extension), $\Sigma \sqsubseteq \Sigma$.

Soundness Because $\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx t : A \ddagger A'$ is well-formed, we have $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Gamma' \vdash t : A'$. Let $\Sigma'' \sqsupseteq \Sigma$. By Lemma 2.155 (preservation of judgments under signature extensions), $\Sigma''; \Gamma \vdash t : A$ and $\Sigma''; \Gamma' \vdash t : A'$.

From Remark 4.15 (reflexivity of heterogeneous equality), $\Sigma''; \Gamma \ddagger \Gamma' \vdash t \equiv \{t\} \equiv t : A \ddagger A'$.

Completeness Assume $\Theta \vDash \Sigma; \Gamma \ddagger \Gamma' \vdash t \approx t : A \ddagger A'$. In particular, $\Theta \vDash \Sigma$.

Let $\Theta' = \Theta$. We have $\Theta'_\Sigma = \Theta$. By assumption, $\Theta' \vDash \Sigma$. Because $\Theta' \vDash \Sigma$, then vacuously $\Theta' \vDash \Sigma; \square$.

□

4.5.2 Metavariable instantiation

Metavariable instantiation is the bread-and-butter of higher-order unification.

Given a unification problem of the form $\Sigma'; \square$ (with Σ' closed), Theorem 4.31 (correctness of unification) states that such a unification problem has a unique solution. The end goal of our unification algorithm is to reduce both the number of unification constraints and the number of uninstantiated metavariables to zero.

Metavariable instantiation reduces both the number of constraints in the problem, and the number of uninstantiated metavariables, thus getting us

closer to a solution to the unification problem. However, metavariable instantiation must be performed in such a way that the body of the metavariable has the appropriate type (soundness) and that potential solutions are not lost (completeness).

Problem 4.33 (Metavariable instantiation)**.** Consider the following candidate for a rule schema:

$$\lceil \Sigma_1, \alpha : A, \Sigma_2; , \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{x}^n \approx t : B_1 \ddagger B_2 \rightsquigarrow \Sigma_1, \alpha := u : A, \Sigma_2; \square \rfloor \quad (\star)$$

This rule schema instantiates α to u using the constraint $\Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{x}^n \approx t : B_1 \ddagger B_2$.

The question is, under which conditions does rule $(\star)$ fulfill Definition 4.26 (rule correctness)?

Sufficient preconditions for a solution to Problem 4.33 are given in Rule schema 2 (metavariable instantiation). In order to specify those preconditions and prove the correctness of the rule, we first need to introduce some new concepts.

Lemma 4.34 (General η-equality for Π-types)**.** *If* $\Sigma; \Gamma \vdash u : \Pi \vec{A}^n B$*, then* $\Sigma; \Gamma \vdash u \equiv \lambda \vec{x}^n.(u^{(+n)}\ @\ \vec{x}) : \Pi \vec{A}^n B$*.*

Proof. By induction on n.

- Case 0: Assume $\Sigma; \Gamma \vdash u : \Pi \vec{A}^0 B$. Note that $(u^{(+0)}\ @\ \varepsilon) \Downarrow u$. Also, $u = \lambda \vec{x}^0.u$. By reflexivity of equality, $\Sigma; \Gamma \vdash u \equiv \lambda \vec{x}^0.(u^{(+0)}\ @\ \varepsilon) : \Pi \vec{A}^0 B$.

- Case $n+1$: Assume $\Sigma; \Gamma \vdash u : \Pi A_0 \Pi \vec{A}^n B$.

  By Lemma 2.62 (context weakening), $\Sigma; \Gamma, A_0 \vdash u^{(+1)} : (\Pi A_0 \Pi \vec{A}^n B)^{(+1)}$, that is, $\Sigma; \Gamma, A_0 \vdash u^{(+1)} : \Pi A_0^{(+1)} (\Pi \vec{A}^n B)^{(+1)+1}$.

  By the typing rules, $\Sigma; \Gamma, x : A_0 \vdash x : A_0^{(+1)}$.

  By Postulate 2 (typing of hereditary application), and Definition 2.31 (hereditary substitution), $(u^{(+1)}\ @\ 0)\Downarrow$ and $\Sigma; \Gamma, A_0 \vdash (u^{(+1)}\ @\ 0) : ((\Pi \vec{A}^n B)^{(+1)+1})[0/0]$, that is, $\Sigma; \Gamma, A_0 \vdash (u^{(+1)}\ @\ 0) : \Pi \vec{A}^n B$.

  (i) Because $(u^{(+1)}\ @\ 0)\Downarrow$, by Definition 2.32 (hereditary elimination), we have two possible cases for u:

  - $u = f$, for some neutral term f. Then by the ETA-ABS rule, $\Sigma; \Gamma \vdash f \equiv \lambda.f^{(+1)}\, 0 : \Pi A_0 \Pi \vec{A}^n B$.
  - $u = \lambda.u_0$, for some term u_0. Then $(u^{(+1)}\ @\ 0) \Downarrow u_0$. By reflexivity, $\Sigma; \Gamma \vdash u \equiv \lambda.(u^{(+1)}\ @\ 0) : \Pi A_0 \Pi \vec{A}^n B$.

  In both cases, $\Sigma; \Gamma \vdash u \equiv \lambda.(u^{(+1)}\ @\ 0) : \Pi A_0 \Pi \vec{A}^n B$.

  (ii) By the induction hypothesis (with $x_0 = 0$), $\Sigma; \Gamma, A_0 \vdash (u^{(+1)}\ @\ 0) \equiv \lambda \vec{x}^n.((u^{(+1)}\ @\ 0)^{(+n)}\ @\ \vec{x}) : \Pi \vec{A}^n B$. By Lemma 2.39 (hereditary substitution and application commute with renaming), Definition 2.33 (iterated hereditary elimination), and Remark 2.30 (properties of renamings), this is the same as $\Sigma; \Gamma, A_0 \vdash (u^{(+1)}\ @\ n) \equiv \lambda \vec{x}^n.((u^{(+(n+1))}\ @\ n\, \vec{x}) : \Pi \vec{A}^n B$.

By the ABS-EQ rule, $\Sigma;\Gamma \vdash \lambda x_0.(u^{(+1)} \mathbin{@} x_0) \equiv \lambda x_0.\lambda \vec{x}^n.(u^{(+(n+1)} \mathbin{@} x_0\, \vec{x}) : \Pi A_0 \Pi \vec{A}^n B$.

By (i), (ii) and transitivity of equality, $\Sigma;\Gamma \vdash u \equiv \lambda x_0.\lambda \vec{x}^n.(u^{(+(n+1))} \mathbin{@} x_0\, \vec{x}) : \Pi A_0 \Pi \vec{A}^n B$.

□

Lemma 4.35 (General η-equality for pairs)**.** *Assume* $\Sigma;\Gamma \vdash u : \Sigma AB$. *Then* $(u \mathbin{@} .\pi_1)\Downarrow$, $(u \mathbin{@} .\pi_2)\Downarrow$ *and* $\Sigma;\Gamma \vdash u \equiv \langle u \mathbin{@} .\pi_1, u \mathbin{@} .\pi_2\rangle : \Sigma AB$.

Proof. By Postulate 3 (typing of hereditary projection), $(u \mathbin{@} .\pi_1)\Downarrow$ and $(u \mathbin{@} .\pi_2)\Downarrow$. By Definition 2.32 (hereditary elimination), there are only two possible cases:

(i) $u = f$, where f is a neutral term. Then $u \mathbin{@} .\pi_1 \Downarrow f\, .\pi_1$ and $(u \mathbin{@} .\pi_2)\Downarrow f\, .\pi_2$. By the ETA-PAIR rule, $\Sigma;\Gamma \vdash f \equiv \langle f\, .\pi_1, f\, .\pi_2\rangle : \Sigma AB$, that is, $\Sigma;\Gamma \vdash u \equiv \langle u \mathbin{@} .\pi_1, u \mathbin{@} .\pi_2\rangle : \Sigma AB$.

(ii) $u = \langle u_1, u_2\rangle$, $u \mathbin{@} .\pi_1 \Downarrow u_1$ and $(u \mathbin{@} .\pi_2) \Downarrow u_2$. By reflexivity, $\Sigma;\Gamma \vdash \langle u_1, u_2\rangle \equiv \langle u_1, u_2\rangle : \Sigma AB$, that is, $\Sigma;\Gamma \vdash u \equiv \langle u \mathbin{@} .\pi_1, u \mathbin{@} .\pi_2\rangle : \Sigma AB$.

□

The following lemma, derived from Miller's pattern condition [41], shows that a term of a function type can be characterized by the result of applying distinct variables to said term.

Lemma 4.36 (Miller's pattern condition)**.** *Let* u, v *be such that* $\Sigma;\cdot \vdash u : \Pi \overrightarrow{A}^n.B$, $\Sigma;\cdot \vdash v : \Pi \overrightarrow{A}^n.B$ *(in particular,* u *and* v *closed). Assume that, for all* $i \in \{1, ..., n\}$, $\Sigma;\Gamma \vdash x_i : A_i[\vec{x}_{1,...,i-1}]$, *and* $\Sigma;\Gamma \vdash u \mathbin{@} \vec{x} \equiv v \mathbin{@} \vec{x} : B[\vec{x}]$, *with all variables in* $\vec{x}$ *pairwise distinct. Then* $\Sigma;\cdot \vdash u \equiv v : \Pi \overrightarrow{A}^n.B$.

Proof. By Lemma 2.62 (context weakening), $\Sigma; \overrightarrow{y : A}^n, \Gamma \vdash u \mathbin{@} \vec{x} \equiv v \mathbin{@} \vec{x} : B[\vec{x}]$.

By induction, we show that for all $m \in \{0, ..., n\}$, there exist $\Gamma^{(m)}$, $\overrightarrow{x^{(m)}}$ with $\left|\Gamma^{(m)}\right| = \left|\overrightarrow{x^{(m)}}\right| = n - m$ such that all variables in $\overrightarrow{x^{(m)}}$ are pairwise distinct, $\Sigma; \overrightarrow{y : A}^n, \Gamma^{(m)} \vdash u \mathbin{@} \vec{y}_{1,...,m}\, \overrightarrow{x^{(m)}} \equiv v \mathbin{@} \vec{y}_{1,...,m}\, \overrightarrow{x^{(m)}} : B[\vec{y}_{1,...,m}, \overrightarrow{x^{(m)}}]$, and, for all k such that $1 \le k \le n - m$, we have $\Sigma; \overrightarrow{y : A}^n, \Gamma^{(m)} \vdash x_k^{(m)} : A_{m+k}[\vec{y}_{1,...,m}, \overrightarrow{x^{(m)}}_{1,...,k-1}]$.

Here, $\vec{y}_{1,...,m} \stackrel{\text{def}}{=} (\left|\Gamma^{(m)}\right| + n - 1), (\left|\Gamma^{(m)}\right| + n - 2), ..., \left|\Gamma^{(m)}\right|$.

By induction on m:

- Case 0: Holds by the assumptions of the theorem, with $\Gamma_0 = \Gamma$, $\overrightarrow{x^{(0)}} = \vec{x}$.
- Case $m + 1$: By the induction hypothesis, $\Sigma; \overrightarrow{y : A}^n, \Gamma_m \vdash u \mathbin{@} \vec{y}_{1,...,m}\, \overrightarrow{x^{(m)}} \equiv v \mathbin{@} \vec{y}_{1,...,m}\, \overrightarrow{x^{(m)}} : B[\vec{y}_{1,...,m}, \overrightarrow{x^{(m)}}]$, and, for all k such that $1 \le k \le n - m$, $\Sigma; \overrightarrow{y : A}^n, \Gamma_m \vdash x_k^{(m)} : A_{m+k}[\vec{y}_{1,...,m}, \overrightarrow{x^{(m)}}_{1,...k-1}]$.

In particular, we have $\Sigma; \overrightarrow{y : A}^{n}, \Gamma_m \vdash x_1^{(m)} : A_{m+1}[\vec{y}_{1,\ldots,m}]$, that is, $\Sigma; \overrightarrow{y : A}^{n}, \Gamma_m \vdash x_1^{(m)} : A_{m+1}^{(+1+|\Gamma_m|+(m-n))}$. By Lemma 2.65 (no extraneous variables in term), $\Gamma^{(m)} = \Gamma_1^{(m)}, x_1^{(m)} : A', \Gamma_2^{(m)}$, where $\left|\Gamma_2^{(m)}\right| = x_1^{(m)}$. By Postulate 13 (context strengthening), we have $\Sigma; \overrightarrow{y : A}^{n}, \Gamma_1^{(m)} \vdash x_1^{(m)} : A^{(+(1+|\Gamma^m|+(n-m)))(-(1+\left|\Gamma_2^{(m)}\right|))}$. By Remark 2.29 (composition of renamings), $\Sigma; \overrightarrow{y : A}^{n}, \Gamma_1^{(m)} \vdash x_1^{(m)} : A^{(+(1+\left|\Gamma_1^{(m)}\right|+(n-m)))}$.

By Lemma 2.78 (variable types say everything), we have that for each k such that $2 \leq k \leq n-m$, $\Sigma; \overrightarrow{y : A}^{n}, \Gamma_1^{(m)}, x_1^{(m)} : A', (\Gamma_2^{(m)})[x_1^{(m)} \mapsto y_{m+1}] \vdash x_k^{(m)} : A_{m+1+k}[\vec{y}_{1,\ldots,m}, \overrightarrow{x^{(m)}}][x_1^{(m)} \mapsto y_{m+1}]$ and $\Sigma; \overrightarrow{y : A}^{n}, \Gamma_1^{(m)}, x_1^{(m)} : A', (\Gamma_2^{(m)})[x_1^{(m)} \mapsto y_{m+1}] \vdash (u \mathbin{@} \vec{y}_{1,\ldots,m}\, \overrightarrow{x^{(m)}})[x_1^{(m)} \mapsto y_{m+1}] \equiv (v \mathbin{@} \vec{y}_{1,\ldots,m}\, \overrightarrow{x^{(m)}})[x_1^{(m)} \mapsto y_{m+1}] : B[\vec{y}_{1,\ldots,m}, \overrightarrow{x^{(m)}}][x_1^{(m)} \mapsto y_{m+1}]$.

Let $\Gamma^{(m+1)} \stackrel{\text{def}}{\equiv} \Gamma_1^{(m)}, (\Gamma_2^{(m)})[x_1^{(m)} \mapsto y_{m+1}]$ and $\overrightarrow{x^{(m+1)}}$ be defined such that $\left|\overrightarrow{x^{(m+1)}}\right| \stackrel{\text{def}}{\equiv} n-m-1$, $x_i^{(m+1)} \stackrel{\text{def}}{\equiv} x_{i+1}^{(m)}$ if $x_{i+1}^{(m)} < x_1^{(m)}$, and $x_i^{(m+1)} \stackrel{\text{def}}{\equiv} x_{i+1}^{(m)} - 1$ otherwise. By construction, all the variables in $x^{(m+1)}$ are pairwise distinct.

Note that u and v are closed terms. By Lemma 2.39 (hereditary substitution and application commute with renaming), and Postulate 13 (context strengthening), for each k such that $1 \leq k \leq n-m-1$ we have $\Sigma; \overrightarrow{y : A}^{n}, \Gamma^{(m+1)} \vdash x_k^{(m+1)} : A_{m+1+k}[\vec{y}_{1,\ldots,m+1}, \overrightarrow{x^{(m+1)}}_{1,\ldots,k-1}]$ and $\Sigma; \overrightarrow{y : A}^{n}, \Gamma^{(m+1)} \vdash u \mathbin{@} \vec{y}_{1,\ldots,m+1}\, \overrightarrow{x^{(m+1)}} \equiv v \mathbin{@} \vec{y}_{1,\ldots,m+1}\, \overrightarrow{x^{(m+1)}} : B[\vec{y}_{1,\ldots,m+1}, \overrightarrow{x^{(m+1)}}]$.

By taking $m = n$ we have $\Sigma; \overrightarrow{y : A}, \Gamma_n \vdash u \mathbin{@} \vec{y} \equiv v \mathbin{@} \vec{y} : B[\vec{y}]$.

Observe that $\textsc{fv}(u \mathbin{@} \vec{y} \equiv v \mathbin{@} \vec{y} : B[\vec{y}]) \subseteq \vec{y}$. By iterated application of Postulate 13 (context strengthening), we have $\Sigma; \overrightarrow{y : A} \vdash u \mathbin{@} \vec{y} \equiv v \mathbin{@} \vec{y} : B[\vec{y}]$.

By the ABS-EQ rule, $\Sigma; \cdot \vdash \lambda\vec{y}.(u \mathbin{@} \vec{y}) \equiv \lambda\vec{y}.(v \mathbin{@} \vec{y}) : \Pi\vec{A}B$. By Lemma 4.34 (general η-equality for Π-types), $\Sigma; \cdot \vdash u \equiv \lambda\vec{y}.(u \mathbin{@} \vec{y}) : \Pi\vec{A}B$ and $\Sigma; \cdot \vdash v \equiv \lambda\vec{y}.(v \mathbin{@} \vec{y}) : \Pi\vec{A}B$. By transitivity of equality, $\Sigma; \cdot \vdash u \equiv v : \Pi\vec{A}B$. □

The term u in Problem 4.33 is based on the right-hand side of the original constraint (t), as we see in Rule schema 2. For u to have the appropriate type (A), the context and types of both sides of the constraint must be consistent.

A sufficient precondition is $\Sigma \vdash \Gamma_1, B_1 \equiv \Gamma_2, B_2\,\mathbf{ctx}$. However, we can define a weaker precondition which is also sufficient, and concerns only the types of those variables which are used in the constraint.

Definition 4.37 (Heterogeneously equal contexts modulo variables)**.** We say that two such contexts Γ_1 and Γ_2 are heterogeneously equal in signature Σ modulo the sets of variables X_1 and X_2 (written $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1,X_2} \Gamma_2$), if Σ is a well-formed signature, Γ_1 and Γ_2 are well-formed contexts such that $|\Gamma_1| = |\Gamma_2|$, and, for each variable x occurring in both X_1 and X_2, the types of x in Γ_1 and Γ_2 are heterogeneously equal.

$$\frac{}{\Sigma \vdash \cdot \equiv\{\ \cdot\ \}\equiv_{\emptyset,\emptyset} \cdot}\ \textsc{empty}$$

$$\frac{0 \notin X_1 \cup X_2 \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1-1,X_2-1} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, \mathrm{Set}\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{unused}$$

$$\frac{0 \in X_2 - X_1 \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1-1,(X_2-1)\cup \mathrm{FV}(A_2)} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, A_2\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used-r}$$

$$\frac{0 \in X_1 - X_2 \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{(X_1-1)\cup \mathrm{FV}(A_1),X_2-1} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, A_1\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used-l}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma_1\ddagger\Gamma_2 \vdash A_1 \equiv\{A\}\equiv A_2 : \mathrm{Set}\ddagger\mathrm{Set} \qquad \begin{array}{l} X_1' = (X_1 - 1) \cup \mathrm{FV}(A_1) \\ X_2' = (X_2 - 1) \cup \mathrm{FV}(A_2) \\ \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1',X_2'} \Gamma_2 \end{array}\end{array}}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, A\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used}$$

Remark. In the rule UNUSED we use Set as the witness type because it is well-formed in any context Γ. Using Bool instead of Set would work equally well.

Lemma 4.38 (Typing in heterogeneously equal contexts)**.** *Let t and u be terms such that $\Sigma;\Gamma_1 \vdash t : B_1$, $\Sigma;\Gamma_2 \vdash u : B_2$, with $|\Gamma_1| = |\Gamma_2|$.*

Furthermore, we have $\Sigma;\Gamma_1\ddagger\Gamma_2 \vdash B_1 \equiv\{B\}\equiv B_2 : \mathrm{Set}\ddagger\mathrm{Set}$ and $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\mathrm{FV}(t)\cup\mathrm{FV}(B_1),\mathrm{FV}(u)\cup\mathrm{FV}(B_2)} \Gamma_2$.

Then $\Sigma;\Gamma \vdash t : B$ and $\Sigma;\Gamma \vdash u : B$.

Proof. We will prove the following stronger property:

> Suppose that $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1,X_2} \Gamma_2$. For every Δ, if $\Sigma;\Gamma_1, \Delta \vdash t : B$ and $\mathrm{FV}(\Delta \vdash t : B) \subseteq X_1$, then $\Sigma;\Gamma, \Delta \vdash t : B$. Also, if $\Sigma;\Gamma_2, \Delta \vdash u : B$ and $\mathrm{FV}(\Delta \vdash u : B) \subseteq X_2$, then $\Sigma;\Gamma, \Delta \vdash u : B$. $\qquad (\star)$

We proceed by induction on the length of Γ (i.e., the structure of the derivation for $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1,X_2} \Gamma_2$).

In the base case, we have $\Gamma_1 = \Gamma_2 = \Gamma = \cdot$. By assumption, $\cdot, \Delta \vdash t : B$ and $\cdot, \Delta \vdash u : B$.

In the inductive step, we have:

$$\begin{array}{lcl} \Gamma_1 & = & \Gamma_1', A_1 \\ \Gamma_2 & = & \Gamma_2', A_2 \\ \Gamma & = & \Gamma', A \end{array}$$

We consider four cases, one for each possible rule in the derivation of $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1,X_2} \Gamma_2$.

- Rule EMPTY: Trivial.
- Rule UNUSED:

$$\frac{0 \notin X_1 \cup X_2 \qquad \Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{X_1-1,X_2-1} \Gamma_2'}{\Sigma \vdash \Gamma_1', A_1 \equiv\{\Gamma', \mathrm{Set}\}\equiv_{X_1,X_2} \Gamma_2', A_2}\ \textsc{unused}$$

From the premises of the rule, $0 \notin X_1$. By assumption, $\textsc{fv}(\Delta \vdash t : B) \subseteq X_1$, which means $0 \notin \textsc{fv}(\Delta \vdash t : B)$.

Also by assumption, $\Gamma_1', A_1, \Delta \vdash t : B$, which, by Lemma 2.71 (variables of irrelevant type), implies $\Gamma_1', \mathrm{Set}, \Delta \vdash t : B$.

By Definition 2.45 (free variables of a scoped and typed term) and the assumption, $\textsc{fv}(\mathrm{Set}, \Delta \vdash t : B) = \textsc{fv}(\Delta \vdash t : B) - 1 \subseteq X_1 - 1$.

From the premises, $\Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{X_1-1,X_2-1} \Gamma_2'$. By the induction hypothesis, $\Gamma', \mathrm{Set}, \Delta \vdash t : B$; i.e. $\Gamma, \Delta \vdash t : B$.

By the same token, we show that, if $\Gamma_2, \Delta \vdash u : B$ with $\textsc{fv}(\Delta \vdash u : B) \subseteq X_2$, then $\Gamma, \Delta \vdash u : B$.

- Rule USED-L:

$$\frac{0 \in X_1 - X_2 \qquad \Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{(X_1-1)\cup\textsc{fv}(A_1),X_2-1} \Gamma_2'}{\Sigma \vdash \Gamma_1', A_1 \equiv\{\Gamma', A_1\}\equiv_{X_1,X_2} \Gamma_2', A_2}\ \textsc{used-l}$$

Assume $\Gamma_1, \Delta \vdash t : B$, i.e. $\Gamma_1', A_1, \Delta \vdash t : B$.

Because $\textsc{fv}(\Delta \vdash t : B) \subseteq X_1$, by Definition 2.45 (free variables of a scoped and typed term), $\textsc{fv}(A_1, \Delta \vdash t : B) = \textsc{fv}(A_1) \cup (\textsc{fv}(\Delta \vdash t : B) - 1) \subseteq (X_1 - 1) \cup \textsc{fv}(A_1)$.

By the induction hypothesis, $\Gamma', A_1, \Delta \vdash t : B$, i.e. $\Gamma, \Delta \vdash t : B$.

Now assume $\Gamma_2, \Delta \vdash u : B$, i.e. $\Gamma_2', A_2, \Delta \vdash u : B$.

By the original assumption, $\textsc{fv}(\Delta \vdash u : B) \subseteq X_2$. By the premises of the rule, $0 \notin X_2$; therefore, $0 \notin \textsc{fv}(\Delta \vdash u : B)$. By Lemma 2.71 (variables of irrelevant type), $\Gamma_2', \mathrm{Set}, \Delta \vdash u : B$.

Also from $\textsc{fv}(\Delta \vdash u : B) \subseteq X_2$ we deduce $\textsc{fv}(\mathrm{Set}, \Delta \vdash u : B) = \textsc{fv}(\mathrm{Set}) \cup (\textsc{fv}(\Delta \vdash u : B) - 1) = \textsc{fv}(\Delta \vdash u : B) - 1 \subseteq X_2 - 1$. By the induction hypothesis, $\Gamma', \mathrm{Set}, \Delta \vdash u : B$.

Finally, by Lemma 2.71, we have $\Gamma', A_1, \Delta \vdash u : B$, i.e. $\Gamma, \Delta \vdash u : B$.

- Rule USED-R: Same as USED-L, swapping "$_1$" and "$_2$", and "t" and "u".

- Rule USED:

$$\frac{\begin{array}{l} X_1' = (X_1 - 1) \cup \textsc{fv}(A_1) \\ X_2' = (X_2 - 1) \cup \textsc{fv}(A_2) \end{array} \qquad \begin{array}{ll} \Sigma; \Gamma_1' \ddagger \Gamma_2' \vdash A_1 \equiv\{A\}\equiv A_2 : \mathrm{Set} \ddagger \mathrm{Set} & \Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{X_1',X_2'} \Gamma_2' \end{array}}{\Sigma \vdash \Gamma_1', A_1 \equiv\{\Gamma', A\}\equiv_{X_1,X_2} \Gamma_2', A_2}\ \textsc{used}$$

Assume that $\Sigma; \Gamma_1', A_1, \Delta \vdash t : B$ with $\text{FV}(\Delta \vdash t : B) \subseteq X_1$.

From the first premise of the rule, we have $\Sigma; \Gamma_1' \vdash A_1 \equiv A : \text{Set}$ and $\text{FV}(A) \subseteq \text{FV}(A_1)$.

By the assumptions and Lemma 2.63 (preservation of judgments by type conversion), $\Sigma; \Gamma_1', A, \Delta \vdash t : B$.

By the assumptions and Definition 2.45 (free variables of a scoped and typed term), $\text{FV}(A, \Delta \vdash t : B) = \text{FV}(A) \cup (\text{FV}(\Delta \vdash t : B) - 1) \subseteq \text{FV}(A_1) \cup (X_1 - 1)$

By the induction hypothesis, $\Sigma; \Gamma', A, \Delta \vdash t : B$, i.e. $\Sigma; \Gamma, \Delta \vdash t : B$.

From the assumptions $\text{FV}(\Delta \vdash u : B) \subseteq X_2$ and $\Sigma; \Gamma_2', A_2, \Delta \vdash u : B$, by the same token, it follows that $\Sigma; \Gamma, \Delta \vdash u : B$.

Now that we have proven $(\star)$, we can use it to prove the original lemma.

By hypothesis, $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash B_1 \equiv\{B\}\equiv B_2 : \text{Set}$.

Assume $\Sigma; \Gamma_1 \vdash t : B_1$. From the hypothesis, we have $\Sigma; \Gamma_1 \vdash B_1 \equiv B : \text{Set}$, with $\text{FV}(B) \subseteq \text{FV}(B_1)$. By the CONV-EQ rule, $\Sigma; \Gamma_1 \vdash t : B$.

By Definition 2.45 (free variables of a scoped and typed term), $\text{FV}(\cdot \vdash t : B) = \text{FV}(t) \cup \text{FV}(B) \subseteq \text{FV}(t) \cup \text{FV}(B_1)$,

Also by hypothesis, $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\text{FV}(t)\cup\text{FV}(B_1),\text{FV}(u)\cup\text{FV}(B_2)} \Gamma_2$. By applying $(\star)$ with $\Delta = \cdot$, we have $\Sigma; \Gamma \vdash t : B$.

By the same reasoning, from $\Sigma; \Gamma_2 \vdash u : B_2$ and using $(\star)$, we deduce $\Sigma; \Gamma \vdash u : B$.

□

Using the notion of heterogeneously equal contexts we can define a correct rule schema for metavariable instantiation.

Rule-Schema 2 (Metavariable instantiation)**.**

$$\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{x}^n \approx t : B_1 \ddagger B_2 \rightsquigarrow \Sigma_1, \alpha := \lambda \vec{y}^n.t' : A, \Sigma_2; \square$$

where

$$\Sigma = \Sigma_1, \alpha : A, \Sigma_2$$

$$t' = t[\vec{x} \mapsto \vec{y}]$$

$$\text{all variables in } \vec{x} \text{ are pair-wise distinct} \tag{1}$$

$$\text{FV}(t) \subseteq \vec{x} \tag{2a}$$

$$\text{CONSTS}(t) \subseteq \text{DECLS}(\Sigma_1) \tag{2b}$$

$$\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash B_1 \equiv\{B\}\equiv B_2 : \text{Set} \ddagger \text{Set} \tag{3a}$$

$$\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\{x_i | i=1,\ldots,n\}\cup\text{FV}(B_1),\text{FV}(t)\cup\text{FV}(B_2)} \Gamma_2 \tag{3b}$$

The vector $\vec{y}$ denotes $(n-1) \ldots 0$. The side conditions ensure that $\lambda \vec{y}.t'$ is a well-typed and unique instantiation for α.

Proof of correctness. Let $\mathcal{C} = \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{x}^n \approx t : B_1 \ddagger B_2$, and $\Sigma' = \Sigma_1, \alpha := \lambda \vec{y}^n.t' : A, \Sigma_2$.

By Definition 4.26 (rule correctness), assuming $\Sigma; \mathcal{C}$ **wf**, it suffices to show:

Well-formedness Because no new constraints are added, the rule is well-formed if Σ' **sig** and $\Sigma' \sqsupseteq \Sigma$.

(i) By well-formedness of the original problem, we have $\Sigma; \Gamma_1 \vdash \alpha\, \vec{x} : B_1$ and $\Sigma; \Gamma_2 \vdash t : B_2$.

By (3a), (3b) and Lemma 4.38 (typing in heterogeneously equal contexts), we have $\Sigma; \Gamma \vdash \alpha\, \vec{x} : B$ and $\Sigma; \Gamma \vdash t : B$. Because $\Sigma; \Gamma \vdash \alpha\, \vec{x} : B$, by Lemma 2.109 (type application inversion), there is B' such that $\Sigma; \Gamma \vdash A \mathbin{\hat{@}} \vec{x} \Downarrow B'$, and $\Sigma; \Gamma \vdash B \equiv B' : \mathrm{Set}$. Because $\Sigma; \Gamma \vdash t : B$, by the CONV rule, $\Sigma; \Gamma \vdash t : B'$. By Lemma 2.120 (typing of metavariable bodies), $\Sigma; \cdot \vdash \lambda \vec{y}^n.t' : A$.

(ii) By the assumption, $\textsc{consts}(t') \subseteq \textsc{decls}(\Sigma_1)$. Because $\Sigma_1; \cdot \vdash A$ **type**, by Lemma 2.72 (no extraneous constants), $\textsc{consts}(A) \subseteq \textsc{decls}(\Sigma_1)$. Because $\Sigma_1 \subseteq \Sigma$, and $\Sigma; \cdot \vdash \lambda \vec{y}^n.t' : A$, by Postulate 12 (signature strengthening), $\Sigma_1; \cdot \vdash \lambda \vec{y}^n.t' : A$.

By Remark 2.5 (signature inversion), we have that Σ_1 **sig**. By Definition 2.4 (well-formed signature) and item (ii), $\Sigma_1, \alpha := \lambda \vec{y}^n.t' : A$ **sig**. By Definition 2.151 (signature extension), this gives $\Sigma_1, \alpha : A \sqsubseteq \Sigma_1, \alpha := \lambda \vec{y}^n.t' : A$. By Corollary 2.156 (horizontal composition of extensions), $\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := \lambda \vec{y}^n.t' : A, \Sigma_2$.

Soundness Take $\Sigma'' \sqsupseteq \Sigma'$. We need to show that $\Sigma'' \mathrel{|\!\approx} \mathcal{C}$.

(i) By rule $\textsc{delta-meta}_0$, $\Sigma'; \Gamma_1 \vdash \alpha \equiv \lambda \vec{y}^n.t' : A$. Because $\Sigma' \vdash \Gamma_1$ **ctx**, by Lemma 2.155 (preservation of judgments under signature extensions), we have $\Sigma''; \Gamma_1 \vdash \alpha \equiv \lambda \vec{y}^n.t' : A$.

(ii) The term α is neutral. By Definition 2.32 (hereditary elimination), $\alpha \mathbin{@} \vec{x} \Downarrow \alpha\, \vec{x}$.

(iii) By Remark 2.35 (iterated application as substitution on body) $(\lambda \vec{y}^n.t') \mathbin{@} \vec{x} \Downarrow t'[\vec{x}]$,

By Lemma 2.40 (correspondence between renaming and substitution), $t'[\vec{x}] \Downarrow t'[\vec{y} \mapsto \vec{x}]$. Because the variables in $\vec{x}$ are pairwise distinct, and so are the variables in $\vec{y}$, we have $t'[\vec{y} \mapsto \vec{x}] = t[\vec{x} \mapsto \vec{y}][\vec{y} \mapsto \vec{x}] = t$. Therefore, we have $(\lambda \vec{y}^n.t' \mathbin{@} \vec{x}) \Downarrow t$.

(iv) By well-formedness of the original problem, $\Sigma; \Gamma_1 \vdash \alpha\, \vec{x} : B_1$. Because $\Sigma'' \sqsupseteq \Sigma' \sqsupseteq \Sigma$, by Lemma 2.155 (preservation of judgments under signature extensions), we also have $\Sigma''; \Gamma_1 \vdash \alpha\, \vec{x} : B_1$. By Lemma 2.109 (type application inversion), this means that there exists B_1' such that $\Sigma''; \Gamma_1 \vdash A \mathbin{\hat{@}} \vec{x} \Downarrow B_1'$ and $\Sigma''; \Gamma_1 \vdash B_1' \equiv B_1$ **type**
By (i), (ii) (iii) and Lemma 2.110 (type of hereditary application), $\Sigma''; \Gamma_1 \vdash \alpha\, \vec{x} \equiv t : B_1'$.

(v) By (iv) and the CONV-EQ rule, $\Sigma''; \Gamma_1 \vdash \alpha\, \vec{x} \equiv t : B_1$.

(vi) By well-formedness of the original problem, $\Sigma; \Gamma_2 \vdash t : B_2$. Because $\Sigma'' \sqsupseteq \Sigma' \sqsupseteq \Sigma$, $\Sigma''; \Gamma_2 \vdash t : B_2$. By reflexivity, $\Sigma''; \Gamma_2 \vdash t \equiv t : B_2$.

(vii) By the premises of the rule, $\textsc{fv}(t) \subseteq \vec{x} = \textsc{fv}(\alpha\, \vec{x})$. Also, trivially, $\textsc{fv}(t) \subseteq \textsc{fv}(t)$.

By (v), (vi), and (vii), $\Sigma''; \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{x}^n \mathrel{\equiv\!\{t\}\!\equiv} t : B_1 \ddagger B_2$.

By Definition 4.17 (constraint satisfaction), $\Sigma'' \mathrel{|\!\approx} \mathcal{C}$.

Completeness Assume that $\Theta \vDash \Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{x}^n \approx t : B_1 \ddagger B_2$ holds; that is, $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma_1, B_1 \equiv \Gamma_2, B_2\, \mathbf{ctx}$ and $\Theta; \Gamma_1 \vdash \alpha\, \vec{x} \equiv t : B_1$.

Take $\Theta' = \Theta$. Because $\Theta \vDash \Sigma$, $\Theta'_\Sigma = \Theta_\Sigma = \Theta$. We need to show that $\Theta \vDash \Sigma'; \square$. Because there are no new constraints, it suffices to show $\Theta \vDash \Sigma'$. Because $\textsc{decls}(\Theta) = \textsc{decls}(\Sigma')$, by Lemma 2.130 (alternative characterization of a compatible metasubstitution), it suffices to show that, for each $D \in \Sigma'$, Θ is compatible with D.

If $D \in \Sigma_1$ or $D \in \Sigma_2$, then $D \in \Sigma$. Because $\Theta \vDash \Sigma$, by Lemma 2.130, Θ is compatible with D.

If $D \notin \Sigma_1$ and $D \notin \Sigma_2$, then $D = (\alpha \coloneqq \lambda \vec{y}^n.t' : A)$. Let u and B be the term and type such that $\alpha \coloneqq u : B \in \Theta$. By Remark 2.129 (alternative characterization of compatibility of a metasubstitution with a declaration), it suffices to show that (i) $\Theta; \cdot \vdash B \equiv A$ and (ii) $\Theta; \cdot \vdash u \equiv \lambda \vec{y}^n.t' : B$:

(i) By the assumption, $\Theta \vDash \Sigma$, with $\alpha : A \in \Sigma$. By Lemma 2.130, and Remark 2.129, $\Theta; \cdot \vdash B \equiv A$ **type**.

(ii) Because $\Sigma \vdash \Gamma_1\, \mathbf{ctx}$ and $\Theta \vDash \Sigma$, we have $\Theta \vdash \Gamma_1\, \mathbf{ctx}$. Because $\alpha \coloneqq u : B \in \Theta$, by the rule $\textsc{delta-meta}$, $\Theta; \Gamma_1 \vdash \alpha \equiv u : B$.

Because the original problem is well-formed, we have $\Sigma; \Gamma_1 \vdash \alpha\, \vec{x} : B_1$. Also, because $\alpha \coloneqq u : B \in \Theta$, we have $\Theta; \Gamma_1 \vdash \alpha \Rightarrow B$. By Lemma 2.109 (type application inversion), $\Sigma; \Gamma \vdash B \mathbin{\hat{@}} \vec{x} \Downarrow B'_1$, and $\Sigma; \Gamma \vdash B'_1 \equiv B_1 : \mathrm{Set}$.

By Lemma 2.110 (type of hereditary application), from $\Theta; \Gamma_1 \vdash \alpha \equiv u : B$ we have $\Theta; \Gamma_1 \vdash \alpha\, \vec{x} \equiv u \mathbin{@} \vec{x} : B_1$.

By assumption, $\Theta; \Gamma_1 \vdash \alpha\, \vec{x} \equiv t : B_1$. By symmetry and transitivity of equality, we have $\Theta; \Gamma_1 \vdash u \mathbin{@} \vec{x} \equiv t : B_1$.

By item (iii) of the soundness proof, $\lambda \vec{y}^n.t' \mathbin{@} \vec{x} \Downarrow t$. Therefore, by reflexivity, $\Theta; \Gamma_1 \vdash (\lambda \vec{y}^n.t') \mathbin{@} \vec{x} \equiv t : B_1$. By symmetry and transitivity of equality, we have $\Theta; \Gamma_1 \vdash u \mathbin{@} \vec{x} \equiv (\lambda \vec{y}^n.t') \mathbin{@} \vec{x} : B_1$.

By Lemma 4.36 (Miller's pattern condition), this gives $\Theta; \Gamma_1 \vdash u \equiv (\lambda \vec{y}^n.t') : A$.

Because all of u, $(\lambda \vec{y}^n.t')$ and A are closed terms, by Postulate 13 (context strengthening), $\Theta; \cdot \vdash u \equiv (\lambda \vec{y}^n.t') : A$. Because $\Theta; \cdot \vdash B \equiv A$ **type**, by the $\textsc{conv-eq}$ rule, $\Theta; \cdot \vdash u \equiv (\lambda \vec{y}^n.t') : B$.

□

4.5.3 Type constructors

When it comes to the judgmental equality, the type formers Π and Σ are injective. That is, two Π types are equal as terms iff the domain and codomain are equal as terms (Postulate 10). Correspondingly, two Σ-types are equal as terms iff their first and second components are equal as terms (Postulate 11).

We can exploit this injectivity property to simplify those constraints where both sides are a Π-type or a Σ-type.

Rule-Schema 3 (Injectivity of Π)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \Pi A B \approx \Pi A' B' : \mathrm{Set} \ddagger \mathrm{Set} \rightsquigarrow \Sigma; \Gamma \ddagger \Gamma' \vdash A \approx A' : \mathrm{Set} \ddagger \mathrm{Set} \wedge \Gamma \ddagger \Gamma', A \ddagger A' \vdash B \approx B' : \mathrm{Set} \ddagger \mathrm{Set}$$

Proof of correctness. Let $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash \Pi A B \approx \Pi A' B' : \mathrm{Set} \ddagger \mathrm{Set}$ and $\vec{\mathcal{D}} = \Gamma \ddagger \Gamma' \vdash A \approx A' : \mathrm{Set} \ddagger \mathrm{Set} \wedge \Gamma \ddagger \Gamma', A \ddagger A' \vdash B \approx B' : \mathrm{Set} \ddagger \mathrm{Set}$.

Assume $\Sigma; \mathcal{C}$ is well formed. By Definition 4.26 (rule correctness), it suffices to show:

Well-formedness We want to show that $\Sigma; \mathcal{D}_1$ and $\Sigma; \mathcal{D}_2$ are both well-formed. For this, it suffices to show that $\Sigma; \Gamma \vdash A : \mathrm{Set}$, $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$, and also $\Sigma; \Gamma' \vdash A' : \mathrm{Set}$, $\Sigma; \Gamma', A' \vdash B' : \mathrm{Set}$.

Because $\Sigma; \mathcal{C}$ is well-formed, we have $\Sigma; \Gamma \vdash \Pi A B : \mathrm{Set}$. By Lemma 2.52 (Π inversion), $\Sigma; \Gamma \vdash A : \mathrm{Set}$ and $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$.

Analogously, $\Sigma; \Gamma' \vdash A' : \mathrm{Set}$ and $\Sigma; \Gamma, A' \vdash B' : \mathrm{Set}$.

Soundness Assume that we have $\Sigma'' \models\!\approx \vec{\mathcal{D}}$ and $\Sigma'' \sqsupseteq \Sigma$. By Definition 4.17 (constraint satisfaction), $\Sigma''; \Gamma \ddagger \Gamma' \vdash A \equiv\!\{A_0\}\!\equiv A' : \mathrm{Set} \ddagger \mathrm{Set}$ and $\Sigma''; \Gamma \ddagger \Gamma', A \ddagger A' \vdash B \equiv\!\{B_0\}\!\equiv B' : \mathrm{Set} \ddagger \mathrm{Set}$.

By the PI-EQ rule, from $\Sigma''; \Gamma \vdash A \equiv A_0 : \mathrm{Set}$ and $\Sigma''; \Gamma, A \vdash B \equiv B_0 : \mathrm{Set}$, we deduce $\Sigma''; \Gamma \vdash \Pi A B \equiv \Pi A_0 B_0 : \mathrm{Set}$. Also, by Definition 2.18 (free variables in a term), from $\mathrm{FV}(A_0) \subseteq \mathrm{FV}(A)$, and $\mathrm{FV}(B_0) \subseteq \mathrm{FV}(B)$, it follows that $\mathrm{FV}(\Pi A_0 B_0) = \mathrm{FV}(A_0) \cup (\mathrm{FV}(B_0) - 1) \subseteq \mathrm{FV}(A) \cup (\mathrm{FV}(B) - 1) = \mathrm{FV}(\Pi A B)$.

Analogously, $\Sigma''; \Gamma' \vdash \Pi A' B' \equiv \Pi A_0 B_0 : \mathrm{Set}$ with $\mathrm{FV}(\Pi A_0 B_0) = \mathrm{FV}(\Pi A' B')$.

Therefore, $\Sigma''; \Gamma \ddagger \Gamma' \vdash \Pi A B \equiv\!\{\Pi A_0 B_0\}\!\equiv \Pi A' B' : \mathrm{Set} \ddagger \mathrm{Set}$. By Definition 4.17, $\Sigma \models\!\approx \mathcal{C}$.

Completeness Assume $\Theta \vDash \Sigma; \mathcal{C}$. By Definition 4.11 (solution to a unification problem), $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, \mathrm{Set} \equiv \Gamma', \mathrm{Set}\ \mathbf{ctx}$ and $\Theta; \Gamma \vdash \Pi A B \equiv \Pi A' B' : \mathrm{Set}$.

Take $\Theta' = \Theta$. Because $\Theta \vDash \Sigma$, $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

(i) Because $\Theta; \Gamma \vdash \Pi A B \equiv \Pi A' B' : \mathrm{Set}$, by Postulate 10 (injectivity of Π), $\Theta; \Gamma \vdash A \equiv A' : \mathrm{Set}$ and $\Theta; \Gamma, A \vdash B \equiv B' : \mathrm{Set}$.

(ii) By the assumption, $\Theta \vdash \Gamma, \mathrm{Set} \equiv \Gamma', \mathrm{Set}\ \mathbf{ctx}$. By (i), $\Theta \vdash \Gamma, A \equiv \Gamma', A'\ \mathbf{ctx}$, and Definition 4.9, $\Theta \vDash \overrightarrow{\mathcal{D}}$.

From the assumption, $\Theta \vDash \Sigma$. By (ii), $\Theta \vDash \overrightarrow{\mathcal{D}}$. Therefore, $\Theta \vDash \Sigma, \overrightarrow{\mathcal{D}}$. Because $\Theta = \Theta'$, we have $\Theta' \vDash \Sigma, \overrightarrow{\mathcal{D}}$.

□

Rule-Schema 4 (Injectivity of Σ)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \Sigma A B \approx \Sigma A' B' : \mathrm{Set} \ddagger \mathrm{Set} \rightsquigarrow \Sigma; \Gamma \ddagger \Gamma' \vdash A \approx A' : \mathrm{Set} \ddagger \mathrm{Set} \wedge \Gamma \ddagger \Gamma', A \ddagger A' \vdash B \approx B' : \mathrm{Set} \ddagger \mathrm{Set}$$

Proof. We follow the same reasoning as in the proof for Rule schema 3 (injectivity of Π). We replace all mentions of Π by Σ, and use the SIGMA-EQ rule and Postulate 11 (injectivity of Σ) instead of PI-EQ and Postulate 10 (injectivity of Π), respectively. □

The following rules are special cases of syntactic equality, and we can in fact do without them. We spell them out here for the sake of completeness.

Rule-Schema 5 (Bool)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \mathrm{Bool} \approx \mathrm{Bool} : \mathrm{Set} \ddagger \mathrm{Set} \rightsquigarrow \Sigma; \square$$

Proof of correctness. This rule schema is a special case of Rule schema 1 (syntactic equality). □

Rule-Schema 6 (Set)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \mathrm{Set} \approx \mathrm{Set} : \mathrm{Set} \ddagger \mathrm{Set} \rightsquigarrow \Sigma; \square$$

Proof of correctness. This rule schema is a special case of Rule schema 1 (syntactic equality). □

4.5.4 Constraint symmetry

The heterogeneous equality is symmetric (Remark 4.16). We can exploit this property to exchange both sides of a constraint.

Rule-Schema 7 (Constraint symmetry)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A' \rightsquigarrow \Sigma; \Gamma' \ddagger \Gamma \vdash u \approx t : A' \ddagger A$$

Proof of correctness. Let $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A'$ and $\mathcal{D} = \Gamma' \ddagger \Gamma \vdash u \approx t : A' \ddagger A$.

Well-formedness Follows from Definition 4.2 (well-formed internal constraint), which is symmetric.

Soundness Assume $\Sigma'' \sqsupseteq \Sigma'$, with $\Sigma'' \models\!\!\approx \mathcal{D}$. That is, $\Sigma''; \Gamma' \ddagger \Gamma \vdash u \equiv\!\!\{v\}\!\!\equiv t : A' \ddagger A$.

From Remark 4.16 (symmetry of heterogeneous equality), we also have $\Sigma''; \Gamma \ddagger \Gamma' \vdash t \equiv\!\!\{v\}\!\!\equiv u : A \ddagger A'$. Therefore, $\Sigma'' \models\!\!\approx \mathcal{C}$.

Completeness Assume $\Theta \vDash \Sigma; \mathcal{C}$. This means $\Theta \vDash \Sigma$ and $\Theta \vDash \mathcal{C}$.

Take $\Theta' \stackrel{\text{def}}{=} \Theta$. Because $\Theta \vDash \Sigma$, $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

Because $\Theta \vDash \mathcal{C}$, by Definition 4.9 (solution to a constraint), this means $\Theta \vdash \Gamma, A \equiv \Gamma', A'$ **ctx**, and $\Theta; \Gamma \vdash t \equiv u : A$.

By Lemma 2.64 (equality of contexts is an equivalence relation), $\Theta \vdash \Gamma', A' \equiv \Gamma, A$ **ctx**.

By symmetry of equality, $\Theta; \Gamma \vdash u \equiv t : A$. By Lemma 2.63 (preservation of judgments by type conversion), $\Theta; \Gamma' \vdash u \equiv t : A'$.

Because $\Theta \vdash \Gamma', A' \equiv \Gamma, A\,\mathbf{ctx}$ and $\Theta; \Gamma' \vdash u \equiv t : A'$, by Definition 4.9, $\Theta \vDash \mathcal{D}$.

Because $\Theta \vDash \Sigma$ and Definition 4.11 (solution to a unification problem), $\Theta \vDash \Sigma; \mathcal{D}$; that is, $\Theta' \vDash \Sigma; \mathcal{D}$.

□

Remark 4.39 (Rule symmetry). By Lemma 4.29 (correctness of problem reduction) and Rule schema 7 (constraint symmetry) means that, for each rule, we get a corresponding mirrored version.

In more detail, for each correct rule in the form $\Sigma; \overrightarrow{\mathcal{C}} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$, we get a rule $\Sigma; \overrightarrow{\mathcal{C}'} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}'}$; where for each $\mathcal{C}_i = \Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A_1 ‡ A_2$ we have $\mathcal{C}'_i \stackrel{\text{def}}{=} \Gamma_2 ‡ \Gamma_1 \vdash u \approx t : A_2 ‡ A_1$, and for each $\mathcal{D}_i = \Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A_1 ‡ A_2$ we have $\mathcal{D}'_i \stackrel{\text{def}}{=} \Gamma_2 ‡ \Gamma_1 \vdash u \approx t : A_2 ‡ A_1$.

4.5.5 Term conversion

Consider the following problem:

$$\begin{gathered} \Sigma; \cdot ‡ \cdot \vdash \mathbb{o} \approx \lambda x.\mathbb{o}\, x : (\mathbb{A} \to \mathbb{A}) ‡ (\mathbb{A} \to \mathbb{A}) \qquad (\star) \\ \Sigma = \mathbb{A} : \mathrm{Set}, \mathbb{o} : \mathbb{A} \to \mathbb{A} \end{gathered}$$

Observe that, by the ETA-ABS rule, $\Sigma; \cdot \vdash \mathbb{o} \equiv \lambda x.\mathbb{o}\, x : \mathbb{A} \to \mathbb{A}$ $(\star\star)$. If we replace the LHS of the constraint in $(\star)$ with the RHS of $(\star\star)$, we can use Rule schema 1 (syntactic equality) to solve problem $(\star)$.

In general, given a constraint $\Sigma; \Gamma ‡ \Gamma' \vdash t \approx u : A ‡ A'$, we may need to replace one of its sides (e.g. t) with a different, but still judgmentally equal term (t'), before we can apply other rule(s) and solve the constraint.

Because of the definition of heterogeneous equality (Definition 4.12), we impose the additional constraint that $\mathrm{FV}(t') \subseteq \mathrm{FV}(t)$. This condition is in particular fulfilled when $\Sigma; \Gamma \vdash t \longrightarrow_{\delta\eta} t' : A$, which by Remark 2.43 (free variables of $\delta\eta$-reduct) implies $\mathrm{FV}(t') \subseteq \mathrm{FV}(t)$, and covers the use cases in the unification algorithm (Algorithm 2).

Rule-Schema 8 (Term conversion)**.**

$$\begin{aligned} \Sigma; \Gamma ‡ \Gamma' \vdash t \approx u : A ‡ A' \rightsquigarrow \Sigma; \Gamma ‡ \Gamma' \vdash t' \approx u : A ‡ A' \\ \textbf{where} \quad \Sigma; \Gamma \vdash t \equiv t' : A \\ \mathrm{FV}(t) \supseteq \mathrm{FV}(t') \end{aligned}$$

Proof of correctness. Let $\mathcal{C} = \Gamma ‡ \Gamma' \vdash t \approx u : A ‡ A'$, and $\mathcal{D} = \Gamma ‡ \Gamma' \vdash t' \approx u : A ‡ A'$.

Well-formedness It suffices to show that $\Sigma; \Gamma \vdash t' : A$. This follows from $\Sigma; \Gamma \vdash t \equiv t' : A$ by Lemma 2.70 (piecewise well-formedness of typing judgments).

Soundness Assume that $\Sigma'' \sqsupseteq \Sigma$, with $\Sigma'' \not\approx \mathcal{D}$. By assumption, $\Sigma''; \Gamma ‡ \Gamma' \vdash t' \equiv\!\{v\}\!\equiv u : A ‡ A'$. In other words, $\Sigma''; \Gamma \vdash t' \equiv v : A$ and $\Sigma''; \Gamma' \vdash u \equiv v : A'$, with $\mathrm{FV}(v) \subseteq \mathrm{FV}(t')$ and $\mathrm{FV}(v) \subseteq \mathrm{FV}(u)$.

By transitivity of equality, $\Sigma''; \Gamma \vdash t \equiv v : A$. Also, $\text{FV}(v) \subseteq \text{FV}(t') \subseteq \text{FV}(t)$.

By Definition 4.12 (heterogeneous equality), $\Sigma''; \Gamma \ddagger \Gamma' \vdash t \equiv\{v\}\equiv u : A \ddagger A'$. Therefore, $\Sigma'' \not\approx \mathcal{C}$.

Completeness Assume $\Theta \vDash \Sigma; \mathcal{C}$. This means $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, A \equiv \Gamma', A'$ **ctx**, and $\Theta; \Gamma \vdash t \equiv u : A$.

Take $\Theta' = \Theta$. Because $\Theta \vDash \Sigma$, $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

By the rule premises, $\Sigma; \Gamma \vdash t \equiv t' : A$. Because $\Theta \vDash \Sigma$, by Definition 2.125 (compatible metasubstitution), $\Theta; \Gamma \vdash t \equiv t' : A$. By assumption, $\Theta; \Gamma \vdash t \equiv u : A$. By transitivity and symmetry of equality, $\Theta; \Gamma \vdash t' \equiv u : A$. By Definition 4.9 (solution to a constraint), $\Theta \vDash \mathcal{D}$.

Because $\Theta \vDash \Sigma$ and $\Theta \vDash \mathcal{D}$, by Definition 4.11 $\Theta \vDash \Sigma; \mathcal{D}$. Because $\Theta' = \Theta$ we have $\Theta' \vDash \Sigma; \mathcal{D}$.

□

4.5.6 Type conversion

By Lemma 2.63 (preservation of judgments by type conversion), we may replace the context and/or the type in a typing or equality judgment by a judgmentally equal one.

In practice, this means that we can consider forms of the context and the type with fewer free variables, metavariables and/or constants when determining if a rule can be applied to a constraint. This may make it easier to fulfill the rule's preconditions.

Rule-Schema 9 (Type and context conversion)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A' \rightsquigarrow \Sigma; \Gamma_0 \ddagger \Gamma' \vdash t \approx u : A_0 \ddagger A' \quad \textbf{where} \quad \Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0 \, \textbf{ctx}$$

Proof of correctness. Let $\mathcal{C} \stackrel{\text{def}}{=} \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A'$ and $\mathcal{D} \stackrel{\text{def}}{=} \Gamma_0 \ddagger \Gamma' \vdash t \approx u : A_0 \ddagger A'$.

Well-formedness By well-formedness of the original problem, $\Sigma; \Gamma' \vdash u : A'$.

Also by hypothesis, $\Sigma; \Gamma \vdash t : A$,

By the premise, $\Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0$ **ctx**. By Lemma 2.63 (preservation of judgments by type conversion), $\Sigma; \Gamma_0 \vdash t : A_0$.

Therefore, $\Sigma; \Gamma_0 \ddagger \Gamma' \vdash t \approx u : A_0 \ddagger A'$ is well-formed.

Soundness Assume $\Sigma'' \sqsupseteq \Sigma$, with $\Sigma'' \not\approx \mathcal{D}$. By Definition 4.17 (constraint satisfaction), this means that $\Sigma''; \Gamma_0 \ddagger \Gamma' \vdash t \equiv\{v\}\equiv u : A_0 \ddagger A'$.

(i) By the rule's premise, $\Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0$ **ctx**. By symmetry of context equality, we have $\Sigma \vdash \Gamma_0, A_0 \equiv \Gamma, A$ **ctx**. By Lemma 2.155 (preservation of judgments under signature extensions), $\Sigma'' \vdash \Gamma_0, A_0 \equiv \Gamma, A$ **ctx**.

(ii) By assumption $\Sigma''; \Gamma_0 \vdash t \equiv v : A_0$. By (i) and Lemma 2.63 (preservation of judgments by type conversion), $\Sigma''; \Gamma \vdash t \equiv v : A$.

(iii) By the assumption, $\textsc{fv}(v) \subseteq \textsc{fv}(t)$.

(iv) By the assumption, $\Sigma''; \Gamma' \vdash u \equiv v : A'$, and $\textsc{fv}(v) \subseteq \textsc{fv}(u)$.

By (ii), (iii), (iv) and Definition 4.12 (heterogeneous equality), $\Sigma''; \Gamma_0 \ddagger \Gamma' \vdash t \equiv\!\{v\}\!\equiv u : A_0 \ddagger A'$. By Definition 4.17, $\Sigma'' \not\approx \mathcal{C}$.

Completeness Assume $\Theta \vDash \Sigma; \mathcal{C}$; that is, $\Theta \vDash \Sigma$ and $\Theta \vDash \mathcal{C}$. By Definition 4.9 (solution to a constraint), we have $\Theta \vdash \Gamma, A \equiv \Gamma', A'\ \mathbf{ctx}$ and $\Theta; \Gamma \vdash t \equiv u : A$.

Let $\Theta' = \Theta$. Because $\Theta \vDash \Sigma$, $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

(i) By the premise, $\Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0\ \mathbf{ctx}$. By assumption, $\Theta \vDash \Sigma$. By Definition 2.125 (compatible metasubstitution), $\Theta \vdash \Gamma, A \equiv \Gamma_0, A_0\ \mathbf{ctx}$. Because $\Theta \vdash \Gamma, A \equiv \Gamma', A'\ \mathbf{ctx}$, by transitivity of context equality, $\Theta \vdash \Gamma_0, A_0 \equiv \Gamma', A'\ \mathbf{ctx}$.

(ii) By assumption, $\Theta; \Gamma \vdash t \equiv u : A$. Because $\Theta \vdash \Gamma, A \equiv \Gamma_0, A_0\ \mathbf{ctx}$, by Lemma 2.63 (preservation of judgments by type conversion), $\Theta; \Gamma_0 \vdash t \equiv u : A_0$.

By (i), (ii) and Definition 4.9, $\Theta \vDash \mathcal{D}$. By Definition 4.11, $\Theta \vDash \Sigma; \mathcal{D}$. Because $\Theta' = \Theta$, we have $\Theta' \vDash \Sigma; \mathcal{D}$.

□

In order for the body of a metavariable to be well-scoped, we may need to rearrange or normalize the signature first.

Rule-Schema 10 (Signature conversion)**.**

$$\Sigma; \square \rightsquigarrow \Sigma'; \square$$
$$\textbf{where} \quad \Sigma \sqsubseteq \Sigma' \textbf{ and } \Sigma' \sqsubseteq \Sigma\, \mathbf{ctx}$$

Proof of correctness.

Well-formedness By the assumption, $\Sigma \sqsubseteq \Sigma'$. By Definition 2.151 (signature extension), $\Sigma' \sqsubseteq \Sigma$ implies Σ' **sig**. Therefore, $\Sigma'; \square\ \mathbf{wf}$.

Soundness Holds vacuously.

Completeness Assume $\Theta \vDash \Sigma$. By Definition 2.125 (compatible metasubstitution), $\textsc{decls}(\Theta) = \textsc{decls}(\Sigma)$. Because $\Sigma \sqsubseteq \Sigma'$ and $\Sigma' \sqsubseteq \Sigma$, by Remark 2.153 (signature extension declarations), $\textsc{decls}(\Sigma) = \textsc{decls}(\Sigma')$.

Because $\Sigma' \sqsubseteq \Sigma$, by Lemma 2.157 (restriction of a metasubstitution to an extended signature), $\Theta_{\Sigma'} \vDash \Sigma'$. Let $\Theta' = \Theta$. Because $\textsc{decls}(\Theta') = \textsc{decls}(\Theta) = \textsc{decls}(\Sigma) = \textsc{decls}(\Sigma')$, by Definition 2.132 (restriction of a metasubstitution to a set of metavariables), $\Theta' = \Theta_{\Sigma'}$. Thus $\Theta' \vDash \Sigma'$.

By the same token, $\Theta'_\Sigma = \Theta$. Thus $\Theta'_\Sigma \vDash \Sigma'$.

□

4.5.7 Type-directed unification

Two functions are judgmentally equal if and only if their bodies are judgmentally equal (Lemma 2.83). Correspondingly, two pairs are equal if and only if their first and second projections are equal (Lemma 2.85).

We can use these properties to simplify constraints where both sides are headed by a λ-abstraction, or by a pair constructor.

Rule-Schema 11 (λ-abstraction)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \lambda.t \approx \lambda.u : \Pi A B \ddagger \Pi A' B' \rightsquigarrow \Sigma; \Gamma \ddagger \Gamma', A \ddagger A' \vdash t \approx u : B \ddagger B'$$

Proof of correctness. Let $\mathcal{C} \stackrel{\text{def}}{=} \Gamma \ddagger \Gamma' \vdash \lambda.t \approx \lambda.u : \Pi A B \ddagger \Pi A' B'$ and let $\mathcal{D} \stackrel{\text{def}}{=} \Gamma \ddagger \Gamma', A \ddagger A' \vdash t \approx u : B \ddagger B'$.

Assume that $\Sigma; \mathcal{C}$ is well-formed.

Well-formedness By Definition 4.5 (well-formed unification problem) and Definition 4.2 (well-formed internal constraint), this means that Σ **sig**, $\Sigma; \Gamma \vdash \lambda.t : \Pi A B$ and $\Sigma; \Gamma' \vdash \lambda.u : \Pi A' B'$.

By assumption, $\Sigma; \Gamma \vdash \lambda.t : \Pi A B$. By Lemma 2.82 (λ inversion), this means $\Sigma; \Gamma, A \vdash t : B$. Analogously, $\Sigma; \Gamma, A' \vdash u : B'$.

By Definition 4.5 and Definition 4.2, $\Sigma; \mathcal{D}$ is well-formed.

Soundness Assume $\Sigma'' \sqsupseteq \Sigma$, with $\Sigma'' \models\!\approx \mathcal{D}$. By Definition 4.17 (constraint satisfaction), this means $\Sigma''; \Gamma \ddagger \Gamma', A \ddagger A' \vdash t \equiv\!\{v\}\!\equiv u : B \ddagger B'$.

By the assumption, $\Sigma''; \Gamma, A \vdash t \equiv v : B$, with $\text{FV}(v) \subseteq \text{FV}(t)$. By ABS-EQ, this means $\Sigma''; \Gamma \vdash \lambda.t \equiv \lambda.v : \Pi A B$. Also, $\text{FV}(\lambda.v) = \text{FV}(v) - 1 \subseteq \text{FV}(t) - 1 = \text{FV}(\lambda.t)$

Analogously, $\Sigma''; \Gamma' \vdash \lambda.u \equiv \lambda.v : \Pi A' B'$, with $\text{FV}(\lambda.v) \subseteq \text{FV}(\lambda.u)$.

Therefore, $\Sigma''; \Gamma \ddagger \Gamma' \vdash \lambda.t \equiv\!\{\lambda.v\}\!\equiv \lambda.u : \Pi A B \ddagger \Pi A' B'$. By Definition 4.17, this means $\Sigma'' \models\!\approx \mathcal{C}$.

Completeness Assume $\Theta \vDash \Sigma; \mathcal{C}$. By Definition 4.11 (solution to a unification problem) and Definition 4.9 (solution to a constraint), this means $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, \Pi A B \equiv \Gamma', \Pi A' B'$ **ctx** and $\Theta; \Gamma \vdash \lambda.t \equiv \lambda.u : \Pi A B$.

Take $\Theta' = \Theta$. Because $\Theta \vDash \Sigma$, $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

(i) By assumption, $\Theta \vdash \Gamma, \Pi A B \equiv \Gamma', \Pi A' B'$ **ctx**; that is $\Theta \vdash \Gamma \equiv \Gamma'$ **ctx** and $\Theta; \Gamma \vdash \Pi A B \equiv \Pi A' B'$ **type**. By Postulate 10 (injectivity of Π), $\Theta; \Gamma \vdash A \equiv A'$ **type** and $\Theta; \Gamma, A \vdash B \equiv B'$ **type**. By Definition 2.16 (equality of contexts) we have $\Theta \vdash \Gamma, A, B \equiv \Gamma', A', B'$ **ctx**.

(ii) By the assumption, $\Theta; \Gamma \vdash \lambda.t \equiv \lambda.u : \Pi A B$. By Lemma 2.83 (injectivity of λ), we get $\Theta; \Gamma, A \vdash t \equiv u : B$.

By (i), (ii) and Definition 4.9 (solution to a constraint), we have $\Theta \vDash \mathcal{D}$. By assumption $\Theta \vDash \Sigma$, so, by Definition 4.11, $\Theta \vDash \Sigma; \mathcal{D}$. Because $\Theta' = \Theta$, $\Theta' \vDash \Sigma; \mathcal{D}$.

□

To unify a pair it suffices to unify each component individually.

Rule-Schema 12 (Pairs)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \langle t_1, t_2 \rangle \approx \langle u_1, u_2 \rangle : \Sigma AB \ddagger \Sigma A'B' \rightsquigarrow$$
$$\Sigma; \Gamma \ddagger \Gamma' \vdash t_1 \approx u_1 : A \ddagger A' \ \wedge \ \Gamma \ddagger \Gamma' \vdash t_2 \approx u_2 : B[t_1] \ddagger B'[u_1]$$
$$\textbf{where} \quad B[t_1]{\Downarrow} \ \textbf{and} \ B'[u_1]{\Downarrow}$$

As we show in the proof below, the rule preconditions $B[t_1]{\Downarrow}$ and $B'[u_1]{\Downarrow}$ are redundant provided $\Sigma; \Gamma \ddagger \Gamma' \vdash \langle t_1, t_2 \rangle \approx \langle u_1, u_2 \rangle : \Sigma AB \ddagger \Sigma A'B'$ **wf**.

Proof of correctness. Let $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash \langle t_1, t_2 \rangle \approx \langle u_1, u_2 \rangle : \Sigma AB \ddagger \Sigma A'B'$, $\mathcal{D}_1 = \Gamma \ddagger \Gamma' \vdash t_1 \approx u_1 : A \ddagger A'$, $\mathcal{D}_2 = \Gamma \ddagger \Gamma' \vdash t_2 \approx u_2 : B[t_1] \ddagger B'[u_1]$.

Assume that $\Sigma; \mathcal{C}$ is well-formed. By Definition 4.26 (rule correctness), it suffices to show:

Well-formedness We need to show that $\Sigma; \mathcal{D}_1$ and $\Sigma; \mathcal{D}_2$ are well-formed.

By assumption, $\Sigma; \Gamma \vdash \langle t_1, t_2 \rangle : \Sigma AB$. By Lemma 2.84 ($\langle,\rangle$-inversion), this means $\Sigma; \Gamma \vdash t_1 : A$, $B[t_1]{\Downarrow}$ and $\Sigma; \Gamma \vdash t_2 : B[t_1]$.

By the same reasoning, $\Sigma; \Gamma' \vdash u_1 : A'$, $B'[u_1]{\Downarrow}$ and $\Sigma; \Gamma' \vdash u_2 : B'[u_1]$.

Therefore, both $\Sigma; \mathcal{D}_1$ and $\Sigma; \mathcal{D}_2$ are well-formed.

Soundness Assume $\Sigma'' \sqsupseteq \Sigma$, and $\Sigma'' \models\!\approx \mathcal{D}_1 \wedge \mathcal{D}_2$. By Definition 4.17, this means that $\Sigma''; \Gamma \ddagger \Gamma' \vdash t_1 \equiv\!\{v_1\}\!\equiv u_1 : A \ddagger A'$ and $\Sigma''; \Gamma \ddagger \Gamma' \vdash t_2 \equiv\!\{v_2\}\!\equiv u_2 : B[t_1] \ddagger B'[u_1]$.

By the assumption, $\Sigma''; \Gamma \vdash t_1 \equiv v_1 : A$ and and $\Sigma''; \Gamma \vdash t_2 \equiv v_2 : B[t_1]$. By the PAIR-EQ rule, this gives $\Sigma''; \Gamma \vdash \langle t_1, t_2 \rangle \equiv \langle v_1, v_2 \rangle : \Sigma AB$.

Also from the assumption, $\mathrm{FV}(v_1) \subseteq \mathrm{FV}(t_1)$ and $\mathrm{FV}(v_2) \subseteq \mathrm{FV}(t_2)$. Therefore, $\mathrm{FV}(\langle v_1, v_2 \rangle) = \mathrm{FV}(v_1) \cup \mathrm{FV}(v_2) \subseteq \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) = \mathrm{FV}(\langle t_1, t_2 \rangle)$.

Analogously, $\Sigma''; \Gamma' \vdash \langle u_1, u_2 \rangle \equiv \langle v_1, v_2 \rangle : \Sigma A'B'$, with $\mathrm{FV}(\langle v_1, v_2 \rangle) \subseteq \mathrm{FV}(\langle u_1, u_2 \rangle)$.

Therefore, $\Sigma''; \Gamma \ddagger \Gamma' \vdash \langle t_1, t_2 \rangle \equiv\!\{\langle v_1, v_2 \rangle\}\!\equiv \langle u_1, u_2 \rangle : \Sigma AB \ddagger \Sigma A'B'$. By Definition 4.17, this means $\Sigma'' \models\!\approx \mathcal{C}$.

Completeness Assume $\Theta \vDash \Sigma; \mathcal{C}$. By Definition 4.11 (solution to a unification problem) and Definition 4.9 (solution to a constraint), this means $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, \Sigma AB \equiv \Gamma', \Sigma A'B'$ **ctx**, and $\Theta; \Gamma \vdash \langle t_1, t_2 \rangle \equiv \langle u_1, u_2 \rangle : \Sigma AB$.

Take $\Theta' \stackrel{\text{def}}{=} \Theta$. Because $\Theta \vDash \Sigma$, by Remark 2.133 (restriction to a compatible signature), $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

(i) By the assumption, $\Theta \vdash \Gamma, \Sigma AB \equiv \Gamma', \Sigma A'B'$ **ctx**. By Definition 2.16 (equality of contexts), this gives $\Theta \vdash \Gamma \equiv \Gamma'$ **ctx** and $\Theta; \Gamma \vdash \Sigma AB \equiv \Sigma A'B'$ **type**. By Postulate 11 (injectivity of Σ), we have $\Theta; \Gamma \vdash A \equiv A'$ **type** and $\Theta; \Gamma, A \vdash B \equiv B'$ **type**.

By Definition 2.16 (equality of contexts), $\Theta \vdash \Gamma, A \equiv \Gamma', A'$ **ctx**.

(ii) By the assumption, $\Theta; \Gamma \vdash \langle t_1, t_2 \rangle \equiv \langle u_1, u_2 \rangle : \Sigma AB$. By Lemma 2.85 (injectivity of $\langle , \rangle$), we get $\Theta; \Gamma \vdash t_1 \equiv u_1 : A$, $B[t_1]\Downarrow$ and $\Theta; \Gamma \vdash t_2 \equiv u_2 : B[t_1]$.

(iii) By (i), $\Theta; \Gamma, A \vdash B \equiv B'$ **type**. By (ii), $\Theta; \Gamma \vdash t_1 \equiv u_1 : A$. By Postulate 4 (congruence of hereditary substitution) and Remark 2.15 (there is only set), $\Theta; \Gamma \vdash B[t_1] \equiv B'[u_1]$ **type**. By Definition 2.16 (equality of contexts), $\Theta \vdash \Gamma, B[t_1] \equiv \Gamma', B'[u_1]$ **ctx**.

By (i), (ii), (iii) and Definition 4.9 (solution to a constraint), we have $\Theta \vDash \mathcal{D}_1$ and $\Theta \vDash \mathcal{D}_2$. By assumption, $\Theta \vDash \Sigma$. Therefore, by Definition 4.11, $\Theta \vDash \Sigma; \mathcal{D}_1 \wedge \mathcal{D}_2$.

□

For the Bool type, two constructors are equal if they are the identical. Because true and false take no arguments, the following rule is subsumed by syntactic equality. We include it for the sake of completeness.

Rule-Schema 13 (Booleans)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash c \approx c : \mathrm{Bool} \ddagger \mathrm{Bool} \rightsquigarrow \Sigma; \square$$

Proof. The rule schema is a special case of Rule schema 1 (syntactic equality). □

4.5.8 Strongly neutral terms

Constraints involving neutral terms are not always straightforward to normalize. Let $\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \mathbb{b} : \mathbb{A}$, and consider the following three examples:

Example 4.40 (Strong neutral unification)**.**

$$\Sigma, \alpha : \mathbb{A}; x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathrm{if}\,(\lambda.\mathbb{A})\; x\; \mathbb{a}\; \alpha \equiv \mathrm{if}\,(\lambda.\mathbb{A})\; x\; \mathbb{a}\; \mathbb{b} : \mathbb{A} \ddagger \mathbb{A}$$

By Definition 4.11 (solution to a unification problem), this problem has a solution $\Theta \stackrel{\text{def}}{=} \Sigma, \alpha := \mathbb{b} : \mathbb{A}$. In effect, we can obtain such a solution by requiring each of the arguments to if on the LHS to be equal to the corresponding argument on the RHS.

$$\begin{aligned} \Sigma, \alpha : \mathbb{A}; x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \lambda.\mathbb{A} &\approx \lambda.\mathbb{A} : (\mathrm{Bool} \to \mathbb{A}) \ddagger (\mathrm{Bool} \to \mathbb{A}) \\ x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash x &\approx x : \mathrm{Bool} \ddagger \mathrm{Bool} \\ x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathbb{a} &\approx \mathbb{a} : \mathbb{A} \ddagger \mathbb{A} \\ x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \alpha &\approx \mathbb{b} : \mathbb{A} \ddagger \mathbb{A} \end{aligned}$$

By applying Rule schema 1 (syntactic equality) and Rule schema 2 (metavariable instantiation), we have $\Sigma, \alpha := \mathbb{b} : \mathbb{A}; \square$. By definition, $\textsc{close}(\Sigma, \alpha := \mathbb{b} : \mathbb{A}) \Downarrow \Theta$. ◀

However, this approach does not by itself lead to a correct rule schema, as, in the general case, solutions may be lost:

Example 4.41 (No solutions)**.** Consider the problem:

$$\Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \cdot \vdash \text{if}\,(\lambda.\mathbb{A})\,\alpha\,\mathbb{b}\,\mathbb{a} \equiv \text{if}\,(\lambda.\mathbb{A})\,\beta\,\mathbb{a}\,\mathbb{b} : \mathbb{A}$$

This problem has a solution, namely $\Theta = \Sigma, \alpha := \text{true} : \text{Bool}, \beta := \text{false} : \text{Bool}$.

Analogously to Example 4.40 (strong neutral unification), we can solve the problem by solving the following problem instead:

$$\begin{aligned}\Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \cdot &\vdash \lambda.\mathbb{A} \approx \lambda.\mathbb{A} : \text{Bool} \to \mathbb{A}\\ \cdot &\vdash \alpha \approx \beta : \text{Bool}\\ \cdot &\vdash \mathbb{a} \approx \mathbb{b} : \mathbb{A}\\ \cdot &\vdash \mathbb{b} \approx \mathbb{a} : \mathbb{A}\end{aligned}$$

However, Θ is no longer a solution of the resulting problem, as this would imply $\Theta; \cdot \vdash \mathbb{a} \equiv \mathbb{b} : \mathbb{A}$. ◀

Even when a solution is found, the solution might not be unique:

Example 4.42 (Non-unique solutions)**.** Consider the problem:

$$\begin{aligned}\Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \cdot &\vdash \beta \approx \text{true} : \text{Bool}\\ \cdot &\vdash \text{if}\,(\lambda.\mathbb{A})\,\alpha\,\mathbb{a}\,\mathbb{a} \equiv \text{if}\,(\lambda.\mathbb{A})\beta\,\mathbb{a}\,\mathbb{a} : \mathbb{A}\end{aligned}$$

This problem has two solutions, namely $\Theta_1 = \Sigma, \alpha := \text{true} : \text{Bool}, \beta := \text{true} : \text{Bool}$, $\Theta_2 = \Sigma, \alpha := \text{false} : \text{Bool}, \beta := \text{true} : \text{Bool}$.

Analogously to Example 4.40 (strong neutral unification), we can solve the problem by solving the following problem instead:

$$\begin{aligned}\Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \cdot &\vdash \beta \approx \text{true} : \text{Bool}\\ \cdot &\vdash \lambda.\mathbb{A} \approx \lambda.\mathbb{A} : \text{Bool} \to \mathbb{A}\\ \cdot &\vdash \alpha \approx \beta : \text{Bool}\\ \cdot &\vdash \mathbb{a} \approx \mathbb{a} : \mathbb{A}\\ \cdot &\vdash \mathbb{a} \approx \mathbb{a} : \mathbb{A}\end{aligned}$$

By applying Rule schema 1 (syntactic equality) and Rule schema 2 (meta-variable instantiation), we have:

$$\Sigma, \beta := \text{true} : \text{Bool}, \alpha := \beta : \text{Bool}; \square$$

And $\textsc{close}(\Sigma, \beta := \text{true} : \text{Bool}, \alpha := \beta : \text{Bool}) \Downarrow \Theta_1$. However, the resulting solution (Θ_1) is not a unique solution to the original problem. ◀

We wish to reduce constraints involving neutral terms, such as the one in Example 4.40, without losing solutions, as in Example 4.41, or sacrificing uniqueness (Example 4.42).

Rule-Schema 14 (Strongly neutral terms)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash h\,\vec{e}^{\,n} \approx h\,\overrightarrow{e'}^{n} : T \ddagger T' \rightsquigarrow \Sigma; \bigwedge_{i \in J} \Gamma \ddagger \Gamma' \vdash t_i \approx u_i : B_i \ddagger B'_i$$

where

$J \subseteq \{1, ..., n\}$

$h\,\vec{e}$ and $h\,\vec{e}'$ are strongly neutral

for each $i \in \{1, ..., n\}$, either:

(i) $i \notin J$, and either $e_i = e'_i = .\pi_1$ or $e_i = e'_i = .\pi_2$,

(ii) $i \in J$, and there exist t_i, u_i, such that $e_i = t_i$, $e'_i = u_i$,

$\Sigma; \Gamma \vdash h \mathbin{\hat{@}} \vec{e}_{1,...,i-1} \Downarrow V_i$ and $\Sigma \vdash V_i \searrow \Pi B_i C_i$, and

$\Sigma; \Gamma' \vdash h \mathbin{\hat{@}} \vec{e}'_{1,...,i-1} \Downarrow V'_i$ and $\Sigma \vdash V_i \searrow \Pi B'_i C'_i$

Proof of correctness. By Lemma 2.106 (type elimination), Lemma 2.98 (equality of WHNF) and the CONV-EQ rule, for each $i \in J$, $\Sigma; \Gamma \vdash h\,\vec{e}_{1,...,i-1} : \Pi B_i C_i$ and $\Sigma; \Gamma' \vdash h\,\vec{e}'_{1,...,i-1} : \Pi B'_i C'_i$.

Let $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash h\,\vec{e}^{\,n} \approx h\,\vec{e}'^{n} : T \ddagger T'$. For each $i \in J$, let $\mathcal{D}_i = \Gamma \ddagger \Gamma' \vdash t \approx u : B_i \ddagger B'_i$ and $\overrightarrow{\mathcal{D}} = \bigwedge_{i \in J} \mathcal{D}_i$,

Well-formedness Assume that $\Sigma; \mathcal{C}$ is a well-formed problem; that is, Σ **sig**, $\Sigma; \Gamma \vdash h\,\vec{e} : T$ and $\Sigma; \Gamma' \vdash h\,\vec{e}' : T'$.

Let $i \in J$. By assumption, $\Sigma; \Gamma \vdash h\,\vec{e}_{1,...,i-1}\,t_i\,\vec{e}_{i+1,...,n} : T$, By the preconditions and Lemma 2.111 (application inversion), $\Sigma; \Gamma \vdash t : B_i$. Analogously, and $\Sigma; \Gamma \vdash u : B'_i$. Therefore $\Sigma; \mathcal{D}_i$ **wf**.

Because Σ **sig**, we have $\Sigma \sqsubseteq \Sigma$. By Definition 4.5, $\Sigma; \overrightarrow{\mathcal{D}}$ is a well-formed problem.

Soundness Let $\Sigma' \sqsupseteq \Sigma$. Assume that, for each $i \in J$, $\Sigma' \mid\!\approx \mathcal{D}_i$, that is, $\Sigma'; \Gamma \ddagger \Gamma' \vdash t_i \equiv\{v_i\}\equiv u_i : B_i \ddagger B'_i$ and $\Sigma'; \mathcal{D}_i$ **wf** $(\star)$.

Consider the term $h\,\overrightarrow{e''}^{n}$, where, for each $i \in \{1, ..., n\}$:

$$\begin{array}{lll} e''_i = v_i & \textbf{if} & i \in J \\ e''_i = .\pi_1 & \textbf{if} & e_i = e'_i = .\pi_1 \\ e''_i = .\pi_2 & \textbf{if} & e_i = e'_i = .\pi_2 \end{array}$$

(i) We show by induction on k that for all k, there are U_k and U'_k such that $\Sigma'; \Gamma \ddagger \Gamma' \vdash h\,\vec{e}_{1,...,k} \equiv\{h\,\overrightarrow{e''}_{1,...,k}\}\equiv h\,\overrightarrow{e'}_{1,...,k} : U_k$, that is, $\Sigma'; \Gamma \vdash h\,\vec{e}_{1,...,k} \equiv h\,\overrightarrow{e''}_{1,...,k} : U_k$, $\Sigma'; \Gamma' \vdash h\,\overrightarrow{e'}_{1,...,k} \equiv h\,\overrightarrow{e''}_{1,...,k} : U'_k$, and $\mathrm{FV}(h\,\overrightarrow{e''}_{1,...,k}) \subseteq \mathrm{FV}(h\,\vec{e}_{1,...,k}) \cap \mathrm{FV}(h\,\overrightarrow{e'}_{1,...,k})$.

- Case 0: Because $\Sigma; \mathcal{C}$ is well-formed, we have $\Sigma; \Gamma \vdash h\,\vec{e} : T$. By Lemma 2.155 (preservation of judgments under signature extensions), we have $\Sigma'; \Gamma \vdash h\,\vec{e} : T$.
  By Corollary 2.77 (uniqueness of typing for heads), there is U_0 such that $\Sigma'; \Gamma \vdash h \Rightarrow U_0$. By the HEAD-EQ rule, $\Sigma'; \Gamma \vdash h \equiv h : U_0$. Analogously, there is U'_0 such that $\Sigma'; \Gamma' \vdash h \equiv h : U'_0$.
  Trivially, $\mathrm{FV}(h) \subseteq \mathrm{FV}(h) \cap \mathrm{FV}(h)$.

- Case $k+1$:
  By the induction hypothesis, $\Sigma;\Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\overrightarrow{e''}_{1,\ldots,k} : U_k$, $\Sigma;\Gamma' \vdash h\,\overrightarrow{e'}_{1,\ldots,k} \equiv h\,\overrightarrow{e''}_{1,\ldots,k} : U'_k$, and $\mathrm{FV}(h\,\overrightarrow{e''}_{1,\ldots,k}) \subseteq \mathrm{FV}(h\,\vec{e}_{1,\ldots,k}) \cap \mathrm{FV}(h\,\overrightarrow{e'}_{1,\ldots,k})$.
  - Case $e''_{k+1} = v_{k+1}$, $e_{k+1} = t_{k+1}$ and $e'_{k+1} = u_{k+1}$: As shown above, $\Sigma;\Gamma \vdash h\,\vec{e}_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$. By Lemma 2.155 (preservation of judgments under signature extensions), $\Sigma';\Gamma \vdash h\,\vec{e}_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$.
    By Corollary 2.76 (uniqueness of typing for equality of neutrals), $\Sigma';\Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\overrightarrow{e''}_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$.
    By $(\star)$, and Definition 4.12 (heterogeneous equality), $\Sigma';\Gamma \vdash t_{k+1} \equiv v_{k+1} : B_{k+1}$. By the APP-EQ$_0$ rule, there is U_{k+1} (with $C_{k+1}[t_{k+1}] \Downarrow U_{k+1}$) such that $\Sigma';\Gamma \vdash h\,\vec{e}_{1,\ldots,k}\,t_{k+1} \equiv h\,\vec{e''}_{1,\ldots,k}\,v_{k+1} : U_{k+1}$.
    Analogously, there exists U'_{k+1} such that $\Sigma';\Gamma \vdash h\,\vec{e'}_{1,\ldots,k}\,t_{k+1} \equiv h\,\vec{e''}_{1,\ldots,k}\,v_{k+1} : U'_{k+1}$.
    Finally, by $(\star)$ and distributivity of $\cap$ over $\cup$,[1] we have $\mathrm{FV}(h\,\vec{e''}_{1,\ldots,k}\,v_{k+1}) = \mathrm{FV}(h\,\vec{e''}_{1,\ldots,k}) \cup \mathrm{FV}(v_{k+1}) \subseteq (\mathrm{FV}(h\,\vec{e}_{1,\ldots,k}) \cap \mathrm{FV}(h\,\vec{e'}_{1,\ldots,k})) \cup (\mathrm{FV}(t_{k+1}) \cap \mathrm{FV}(u_{k+1})) \subseteq (\mathrm{FV}(h\,\vec{e}_{1,\ldots,k}) \cup \mathrm{FV}(t_{k+1})) \cap (\mathrm{FV}(h\,\vec{e'}_{1,\ldots,k}) \cup \mathrm{FV}(u_{k+1})) = \mathrm{FV}(h\,\vec{e}_{1,\ldots,k}\,t_{k+1}) \cap \mathrm{FV}(h\,\vec{e'}_{1,\ldots,k}\,u_{k+1})$.
  - Case $e''_{k+1} = e_{k+1} = e'_{k+1} = .\pi_1$: By Lemma 2.113 (projection inversion), there are U, V such that $\Sigma';\Gamma \vdash h\,\vec{e}_{1,\ldots,k} : \Sigma U V$. By Corollary 2.76 (uniqueness of typing for equality of neutrals), $\Sigma';\Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\vec{e''}_{1,\ldots,k} : \Sigma U V$. Let $U_{k+1} \stackrel{\text{def}}{=} U$. By the PROJ1-EQ rule, $\Sigma';\Gamma \vdash h\,\vec{e}_{1,\ldots,k}.\pi_1 \equiv h\,\vec{e''}_{1,\ldots,k}.\pi_1 : U_{k+1}$. Analogously, there exists U'_{k+1} such that $\Sigma';\Gamma' \vdash h\,\vec{e'}_{1,\ldots,k}.\pi_1 \equiv h\,\vec{e''}_{1,\ldots,k}.\pi_1 : U'_{k+1}$.
    By the induction hypothesis, $\mathrm{FV}(h\,\vec{e''}_{1,\ldots,k}.\pi_1) = \mathrm{FV}(h\,\vec{e''}_{1,\ldots,k}) \subseteq \mathrm{FV}(h\,\vec{e}_{1,\ldots,k}) \cap \mathrm{FV}(h\,\vec{e'}_{1,\ldots,k}) = \mathrm{FV}(h\,\vec{e}_{1,\ldots,k}.\pi_1) \cap \mathrm{FV}(h\,\vec{e'}_{1,\ldots,k}.\pi_1)$.
  - Case $e''_{k+1} = e_{k+1} = e'_{k+1} = .\pi_2$: As in the previous case, $\Sigma;\Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\vec{e''}_{1,\ldots,k} : \Sigma U V$. By Remark 2.36 (hereditary substitution by a neutral term), there exists U_{k+1} such that $V[h\,\vec{e}_{1,\ldots,k}] \Downarrow U_{k+1}$. By the PROJ2-EQ rule, $\Sigma;\Gamma \vdash h\,\vec{e}_{1,\ldots,k}.\pi_2 \equiv h\,\vec{e''}_{1,\ldots,k}.\pi_2 : U_{k+1}$.
    Analogously, there exists U'_{k+1} such that $\Sigma';\Gamma' \vdash h\,\vec{e'}_{1,\ldots,k}.\pi_2 \equiv h\,\vec{e''}_{1,\ldots,k}.\pi_2 : U'_{k+1}$.
    Finally, by the induction hypothesis, $\mathrm{FV}(h\,\vec{e''}_{1,\ldots,k}.\pi_2) = \mathrm{FV}(h\,\vec{e''}_{1,\ldots,k}) \subseteq \mathrm{FV}(h\,\vec{e}_{1,\ldots,k}) \cap \mathrm{FV}(h\,\vec{e'}_{1,\ldots,k}) = \mathrm{FV}(h\,\vec{e}_{1,\ldots,k}.\pi_2) \cap \mathrm{FV}(h\,\vec{e'}_{1,\ldots,k}.\pi_2)$.

(ii) By taking $k = n$ in (i), we have $\Sigma';\Gamma \vdash h\,\vec{e} \equiv h\,\vec{e''} : U_n$ and $\Sigma';\Gamma \vdash h\,\vec{e'} \equiv h\,\vec{e''} : U'_n$, with $\mathrm{FV}(h\,\vec{e''}) \subseteq \mathrm{FV}(h\,\vec{e}) \cap \mathrm{FV}(h\,\vec{e'})$.

Because the original problem is well-formed, we have $\Sigma';\Gamma \vdash h\,\vec{e} : T$ and $\Sigma';\Gamma \vdash h\,\vec{e'} : T'$. By Corollary 2.76 (uniqueness of typing for

[1] Given A, B, C and D sets, we have $(A \cap C) \cup (B \cap D) \subseteq (A \cap C) \cup (B \cap D) \cup (A \cap D) \cup (B \cap C) = (A \cup B) \cap (C \cup D)$.

equality of neutrals), $\Sigma'; \Gamma \vdash h\,\vec{e} \equiv h\,\vec{e}'' : T$ and $\Sigma'; \Gamma \vdash h\,\vec{e}' \equiv h\,\vec{e}'' : T'$.

By Definition 4.12 (heterogeneous equality), $\Sigma'; \Gamma ‡ \Gamma' \vdash h\,\vec{e} \equiv \{h\,\vec{e}''\} \equiv h\,\vec{e}' : T ‡ T'$. Therefore, $\Sigma' \models\!\approx \mathcal{C}$.

Completeness First, note that, by Remark 2.159 (prefixes of strongly neutral terms), because $h\,\vec{e}$ and $h\,\vec{e}'$ are strongly neutral, we have that for all k, $h\,\vec{e}_{1,\ldots,k}$ and $h\,\vec{e}'_{1,\ldots,k}$ are also strongly neutral. $(\star)$

Assume that $\Theta \vDash \Sigma; \mathcal{C}$. By Definition 4.11 (solution to a unification problem), assume that $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, T \equiv \Gamma', T'$ **ctx** and $\Theta; \Gamma \vdash h\,\vec{e} \equiv h\,\vec{e}' : T$.

Take $\Theta' = \Theta$. By Remark 2.17 (context equality inversion), $\Theta \vdash \Gamma \equiv \Gamma'$ **ctx**.

By induction on k, we show that for every k, if there exists U such that $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\vec{e}'_{1,\ldots,k} : U$, then for every $i \in J$, $i \leq k$, we have $\Theta \vDash \mathcal{D}_i$.

- Case 0: Vacuously true ($i \in \{1, ..., 0\} = \emptyset$).
- Case $k+1$: Assume that there exists U_{k+1} such that $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k+1} \equiv h\,\vec{e}'_{1,\ldots,k+1} : U_{k+1}$. By the rule preconditions, we have two possible cases for e_{k+1} and e'_{k+1}:
  - Case $e_{k+1} = t_{k+1}$, $e'_{k+1} = u_{k+1}$, and $k+1 \in J$: Assume there exists U such that $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k+1} \equiv h\,\overrightarrow{e'}_{1,\ldots,k+1} : U$.
    (i) By Lemma 2.163 (injectivity of elimination for strongly neutral terms), there exist B^0_{k+1}, C^0_{k+1} such that $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\vec{e}'_{1,\ldots,k} : \Pi B^0_{k+1} C^0_{k+1}$ and $\Theta; \Gamma \vdash t_{k+1} \equiv u_{k+1} : B^0_{k+1}$.
    (ii) Because $\Theta \vDash \Sigma$, we have $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$.
    By (i) and Lemma 2.75 (uniqueness of typing for neutrals), we have $\Theta; \Gamma \vdash \Pi B^0_{k+1} C^0_{k+1} \equiv \Pi B_{k+1} C_{k+1}$ **type**. By Postulate 10 (injectivity of Π), we have $\Theta; \Gamma \vdash B^0_{k+1} \equiv B_{k+1}$ **type**.
    By the CONV-EQ rule, we have $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\vec{e}'_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$ and $\Theta; \Gamma \vdash t_{k+1} \equiv u_{k+1} : B_{k+1}$.
    (iii) By (ii) and Lemma 2.70 (piecewise well-formedness of typing judgments), $\Theta; \Gamma \vdash h\,\overrightarrow{e'}_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$. By assumption, $\Theta \vDash \Sigma$ and $\Sigma; \Gamma \vdash h\,\vec{e}'_{1,\ldots,k} : \Pi B'_{k+1} C'_{k+1}$. By Definition 2.125 (compatible metasubstitution), $\Theta; \Gamma \vdash h\,\vec{e}'_{1,\ldots,k} : \Pi B'_{k+1} C'_{k+1}$. By Lemma 2.75 (uniqueness of typing for neutrals), $\Theta; \Gamma \vdash \Pi B_{k+1} C_{k+1} \equiv \Pi B'_{k+1} C'_{k+1}$ **type**. By Postulate 10 (injectivity of Π), $\Theta; \Gamma \vdash B_{k+1} \equiv B'_{k+1}$ **type**.

    Take $i \in \{1, ..., k+1\} \cap J$. If $i = k+1$, then, by (ii), $\Theta; \Gamma \vdash t_{k+1} \equiv u_{k+1} : B_{k+1}$, and by (iii), $\Theta; \Gamma \vdash B_{k+1} \equiv B'_{k+1}$ **type**. Therefore $\Theta \vDash \mathcal{D}_i$.
    If $i \leq k$, then by (ii), $\Theta; \Gamma \vdash h\,\vec{e}_{1,\ldots,k} \equiv h\,\vec{e}'_{1,\ldots,k} : \Pi B_{k+1} C_{k+1}$. By the induction hypothesis, $\Theta \vDash \mathcal{D}_i$.

– Case $e_{k+1} = e'_{k+1} = .\pi_1$, or $e_{k+1} = e'_{k+1} = .\pi_2$, and $k+1 \notin J$: By Lemma 2.163, $\Theta; \Gamma \vdash h\,\vec{e}_{1,\dots,k} \equiv h\,\vec{e}'_{1,\dots,k} : T_0$. Take $i \in \{1, ..., k+1\}$. If $i \in J$, then necessarily $i \leq k$. By the induction hypothesis, $\Theta \vDash \mathcal{D}_i$.

By $(\star)$, $\Theta; \Gamma \vdash h\,\vec{e}_{1,\dots,n} \equiv h\,\vec{e}'_{1,\dots,n} : T$. Note that $\forall i \in J, i \leq n$. By the proven property, taking $k := n$, we have that for all $i \in J$, $\Theta \vDash \mathcal{D}_i$.

By the assumption, $\Theta \vDash \Sigma$. Therefore, by Definition 4.11, $\Theta \vDash \Sigma; \overrightarrow{\mathcal{D}}$.

□

4.5.9 Metavariable argument killing

In some cases, we may be able to deduce that the body of a metavariable cannot depend on some of its arguments.

By including this information in the signature, we can simplify existing constraints. This may allow us to instantiate more metavariables.

Example 4.43 (Good pruning)**.** Consider the following problem, where $\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{o} : \mathbb{A}, \mathbb{F} : \mathbb{A} \to \mathbb{A}$:

$$\begin{aligned} \Sigma, \alpha : \text{Bool} \to \mathbb{A}, \beta : \mathbb{A}; x : \text{Bool} \ddagger \text{Bool} \vdash \quad \beta \quad &\approx \mathbb{F}\,(\alpha\, x) : \mathbb{A} \ddagger \mathbb{A} \\ \cdot \vdash \alpha\,\text{true} &\approx \mathbb{o} \quad : \mathbb{A} \ddagger \mathbb{A} \end{aligned}$$

This problem has solution $\Theta = \Sigma, \alpha := \lambda x.\mathbb{o} : \text{Bool} \to \mathbb{A}, \beta := \mathbb{F}\mathbb{o} : \mathbb{A}$. However, there is no clear way of finding this solution with the rules described so far:

(i) In the first constraint, $x \in \text{FV}(\mathbb{F}\,(\alpha\, x))$, but x is not in the arguments of β; and in the second constraint, α has a non-variable argument; therefore, the Rule schema 2 (metavariable instantiation) does not apply to either of them.

(ii) In both of the constraints, there is at least one side which is not a strongly neutral term. Therefore, Rule schema 14 (strongly neutral terms) does not apply to any of them.

(iii) Finally, because none of the terms in the constraints can be reduced further, there is no clear way in which Rule schema 8 (term conversion) or Rule schema 9 (type and context conversion) could change the constraints so that any of the above-mentioned rules would apply.

Observe that the variable x does not appear in the arguments of β. Thus, we may (correctly) assume that x is not actually used by α. Under this assumption we may "kill" the argument of α as follows:

$$\begin{aligned} \Sigma, \gamma : \mathbb{A}, \alpha := \lambda x.\gamma : \text{Bool} \to \mathbb{A}, \beta : \mathbb{A}; x : \text{Bool} \ddagger \text{Bool} \vdash \beta &\approx \mathbb{F}\,(\alpha\, x) : \mathbb{A} \ddagger \mathbb{A} \\ \cdot \vdash \alpha\,\text{true} &\approx \mathbb{o} : \mathbb{A} \ddagger \mathbb{A} \end{aligned}$$

By applying Rule schema 8 (term conversion) twice, we obtain:

$$\begin{aligned}\Sigma, \gamma : \mathbb{A}, \alpha := \lambda x.\gamma : \mathrm{Bool} \to \mathbb{A}, \beta : \mathbb{A}; x : \mathrm{Bool}\ddagger\mathrm{Bool} \vdash \beta \approx \mathbb{F}\,\gamma : \mathbb{A}\ddagger\mathbb{A} \\ \cdot \vdash \gamma \approx \mathbb{o} \quad : \mathbb{A}\ddagger\mathbb{A}\end{aligned}$$

And now, by applying Rule schema 2 (metavariable instantiation) twice, we obtain:

$$\Sigma, \gamma := \mathbb{o} : \mathbb{A}, \alpha := \lambda x.\gamma : \mathrm{Bool} \to \mathbb{A}, \beta := \mathbb{F}\,\mathbb{o} : \mathbb{A}; \square$$

Finally, by Definition 2.143 (closing metasubstitution), $\textsc{close}(\Sigma, \gamma := \mathbb{o} : \mathbb{A}, \alpha := \lambda x.\gamma : \mathrm{Bool} \to \mathbb{A}, \beta := \mathbb{F}\,\mathbb{o} : \mathbb{A}) \Downarrow \Theta'$, and $\Theta'_{\Sigma,\alpha:\mathrm{Bool}\to\mathbb{A},\beta:\mathbb{A}} = \Theta$. ◀

Example 4.44 (Bad pruning)**.** The approach in Example 4.43 is not always correct.

Consider the following problem, where $\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{o} : \mathbb{A}, \mathbb{F} : \mathbb{A} \to \mathbb{A}$:

$$\begin{aligned}\Sigma, \alpha : \mathbb{A} \to \mathbb{A}, \beta : \mathbb{A}, \gamma : \mathbb{A} \to \mathbb{A}\ ;\ x : \mathrm{Bool}\ddagger\mathrm{Bool} \vdash \beta \approx \gamma\,(\alpha\,x) : \mathbb{A}\ddagger\mathbb{A} \\ \wedge \cdot \vdash \alpha\,x \approx x : \mathbb{A}\ddagger\mathbb{A} \\ \wedge \cdot \vdash \gamma\,x \approx \mathbb{o} : \mathbb{A}\ddagger\mathbb{A}\end{aligned}$$

The problem has the solution $\Theta \stackrel{\text{def}}{=} \Sigma, \alpha := \lambda x.x : \mathbb{A} \to \mathbb{A}, \beta := \mathbb{o} : \mathbb{A}, \gamma := \lambda x.\mathbb{o} : \mathbb{A} \to \mathbb{A}$.

If we kill the first argument of α, we obtain the following problem:

$$\begin{aligned}\Sigma, \delta : \mathbb{A}, \alpha := \lambda x.\delta : \mathbb{A} \to \mathbb{A}, \beta : \mathbb{A}, \gamma : \mathbb{A} \to \mathbb{A}; x : \mathrm{Bool}\ddagger\mathrm{Bool} \vdash \beta \approx \gamma\,\delta : \mathbb{A}\ddagger\mathbb{A} \\ \cdot \vdash \delta \approx x : \mathbb{A}\ddagger\mathbb{A} \\ \cdot \vdash \gamma\,x \approx \mathbb{o} : \mathbb{A}\ddagger\mathbb{A}\end{aligned}$$

Because metavariables can only be instantiated to closed terms, the constraint $\cdot \vdash \delta \approx x : \mathbb{A}\ddagger\mathbb{A}$ is unsolvable. The resulting problem does not have the solution Θ. Therefore, it was not correct to kill the argument. ◀

We want to "kill" metavariable arguments in cases such as Example 4.43, while avoiding cases such as Example 4.44.

In this section we introduce a notion of killing arguments, and use it for specifying two correct rule schemas; namely Rule schema 16 (generalized metavariable intersection) and Rule schema 17 (metavariable pruning).

Definition 4.45 (Metavariable argument killing: $\Sigma \vdash \textsc{kill}(\alpha, n) \mapsto \Sigma'$)**.** We say that killing the n-th argument of metavariable α in signature Σ yields signature Σ' (written $\Sigma \vdash \textsc{kill}(\alpha, n) \mapsto \Sigma'$), if all the following hold:

(i) $n \in \mathbb{N}$, $n \geq 1$,

(ii) Σ **sig**, $\Sigma = \Sigma_1, \alpha : T, \Sigma_2$ for some Σ_1, Σ_2 and T.

(iii) $\Sigma' = \Sigma_1, \beta : \Pi \vec{A}^{n-1} U, \alpha := \lambda \vec{x}^{n-1}.\lambda y.\beta\, \vec{x} : T, \Sigma_2$ for some β, $\vec{A}$ and U with with $\beta \notin \textsc{decls}(\Sigma)$; and

(iv) $\Sigma_1; \cdot \vdash T \equiv \Pi \vec{A}^{n-1} \Pi B U^{(+1)}$ **type** for some B.

Lemma 4.46 (Well-formedness of killing)**.** *Assume* $\Sigma \vdash \textsc{kill}(\alpha, n) \mapsto \Sigma'$, *where* $\Sigma = \Sigma_1, \alpha : T, \Sigma_2$, *and* $\Sigma' = \Sigma_1, \beta : T', \alpha := t : T, \Sigma_2$ *for some* Σ_1, Σ_2, T, T' *and* t. *Then* Σ' **sig** *and* $\Sigma \sqsubseteq \Sigma'$.

Proof. Follows by construction, Postulate 13 (context strengthening), and the typing rules.

(i) By the assumption Σ **sig**; by Remark 2.5 (signature inversion), Σ_1 **sig**.

(ii) Also by the assumption, $\Sigma_1; \cdot \vdash T \equiv \Pi \vec{A}^{n-1} \Pi B U^{(+1)}$ **type**. By Lemma 2.70 (piecewise well-formedness of typing judgments) and Lemma 2.52 (Π inversion), gives $\Sigma_1; \vec{A}^{n-1}, B \vdash U^{(+1)}$ **type**. By Postulate 13 (context strengthening), $\Sigma_1; \vec{A}^{n-1} \vdash U$ **type**. By repeated application of the PI rule, $\Sigma_1; \cdot \vdash \Pi \vec{A}^{n-1}.U$ **type**, that is, $\Sigma_1; \cdot \vdash T'$ **type**. By Definition 4.45 (metavariable argument killing), $\beta \notin \textsc{decls}(\Sigma) \supseteq \textsc{decls}(\Sigma_1)$. Therefore, $\Sigma_1, \beta : T'$ **sig**. By Definition 2.151 (signature extension), $\Sigma_1 \sqsubseteq \Sigma_1, \beta : T'$. By Corollary 2.156 (horizontal composition of extensions), $\Sigma_1, \alpha : T, \Sigma_2 \sqsubseteq \Sigma_1, \beta : T', \alpha : T, \Sigma_2$.

(iii) By Lemma 2.69 (signature weakening), $\Sigma_1, \beta : T'; \cdot \vdash T$ **type**.

By the typing rules, $\Sigma_1, \beta : T'; \overrightarrow{x : A}^{n-1} \vdash \beta\, \vec{x} : U$. By Lemma 2.62 (context weakening), $\Sigma_1, \beta : T'; \overrightarrow{x : A}^{n-1}, y : B \vdash \beta\, \vec{x} : U^{(+1)}$. By the ABS rule, $\Sigma_1, \beta : T'; \cdot \vdash \lambda \vec{x}^{n-1}.\lambda y.\beta\, \vec{x} : \Pi \vec{A}^{n-1} \Pi B U$, that is, $\Sigma_1, \beta : T'; \cdot \vdash t : \Pi \vec{A}^{n-1} \Pi B U$.

By the assumption, $\Sigma_1; \cdot \vdash T \equiv \Pi \vec{A}^{n-1} \Pi B U$ **type**. Therefore, by the CONV-EQ rule, $\Sigma_1, \beta : T'; \cdot \vdash t : T$. Therefore, $\Sigma_1, \beta : T', \alpha := t : T$ **sig**.

By Definition 2.151 (signature extension), $\Sigma_1, \beta : T', \alpha : T \sqsubseteq \Sigma_1, \beta : T', \alpha := t : T$. By Corollary 2.156 (horizontal composition of extensions), $\Sigma_1, \beta : T', \alpha : T, \Sigma_2 \sqsubseteq \Sigma_1, \beta : T', \alpha := t : T, \Sigma_2$.

By (ii), (iii) and composition rule in Definition 2.151 (signature extension), $\Sigma \sqsubseteq \Sigma'$. □

Killing an argument of a metavariable α preserves a solution if the body of the metavariable in that solution does not depend on the killed argument.

Lemma 4.47 (Completeness of killing)**.** *Assume that* $\Sigma \vdash \textsc{kill}(\alpha, n) \mapsto \Sigma'$ *where for some* T_α, T_β *and* t_α, *we have* $\Sigma = \Sigma_1, \alpha : T_\alpha, \Sigma_2$ *and* $\Sigma' = \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T_\alpha, \Sigma_2$.

Also, let Θ *be a metasubstitution such that* $\Theta \vDash \Sigma$.

If there is v *and* T *such that* $\Theta; \cdot \vdash \alpha \equiv \lambda^n.v : T$, *with* $0 \notin \textsc{fv}(v)$, *then there are* u_β *and* U_β *such that, for* $\Theta' = \Theta, \beta := u_\beta : U_\beta$, *we have:*

(i) Θ' **wf**,

(ii) $\Theta'_\Sigma = \Theta$,

(iii) $\Theta \sqsubseteq \Theta'$, *and*

(iv) $\Theta' \vDash \Sigma'$.

Proof. By Lemma 2.75 (uniqueness of typing for neutrals), Lemma 2.70 (piecewise well-formedness of typing judgments) and the CONV-EQ rule, $\Theta; \cdot \vdash \alpha \equiv \lambda^n.v : T_\alpha$.

By Definition 4.45 (metavariable argument killing) and Lemma 2.69 (signature weakening), $\Sigma; \cdot \vdash T_\alpha \equiv \Pi\vec{U}^n V$ **type** for some U and V, with $0 \notin \mathrm{FV}(V)$. By Definition 2.125 (compatible metasubstitution), $\Theta; \cdot \vdash T_\alpha \equiv \Pi\vec{U}^n V$ **type**.

By the CONV-EQ rule, $\Theta; \cdot \vdash \alpha \equiv \lambda^n.v : \Pi\vec{U}^n V$.

By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Theta; \cdot \vdash \lambda^n.v : \Pi\vec{U}^n V$. By Corollary 2.58 (iterated λ-inversion) and Postulate 13 (context strengthening), we have $\Theta; \vec{U}_{1,\dots,n-1} \vdash v^{(-1)} : V^{(-1)}$. By the PI rule, $\Theta; \cdot \vdash \lambda^{n-1}.v^{(-1)} : \Pi\overrightarrow{U}_{1,\dots,n-1}(V^{(-1)})$. Take $T_\beta \stackrel{\text{def}}{\equiv} \Pi\overrightarrow{U}_{1,\dots,n-1}(V^{(-1)})$. By Remark 2.142 (metavariable-free term), we have u_β and U_β such that $\Theta; \cdot \vdash U_\beta \equiv T_\beta$ **type**, $\Theta; \cdot \vdash u_\beta : U_\beta$, $\Theta; \cdot \vdash u_\beta \equiv \lambda^{n-1}.v^{(-1)} : U_\beta$, and $\mathrm{METAS}(u_\beta) = \mathrm{METAS}(U_\beta) = \emptyset$.

(i) Θ' **wf**: By assumption, Θ **wf**. By the SUBST-META rule, Θ' **wf**.

(ii) $\Theta'_\Sigma = \Theta$: By construction.

(iii) $\Theta \sqsubseteq \Theta'$: By Definition 2.151 (signature extension).

(iv) $\Theta' \vDash \Sigma'$: By Lemma 2.130 (alternative characterization of a compatible metasubstitution), it suffices to show that, for each declaration $D \in \Sigma'$, Θ' is compatible with D. Because $\Theta \vDash \Sigma$ and $\Theta \subseteq \Theta'$, by Remark 2.128 (compatibility with a declaration as a judgment) and Remark 2.137 (metasubstitution weakening), it suffices to consider the two declarations that are in Σ', but not in Σ.

1. Θ' compatible with $\alpha := t_\alpha : T_\alpha$: Because $\Theta; \cdot \vdash \alpha \equiv \lambda\vec{z}^n.v : \Pi\overrightarrow{U}^n V$, $\Theta \subseteq \Theta'$ and Remark 2.137 (metasubstitution weakening), we have $\Theta'; \cdot \vdash \alpha \equiv \lambda\vec{z}^n.v : \Pi\overrightarrow{U}^n V$.

   By DELTA-META$_0$, transitivity and CONV-EQ, $\Theta'; \cdot \vdash \beta \equiv \lambda^{n-1}.v^{(-1)} : \Pi\overrightarrow{U}_{1,\dots,n-1}(V^{(-1)})$. By Lemma 4.34 (general η-equality for Π-types), symmetry and transitivity, $\Theta'; \cdot \vdash \lambda\vec{x}^{n-1}.\beta\,\vec{x} \equiv \lambda^{n-1}.v^{(-1)} : \Pi\overrightarrow{U}_{1,\dots,n-1}(V^{(-1)})$. By Lemma 2.59 (abstraction equality inversion), $\Theta'; \overrightarrow{x : U}_{1,\dots,n-1} \vdash \beta\,\vec{x}_{1,\dots,n-1} \equiv v^{(-1)} : V^{(-1)}$. By Lemma 2.62 (context weakening) $\Theta'; \overrightarrow{x : U}_{1,\dots,n-1}, y : U_n \vdash \beta\,\vec{x}_{1,\dots,n-1} \equiv v : V$. By Lemma 2.59 (abstraction equality inversion), $\Theta'; \cdot \vdash \lambda\vec{x}^{n-1}.\lambda y.\beta\,\vec{x} \equiv \lambda\vec{z}^n.v : \Pi\overrightarrow{U}V$.

   By symmetry and transitivity, $\Theta'; \cdot \vdash \alpha \equiv \lambda\vec{x}^{n-1}.\lambda y.\beta\,\vec{x} : \Pi\overrightarrow{U}V$. By Definition 4.45 (metavariable argument killing), $t_\alpha = \lambda\vec{x}^{n-1}.\lambda y.\beta\,\vec{x}$. Therefore, $\Theta'; \cdot \vdash \alpha \equiv t_\alpha : \Pi\overrightarrow{U}V$.

Because $\Theta \vDash \Sigma$, we have $\Theta; \cdot \vdash \alpha : T_\alpha$. By Lemma 2.75 (uniqueness of typing for neutrals) and Remark 2.137 (metasubstitution weakening), $\Theta'; \cdot \vdash T_\alpha \equiv \Pi \vec{U} V$ **type**. By the CONV-EQ rule, $\Theta'; \cdot \vdash \alpha \equiv t_\alpha : T_\alpha$.

2. Θ' compatible with $\beta : T_\beta$: By construction, $\beta : U_\beta \in \Theta'$. By the META_1 and HEAD rules, $\Theta'; \cdot \vdash \beta : U_\beta$.

   As shown above, $\Theta; \cdot \vdash U_\beta \equiv T_\beta$ **type**. Because $\Theta' \sqsupseteq \Theta$, by Lemma 2.155 (preservation of judgments under signature extensions), $\Theta'; \cdot \vdash U_\beta \equiv T_\beta$ **type**. By the CONV rule, $\Theta'; \cdot \vdash \beta : T_\beta$.

□

Metavariable intersection

One case where we may kill metavariable arguments is when both sides are headed by the same metavariable, but some of the arguments differ. We first prove the following lemma:

Lemma 4.48 (Intersection)**.** *Assume that* $\Theta; \Gamma \vdash \alpha\, \vec{f} \equiv \alpha\, \vec{f'} : A$*, where* $\vec{f} = \overrightarrow{f_1}^{n}\, \overrightarrow{f_2}^{1+m}$*,* $\vec{f'} = \overrightarrow{f'_1}^{n}\, \overrightarrow{f'_2}^{1+m}$*, and, for all* $f \in \vec{f}$*, or* $f \in \vec{f'}$*,* f *is irreducible. Also, assume that* $f_{n+1} = x\, \vec{e}$*,* $f'_{n+1} = y\, \vec{e'}$*, and* $x \neq y$*.*

Then there are v *and* T *such that* $\Theta; \cdot \vdash \alpha \equiv \lambda^{n+1}.v : T$*, where* $0 \notin \text{FV}(v)$*.*

Proof. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Theta; \Gamma \vdash \alpha\, \vec{f} : A$. By Lemma 2.112 (iterated application inversion), $\Theta; \cdot \vdash \alpha : \Pi \vec{U}^{n+1+m} V$ for some $\vec{U}$ and V. By Lemma 4.34 (general η-equality for Π-types), $\Theta; \cdot \vdash \alpha \equiv \lambda \vec{z}^{n+1+m}.\alpha\, \vec{z}^{n+1+m} : \Pi U^{n+1+m} V$.

By Lemma 2.166 (reduction at Π-type), there is v such that $\Theta; \cdot \vdash \lambda \vec{z}^{n+1+m}.\alpha\, \vec{z}^{n+1+m} \longrightarrow^\star_{\delta\eta} \lambda \vec{z}^{n+1+m}.v : \Pi \vec{U} V$, and $\Theta; \cdot \vdash \lambda \vec{z}^{n+1+m}.v \not\longrightarrow_{\delta\eta} : \Pi \vec{U} V$,

By Lemma 2.86 (equality of $\delta\eta$-reduct), it suffices to show that $z_{n+1} = m \notin \text{FV}(v)$.

By the assumption, $\Theta; \Gamma \vdash \alpha\, \overrightarrow{f_1}^{n}\, (x\, \vec{e})\, \overrightarrow{f_2}^{m} \equiv \alpha\, \overrightarrow{f'_1}^{n}\, (y\, \vec{e'})\, \overrightarrow{f'_2}^{m} : A$. By Lemma 2.86 (equality of $\delta\eta$-reduct) and transitivity, $\Theta; \cdot \vdash \alpha \equiv \lambda \vec{z}^{n+1+m}.v : \Pi \overrightarrow{z : U}^{n+1+m} V$. In particular, by Lemma 2.62 (context weakening) and Remark 2.30 (properties of renamings), $\Theta; \Gamma \vdash \alpha \equiv \lambda \vec{z}^{n+1+m}.v : \Pi \overrightarrow{z : U}^{n+1+m} V$. By Remark 2.35 (iterated application as substitution on body), $(\lambda \vec{z}^{n+1+m}.v)\ @\ \vec{f} \Downarrow v[\vec{f}/\vec{z}]$ and $(\lambda \vec{z}^{n+1+m}.v)\ @\ \vec{f'} \Downarrow v[\vec{f'}/\vec{z}]$.

By Lemma 2.79 (typing and congruence of elimination), $\Theta; \Gamma \vdash \alpha\, \vec{f} \equiv v[\vec{f}/\vec{z}] : A$ and $\Theta; \Gamma \vdash \alpha\, \vec{f'} \equiv v[\vec{f'}/\vec{z}] : A$.

By transitivity and symmetry of equality, $\Theta; \Gamma \vdash v[\vec{f}/\vec{z}] \equiv v[\vec{f'}/\vec{z}] : A$.

By the APP rule, $\Theta; \Gamma \vdash \alpha\, \vec{f} : V[\vec{f}/\vec{z}]$. By Lemma 2.75 (uniqueness of typing for neutrals), $\Theta; \Gamma \vdash A \equiv V[\vec{f}/\vec{z}]$ **type**. By the CONV-EQ rule, $\Theta; \Gamma \vdash v[\vec{f}/\vec{z}] \equiv v[\vec{f'}/\vec{z}] : V[\vec{f}/\vec{z}]$.

Assume that $z_{n+1} \in \text{FV}(v)$. which by Lemma 2.176 (injectivity of normal forms with respect to irreducibles), means that $x = y$. This is a contradiction with the theorem premises. Therefore, $z_{n+1} \notin \text{FV}(v)$. □

Rule-Schema 15 (Metavariable intersection)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}^{\,n}\, x\, \vec{g}^{\,m} \approx \alpha\, \overrightarrow{f'}^{\,n}\, y\, \overrightarrow{g'}^{\,m} : A \ddagger A'$$
$$\rightsquigarrow\ \Sigma'; \Gamma \ddagger \Gamma' \vdash \beta\, \vec{f}\, \vec{g} \approx \beta\, \overrightarrow{f'}\, \overrightarrow{g'} : A \ddagger A'$$

where

$$x \neq y \tag{1}$$

all terms in $\vec{f}$, $\overrightarrow{f'}$, $\vec{g}$, $\overrightarrow{g'}$ are irreducible (2)

$$\Sigma \vdash \textsc{kill}(\alpha, n+1) \mapsto \Sigma', \text{ where } \Sigma = \Sigma_1, \alpha : U, \Sigma_2$$
$$\text{and } \Sigma' = \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2 \tag{3}$$

Condition 3 ensures that the resulting problem is well formed, while conditions 1 and 2 ensure the resulting problem has the same solutions as the original problem.

Proof of correctness. Let $\mathcal{C} \stackrel{\text{def}}{=} \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}\, x\, \vec{g} \approx \alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'} : A \ddagger A'$ and $\mathcal{D} \stackrel{\text{def}}{=} \Gamma \ddagger \Gamma' \vdash \beta\, \vec{f}\, \vec{g} \approx \beta\, \overrightarrow{f'}\, \overrightarrow{g'} : A \ddagger A'$.

Assume Σ **sig** and $\Sigma; \mathcal{C}$ **wf**.

Well-formedness By Lemma 4.46 (well-formedness of killing), we have that Σ' **sig**, and $\Sigma' \sqsupseteq \Sigma$.

Because $\Sigma; \mathcal{C}$ **wf**, we have $\Sigma; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} : A$. By Lemma 2.155 (preservation of judgments under signature extensions), $\Sigma'; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} : A$. By Definition 4.45 (metavariable argument killing) and the DELTA-META rule, $\Sigma'; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \beta\, \vec{f}\, \vec{g} : A$. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma'; \Gamma \vdash \beta\, \vec{f}\, \vec{g} : A$.

Analogously, $\Sigma'; \Gamma' \vdash \beta\, \overrightarrow{f'}\, \overrightarrow{g'} : A'$.

Therefore, $\Sigma'; \mathcal{D}$ **wf**.

Soundness Assume $\Sigma'' \sqsupseteq \Sigma'$, with $\Sigma'' \not\models \mathcal{D}$. That is, $\Sigma''; \mathcal{D}$ **wf**, and there is a term v such that $\Sigma''; \Gamma \vdash \beta\, \vec{f}\, \vec{g} \equiv v : A$, $\Sigma''; \Gamma' \vdash \beta\, \overrightarrow{f'}\, \overrightarrow{g'} \equiv v : A'$, $\textsc{fv}(v) \subseteq \textsc{fv}(\beta\, \vec{f}\, \vec{g})$ and $\textsc{fv}(v) \subseteq \textsc{fv}(\beta\, \overrightarrow{f'}\, \overrightarrow{g'})$.

By the same reasoning as in the well-formedness proof, $\Sigma'; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \beta\, \vec{f}\, \vec{g} : A$. By Lemma 2.155 (preservation of judgments under signature extensions), $\Sigma''; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \beta\, \vec{f}\, \vec{g} : A$. By transitivity, $\Sigma''; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv v : A$.

By Definition 2.18 (free variables in a term), $\textsc{fv}(\beta\, \vec{f}\, \vec{g}) \subseteq \textsc{fv}(\alpha\, \vec{f}\, x\, \vec{g})$; therefore, $\textsc{fv}(v) \subseteq \textsc{fv}(\alpha\, \vec{f}\, x\, \vec{g})$.

Analogously, $\Sigma''; \Gamma' \vdash \alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'} \equiv v : A'$ and $\textsc{fv}(v) \subseteq \textsc{fv}(\alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'})$.

By Definition 4.12 (heterogeneous equality), $\Sigma''; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv\!\{v\}\!\equiv \alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'} : A \ddagger A'$.

By Lemma 2.70 (piecewise well-formedness of typing judgments), Σ'' **sig**, $\Sigma''; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} : A$ and $\Sigma''; \Gamma' \vdash \alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'} : A'$. Note that $|\Gamma| = |\Gamma'|$. By Definition 4.2 (well-formed internal constraint), $\Sigma''; \mathcal{C}$ **wf**.

Therefore, $\Sigma'' \not\models \mathcal{C}$.

Completeness Assume that $\Theta \vDash \Sigma; \mathcal{C}$. This means that $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, A \equiv \Gamma', A'$ **ctx** and $\Theta; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'} : A$.

(i) By Definition 4.45 (metavariable argument killing), there exist $\overrightarrow{U'}^{n+1}$ and V, $0 \notin \textsc{fv}(V)$, and $\Sigma_1; \cdot \vdash U \equiv \Pi \overrightarrow{U'}^{n+1} V$ **type** such that $\Sigma_1 \subseteq \Sigma$. By Lemma 2.69 (signature weakening) and $\Theta \vDash \Sigma$, we have $\Theta; \cdot \vdash U \equiv \Pi \overrightarrow{U'}^{n+1} V$ **type**. By the CONV rule, $\Theta; \cdot \vdash \alpha : \Pi \overrightarrow{U'}^{n+1} V$.
By Lemma 4.48 (intersection), there exists u_0 such that $\Theta; \cdot \vdash \alpha \equiv \lambda \vec{x}^n . \lambda y . u_0 : \Pi \overrightarrow{U'}^{n+1} V$, and $y \notin \textsc{fv}(u_0)$. By Lemma 4.47 (completeness of killing), there is $\Theta' \vDash \Sigma'$ with $\Theta'_\Sigma = \Theta$.

(ii) By (i) and Remark 2.134 (subsumption of restriction), $\Theta' \supseteq \Theta$. By Lemma 2.69 (signature weakening), $\Theta' \vdash \Gamma, A \equiv \Gamma', A'$ **ctx** and $\Theta'; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \alpha\, \overrightarrow{f'}\, y\, \overrightarrow{g'} : A$.

(iii) By the same reasoning as in the well-formedness proof, $\Sigma'; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \beta\, \vec{f}\, \vec{g} : A$. Because $\Theta' \vDash \Sigma'$, we have $\Theta'; \Gamma \vdash \alpha\, \vec{f}\, x\, \vec{g} \equiv \beta\, \vec{f}\, \vec{g} : A$.
Analogously, $\Theta'; \Gamma' \vdash \alpha\, \overrightarrow{f'}\, x\, \overrightarrow{g'} \equiv \beta\, \overrightarrow{f'}\, \overrightarrow{g'} : A'$. By (ii) and Lemma 2.63 (preservation of judgments by type conversion), $\Theta'; \Gamma \vdash \alpha\, \overrightarrow{f'}\, x\, \overrightarrow{g'} \equiv \beta\, \vec{f'}\, \overrightarrow{g'} : A$. By transitivity of equality, $\Theta'; \Gamma \vdash \beta\, \vec{f}\, \vec{g} \equiv \beta\, \overrightarrow{f'}\, \overrightarrow{g'} : A$.

By (ii) and (iii), $\Theta' \vDash \mathcal{D}$. By (i), $\Theta' \vDash \Sigma'; \mathcal{D}$.

□

The proof above (via Lemma 4.48), does not use the fact that x and y are variables; only that they are irreducible terms with distinct heads.

Therefore, we could prove the correctness of the following, more general version of the rule using the same reasoning steps:

Rule-Schema 16 (Generalized metavariable intersection)**.**

$$\frac{\Sigma_1, \alpha : U, \Sigma_2; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}^n \approx \alpha\, \vec{g}^n : A \ddagger A' \quad \rightsquigarrow}{\Sigma_1, \beta : T, \alpha := u : U, \Sigma_2; \Gamma \ddagger \Gamma' \vdash \beta\, \vec{f}_{1,\ldots,i-1}\, \vec{f}_{i+1,\ldots,n} \approx \beta\, \vec{g}_{1,\ldots,i-1}\, \vec{g}_{i+1,\ldots,n} : A \ddagger A'}$$

where

$$i \in \{1, \ldots, n\}$$

$$f_i = h\, \vec{e},\ g_i = h'\, \vec{e}',\ h \neq h'$$

all terms in $\vec{f}$ and $\vec{g}$ are irreducible

$$\Sigma_1, \alpha : U, \Sigma_2 \vdash \textsc{kill}(\alpha, i) \mapsto \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2$$

Metavariable pruning

Another situation where we can kill an argument of a metavariable is when it is headed by *a variable which is not free on the other side of the constraint.* For completeness, the metavariable must occur in a rigid position, and all arguments to the metavariable must be irreducible.

Lemma 4.49 (Pruning)**.** *Let* $\vec{f} = \overrightarrow{f_1}^n\ (y^{(+k)}\ \vec{e})\ \overrightarrow{f_2}^m$ *where, for all* $f \in \overrightarrow{f_1}$ *or* $f \in \overrightarrow{f_2}$*, f is irreducible, and let α be a metavariable.*

Assume that $t_2[\![\alpha\ \vec{f}]\!]^k$*,* $\Theta;\Gamma \vdash t_1 \equiv t_2 : A$*, and* $y \notin \mathrm{FV}(t_1)$*.*
Then there exist Δ and B such that $|\Delta| = k$ $\Theta;\Gamma,\Delta \vdash \alpha\ \vec{f} : B$*,*
and there exist v_0 and T such that $\Theta;\cdot \vdash \alpha \equiv \lambda^{n+1}.v_0 : T_0$ *and* $0 \notin \mathrm{FV}(v_0)$*.*

Proof. By Lemma 2.170 (typing of rigid occurrences), there exist Δ and B such that $|\Delta| = k$ and $\Theta;\Gamma,\Delta \vdash \alpha\ \vec{f} : B$.

By Lemma 2.112 (iterated application inversion), there exist $\vec{U}$ and V such that $\Theta;\cdot \vdash \alpha : \Pi\vec{U}^{n+1+m}V$.

By Lemma 2.166 (reduction at Π-type), there is v such that $\Theta;\cdot \vdash \lambda\vec{z}^{n+1+m}.v\not\longrightarrow_{\delta\eta} : \Pi\vec{U}V$ and $\Theta;\cdot \vdash \lambda\vec{z}^{n+1+m}.\alpha\ \vec{z}^{n+1+m} \longrightarrow^\star_{\delta\eta} \lambda\vec{z}^{n+1+m}.v : \Pi\vec{U}^{n+1+m}V$, with $\vec{z}^{n+1+m} = ((n+1+m),...,1,0)$. Take $v_0 = \lambda^m.v$, and $T_0 := \Pi\vec{U}^{n+1+m}V$. By the ETA-ABS rule, transitivity of equality and Lemma 2.86 (equality of δη-reduct), $\Theta;\cdot \vdash \alpha \equiv \lambda^{n+1}.v_0 : T_0$.

It suffices to show that $0 \notin \mathrm{FV}(v_0)$, i.e. $z_{n+1} \notin \mathrm{FV}(v)$.

By the hypothesis, $\Theta;\Gamma \vdash t_1 \equiv t_2 : A$. By Remark 2.88 (existence of a common normal form), there is a term r such that $\Theta;\Gamma \vdash t_1 \longrightarrow^\star_{\delta\eta} r : A$, $\Theta;\Gamma \vdash t_2 \longrightarrow^\star_{\delta\eta} r : A$, and $\Theta;\Gamma \vdash r\not\longrightarrow_{\delta\eta} : A$.

Because $t_2[\![\alpha\ \vec{f}]\!]^k$, by Lemma 2.170 (typing of rigid occurrences), $\Theta;\Gamma \vdash t_2[\![\Delta \vdash \alpha\ \vec{f} : B']\!] : A$ for some Δ and B' with $|\Delta| = k$.

We have $\Theta;\Gamma \vdash t_2 \longrightarrow^\star_{\delta\eta} r : A$. By Remark 2.43 (free variables of δη-reduct) and Lemma 2.172 (free variables in reduction of rigid occurrences), there is u' such that $\Theta;\Gamma,\Delta \vdash \alpha\ \vec{f} \equiv u' : B'$ and $\mathrm{FV}(u') - |\Delta| \subseteq \mathrm{FV}(r)$. Because $y \notin \mathrm{FV}(r)$, this means $y + |\Delta| \notin \mathrm{FV}(u')$.

By Postulate 6 (congruence of hereditary application), $\Theta;\Gamma,\Delta \vdash \alpha\ \vec{f} \equiv v[\vec{f}] : V[\vec{f}]$. By Lemma 2.75 (uniqueness of typing for neutrals) and the CONV-EQ rule, $\Theta;\Gamma,\Delta \vdash \alpha\ \vec{f} \equiv v[\vec{f}] : B'$. By transitivity and symmetry, $\Theta;\Gamma,\Delta \vdash v[\vec{f}] \equiv u' : B'$.

We proceed by contradiction; assume $z_{n+1} \in \mathrm{FV}(v)$. Because $f_{n+1} = y^{(+|\Delta|)}\ \vec{e}$, by Lemma 2.175 (preservation of irreducibles by normal forms), this means $y^{(+|\Delta|)} \in \mathrm{FV}(u')$, which is a contradiction. Therefore, $z_{n+1} \notin \mathrm{FV}(v)$. □

Rule-Schema 17 (Metavariable pruning)**.**

$$\Sigma;\Gamma\ddagger\Gamma' \vdash v \approx t : A\ddagger A' \rightsquigarrow \Sigma';\Gamma\ddagger\Gamma' \vdash v \approx t : A\ddagger A'$$

where

$$t[\![\alpha\ \overrightarrow{f_1}^n\ (y\ \vec{e})\ \overrightarrow{f_2}^m]\!]^k \text{ for } k \in \mathbb{N} \tag{1}$$

$$y \notin \mathrm{FV}(v) \tag{2}$$

$$\text{every } f \in \overrightarrow{f_1} \text{ or } f \in \overrightarrow{f_2} \text{ is irreducible} \tag{3}$$

$$\Sigma \vdash \mathrm{KILL}(\alpha, n+1) \mapsto \Sigma', \text{ where } \Sigma = \Sigma_1, \alpha : U, \Sigma_2 \text{ and } \Sigma' = \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2 \tag{4}$$

Proof of correctness. Let $\mathcal{C} \stackrel{\text{def}}{=} \Gamma\ddagger\Gamma' \vdash v \approx t : A\ddagger A'$.

Assume Σ **sig**, $\Sigma;\mathcal{C}$ **wf**.

Well-formedness By the assumption and Lemma 4.46 (well-formedness of killing), we have Σ' **sig** and $\Sigma' \sqsupseteq \Sigma$.

Also by assumption, $\Sigma; \mathcal{C}$. By Lemma 2.155 (preservation of judgments under signature extensions), and Remark 4.7 (well-formed unification constraint is a judgment), $\Sigma'; \mathcal{C}$ **wf**.

Soundness Assume $\Sigma'' \sqsupseteq \Sigma'$ and $\Sigma'' \vDash \mathcal{C}$. Then we have $\Sigma'' \vDash \mathcal{C}$.

Completeness Assume that $\Theta \vDash \Sigma; \mathcal{C}$. This means $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma, A \equiv \Gamma', A'$ **ctx** and $\Theta; \Gamma \vdash v \equiv t : A$.

By Lemma 4.49 (pruning), taking $t_1 := v$, $t_2 := t$, and $\vec{f} := \overrightarrow{f_1}^n\ (y\ \vec{e})\ \overrightarrow{f_2}^m$, there exists u_0 and U' such that $\Theta; \cdot \vdash \alpha \equiv \lambda \vec{z}^{n+1}.u_0 : U'$, with $z_{n+1} \notin \mathrm{FV}(u_0)$.

By Lemma 4.47 (completeness of killing), there is Θ' with $\Theta'_\Sigma = \Theta$ and $\Theta' \vDash \Sigma'$. By Remark 2.134 (subsumption of restriction), $\Theta' \supseteq \Theta$. By Lemma 2.69 (signature weakening), $\Theta' \vdash \Gamma, A \equiv \Gamma', A'$ **ctx** and $\Theta'; \Gamma \vdash v \equiv t : A$; that is, $\Theta' \vDash \mathcal{C}$.

Therefore, $\Theta' \vDash \Sigma'; \mathcal{C}$.

□

4.5.10 Metavariable argument currying

The pattern condition for metavariable instantiation states that all the arguments of a metavariable must be variables. Abel and Pientka [3] observed that we can relax this condition when some of the arguments of the metavariable are record constructors. In this case, we can consider each of the fields of the record as a separate argument when evaluating the pattern condition.

In our formulation, this means that a metavariable argument of type $y : \Sigma UV$ can be expanded into two arguments $y_1 : U$ and $y_2 : V[y_1]$.

Rule-Schema 18 (Metavariable argument currying)**.**

$$\Sigma_1, \alpha : T, \Sigma_2; \square \rightsquigarrow \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T, \Sigma_2; \square$$

where

β is fresh in Σ

$$\Sigma_1; \cdot \vdash T \equiv \Pi \vec{A}^n \Pi(\Sigma UV)B\ \textbf{type}$$

$$B((+2)+1)[\langle 1, 0\rangle/0] \Downarrow B'$$

$$T_\beta = \Pi \vec{A}^n \Pi U \Pi V B'$$

$$t_\alpha = \lambda \vec{x}^n\ y.\beta\ \vec{x}^n\ (y\ .\pi_1)\ (y\ .\pi_2)$$

Proof of correctness. Let $\Sigma = \Sigma_1, \alpha : T, \Sigma_2$, $\Sigma' = \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T, \Sigma_2$, and $T_\alpha = \Pi \vec{A}^n \Pi(\Sigma UV)B$.

Well-formedness Assume Σ **sig**. We want to show that $\Sigma \sqsubseteq \Sigma'$.

- $\Sigma_1, \beta : T_\beta$ **sig**:
  It suffices to show that the premises of the following rule are fulfilled:

$$\frac{\Sigma_1\ \mathbf{sig} \qquad \beta \text{ is fresh in } \Sigma_1 \qquad \Sigma_1;\cdot \vdash T_\beta\ \mathbf{type}}{\Sigma_1, \beta : T_\beta\ \mathbf{sig}}\ \text{META-DECL}$$

By assumption, Σ_1 **sig**. By assumption β is fresh in Σ, and therefore also in Σ_1. It remains to show that $\Sigma_1;\cdot \vdash T_\beta$ **type**.

(i) By the rules premises, $\Sigma_1;\cdot \vdash T \equiv T_\alpha$ **type**. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma_1;\cdot \vdash T_\alpha$ **type** (that is, $\Sigma_1;\cdot \vdash \Pi\vec{A}^n\Pi(\Sigma UV)B$ **type**) By Lemma 2.52 (Π inversion), this gives $\Sigma_1; \vec{A}, \Sigma UV \vdash B$ **type**.

(ii) By Lemma 2.70 (piecewise well-formedness of typing judgments) and Lemma 2.53 (Σ inversion), we have $\Sigma; \vec{A} \vdash U : \mathrm{Set}$ and $\Sigma; \vec{A}, U \vdash V : \mathrm{Set}$.

(iii) By (ii), $\Sigma_1 \vdash \vec{A}, U, V\ \mathbf{ctx}$. By (i) and Lemma 2.62 (context weakening), $\Sigma_1; \vec{A}, U, V, (\Sigma UV)^{(+2)} \vdash B^{((+2)+1)} : \mathrm{Set}$.

(iv) Note that $\Sigma_1; \vec{A}, U, V \vdash 1 : U^{(+2)}$, $\Sigma_1; \vec{A}, U, V \vdash 1 : V^{(+1)}$, $V^{(+1)} = V^{((+2)+1)}[1/0]$, and $\Sigma UV^{(+2)} = \Sigma(U^{(+2)})(V^{((+2)+1)})$. By the PAIR rule, $\Sigma_1; \vec{A}, U, V \vdash \langle 1, 0\rangle : \Sigma UV^{(+2)}$.

By (iii), (iv) and Postulate 1 (typing of hereditary substitution), $\Sigma_1; \vec{A}, U, V \vdash B^{((+2)+1)}[\langle 1, 0\rangle/0] : \mathrm{Set}$; that is $\Sigma_1; \vec{A}, U, V \vdash B' : \mathrm{Set}$. By successive applications of the PI rule, we have $\Sigma_1;\cdot \vdash \Pi\vec{A}\Pi U\Pi V B' : \mathrm{Set}$; that is, $\Sigma_1;\cdot \vdash T_\beta$ **type**.

- $\Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T$ **sig**: It suffices to show that the premises of the META-INST rule are fulfilled:

$$\frac{\begin{array}{c}\Sigma_1, \beta : T_\beta\ \mathbf{sig} \\ \alpha \text{ fresh in } \Sigma_1, \beta : T_\beta \\ \Sigma_1, \beta : T_\beta;\cdot \vdash T\ \mathbf{type} \\ \Sigma_1, \beta : T_\beta;\cdot \vdash t_\alpha : T\end{array}}{\Sigma_1, \beta : T_\beta, \alpha := t : T\ \mathbf{sig}}\ \text{META-INST}$$

As shown above, $\Sigma_1, \beta : T_\beta$ **sig**. Because Σ **sig**, we have that α is fresh in Σ_1. Because β is fresh in Σ, $\alpha \neq \beta$. Therefore, α is fresh in $\Sigma_1, \beta : T_\beta$. Because $\Sigma_1;\cdot \vdash T$ **type** and $\Sigma_1 \subseteq \Sigma_1, \beta : T_\beta$, by Lemma 2.69 (signature weakening), we also have $\Sigma_1, \beta : T_\beta;\cdot \vdash T$ **type**.

It remains to show that $\Sigma_1, \beta : T_\beta;\cdot \vdash t_\alpha : T$.

(a) Let $(x_1, ..., x_n, y) = (n, n-1, ..., 1, 0)$. Note that $\mathrm{FV}(\Pi\vec{A}\Pi U \Pi V B) = \emptyset$, and therefore, $A_1^{(+(n+1))} = A_1$, $A_2^{(+n)} = A_2[x_1]$, ..., $A_n^{(+2)} = A_2[\vec{x}_{1,...,n-1}]$, $U^{(+1)} = U[\vec{x}_{1,...,n}]$ and $V^{(+1)+1}[(y\,.\pi_1)] = V[\vec{x}_{1,...,n}, (y\,.\pi_1)]$,

By the the VAR rule, $\Sigma_1, \beta : T_\beta, \vec{x : A}, y : \Sigma UV \vdash x_1 : A_1$, $\Sigma_1, \beta : T_\beta, \vec{x : A}, y : \Sigma UV \vdash x_2 : A_2[x_1]$, ..., and $\Sigma_1, \beta : T_\beta, \vec{x : A}, y : \Sigma UV \vdash x_n : A_n[x_{1,...,n-1}]$.

By the PROJ1 and the PROJ2 rule, $\Sigma_1, \beta : T_\beta, \vec{x : A}, y : \Sigma UV \vdash y\,.\pi_1 : U[\vec{x}_{1,...,n}]$ and $\Sigma_1, \beta : T_\beta, \vec{x : A}, y : \Sigma UV \vdash y\,.\pi_2 : V^{(+1)+1}[y\,.\pi_1]$.

By repeated application of the APP rule, we have $\Sigma_1, \beta : T_\beta; x : \vec{A}, y : \Sigma UV \vdash \beta\, \vec{x}\, (y\, .\pi_1)\, (y\, .\pi_2) : B'[\vec{x}, (y\, .\pi_1), (y\, .\pi_2)]$.
By Lemma 2.40 (correspondence between renaming and substitution) and Postulate 5 (hereditary substitution commutes), $B'[\vec{x}, (y\, .\pi_1), (y\, .\pi_2)] = B((+1)+1)[\langle 0\, .\pi_1, 0\, .\pi_2\rangle/0]$.
By the ETA-PAIR rule, $\Sigma_1, \beta : T_\beta; \vec{A}, \Sigma UV \vdash 0 \equiv \langle 0\, .\pi_1, 0\, .\pi_2\rangle : \Sigma UV$. By Postulate 4 (congruence of hereditary substitution), $\Sigma_1, \beta : T_\beta; \overrightarrow{x : A}, y : \Sigma UV \vdash B'[\vec{x}, (y\, .\pi_1), (y\, .\pi_2)] \equiv B((+1)+1)[0/0]$ **type**. By definition, $B((+1)+1)[0/0] = B$. Therefore, $\Sigma_1, \beta : T_\beta; \overrightarrow{x : A}, y : \Sigma UV \vdash \beta\, \vec{x}\, (y\, .\pi_1)\, (y\, .\pi_2) : B$.
By the ABS rule, $\Sigma_1, \beta : T_\beta; \cdot \vdash \lambda \vec{x}^n\, y.\beta\, \vec{x}\, (y\, .\pi_1)\, (y\, .\pi_2) : \Pi\overrightarrow{A\Pi}(\Sigma UV)B$.

(b) By assumption, we have $\Sigma_1; \cdot \vdash T \equiv \Pi\overrightarrow{A\Pi}(\Sigma UV)B$ **type**. By Lemma 2.69 (signature weakening), $\Sigma_1, \beta : T_\beta; \cdot \vdash T \equiv \Pi\overrightarrow{A\Pi}(\Sigma UV)B$ **type**

(c) By (a) and the CONV rule, $\Sigma_1, \beta : T_\beta; \cdot \vdash \lambda \vec{x}^n\, y.(\beta\, \vec{x}^n\, (y\, .\pi_1)\, (y\, .\pi_2)) : T$.

By Definition 2.151, $\Sigma_1, \alpha : T \sqsubseteq \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T$. By Corollary 2.156, $\Sigma_1, \alpha : T, \Sigma_2 \sqsubseteq \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T, \Sigma_2$.

Soundness Assume Σ'' **sig** such that $\Sigma'' \sqsupseteq \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T, \Sigma_2$. Vacuously, $\Sigma'' \mid\approx \square$.

Completeness Assume $\Theta \vDash \Sigma$. We want to find Θ' such that $\Theta'_\Sigma = \Theta$, and $\Theta' \vDash \Sigma'$.

(i) Because $\alpha : T \in \Sigma$, by Lemma 2.130 (alternative characterization of a compatible metasubstitution) and Remark 2.129 (alternative characterization of compatibility of a metasubstitution with a declaration), there are u and T' such that $\alpha := u : T' \in \Theta$ and $\Theta; \cdot \vdash T' \equiv T$ **type**.

(ii) By (i) and Remark 2.73 (signature piecewise well-formed), $\Theta; \cdot \vdash u : T'$. Because $\Theta \vDash \Sigma$, $\Theta; \cdot \vdash T \equiv \Pi\overrightarrow{A^n}\Pi(\Sigma UV)B$ **type**. By (i) and transitivity of type equality, $\Theta; \cdot \vdash T' \equiv \Pi\overrightarrow{A^n}\Pi(\Sigma UV)B$ **type**. By the CONV rule, $\Theta; \cdot \vdash u : \Pi\overrightarrow{A^n}\Pi(\Sigma UV)B$.

(iii) By (ii) and Lemma 2.62 (context weakening), $\Theta; \vec{A}, U, V \vdash u : \Pi\overrightarrow{A\Pi}(\Sigma UV)B$.
Let $\vec{x}, y_1, y_2 = n+1, ..., 1, 0$. Note that $(\Sigma UV)^{(+2)} = (\Sigma UV)[\vec{x}]$. Therefore, by the VAR and PAIR rules, $\Theta; \vec{A}, U, V \vdash \langle 1, 0\rangle : (\Sigma UV)[\vec{x}]$.
By Postulate 2 (typing of hereditary application), $(u \mathbin{@} \vec{x}\, \langle y_1, y_2\rangle)\Downarrow$ and $\Theta; x : \vec{A}, y_1 : U, y_2 : V \vdash u \mathbin{@} \vec{x}\, \langle y_1, y_2\rangle : B[\vec{x}, \langle y_1, y_2\rangle]$. By Lemma 2.40 (correspondence between renaming and substitution), $B[\vec{x}, \langle y_1, y_2\rangle] = B^{(+2)+1}[\langle y_1, y_2\rangle] = B'$.
Let $v = \lambda \vec{x}^n.\lambda y_1.\lambda y_2.(u \mathbin{@} \vec{x}\, \langle y_1, y_2\rangle)$. By the ABS rule, $\Theta; \cdot \vdash v : \Pi\vec{A}\Pi U\Pi V B'$, that is, $\Theta; \cdot \vdash v : T_\beta$.

Because $\alpha := u : T' \in \Theta$ and $\Theta\,\mathbf{wf}$, we have $\textsc{metas}(u) = \emptyset$. By construction, this implies $\textsc{metas}(v) = \emptyset$.

(iv) By Lemma 2.141 (existence of meta-free normal form), let T'_β be a term such that $\Theta \vdash T_\beta \mathrel{\hat{\searrow}} T'_\beta$. In particular, $\Theta; \cdot \vdash T_\beta \equiv T'_\beta : \mathrm{Set}$ and $\textsc{metas}(T'_\beta) = \emptyset$. By (iii) and the CONV rule, $\Theta; \cdot \vdash v : T'_\beta$.

Let $\Theta' = (\Theta, \beta := v : T'_\beta)$.

By (iv) and Definition 2.122 $\Theta'\,\mathbf{wf}$. Because $\Theta'_\Sigma = \Theta$, we have $\Theta \sqsubseteq \Theta'$. It remains to show that $\Theta' \vDash \Sigma'$.

Let D be a declaration, $D \in \Sigma'$. There are three possible cases:

- $D \in \Sigma_1$ or $D \in \Sigma_2$: Then $D \in \Sigma$. By Remark 2.131, Θ' is compatible with D.
- $D = \beta : T_\beta$: We have $\beta := v : T'_\beta \in \Theta'$. By (iv), $\Theta; \cdot \vdash T_\beta \equiv T'_\beta : \mathrm{Set}$. By Lemma 2.69, we also have $\Theta'; \cdot \vdash T_\beta \equiv T'_\beta : \mathrm{Set}$, which, by Remark 2.15, $\Theta'; \cdot \vdash T_\beta \equiv T'_\beta\ \mathbf{type}$. By Remark 2.129 (alternative characterization of compatibility of a metasubstitution with a declaration), Θ' is compatible with D.
- $D = \alpha := t_\alpha : T$: We have $\alpha := u : T' \in \Theta$.

  Note that $\beta := v : T'_\beta \in \Theta'$, where $v = \lambda \vec{x}^n.\lambda y_1.\lambda y_2.(u\ @\ \vec{x}\ \langle y_1, y_2\rangle)$ and $\Theta; \cdot \vdash \Pi\overrightarrow{A}\Pi U \Pi V B' \equiv T'_\beta : \mathrm{Set}$. Also, $t_\alpha = \lambda \vec{x}^n\ y.\beta\ \vec{x}^n\ (y\ .\pi_1)\ (y\ .\pi_2)$.

  By the DELTA-META rule and Postulate 9 (commuting of hereditary substitution and application), $\Theta'; \overrightarrow{x : A}, y : \Sigma U V \vdash \beta\ \vec{x}^n\ (y\ .\pi_1)\ (y\ .\pi_2) \equiv u\ @\ \vec{x}\ \langle y\ .\pi_1, y\ .\pi_2\rangle : B'[\vec{x}, (y\ .\pi_1), (y\ .\pi_2)]$. The premises of Postulate 9 are fulfilled by the fact that $\Theta; \cdot \vdash T' \equiv \Pi\vec{A}\Pi U V\ \mathbf{type}$ and the same reasoning as above.

  By the Postulate 4 (congruence of hereditary substitution) and the ETA-PAIR and CONV-EQ rules, we have $\Theta'; \overrightarrow{x : A}, y : \Sigma U V \vdash u\ @\ \vec{x}\ \langle y\ .\pi_1, y\ .\pi_2\rangle \equiv \beta\ \vec{x}^n\ (y\ .\pi_1)\ (y\ .\pi_2) : B$.

  By the ETA-PAIR rule and Postulate 6 (congruence of hereditary application), $\Theta'; \overrightarrow{x : A}, y : \Sigma U V \vdash u\ @\ \vec{x}\ y \equiv \beta\ \vec{x}^n\ (y\ .\pi_1)\ (y\ .\pi_2) : B$. By repeated application of the ABS-EQ rule, Lemma 4.34 (general η-equality for Π-types), and transitivity of equality, $\Theta'; \cdot \vdash u \equiv t_\alpha : T'$.

  By (i) we have $\Theta; \cdot \vdash T' \equiv T\ \mathbf{type}$. Because $\Theta \sqsubseteq \Theta'$, $\Theta'; \cdot \vdash T' \equiv T\ \mathbf{type}$.

  Therefore, Θ' is compatible with D.

By Lemma 2.130 (alternative characterization of a compatible metasubstitution), $\Theta' \vDash \Sigma'$.

□

4.5.11 Metavariable η-expansion

Rule schema 2 (metavariable instantiation) cannot be applied if the metavariable has projections among its eliminators. Abel and Pientka [3] show how these eliminators can be removed.

Example 4.50. Consider the following problem:

$$\begin{aligned}
&\mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \mathbb{b} : \mathbb{A}, \alpha : \mathbb{A} \to \mathbb{A} \to \mathbb{A} \times \mathbb{A};\\
&x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha\, x\, y\, .\pi_1 \approx x : \mathbb{A} \ddagger \mathbb{A},\\
&x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha\, x\, y\, .\pi_2 \approx y : \mathbb{A} \ddagger \mathbb{A}
\end{aligned}$$

Rule schema 2 (metavariable instantiation) cannot be applied to any of the constraints, because the metavariable α is applied to eliminators which are not variables ($.\pi_1$ and $.\pi_2$, respectively). However, if we instantiate α with $\alpha := \lambda x.\lambda y.\langle \alpha_1\, x\, y, \alpha_2\, x\, y\rangle : \mathbb{A} \to \mathbb{A} \to \mathbb{A} \times \mathbb{A}$ (where $\alpha_1 : \mathbb{A} \to \mathbb{A} \to \mathbb{A}$ and $\alpha_2 : \mathbb{A} \to \mathbb{A} \to \mathbb{A}$ are fresh metavariables) and apply Rule schema 8 (term conversion) to normalize the constraints, then the problem becomes:

$$\begin{aligned}
&\mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \mathbb{b} : \mathbb{A},\\
&\alpha_1 : \mathbb{A} \to \mathbb{A} \to \mathbb{A},\\
&\alpha_2 : \mathbb{A} \to \mathbb{A} \to \mathbb{A},\\
&\alpha := \lambda x.\lambda y.\langle \alpha_1\, x\, y, \alpha_2\, x\, y\rangle : \mathbb{A} \to \mathbb{A} \to \mathbb{A} \times \mathbb{A};\\
&x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha_1\, x\, y \approx x : \mathbb{A} \ddagger \mathbb{A},\\
&x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha_2\, x\, y \approx y : \mathbb{A} \ddagger \mathbb{A}
\end{aligned}$$

Then Rule schema 2 can be applied both constraints.

◀

Example 4.51. Consider the following unification problem:

$$\begin{aligned}
&\mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A},\\
&\mathbb{B} : \mathbb{A} \to \mathrm{Set}, \mathbb{b} : \mathbb{B}\, \mathbb{a},\\
&\alpha : \mathbb{A} \to \mathbb{B}\, \mathbb{a} \to \Sigma \mathbb{A}(\mathbb{B}\, \mathbb{a});\\
&\cdot \vdash \alpha\, \mathbb{a}\, \mathbb{b}\, .\pi_1 \approx \mathbb{a} : \mathbb{A} \ddagger \mathbb{A} \wedge\\
&x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{B}\, \mathbb{a} \ddagger \mathbb{B}\, \mathbb{a} \vdash \alpha\, x\, y \approx \langle x, y\rangle : \Sigma \mathbb{A}(\mathbb{B}\, \mathbb{a}) \ddagger \Sigma \mathbb{A}(\mathbb{B}\, (\alpha\, \mathbb{a}\, \mathbb{b}\, .\pi_1))
\end{aligned}$$

In order to instantiate α by applying Rule schema 2 (metavariable instantiation), we need to have $\Sigma; \Gamma \ddagger \Gamma' \vdash \Sigma \mathbb{A}(\mathbb{B}\, \mathbb{a}) \equiv \Sigma \mathbb{A}(\mathbb{B}\, (\alpha\, \mathbb{a}\, \mathbb{b}\, .\pi_1)) : \mathrm{Set} \ddagger \mathrm{Set}$.

By η-expanding α as in the previous example and then applying Rule schema 12 (pairs), we obtain the following problem:

$$
\begin{array}{l}
\mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \\
\mathbb{B} : \mathbb{A} \to \mathrm{Set}, \mathbb{b} : \mathbb{B}\,\mathbb{a}, \\
\alpha_1 : \mathbb{A} \to \mathbb{B}\,\mathbb{a} \to \mathbb{A}, \\
\alpha_2 : \mathbb{A} \to \mathbb{B}\,\mathbb{a} \to \mathbb{B}\,\mathbb{a}, \\
\alpha := \lambda x.\lambda y.\langle \alpha_1\ x\ y, \alpha_2\ x\ y\rangle : \mathbb{A} \to \mathbb{B}\,\mathbb{a} \to \Sigma\mathbb{A}(\mathbb{B}\,\mathbb{a}); \\
\cdot \vdash \alpha_1\ \mathbb{a}\ \mathbb{b} \approx \mathbb{a} : \mathbb{A} \wedge \\
x : \mathbb{A}\ddagger\mathbb{A}, y : \mathbb{B}\,\mathbb{a}\ddagger\mathbb{B}\,\mathbb{a} \vdash \alpha_1\ x\ y \approx x : \mathbb{A}\ddagger\mathbb{A} \wedge \\
x : \mathbb{A}\ddagger\mathbb{A}, y : \mathbb{B}\,\mathbb{a}\ddagger\mathbb{B}\,\mathbb{a} \vdash \alpha_2\ x\ y \approx y : \mathbb{B}\,\mathbb{a}\ddagger\mathbb{B}\,(\alpha_1\ \mathbb{a}\ \mathbb{b})
\end{array}
$$

Then, by applying Rule schema 2 (metavariable instantiation), we can instantiate α_1 to $\lambda x.\lambda y.x$. Then the rest of the constraints can be solved using Rule schema 8 (term conversion), Rule schema 9 (type and context conversion), Rule schema 1 (syntactic equality) and Rule schema 2 (metavariable instantiation).

◀

Examples 4.50 and 4.51 may be generalized as the following rule:

Rule-Schema 19 (Metavariable η-expansion)**.**

$$
\begin{array}{l}
\Sigma_1, \alpha : U, \Sigma_2; \square \rightsquigarrow \\
\Sigma_1, \alpha_1 : \Pi\overrightarrow{T}.A, \alpha_2 : \Pi\overrightarrow{T}.B[\alpha_1\ \vec{x}], \alpha := \lambda\vec{x}^n.\langle \alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : U, \Sigma_2; \square \\
\quad \textbf{where} \\
\Sigma = \Sigma_1, \alpha : U, \Sigma_2 \\
\Sigma_1; \cdot \vdash U \equiv \Pi\overrightarrow{T}^n.\Sigma AB\ \textbf{type} \\
\alpha_1 \text{ and } \alpha_2 \text{ fresh in } \Sigma,\ \alpha_1 \neq \alpha_2
\end{array}
$$

Remark. Rule schema 19 may be applied to any metavariable, regardless of whether it occurs in a constraint or not.

Proof of correctness. Let

$$
\begin{array}{l}
\Sigma^\circ \stackrel{\text{def}}{=} \Sigma_1, \alpha_1 : \Pi\overrightarrow{T}.A, \\
\Sigma^\mathrm{a} \stackrel{\text{def}}{=} \Sigma^\circ, \alpha_2 : \Pi\overrightarrow{T}.B[\alpha_1\ \vec{x}], \\
\Sigma^\mathrm{b} \stackrel{\text{def}}{=} \Sigma^\mathrm{a}, \alpha := \lambda\vec{x}^n.\langle \alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : U \text{ and} \\
\Sigma' \stackrel{\text{def}}{=} \Sigma^\mathrm{b}, \Sigma_2.
\end{array}
$$

Well-formedness It suffices to show Σ **sig**, and $\Sigma \sqsubseteq \Sigma'$.

- Σ° **sig**:
  We use the META-DECL rule:

$$
\frac{\Sigma_1\ \textbf{sig} \qquad \alpha_1 \text{ is fresh in } \Sigma_1 \qquad \Sigma_1; \cdot \vdash \Pi\overrightarrow{T}.A\ \textbf{type}}{\Sigma^\circ\ \textbf{sig}}\ \text{META-DECL}
$$

By induction on the derivation of Σ **sig**, we have Σ_1 **sig**.

By the rule preconditions, α_1 is fresh in Σ, and thus in Σ_1.

Also by the rule preconditions, $\Sigma_1; \cdot \vdash U \equiv \Pi\overrightarrow{T}^n.\Sigma AB$ **type**. By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma_1; \cdot \vdash \Pi\overrightarrow{T}^n.\Sigma AB$ **type**.

By Lemma 2.52 (Π inversion), we have $\Sigma_1; \overrightarrow{T} \vdash \Sigma AB$ **type**. By Lemma 2.53 (Σ inversion), this gives $\Sigma_1; \vec{T} \vdash A$ **type**. By Remark 2.15 (there is only set), $\Sigma_1; \vec{T} \vdash A : \mathrm{Set}$. By repeatedly applying the PI rule, we have $\Sigma_1; \cdot \vdash \Pi\overrightarrow{T}.A : \mathrm{Set}$, which means $\Sigma_1; \cdot \vdash \Pi\overrightarrow{T}.A$ **type**.

- Σ^{a} **sig**:

  We use the META-DECL rule:

$$\frac{\Sigma^\circ \textbf{ sig} \qquad \alpha_2 \text{ is fresh in } \Sigma^\circ \qquad \Sigma^\circ; \cdot \vdash \Pi\overrightarrow{x : T}.B[\alpha_1\ \vec{x}] \textbf{ type}}{\Sigma^{\mathrm{a}} \textbf{ sig}}\ \textsc{meta-decl}$$

  By the rule preconditions, α_2 is fresh in Σ, and therefore also in Σ°.

  By the META$_1$ and HEAD rules, $\Sigma^\circ; \overrightarrow{x : T} \vdash \alpha_1 : \Pi\overrightarrow{T}.A$. By repeatedly applying the APP rule, $\Sigma^\circ; \overrightarrow{x : T} \vdash \alpha_1\ \vec{x} : A$.

  As shown in the proof of Σ° **sig**, $\Sigma_1; \overrightarrow{T} \vdash \Sigma AB : \mathrm{Set}$. By Lemma 2.53 ($\Sigma$ inversion), this gives $\Sigma_1; \overrightarrow{T}, A \vdash B : \mathrm{Set}$.

  Because $\Sigma^\circ; \overrightarrow{x : T} \vdash \alpha_1\ \vec{x} : A$ and $\Sigma_1; \vec{T}, A \vdash B : \mathrm{Set}$, by Postulate 1 (typing of hereditary substitution), $\Sigma^\circ; \overrightarrow{x : T} \vdash B[\alpha_1\ \vec{x}] : \mathrm{Set}$.

  By repeatedly applying the PI rule, we have $\Sigma^\circ; \cdot \vdash \Pi\overrightarrow{T}.B[\alpha_1\ \vec{x}] : \mathrm{Set}$ which means $\Sigma^\circ; \cdot \vdash \Pi\overrightarrow{T}.B[\alpha_1\ \vec{x}]$ **type**.

- Σ^{b} **sig**:

$$\frac{\Sigma^{\mathrm{a}} \textbf{ sig} \qquad \Sigma^{\mathrm{a}}; \cdot \vdash U \textbf{ type} \qquad \alpha \text{ fresh in } \Sigma^{\mathrm{a}} \qquad \Sigma^{\mathrm{a}}; \cdot \vdash \lambda\vec{x}.\langle \alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : U}{\Sigma^{\mathrm{b}} \textbf{ sig}}\ \textsc{meta-inst}$$

  By Definition 2.4 (well-formed signature), Σ **sig** implies $\Sigma_1; \cdot \vdash U$ **type**. By Lemma 2.69 (signature weakening), $\Sigma^{\mathrm{a}}; \cdot \vdash U$ **type**.

  Finally, by the typing rules (in particular, ABS, PAIR, META$_1$ and APP) $\Sigma^{\mathrm{a}}; \cdot \vdash \lambda\vec{x}^n.\langle \alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : \Pi\overrightarrow{T}^n.\Sigma AB$. From the premises, $\Sigma_1; \cdot \vdash U \equiv \Pi\overrightarrow{T}^n.\Sigma AB : \mathrm{Set}$. By Lemma 2.69 (signature weakening), $\Sigma^{\mathrm{a}}; \cdot \vdash U \equiv \Pi\overrightarrow{T}^n.\Sigma AB : \mathrm{Set}$. By the CONV rule, $\Sigma^{\mathrm{a}}; \cdot \vdash \lambda\vec{x}^n.\langle \alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : U$.

Thus, Σ^{b} **sig**. By Definition 2.151, $\Sigma_1, \alpha : U \sqsubseteq \Sigma^{\mathrm{b}}$. By Corollary 2.156 (horizontal composition of extensions), $\Sigma_1, \alpha : U, \Sigma_2 \sqsubseteq \Sigma^{\mathrm{b}}, \Sigma_2$; that is, $\Sigma \sqsubseteq \Sigma'$.

Soundness Assume Σ'' **sig**, $\Sigma'' \sqsupseteq \Sigma'$. Vacuously, $\Sigma'' \not\approx \square$.

Completeness Assume $\Theta \vDash \Sigma; \square$; that is, $\Theta \vDash \Sigma$.

(i) Because $\Theta \vDash \Sigma$, by Lemma 2.130 (alternative characterization of a compatible metasubstitution) and Remark 2.129 (alternative characterization of compatibility of a metasubstitution with a declaration), we have $\alpha := u : U' \in \Theta$, $\Theta; \cdot \vdash U' \equiv U$ **type** and $\Theta; \cdot \vdash u : U'$.

(ii) By the rule preconditions, $\Sigma_1; \cdot \vdash U \equiv \Pi\overrightarrow{T}\Sigma AB$ **type**. By Lemma 2.69 (signature weakening), $\Sigma; \cdot \vdash U \equiv \Pi\overrightarrow{T}\Sigma AB$ **type**. Because $\Theta \vDash \Sigma$, we have $\Theta; \cdot \vdash U \equiv \Pi\overrightarrow{T}\Sigma AB$ **type**. By (i), transitivity of equality and CONV, $\Theta; \cdot \vdash u : \Pi\overrightarrow{T}\Sigma AB$.

(iii) By (ii) and Lemma 2.62 (context weakening), $\Theta; \overrightarrow{T} \vdash u : \Pi\overrightarrow{T}\Sigma AB$. By Postulate 2 (typing of hereditary application) $(u \mathbin{@} \vec{x})\Downarrow$ and $\Theta; \overrightarrow{x : T} \vdash u \mathbin{@} \vec{x} : \Sigma AB$.

By Postulate 3 (typing of hereditary projection), we have $(u \mathbin{@} \vec{x}\,.\pi_1)\Downarrow$ and $\Theta; \overrightarrow{x : T} \vdash u \mathbin{@} \vec{x}\,.\pi_1 : A$. By the same token, we have $(u \mathbin{@} \vec{x}\,.\pi_2)\Downarrow$, $B[u \mathbin{@} \vec{x}\,.\pi_1]\Downarrow$ and $\Theta; \overrightarrow{x : T} \vdash u \mathbin{@} \vec{x}\,.\pi_2 : B[u \mathbin{@} \vec{x}\,.\pi_1]$.

(iv) Let $u_1 = \lambda\vec{x}^n.(u \mathbin{@} \vec{x}\,.\pi_1)$ and $u_2 = \lambda\vec{x}^n.(u \mathbin{@} \vec{x}\,.\pi_2)$. By (iii) and the ABS rule, we have $\Theta; \cdot \vdash u_1 : \Pi\overrightarrow{x : T}^n.A$ and $\Theta; \cdot \vdash u_2 : \Pi\overrightarrow{x : T}^n.B[u \mathbin{@} \vec{x}\,.\pi_1]$.

(v) By Lemma 2.141, there are U_1 and U_2 such that $\Theta; \cdot \vdash \Pi\overrightarrow{x : T}A \equiv U_1 : \mathrm{Set}$ and $\Theta; \cdot \vdash \Pi\overrightarrow{x : T}B[u \mathbin{@} \vec{x}\,.\pi_1] \equiv U_2 : \mathrm{Set}$, with $\mathrm{METAS}(U_1) = \emptyset$ and $\mathrm{METAS}(U_2) = \emptyset$.

By Remark 2.15 (there is only set) and the CONV rule, $\Theta; \cdot \vdash u_1 : U_1$ and $\Theta; \cdot \vdash u_2 : U_2$.

Let $\Theta' = \Theta, \alpha_1 := u_1 : U_1, \alpha_2 := u_2 : U_2$. Note that $\mathrm{DECLS}(\Theta) = \mathrm{DECLS}(\Sigma)$. By the rule premises, α_1 is fresh in Θ and α_2 is fresh in $(\Theta, \alpha_1 := u_1 : U_1)$. By (i), (iv) and (v), $\mathrm{METAS}(u_1) = \mathrm{METAS}(u_2) = \mathrm{METAS}(U_1) = \mathrm{METAS}(U_2) = \emptyset$, $\Theta; \cdot \vdash u_1 : U_1$ and $\Theta; \cdot \vdash u_2 : U_2$. By Lemma 2.69 (signature weakening), $\Theta, \alpha_1 := u_1 : U_1; \cdot \vdash u_2 : U_2$. By Definition 2.122 (well-formed metasubstitution), Θ' **wf**. Because $\Theta'_\Sigma = \Theta$, we have $\Theta \subseteq \Theta'$.

Let $D \in \Sigma'$. There are four possible cases:

1. $D \in \Sigma_1$ or $D \in \Sigma_2$: Then $D \in \Sigma$. By Remark 2.131, Θ' is compatible with D.
2. $D = \alpha_1 : \Pi\overrightarrow{x : T}^n.A$: We have $\alpha_1 := u_1 : U_1 \in \Theta'$. By (v), $\Theta; \cdot \vdash U_1 \equiv \Pi\overrightarrow{x : T}^n.A$ **type**. By Lemma 2.69, $\Theta'; \cdot \vdash U_1 \equiv \Pi\overrightarrow{x : T}^n.A$ **type** Therefore, Θ' is compatible with D.
3. $D = \alpha_2 : \Pi\overrightarrow{x : T}^n.B[\alpha_1\,\vec{x}]$: We have $\alpha_2 := u_2 : U_2 \in \Theta'$. It suffices to show $\Theta'; \cdot \vdash \Pi\overrightarrow{x : T}^n.B[\alpha_1\,\vec{x}] \equiv U_2$ **type**.

   Because $\Theta \vDash \Sigma$, we have $\Theta; \cdot \vdash u \equiv \lambda\vec{x}.\langle\alpha_1\,\vec{x}, \alpha_2\,\vec{x}\rangle : U$. By Definition 2.32 (hereditary elimination) and Postulate 7 (congruence of hereditary projection), we have $\Theta; \cdot \vdash u \mathbin{@} \vec{x}\,.\pi_1 \equiv \alpha_1\,\vec{x} : A[\vec{x}]$. Because $\Theta \subseteq \Theta'$, by Remark 2.137 (metasubstitution weakening), $\Theta'; \cdot \vdash u \mathbin{@} \vec{x}\,.\pi_1 \equiv \alpha_1\,\vec{x} : A[\vec{x}]$.

By Postulate 4 and reflexivity, $\Theta'; \overrightarrow{x : T}^n \vdash B[\alpha_1\ \vec{x}] \equiv B[u\ @\ \vec{x}\ .\pi_1] : U_1$. By the PI-EQ rule, $\Theta'; \cdot \vdash \Pi\overrightarrow{x : T}^n.B[\alpha_1\ \vec{x}] \equiv \Pi\overrightarrow{x : T}^n.B[u\ @\ \vec{x}\ .\pi_1] : \mathrm{Set}$. By (v), Lemma 2.69, symmetry and transitivity, $\Theta'; \cdot \vdash U_2 \equiv \Pi\overrightarrow{x : T}^n.B[\alpha_1\ \vec{x}] : \mathrm{Set}$. By remark 2.15, $\Theta'; \cdot \vdash U_2 \equiv \Pi\overrightarrow{x : T}^n.B[\alpha_1\ \vec{x}]$ **type**.

By Remark 2.129 (alternative characterization of compatibility of a metasubstitution with a declaration), Θ' is compatible with D.

4. $D = \alpha := t_\alpha : U$, where $t_\alpha = \lambda\vec{x}^n.\langle\alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle$. We need to show that $\Theta'; \cdot \vdash \alpha \equiv t_\alpha : U$.

   Note that $\Sigma; \cdot \vdash U \equiv \Pi\overrightarrow{T}^n.\Sigma AB : \mathrm{Set}$. By $\Theta \vDash \Sigma$, $\Theta \subseteq \Theta'$ and transitivity of equality we have $\Theta'; \cdot \vdash U' \equiv \Pi\overrightarrow{T}^n.\Sigma AB$ **type**. Also, $\alpha := u : U' \in \Theta'$. Therefore, it suffices to show that $\Theta'; \cdot \vdash u \equiv t_\alpha : U'$.

   By the definition of Θ', we have $\Theta'; \cdot \vdash \alpha_1 \equiv u_1 : \Pi\overrightarrow{T}^n.A$, that is $\Theta'; \cdot \vdash \alpha_1 \equiv \lambda\vec{x}^n.u\ @\ \vec{x}\ .\pi_1 : \Pi\overrightarrow{T}^n.A$. Similarly, $\Theta'; \cdot \vdash \alpha_2 \equiv u_2 : \Pi\overrightarrow{x : T}^n.B[u_1\ @\ \vec{x}\ .\pi_1]$ and thus $\Theta'; \cdot \vdash \alpha_2 \equiv \lambda\vec{x}^n.u\ @\ \vec{x}\ .\pi_2 : \Pi\overrightarrow{x : T}^n.B[u_1\ @\ \vec{x}\ .\pi_1]$. By Postulate 6 (congruence of hereditary application), Lemma 2.75 (uniqueness of typing for neutrals) the PAIR-EQ rule, the ETA-PAIR rule $\Theta'; \overrightarrow{x : T} \vdash \langle u\ @\ \vec{x}\ .\pi_1, u\ @\ \vec{x}\ .\pi_2\rangle \equiv \langle\alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : \Sigma AB$.

   By Lemma 4.35 (general η-equality for pairs), $\Theta'; \overrightarrow{x : T} \vdash u\ @\ \vec{x} \equiv \langle\alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : \Sigma AB$.

   By the ABS-EQ rule and Lemma 4.34 (general η-equality for Π-types), $\Theta'; \cdot \vdash u \equiv \lambda\vec{x}^n.\langle\alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : \Pi\overrightarrow{T}\Sigma AB$. By the CONV-EQ rule, $\Theta'; \cdot \vdash u \equiv \lambda\vec{x}^n.\langle\alpha_1\ \vec{x}, \alpha_2\ \vec{x}\rangle : U'$.

Therefore, by Lemma 2.130 (alternative characterization of a compatible metasubstitution), $\Theta' \vDash \Sigma'$.

□

4.5.12 Context variable currying

In the same way that one can remove Σ-types from metavariable arguments, Abel and Pientka [3] show how one can remove Σ-types from constraint contexts. This will help us instantiate metavariables whose arguments contain projections.

Example 4.52. Consider the following problem:

$$\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set}, \alpha : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B};$$
$$x : (\mathbb{A} \times (\mathbb{A} \to \mathbb{B}))‡(\mathbb{A} \times (\mathbb{A} \to \mathbb{B})) \vdash \alpha\ (x\ .\pi_1)\ (x\ .\pi_2) \approx (x\ .\pi_2)\ (x\ .\pi_1) : \mathbb{B}‡\mathbb{B}$$

Rule schema 2 (metavariable instantiation) may not be applied, as the arguments of α are not variables. However, because x is a pair, we could consider each of the components as a distinct variable, and reformulate the constraint as follows:

$$\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set}, \alpha : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B};$$
$$x_1 : \mathbb{A}‡\mathbb{A}, x_2 : (\mathbb{A} \to \mathbb{B})‡(\mathbb{A} \to \mathbb{B}) \vdash \alpha\ x_1\ x_2 \approx x_2\ x_1 : \mathbb{B}‡\mathbb{B}$$

By applying rule Rule schema 2, we have

$$\mathbb{A} : \mathrm{Set}, \mathbb{B} : \mathrm{Set}, \alpha := \lambda x.\lambda y.(y\ x) : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B}; \square$$

◀

The technique in Example 4.52 can be generalized as the following rule:

Rule-Schema 20 (Context variable currying)**.**

$$\begin{array}{l}\Sigma; \Gamma_1 \ddagger \Gamma_2, x : \Sigma U_1 V_1 \ddagger \Sigma U_2 V_2, \Delta_1 \ddagger \Delta_2 \vdash t_1 \approx t_2 : A_1 \ddagger A_2 \rightsquigarrow \\ \Sigma; \Gamma_1 \ddagger \Gamma_2, x_1 : U_1 \ddagger U_2, x_2 : V_1 \ddagger V_2, \Delta_1' \ddagger \Delta_2' \vdash t_1' \approx t_2' : A_1' \ddagger A_2' \quad \textbf{where} \\ (\Delta_1 \vdash t_1 : A_1)^{((+2)+1)}[\langle 1, 0\rangle/0] \Downarrow (\Delta_1' \vdash t_1' : A_1') \\ (\Delta_2 \vdash t_2 : A_2)^{((+2)+1)}[\langle 1, 0\rangle/0] \Downarrow (\Delta_2' \vdash t_2' : A_2')\end{array}$$

Informally, we have $\lceil \Delta_1' = \Delta_1[\langle x_1, x_2\rangle/x] \rfloor$, $\lceil t_1' = t_1[\langle x_1, x_2\rangle/x] \rfloor$, etc.

Before showing the correctness of Rule schema 20, we introduce two new lemmas:

Lemma 4.53 (Free variables in substitution by pair)**.** *Let t be a term such that, for some r, $t[\langle x, y\rangle/z] \Downarrow r$. Then:*

- *If $z \notin \mathrm{FV}(t)$, then $\mathrm{FV}(r) = \mathrm{FV}_z(t)$.*
- *If $z \in \mathrm{FV}(t)$, then $\mathrm{FV}_z(t) \subseteq \mathrm{FV}(r) \subseteq \mathrm{FV}_z(t) \cup \{x, y\}$.*

Proof. By induction on the derivation for $t[\langle x, y\rangle/z] \Downarrow r$, and Remark 2.28 (renaming and free variables). □

Lemma 4.54 (Free variables in substitution by irreducible)**.** *Let t be a term such that, for some r, $t[x\ e/z] \Downarrow r$, where $e = .\pi_1$ or $e = .\pi_2$. Then:*

- *If $z \notin \mathrm{FV}(t)$, then $\mathrm{FV}(r) = \mathrm{FV}_z(t)$.*
- *If $z \in \mathrm{FV}(t)$, then $\mathrm{FV}_z(t) \subseteq \mathrm{FV}(r) \subseteq \mathrm{FV}_z(t) \cup \{x\}$.*

Proof. By induction on the derivation for $t[x\ e/z] \Downarrow r$, and Remark 2.28 (renaming and free variables). □

Proof of correctness for Rule schema 20. Let $\mathcal{C} \stackrel{\text{def}}{=} \Gamma_1 \ddagger \Gamma_2, x : \Sigma U_1 V_1 \ddagger \Sigma U_2 V_2$, $\Delta_1 \ddagger \Delta_2 \vdash t_1 \approx t_2 : A_1 \ddagger A_2$ and $\mathcal{D} \stackrel{\text{def}}{=} \Gamma_1 \ddagger \Gamma_2, x_1 : U_1 \ddagger U_2, x_2 : V_1 \ddagger V_2, \Delta_1' \ddagger \Delta_2' \vdash t_1' \approx t_2' : A_1' \ddagger A_2'$.

Well-formedness Assume that Σ **sig**, and $\Sigma; \mathcal{C}$ is well-formed. By assumption Σ **sig**. By Definition 2.151, $\Sigma \sqsubseteq \Sigma$.

(i) By the assumption, $\Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 : A_1$. Because $\Sigma; \mathcal{C}$, by Lemma 2.70 (piecewise well-formedness of typing judgments), we have $\Sigma; \Gamma_1$ **ctx**.

Because $\Sigma; \Gamma_1 \vdash \Sigma U_1 V_1$ **type**, by Lemma 2.53 (Σ inversion) we also have that $\Sigma; \Gamma_1 \vdash U_1$ **type** and $\Sigma; \Gamma_1, U_1 \vdash V_1$ **type**. Therefore, $\Sigma \vdash \Gamma_1, U_1, V_1$ **ctx**.

(ii) By Lemma 2.62 (context weakening), $\Sigma; \Gamma_1, U_1, V_1, (\Sigma U_1 V_1, \Delta_1 \vdash t_1 : A_1)^{(+2)}$, that is, $\Sigma; \Gamma_1, U_1, V_1, (\Sigma U_1 V_1)^{(+2)}, (\Delta_1 \vdash t_1 : A_1)^{(+2)+1}$.

(iii) By the VAR, HEAD and PAIR rules, $\Sigma; \Gamma_1, U_1, V_1 \vdash \langle 1, 0\rangle : (\Sigma U_1 V_1)^{(+2)}$. By (ii) and Postulate 1 (typing of hereditary substitution), $\Sigma; \Gamma_1, U_1, V_1, (\Delta_1 \vdash t_1 : A_1)^{(+2)+1}[\langle 1, 0\rangle/0]$, i.e. $\Sigma; \Gamma_1, U_1, V_1, \Delta_1' \vdash t_1' : A_1'$.

(iv) Analogously, because $\Gamma_2, \Sigma U_2 V_2, \Delta_2 \vdash t_2 : A_2$, we have $\Sigma; \Gamma_2, U_2, V_2, \Delta_2' \vdash t_2' : A_2'$.

By (iii) and (iv), $\Sigma; \mathcal{D}\,\mathbf{wf}$.

Soundness Let $\Sigma'' \sqsupseteq \Sigma$ such that $\Sigma'' \not\approx \mathcal{D}$. That is, there exists a term v' such that $\Sigma''; \Gamma_1, U_1, V_1, \Delta_1' \vdash t_1' \equiv v' : A_1'$, $\Sigma''; \Gamma_2, U_2, V_2, \Delta_2' \vdash t_2' \equiv v' : A_2'$, $\mathrm{FV}(v') \subseteq \mathrm{FV}(t_1')$ and $\mathrm{FV}(v') \subseteq \mathrm{FV}(t_2')$.

Trivially $\Sigma'' \sqsupseteq \Sigma$. It suffices to show $\Sigma'' \not\approx \mathcal{C}$; that is, finding a v such that $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv v : A_1$, $\Sigma''; \Gamma_2, \Sigma U_2 V_2, \Delta_2 \vdash t_2 \equiv v : A_2$, $\mathrm{FV}(v) \subseteq \mathrm{FV}(t_1)$ and $\mathrm{FV}(v) \subseteq \mathrm{FV}(t_2)$.

(i) By Lemma 2.70 (piecewise well-formedness of typing judgments) and Remark 2.13 (context inversion), $\Sigma''; \Gamma_1 \vdash U_1\,\mathbf{type}$ and $\Sigma''; \Gamma_1, U_1 \vdash V_1\,\mathbf{type}$. By the SIGMA rule, $\Sigma''; \Gamma_1 \vdash \Sigma U_1 V_1\,\mathbf{type}$.

(ii) By the ETA-PAIR rule, $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0 \equiv \langle 0\,.\pi_1, 0\,.\pi_2\rangle : (\Sigma U_1 V_1)^{(+1)}$. By Lemma 2.62 (context weakening), $\Sigma; \Gamma, \Sigma U_1 V_1, (\Sigma U_1 V_1)^{(+1)}, (\Delta_1 \vdash t_1 : A_1)^{(+1)+1}$.

By Lemma 2.40 (correspondence between renaming and substitution), $(\Delta_1 \vdash t_1 : A_1)^{(+1)+1}[0/0] \Downarrow (\Delta_1 \vdash t_1 : A_1)$.

By Lemma 2.62 and Postulate 4 (congruence of hereditary substitution), there exist Δ_1^{a}, t_1^{a}, A_1^{a} such that $(\Delta_1^{\mathrm{a}} \vdash t_1^{\mathrm{a}} : A_1^{\mathrm{a}}) = (\Delta_1 \vdash t_1 : A_1)^{(+1)+1}[\langle 0\,.\pi_1, 0\,.\pi_2\rangle]$.

By Postulate 4 (congruence of hereditary substitution), $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash \Delta_1 \equiv \Delta_1^{\mathrm{a}}\,\mathbf{ctx}$, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash A_1 \equiv A_1^{\mathrm{a}}\,\mathbf{type}$, and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv t_1^{\mathrm{a}} : A_1$.

(iii) By Lemma 2.70 (piecewise well-formedness of typing judgments), $\Sigma''; \Gamma_1 \vdash \Sigma U_1 V_1\,\mathbf{type}$. By Lemma 2.53 ($\Sigma$ inversion) and Lemma 2.62 (context weakening), $\Sigma''; \Gamma_1, \Sigma U_1 V_1, U_1^{(+1)} \vdash V_1^{(+1)+1}\,\mathbf{type}$. By the VAR, HEAD and PROJ1 rules, $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0\,.\pi_1 : U_1^{(+1)}$. By Postulate 1 (typing of hereditary substitution), $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash V_1^{(+1)+1}[0\,.\pi_1]\,\mathbf{type}$.

By Lemma 2.62 (context weakening), $\Sigma''; \Gamma_1, \Sigma U_1 V_1, V_1^{(+1)+1}[0\,.\pi_1]$, $(\Sigma U_1 V_1, \Delta_1 \vdash t_1 : A_1)^{(+2)}$, that is, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, V_1^{(+1)+1}[0\,.\pi_1]$, $(\Sigma U_1 V_1)^{(+2)}, (\Delta_1 \vdash t_1 : A_1)^{(+2)+1}$.

By the VAR, HEAD, and PROJ2 typing rules, $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0\,.\pi_2 : V_1^{(+1)+1}[0\,.\pi_1]$. By the VAR, HEAD, PAIR and PROJ1 rules, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, V_1^{(+1)+1}[0\,.\pi_1] \vdash \langle 1\,.\pi_1, 0\rangle : \Sigma U_1 V_1^{(+2)}$. By Definition 2.31 (hereditary substitution), $\langle 1\,.\pi_1, 0\rangle[0\,.\pi_2] \Downarrow \langle 0\,.\pi_1, 0\,.\pi_2\rangle$ and $\Sigma U_1 V_1^{(+2)}[0\,.\pi_2] \Downarrow (\Sigma U_1 V_1)^{(+1)}$. By Postulate 1 (typing of hereditary substitution), $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash \langle 0\,.\pi_1, 0\,.\pi_2\rangle : (\Sigma U_1 V_1)^{(+1)}$.

Let $(\Delta_1^{\mathrm{b}} \vdash t_1^{\mathrm{b}} : A_1^{\mathrm{b}}) = (\Delta_1 \vdash t_1 : A_1)^{(+2)+1}[\langle 1\,.\pi_1, 0\rangle][0\,.\pi_2]$.

Note that $1 \notin \mathrm{FV}((\Delta_1 \vdash t_1 : A_1)^{(+2)+1})$. By Remark 2.49 (strengthening by substitution), this means $(\Delta_1 \vdash t_1 : A_1)^{((+2)+1)}[1\,.\pi_2/1] = (\Delta_1 \vdash t_1 : A_1)^{((+1)+1)}$.

As shown earlier, $(\Delta_1^{\mathrm{a}} \vdash t_1^{\mathrm{a}} : A_1^{\mathrm{a}}) = (\Delta_1 \vdash t_1 : A_1)^{((+1)+1)}[\langle 0\,.\pi_1, 0\,.\pi_2\rangle]$.

By Postulate 5 (hereditary substitution commutes), we have $\Sigma'' \vdash \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}} \equiv \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{a}}\ \mathbf{ctx}$, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}} \vdash A_1^{\mathrm{b}} \equiv A_1^{\mathrm{a}}\ \mathbf{type}$, and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}} \vdash t_1^{\mathrm{b}} \equiv t_1^{\mathrm{a}} : A_1^{\mathrm{b}}$.

By (ii), Lemma 2.63 (preservation of judgments by type conversion), Lemma 2.63 (preservation of judgments by type conversion), and the CONV-EQ rule, we have $\Sigma'' \vdash \Gamma_1, \Sigma U_1 V_1, \Delta_1 \equiv \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}}\ \mathbf{ctx}$, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash A_1 \equiv A_1^{\mathrm{b}}\ \mathbf{type}$, and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv t_1^{\mathrm{b}} : A_1$.

(iv) Note that $\Sigma''; \Gamma_1, \Sigma U_1 V_1, U_1^{(+1)}, V_1^{(+1)+1}, (\Sigma U_1 V_1, \Delta_1 \vdash t_1 : A_1)^{(+2)}$.

By the typing rules, $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0\,.\pi_1 : U_1^{(+1)}$,
and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, U_1^{(+1)}, V_1^{(+1)+1} \vdash \langle 1, 0\rangle : \Sigma U_1 V_1^{(+2)}$,
Let $(\Delta_1^{\mathrm{e}} \vdash t_1^{\mathrm{e}} : A_1^{\mathrm{e}}) \stackrel{\text{def}}{=} (\Delta_1 \vdash t_1 : A_1)^{(+2)+1}[\langle 1, 0\rangle][1\,.\pi_1/1]$, and let $(\Delta_1^{\mathrm{f}} \vdash t_1^{\mathrm{f}} : A_1^{\mathrm{f}}) \stackrel{\text{def}}{=} (\Delta_1 \vdash t_1 : A_1)^{(+2)+1}[2\,.\pi_1/2][\langle 1\,.\pi_1, 0\rangle]$. By Remark 2.49 (strengthening by substitution), $(\Delta_1^{\mathrm{f}} \vdash t_1^{\mathrm{f}} : A_1^{\mathrm{f}}) = (\Delta_1 \vdash t_1 : A_1)^{(+1)+1}[\langle 1\,.\pi_1, 0\rangle]$.

Observe that $1 \notin \mathrm{FV}((\Sigma U_1 V_1, \Delta_1 \vdash t_1 : A_1)^{(+2)})$. By Postulate 5 (hereditary substitution commutes), Lemma 2.63 (preservation of judgments by type conversion) and the CONV-EQ rule, we have $\Sigma'' \vdash \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{e}} \equiv \Delta_1^{\mathrm{f}}\ \mathbf{ctx}$, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{e}} \vdash A_1^{\mathrm{e}} \equiv A_1^{\mathrm{f}}\ \mathbf{type}$, and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{e}} \vdash t_1^{\mathrm{e}} \equiv t_1^{\mathrm{f}} : A_1^{\mathrm{f}}$.

Let $(\Delta_1^{\mathrm{c}} \vdash t_1^{\mathrm{c}} : A_1^{\mathrm{c}}) = (\Delta_1^{\mathrm{f}} \vdash t_1^{\mathrm{f}} : A_1^{\mathrm{f}})[0\,.\pi_2/0]$.

As earlier, let $(\Delta_1^{\mathrm{b}} \vdash t_1^{\mathrm{b}} : A_1^{\mathrm{b}}) \stackrel{\text{def}}{=} (\Delta_1^{\mathrm{e}} \vdash t_1^{\mathrm{e}} : A_1^{\mathrm{e}})[0\,.\pi_2/0]$.

By the typing rules and reflexivity, $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0\,.\pi_2 \equiv 0\,.\pi_2 : V^{(+1)+1}[0\,.\pi_1]$. By (iii) and Postulate 4 (congruence of hereditary substitution), $\Sigma'' \vdash \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}} \equiv \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{c}}\ \mathbf{ctx}$,
$\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}} \vdash A_1^{\mathrm{c}} \equiv A_1^{\mathrm{c}}\ \mathbf{type}$,
and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{b}} \vdash t_1^{\mathrm{c}} \equiv t_1^{\mathrm{c}} : A_1^{\mathrm{b}}$.

Analogously to the previous steps,
$\Sigma'' \vdash \Gamma_1, \Sigma U_1 V_1, \Delta_1 \equiv \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{c}}\ \mathbf{ctx}$,
$\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash A_1 \equiv A_1^{\mathrm{c}}\ \mathbf{type}$,
and $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv t_1^{\mathrm{c}} : A_1$.

(v) By the assumptions and Lemma 2.62 (context weakening), $\Sigma''; \Gamma_1, \Sigma U_1 V_1, (U_1, V_1, \Delta_1' \vdash t_1' \equiv v : A_1')^{(+1)}$, that is, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, U_1^{(+1)}, V_1^{(+1)+1}, (\Delta_1' \vdash t_1' \equiv v : A_1')^{(+1)+2}$.

By the typing rules, $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0\,.\pi_1 : U_1^{(+1)}$. By Postulate 4 (congruence of hereditary substitution), $\Sigma''; \Gamma_1, \Sigma U_1 V_1, V_1^{(+1)+1}[0\,.\pi_1], (\Delta_1' \vdash t_1' \equiv v : A_1')^{(+1)+2}[1\,.\pi_1/1]$.

By the typing rules, we have $\Sigma''; \Gamma_1, \Sigma U_1 V_1 \vdash 0\,.\pi_2 : V_1^{(+1)+1}[0\,.\pi_1]$. By Postulate 4 (congruence of hereditary substitution), $\Sigma''; \Gamma_1, \Sigma U_1 V_1, (\Delta_1' \vdash t_1' \equiv v : A_1')^{(+1)+2}[1\,.\pi_1/1][0\,.\pi_2/0]$.

Let $v = v'^{(+1)+2+|\Delta_1|}[1\,.\pi_1^{(+|\Delta_1|)}/1 + |\Delta_1|][0\,.\pi_2^{(+|\Delta_1|)}/|\Delta_1|]$. By the earlier definitions, $\Delta_1^{\mathrm{c}} \vdash t_1^{\mathrm{c}} \equiv v : A_1^{\mathrm{c}}) = (\Delta_1' \vdash t_1' \equiv v' : A_1')^{(+1)+2}[1\,.\pi_1/1][0\,.\pi_2/0]$.

By Postulate 4 (congruence of hereditary substitution), $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1^{\mathrm{c}} \vdash t_1^{\mathrm{c}} \equiv v : A_1^{\mathrm{c}}$.

By transitivity, $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv v : A_1$.

(vi) So far, we have shown $\Sigma''; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv v : A_1$. Analogously (replacing contexts, terms and types of the form $\square_1$ by those of the form $\square_2$), we have $\Sigma''; \Gamma_2 : \Sigma U_2 V_2, \Delta_2 \vdash t_2 \equiv v : A_2$.

(vii) It remains to show that $\mathrm{FV}(v) \subseteq \mathrm{FV}(t_1)$ and $\mathrm{FV}(v) \subseteq \mathrm{FV}(t_2)$.

We have $v = v'^{(+1)+2+|\Delta_1|}[1\,.\pi_1^{(+|\Delta_1|)}/1 + |\Delta_1|][0\,.\pi_2^{(+|\Delta_1|)}/|\Delta_1|]$
$t_1' = t_1^{((+2)+1+|\Delta_1|)}[\langle 1, 0\rangle^{(+|\Delta_1|)}/|\Delta_1|]$.

By the assumption $\mathrm{FV}(v') \subseteq \mathrm{FV}(t_1')$. Assume $z \in \mathrm{FV}(v)$. There are three possible cases:

- Case $z < |\Delta_1|$. We have $z \in \mathrm{FV}(v'^{(+1)+2+|\Delta_1|}[1\,.\pi_1^{(+|\Delta_1|)}/(1 + |\Delta_1|)][0\,.\pi_2^{(+|\Delta_1|)}/|\Delta_1|])$. By Lemma 2.51 (free variables in hereditary substitution), we have $z \in \mathrm{FV}_{|\Delta_1|}(v'^{(+1)+2+|\Delta_1|}[(1\,.\pi_1)^{(+|\Delta_1|)}/(1 + |\Delta_1|)])$. Then, because $z < |\Delta_1|$, we have $z \in \mathrm{FV}(v'^{(+1)+2+|\Delta_1|}[1\,.\pi_1^{(+|\Delta_1|)}/(1 + |\Delta_1|)])$.
  By the same token, $z \in \mathrm{FV}(v'^{((+1)+2+|\Delta_1|)})$. By Remark 2.28 (renaming and free variables), because $z < |\Delta_1|$, $z \in \mathrm{FV}(v')$.
  By the assumption, $z \in \mathrm{FV}(t_1')$. By Lemma 4.53 (free variables in substitution by pair), $z \in \mathrm{FV}(t_1') \subseteq \mathrm{FV}_{|\Delta_1|}(t_1^{((+2)+1+|\Delta_1|)}) \cup \mathrm{FV}(\langle 1, 0\rangle^{(+|\Delta_1|)})$. Because $z \notin \mathrm{FV}(\langle 1, 0\rangle^{(+|\Delta_1|)})$, then $z \in \mathrm{FV}_{|\Delta_1|}(t_1^{((+2)+1+|\Delta_1|)})$. Because $z < |\Delta_1|$, $z \in \mathrm{FV}(t_1^{((+2)+1+|\Delta_1|)})$. Because $z < |\Delta_1|$, by Remark 2.28 (renaming and free variables), $z \in \mathrm{FV}(t_1)$.
- Case $z > |\Delta_1|$. We have $z \in \mathrm{FV}(v'^{((+1)+2+|\Delta_1|)}[(1\,.\pi_1)^{(+|\Delta_1|)}/1 + |\Delta_1|][0\,.\pi_2^{(+|\Delta_1|)}/|\Delta_1|])$. By Lemma 2.51 (free variables in hereditary substitution), $z \in \mathrm{FV}_{|\Delta_1|}(v'^{(+1)+2+|\Delta_1|}[(1\,.\pi_1)^{(+|\Delta_1|)}/(1 + |\Delta_1|)])$. Because $z > |\Delta_1|$, this means $z + 1 \in \mathrm{FV}(v'^{(+1)+2+|\Delta_1|}[(1\,.\pi_1)^{(+|\Delta_1|)}/(1 + |\Delta_1|)])$.
  By the same token, $z+2 \in \mathrm{FV}(v'^{((+1)+2+|\Delta_1|)})$. By Remark 2.28 (renaming and free variables), because $z+2 > |\Delta_1|+2$, $z+1 \in \mathrm{FV}(v')$.
  By the assumption, $z + 1 \in \mathrm{FV}(t_1')$. By Lemma 4.53 (free variables in substitution by pair), $z \in \mathrm{FV}(t_1') \subseteq \mathrm{FV}_{|\Delta_1|}(t_1^{((+2)+1+|\Delta_1|)}) \cup \mathrm{FV}(\langle 1, 0\rangle^{(+|\Delta_1|)})$. Because $z+1 > |\Delta_1|+1$, $z \notin \mathrm{FV}(\langle 1, 0\rangle^{(+|\Delta_1|)})$. $z \in \mathrm{FV}_{|\Delta_1|}(t_1^{((+2)+1+|\Delta_1|)})$. Because $z+1 > |\Delta_1|$, $z+2 \in \mathrm{FV}(t_1^{((+2)+1+|\Delta_1|)})$. Because $z+2 > |\Delta_1|+1$, by Remark 2.28 (renaming and free variables), $z \in \mathrm{FV}(t_1)$.
- Case $z = |\Delta_1|$. We have $z \in \mathrm{FV}(v'^{((+1)+2+|\Delta_1|)}[1\,.\pi_1^{(+|\Delta_1|)}/1 + |\Delta_1|][0\,.\pi_2^{(+|\Delta_1|)}/|\Delta_1|])$.
  By Lemma 4.54 (free variables in substitution by irreducible), $\{z, z+1\} \cap \mathrm{FV}(v'^{(+1)+2+|\Delta_1|}[1\,.\pi_1^{(+|\Delta_1|)}/1 + |\Delta_1|]) \neq \emptyset$.

Again, by Lemma 4.54 (free variables in substitution by irreducible), $\{z, z+1, z+2\} \cap \text{FV}(v'^{(+1)+2+|\Delta_1|}) \neq \emptyset$. By Remark 2.28 (renaming and free variables), and noting that $\text{FV}(v'^{(+1)+2+|\Delta_1|}) \subseteq \{0, 1, ..., |\Delta_1|+1, ..., |\Delta_1|+3, ...\}$. we have $\{z, z+1\} \cap \text{FV}(v') \neq \emptyset$. Because $\text{FV}(v') \subseteq \text{FV}(t_1')$, we have $\{z, z+1\} \subseteq \text{FV}(t_1')$.
We show that $z \in \text{FV}(t_1)$ by contradiction. Assume $z \notin \text{FV}(t_1)$. By Remark 2.28 (renaming and free variables), $z \notin t_1^{((+2)+1+|\Delta_1|)}$. By Lemma 4.53 (free variables in substitution by pair), $\text{FV}(t_1') = \text{FV}_{|\Delta_1|}(t_1^{((+2)+1+|\Delta_1|)})$. Therefore, because $\{z, z+1\} \subseteq \text{FV}(t_1')$, $\{z, z+1\} \subseteq \text{FV}_{|\Delta_1|}(t_1^{((+2)+1+|\Delta_1|)})$. By definition of $\text{FV}_{|\Delta_1|}(\cdot)$, $\{z+1, z+2\} \subseteq \text{FV}(t_1^{((+2)+1+|\Delta_1|)})$. However, by definition, $\text{FV}(t_1^{((+2)+1+|\Delta_1|)}) \subseteq \{0, 1, ..., |\Delta_1|, ..., |\Delta_1|+3, ...\}$, which is a contradiction.
Therefore, $z \in \text{FV}(t_1)$.

Analogously, $\text{FV}(v) \subseteq \text{FV}(t_2)$.

By Definition 4.12 (heterogeneous equality), $\Sigma; \Gamma_1 \ddagger \Gamma_2, x : \Sigma U_1 V_1 \ddagger \Sigma U_2 V_2, \Delta_1 \ddagger \Delta_2 \vdash t_1 \equiv \{v\} \equiv t_2 : A_1 \ddagger A_2$; that is, $\Sigma \vDash \mathcal{C}$.

Completeness Assume that $\Theta; \Sigma \vDash \mathcal{C}$. Then we have $\Theta \vDash \Sigma$, $\Theta \vdash (\Gamma_1, \Sigma U_1 V_1, \Delta_1, A_1) \equiv (\Gamma_2, \Sigma U_2 V_2, \Delta_2, A_2)\,\mathbf{ctx}$, and $\Theta; \Gamma_1, \Sigma U_1 V_1, \Delta_1 \vdash t_1 \equiv t_2 : A_1$.

Take $\Theta' = \Theta$.

By Remark 2.13 (context inversion), Postulate 11 (injectivity of Σ) and Definition 2.16 (equality of contexts), $\Theta \vdash \Gamma_1, U_1 \equiv \Gamma_2, U_2\,\mathbf{ctx}$ and $\Theta \vdash \Gamma_1, U_1, V_1 \equiv \Gamma_2, U_2, V_2\,\mathbf{ctx}$.

By Lemma 2.62 (context weakening) $\Theta \vdash \Gamma_1, U_1, V_1, \Sigma U_1 V_1, \Delta_1, A_1^{(+2)} \equiv \Gamma_2, U_2, V_2, \Sigma U_2 V_2, \Delta_2, A_2^{(+2)}\,\mathbf{ctx}$. By Postulate 4 (congruence of hereditary substitution), $\Theta \vdash \Gamma_1, U_1, V_1, \Delta_1', A_1' \equiv \Gamma_2, U_2, V_2, \Delta_2', A_2'\,\mathbf{ctx}$.

Similarly, we have $\Theta; \Gamma_1, U_1, V_1, \Delta_1' \vdash t_1' \equiv t_2' : A_1'$.

Therefore, $\Theta' \vDash \Sigma; \mathcal{D}$, and by Remark 2.133 (restriction to a compatible signature), $\Theta'_\Sigma = \Theta_\Sigma = \Theta$.

□

Remark 4.55. Note that, if $\Sigma; \mathcal{C}\,\mathbf{wf}$, then by Postulate 1 (typing of hereditary substitution), Δ_1', Δ_2', t_1', t_2', A_1' and A_2' exist uniquely. This means that the rule's preconditions always hold.

4.6 Beyond correctness

In Section 4.5 (a reduction rule toolkit) we give a collection of rules and prove their correctness. This set of rules has an additional property which is orthogonal to the correctness, but still desirable; namely, the open-world assumption. We describe this assumption in more detail in Section 4.6.1.

On the other hand, the correctness theorem only partially specifies conditions under which a solution exists. As explained in Section 3.3, whether a solution exists is undecidable in general. However, we can also partially characterize those conditions under which the original problem is unsolvable (Section 4.6.2).

4.6.1 Open-world assumption

Either for the sake of performance, or in an interactive setting, it may be desirable to type-check a program incrementally. In our setting, this means that rule schemas applied to a problem should remain correct if the problem is extended with additional declarations or constraints. The general idea that inferences should remain valid even if new knowledge is added to the system is called the open-world assumption [31]. In our application, adding new knowledge to the system corresponds to extending the signature with additional atom declarations, and/or introducing new constraints.

Definition 4.26 (rule correctness) does not entail the open-world assumption. For example, consider the problem $\Sigma_1; \square$, where Σ_1 is defined as follows:

$$\Sigma_1 \stackrel{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \alpha : \mathbb{A}$$

And let Σ_1' be defined as follows:

$$\Sigma_1' \stackrel{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \alpha := \mathbb{a} : \mathbb{A}$$

Because, in the empty context, $\mathbb{a}$ is the only value of type $\mathbb{A}$, the rule $\Sigma_1; \square \rightsquigarrow \Sigma_1'; \square$ is in fact a correct rule (we will not prove this).

Now, let $\Theta_1 \stackrel{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \alpha := \mathbb{a} : \mathbb{A}$. Then, Θ_1 is a unique solution to $\Sigma_1'; \square$, and also to $\Sigma_1; \square$ (we will not prove this either).

Consider extending the original problem with an additional constant and an additional constraint, yielding a well-formed problem $(\Sigma_2; \vec{\mathcal{C}})$:

$$\Sigma_2; \vec{\mathcal{C}} \stackrel{\text{def}}{=} \Sigma_1, \mathbb{b} : \mathbb{A}; \cdot \ddagger \cdot \vdash \alpha \equiv \mathbb{b} : \mathbb{A} \ddagger \mathbb{A}$$

If we generalized the inference $\Sigma_1; \square \rightsquigarrow \Sigma_1'; \square$ to the extended problem $\Sigma_2; \vec{\mathcal{C}}$, we would obtain the unification rule $(\Sigma_1, \mathbb{b} : \mathbb{A}; \vec{\mathcal{C}} \rightsquigarrow \Sigma_1', \mathbb{b} : \mathbb{A}; \vec{\mathcal{C}})$. Applying this rule results in the problem $(\Sigma_1, \mathbb{b} : \mathbb{A}; \vec{\mathcal{C}})$, which has one unique solution $\Theta_2 \stackrel{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \mathbb{b} : \mathbb{A}, \alpha := \mathbb{b} : \mathbb{A}$.

By definition, $(\Theta_2)_\Sigma = \Theta_2$. However, $\Theta_2; \cdot \vdash \mathbb{a} \not\equiv \mathbb{b} : \mathbb{A}$, which means Θ_2 is not a solution for $\Sigma_1', \mathbb{b} : \mathbb{A}; \vec{\mathcal{C}}$. The rule $(\Sigma_1, \mathbb{b} : \mathbb{A}; \vec{\mathcal{C}} \rightsquigarrow \Sigma_1', \mathbb{b} : \mathbb{A}; \vec{\mathcal{C}})$ is thus not complete, and thus not correct.

Because it does not remain correct under extensions, we say that the rule $\Sigma_1; \square \rightsquigarrow \Sigma_1'; \square$ does not fulfill the open-world assumption.

However, all the rule schemas that we define in Section 4.5 (a reduction rule toolkit) do fulfill the open world assumption.

Remark 4.56 (Open-world assumption for rule schemas). Suppose $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \vec{\mathcal{D}}$. Let Σ_1 be such that Σ, Σ_1 **sig** and $\text{DECLS}(\Sigma_1) \cap \text{DECLS}(\Sigma') = \emptyset$. Then $\Sigma, \Sigma_1; \vec{\mathcal{C}} \rightsquigarrow \Sigma', \Sigma_1; \vec{\mathcal{D}}$.

Proof. By case analysis. By construction and applying Lemma 2.69 (signature weakening) to the preconditions, each of the rule schemas that we define contains the extended version of each of its rules. □

This open-world assumption can be generalized to sequences of rule applications:

Remark 4.57 (Open-world assumption for problem reduction). Suppose $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^\star \Sigma'; \vec{\mathcal{D}}$. Let Σ_1 be such that Σ, Σ_1 **sig** and $\text{DECLS}(\Sigma_1) \cap \text{DECLS}(\Sigma') = \emptyset$. and let $\vec{\mathcal{E}}$ be such that $\Sigma, \Sigma_1; \vec{\mathcal{E}}$ **wf**. Then, $\Sigma, \Sigma_1; \vec{\mathcal{C}}, \vec{\mathcal{E}} \rightsquigarrow^\star \Sigma', \Sigma_1; \vec{\mathcal{D}}, \vec{\mathcal{E}}$.

Proof. By induction on the derivation for $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^\star \Sigma'; \vec{\mathcal{D}}$, using Definition 4.28 (problem reduction) and Remark 4.56 (open-world assumption for rule schemas) at each step. □

4.6.2 Unsolvable problems

It might be the case that a problem cannot be solved; for instance, because one of its constraints is unsolvable.

Definition 4.58 (Unsolvable problem)**.** A problem $\Sigma; \overrightarrow{\mathcal{C}}$ is unsolvable if there does not exist Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$.

Correct rules preserve problem unsolvability. This means that the unification algorithm (Section 5.1) may use the rules in Section 4.5 (a reduction rule toolkit) not only to find a solution, but also to assess whether a solution exists at all.

Lemma 4.59 (Preservation of unsolvability)**.** *If $\Sigma; \overrightarrow{\mathcal{C}} \rightsquigarrow^n \Sigma'; \overrightarrow{\mathcal{D}}$, and there is no solution Θ' such that $\Theta' \vDash \Sigma'; \overrightarrow{\mathcal{D}}$, then there is no solution Θ such that $\Theta \vDash \Sigma; \overrightarrow{\mathcal{C}}$.*

Proof. Proceed by induction on n.

- Case 0: Proceed by contradiction; assume there exists Θ such that $\Theta \vDash \Sigma; \overrightarrow{\mathcal{C}}$. Then $\Theta' = \Theta$ fulfills $\Theta' \vDash \Sigma'; \overrightarrow{\mathcal{D}}$, which is a contradiction. Therefore, there is no Θ such that $\Theta \vDash \Sigma; \overrightarrow{\mathcal{C}}$.

- Case $1 + n$: Then we have $\Sigma; \overrightarrow{\mathcal{C}} \rightsquigarrow^n \Sigma''; \vec{\mathcal{E}} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$. By Lemma 4.29 (correctness of problem reduction), $\Sigma''; \vec{\mathcal{E}} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ is a correct rule; in particular, it is complete. Proceed by contradiction; assume there exists Θ'' such that $\Theta'' \vDash \Sigma''; \vec{\mathcal{E}}$. By completeness, there exists Θ' such that $\Theta'' = \Theta'_{\Sigma''}$ and $\Theta' \vDash \Sigma'; \overrightarrow{\mathcal{D}}$. This is a contradiction; therefore, there is no Θ'' such that $\Theta'' \vDash \Sigma''; \vec{\mathcal{E}}$. By the induction hypothesis, there is no Θ such that $\Theta \vDash \Sigma''; \vec{\mathcal{C}}$.

□

In general, deciding whether a problem $\Sigma; \overrightarrow{\mathcal{C}}$ has a solution is undecidable (Section 3.3). However, Lemma 4.60 shows that unsolvability is decidable in some cases.

Lemma 4.60 (Partial characterization of unsolvable problems)**.** *Consider the following classes of terms:*

$$
\begin{aligned}
T_1 &\stackrel{\text{def}}{\equiv} \{c\} \\
T_2 &\stackrel{\text{def}}{\equiv} \{f \mid f \textit{ is strongly neutral}\} \\
T_3 &\stackrel{\text{def}}{\equiv} \{\Pi AB\} \\
T_4 &\stackrel{\text{def}}{\equiv} \{\Sigma AB\} \\
T_5 &\stackrel{\text{def}}{\equiv} \{\mathrm{Bool}\} \\
T_6 &\stackrel{\text{def}}{\equiv} \{\mathrm{Set}\}
\end{aligned}
$$

Let $\Sigma; \vec{\mathcal{C}}$ be a problem, and $\mathcal{C} \in \vec{\mathcal{C}}$ a constraint, $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A'$. Suppose than any of the following hold:

(i) $t \in T_i$, $u \in T_j$, $i \neq j$.

(ii) $t = c_1$, $u = c_2$, *for some* c_1, c_2; *and* $c_1 \neq c_2$.

(iii) $t = h_1\, \overrightarrow{e_1}$, $u = h_2\, \overrightarrow{e_2}$, t *and* u *strongly neutral, and* $h_1 \neq h_2$.

Then there is no Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$.

Proof. Assume there is Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$. Then, in particular, $\Theta; \Gamma \vdash t \equiv u : A$. By Postulate 14 (existence of a common reduct), there exists r such that $\Theta; \Gamma \vdash t \longrightarrow^{\star}_{\delta\eta} r : A$ and $\Theta; \Gamma \vdash u \longrightarrow^{\star}_{\delta\eta} r : A$. However, in cases (i) and (ii), the existence of such an r leads to a contradiction.

In case (iii), by Lemma 2.163 (injectivity of elimination for strongly neutral terms), $\Theta; \Gamma \vdash t \equiv u : A$ implies $h_1 = h_2$, which is a contradiction. □

Example 4.61. By Lemma 4.60, the following problem is not solvable.

$$\cdot \,;\, \cdot \vdash \mathrm{true} \approx \mathrm{false} : \mathrm{Bool}$$

◀

However, as expected from the undecidability of the problem, not all unsolvable problems can be detected.

Example 4.62 (Unsolvable problem)**.** The following problem is unsolvable, but this cannot be determined by Lemma 4.59 and Lemma 4.60.

$$
\begin{aligned}
\alpha : \mathrm{Bool} \to \mathrm{Bool} \,;\, \cdot \vdash \alpha\, \mathrm{true} \approx \mathrm{true} : \mathrm{Bool} \;\wedge \\
\cdot \vdash \alpha\, \mathrm{true} \approx \mathrm{false} : \mathrm{Bool}
\end{aligned}
$$

◀

4.7 Extensibility and limitations thereof

The system is designed to allow addition of new unification rules; it suffices to show that each of the new rules fulfill Definition 4.26 (rule correctness).

However, adding new typing constructs with new reduction rules can break completeness. The case of the unit type with η-equality is described below.

4.7.1 Singleton types with η-equality

If a type Unit has a single element $\langle\rangle$, then an η-rule for that type would have the form:

$$\frac{\Sigma; \Gamma \vdash f : \text{Unit}}{\Sigma; \Gamma \vdash f \longrightarrow_{\eta} \langle\rangle : \text{Unit}}$$

Adding such a rule to the system is convenient, among other things, because, if one has a metavariable $\alpha : \Pi \vec{A}^n.\text{Unit}$, one can instantiate it directly as $\alpha := \lambda \vec{x}^n.\langle\rangle : \Pi \vec{A}^n.\text{Unit}$. If we consider the unit type above as a record type with no fields, this is a generalization of Rule schema 19 (metavariable η-expansion).

However, this rule has ramifications on the theory. In particular, one would have $\cdot; x : \text{Bool} \to \text{Unit} \vdash x\,\text{true} \equiv x\,\text{false} : \text{Unit}$ without true = false, which means that Lemma 2.163 (injectivity of elimination for strongly neutral terms) and the subsequent corollaries do not hold. In other words, whether a term is strongly neutral or irreducible would depend not only on the syntax of the terms, but also on their types.

Agda's implementation of η-expansion for singletons shares the same issues [7], and is thus incorrect. A correct implementation would require additional bookkeeping which may result in decreased performance, but η-equality for singleton types has use cases that cannot be straightforwardly subsumed by other existing features.

Our theoretical development does not support such a singleton type. However, we support singleton types in our prototype Tog$^+$ so that we can run the case study in Section 5.6. We weaken the Rule schema 14 (strongly neutral terms) and Rule schema 17 (metavariable pruning) rule with the aim of preserving completeness. More specifically, Rule schema 14 is only applied when the type of either side of the constraint is known not to be a singleton, and Rule schema 17 is not applied if the metavariable only occurs rigidly in subterms which, because of their syntax, may possibly have singleton type. While this works for our case study, we do not know how these fixes would scale to the whole corpus of existing Agda code.

Chapter 5

Evaluation and conclusions

In order to assess the practical usefulness of the unification rules described in Section 4.5 (a reduction rule toolkit), we implemented a unifier based on these rules on a prototype type checker for a dependently-typed language. The type-checker implementation is Tog, which is in turn based on the design by Mazzoli and Abel [36]. We extend the unification algorithm in Tog with the unification rules for twin types. We name the resulting implementation Tog$^+$(pronounced [to:g ti:]).

We evaluate Tog$^+$on examples covering the different constructions in the language.

5.1 Unification algorithm

In this section we give an overview of how the rules in Section 4.5 (a reduction rule toolkit) can be combined in order to find a solution to a unification problem. We choose the order in which the rules are applied with the goal of minimizing the need for normalizing terms, but we do not claim that our algorithm is optimal in any sense.

The algorithm is given as a series of functions in pseudocode. The values assigned can mutate inside a function, but not across function boundaries; there is also no global shared state. The entry point to the algorithm is the Solve function (Algorithm 2), which receives the constraints from the elaboration. Solve calls Refine (Algorithm 3) iteratively until all the constraints are solved or stuck. In turn, Refine uses Assign (Algorithm 4) in order to perform the metavariable instantiations which are ultimately needed to achieve a closed signature.

Remember that the elaboration algorithm (Algorithm 1) produces a signature Σ and a list of constraints $\overrightarrow{\mathcal{C}^\star}$. The function Solve is called with the signature (Σ) from the elaboration, the current list of unsolved internal constraints (initially empty) and all the constraints from the elaboration ($\overrightarrow{\mathcal{C}^\star}$). One thus obtains $(\Sigma', \overrightarrow{\mathcal{D}}) := \textsc{Solve}(\Sigma, \square, \overrightarrow{\mathcal{C}^\star})$.

The resulting signature (Σ') may then be fed back into the elaborator, which will return an extended signature ($\Sigma^{(1)} \sqsupseteq \Sigma'$) and/or additional elaborator constraints $\overrightarrow{\mathcal{C}^{\star(1)}}$. One then obtains $(\Sigma'^{(1)}, \overrightarrow{\mathcal{D}^{(1)}}) := \textsc{Solve}(\Sigma^{(1)}, \overrightarrow{\mathcal{D}^{(1)}}, \overrightarrow{\mathcal{C}^{\star(1)}})$.

This process can be repeated as long as there are new elaborator constraints, lets say, for n steps. If the final signature $\Sigma'^{(n)}$ has no uninstantiated metavariables, and the final list of constraints is empty ($\mathcal{D}^{(n)} = \Box$), then we are under the hypotheses of Theorem 4.31 (correctness of unification), which means that the elaborated program type-checks. A solution Θ may be obtained as $\textsc{close}(\Sigma'^{(n)}) \Downarrow \Theta$, but this is not necessary in order to know that the program is type-correct.

By Remark 2.96 (WHNF reduction is deterministic), the t' such that $\Sigma \vdash t \searrow t'$ is unique. This is necessary for the algorithms to be deterministic, in particular, for the auxiliary functions EtaContractWhnf and EtaExpandDefHeaded. Determinism is important from a user-experience point of view: at least for a given version of the unification algorithm, we want a given program to either type check or not type check consistently across different calls to the algorithm.

Note that, by Lemma 2.98 (equality of WHNF) we have that if $\Sigma; \Gamma \vdash t : A$ and $\Sigma \vdash t \searrow t'$, then $\Sigma; \Gamma \vdash t \equiv t' : A$. By Remark 2.102 (preservation of free variables by WHNF), $\textsc{fv}(t') \subseteq \textsc{fv}(t)$. These are precisely the preconditions in Rule schema 8 (term conversion).

Algorithm 2: SOLVE

input : Current signature Σ
Current internal constraints $\overrightarrow{\mathcal{C}}$
New constraints from the elaborator $\overrightarrow{\mathcal{C}^\star}$
output : Current signature Σ'
Updated internal constraints $\overrightarrow{\mathcal{D}}$

$\Sigma' := \Sigma$
$\overrightarrow{\mathcal{D}} := \square$
// Prepend the new constraints to the existing ones:
foreach $(\Gamma \vdash t : A \cong u : B) \in |\overrightarrow{\mathcal{C}^\star}|$ **do**
  // Definition 4.22 (elaboration into internal constraints)
  $\overrightarrow{\mathcal{D}} := \overrightarrow{\mathcal{D}} \ \wedge \ \Gamma \ddagger \Gamma \vdash A \approx B : \mathrm{Set} \ddagger \mathrm{Set} \ \wedge \ \Gamma \ddagger \Gamma \vdash t \approx u : A \ddagger B$
$\overrightarrow{\mathcal{D}} := \overrightarrow{\mathcal{D}} \ \wedge \ \overrightarrow{\mathcal{C}}$
// Solve constraints until progress is no longer made:
do
  progress := false
  $\overrightarrow{\mathcal{D}^{\mathrm{new}}} := \square$
  foreach $\mathcal{C} \in \overrightarrow{\mathcal{D}}$ **do**
    if UNBLOCKED$(\Sigma'; \mathcal{C})$ **then**
      progress := true
      /* Update the signature and collect the constraints
         produced by REFINE */
      $\Sigma', \overrightarrow{\mathcal{D}_{\mathrm{res}}} := \mathrm{REFINE}(\Sigma', \mathcal{C})$
      $\overrightarrow{\mathcal{D}^{\mathrm{new}}} := \overrightarrow{\mathcal{D}^{\mathrm{new}}} \ \wedge \ \overrightarrow{\mathcal{D}^{\mathrm{res}}}$
  /* Attempt to unblock constraints by η-expanding
     uninstantiated metavariables */
  metas := $[all\ \alpha\ such\ that\ \alpha : T \in \Sigma']$
  foreach $\alpha \in$ metas **do**
    Take Σ_1, Σ_2 such that $\Sigma' = \Sigma_1, \alpha : T, \Sigma_2$
    if *by Rule schema 19 (metavariable η-expansion),*
     $\Sigma_1, \alpha : T, \Sigma_2; \square \rightsquigarrow \Sigma''; \square$ **then**
      progress := true
      $\Sigma' := \Sigma''$
  $\overrightarrow{\mathcal{D}} := \overrightarrow{\mathcal{D}} \ \wedge \ \overrightarrow{\mathcal{D}^{\mathrm{new}}}$
while progress = true

Algorithm 3: REFINE

input : Current signature Σ
Constraint $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger B$
output : Updated signature Σ'
Constraints $\overrightarrow{\mathcal{D}}$

if *by Rule schema 1 (syntactic equality),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
// Normalize terms before checking whether other rules apply, using Rule schema 8 (term conversion) (and Rule schema 7 (constraint symmetry))
$t :=$ t' such that $\Sigma \vdash t \searrow t'$;
$u :=$ u' such that $\Sigma \vdash u \searrow u'$;
// The if case of Definition 2.158 (strongly neutral term) has a condition on the number of arguments which may be fulfilled by η-expanding the terms
$t :=$ ETAEXPANDDEFHEADED(t);
$u :=$ ETAEXPANDDEFHEADED(u);
if t *or* u *are neutral terms* **then**
  if *by Rule schema 14 (strongly neutral terms),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if $t = \alpha\, \overrightarrow{e_1}$ *or* $u = \alpha\, \overrightarrow{e_2}$ *for some* α **then**
    // We η-contract to reduce sizes of terms
    t := ETACONTRACTWHNF(t);
    u := ETACONTRACTWHNF(u);
    if $t = \alpha\, \overrightarrow{e_1}$ *and* $u = \alpha\, \overrightarrow{e_2}$ *for some* α **then**
      // For simplicity, we do not use the full generality of the intersection rule.
      if $\overrightarrow{e_1} = \vec{x}^n$ *and* $\overrightarrow{e_2} = \vec{y}^n$ **then**
        $\Sigma'; \overrightarrow{\mathcal{D}} :=$ INTERSECT$(\Sigma; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{x} \approx \alpha\, \vec{y} : A \ddagger B)$;
        return
    else if $\Sigma' :=$ ASSIGN$(\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger B)$ *and* $\Sigma' \neq$ failure **then**
      $\overrightarrow{\mathcal{D}} := \square$;
      return
else
  $t : A :=$ ETAEXPAND$(\Sigma; \Gamma \vdash t : A)$;
  $u : B :=$ ETAEXPAND$(\Sigma; \Gamma' \vdash u : B)$;
  if *by Rule schema 11 (*λ*-abstraction),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if *by Rule schema 12 (pairs),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if *by Rule schema 13 (booleans),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if *by Rule schema 3 (injectivity of* Π*),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if *by Rule schema 4 (injectivity of* Σ*),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if *by Rule schema 5 (*Bool*),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
  if *by Rule schema 6 (*Set*),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$ **then return**;
$\Sigma' := \Sigma$;
$\overrightarrow{\mathcal{D}} := \mathcal{C}$;

Algorithm 4: ASSIGN

[t] **input** : Current signature Σ^0
Constraint $\mathcal{C}^0$
output : Result signature Σ^{res}
Result constraint $\mathcal{D}$

for *two iterations* **do**
  $\Sigma := \Sigma^{(0)}$;
  Set $\mathcal{C} \stackrel{\text{def}}{=} \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B$;
  // This implicitly assigns Γ_1, Γ_2, t, u, A and B
  $\mathcal{C} := \mathcal{C}^{(0)}$;
  if $t = \alpha\, \vec{e}^N$ **then**
    $\Sigma; \mathcal{C} :=$ CHECKPATTERNCONDITION$(\Sigma, \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \vec{e}^N \approx u : A \ddagger B)$;
    if $t = \alpha\, \vec{x}$ **then**
      // The following statement implicitly updates t
      $\mathcal{C} :=$ PRUNE$(\mathcal{C})$;
      if FV$(t) \subseteq \vec{x}$ **then**
        // For brevity, we leave the actual sequence of operations unspecified. As discussed in Section 5.8, only an occurs check is implemented in practice.
        Reorder and normalize Σ^{res} using Rule schema 10 (signature conversion), and normalize t so that $\Sigma = \Sigma_1, \alpha : T, \Sigma_2$ for some Σ_1, Σ_2 and T, and CONSTS$(t) \subseteq$ DECLS(Σ_1);
        if *by Rule schema 2 (metavariable instantiation),* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \square$ **then**
          $\Sigma^{\text{res}} := \Sigma'$;
          $\mathcal{D} := \square$;
          return
    // If we couldn't instantiate α, we can at least attempt to kill some its arguments by using information on the free variables of u
    // We apply Rule schema 7 (constraint symmetry) so that u is on the LHS of the constraint, as required by the implementation of PRUNE
    $\Sigma; \mathcal{C} :=$ PRUNE$(\Sigma; \mathcal{C}')$ where $\Sigma; \mathcal{C} \rightsquigarrow \Sigma; \mathcal{C}'$ by Rule schema 7 (constraint symmetry);
  else
    $\mathcal{C} := \mathcal{C}'$ where $\Sigma; \mathcal{C} \rightsquigarrow \Sigma; \mathcal{C}'$ by Rule schema 7;
// The assignment was unsuccessful
$\Sigma^{\text{res}} := \Sigma$;
$\mathcal{D} := \mathcal{C}$;
return

Algorithm 5: CheckPatternCondition

input : Current signature $\Sigma^{(0)}$
Constraint $\mathcal{C}^{(0)}$ of the form $\Gamma_1 ‡ \Gamma_2 \vdash \alpha\, \vec{e} \approx t : A ‡ B$
output : Signature Σ^{res}
Constraint $\mathcal{D}$

```
// Remove projections and record constructors
```
$\Sigma := \Sigma^{(0)}$;
Set $\mathcal{C} \stackrel{\text{def}}{=} \Gamma_1 ‡ \Gamma_2 \vdash \alpha\, \vec{e}^{\,N} \approx t : A ‡ B$;
```
// This implicitly assigns Γ1, Γ2, α, e⃗, t, A and B
```
$\mathcal{C} := \mathcal{C}^{(0)}$;
$i := N$;
while $i \geq 1$ **do**
  if $e_i = \langle v_1, v_2 \rangle$, $\Sigma = \Sigma_1, \alpha : T, \Sigma_2$ *and by Rule schema 18 (metavariable argument currying) applied with* $n = i$, *we have* $\Sigma; \square \rightsquigarrow \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T, \Sigma_2; \square$ **then**
    $\Sigma := \Sigma_1, \beta : T_\beta, \alpha := t_\alpha : T, \Sigma_2$;
    $\alpha := \beta$;
    $\vec{e} := \vec{e}_{1,\ldots,i-1}\, v_1\, v_2\, \vec{e}_{i+1,\ldots,N}$;
    $N := N + 1$;
    $i := i + 1$;
  else if $e_i = x\, .\pi_1\, \overrightarrow{e'}$ *or* $e_i = x\, .\pi_2\, \overrightarrow{e'}$ *and, by applying Rule schema 20 (context variable currying) to* x, *we have* $\Sigma; \mathcal{C} \rightsquigarrow \Sigma'; \mathcal{C}'$ **then**
    $\Sigma := \Sigma'$;
    $\mathcal{C} := \mathcal{C}'$;
  else if $e_i = x$ *for some* x **then**
    $i := i - 1$
  else
    ```
    // Constraint not solvable; reset and abort
    ```
    $\Sigma^{\text{res}} := \Sigma$;
    $\mathcal{D} := \mathcal{C}$;
    return
```
// Now we have e⃗ = x⃗ for some x⃗
```
$\Sigma^{\text{res}} := \Sigma'$;
$\mathcal{D} := \mathcal{C}'$;
return

Algorithm 6: PRUNE

input : Current signature $\Sigma^{(0)}$
Constraint $\mathcal{C} \stackrel{\text{def}}{=} \Gamma_1 \ddagger \Gamma_2 \vdash u \approx t : A \ddagger B$
output : Updated signature Σ^{res}
Constraint $\mathcal{D} \stackrel{\text{def}}{=} \Gamma_1 \ddagger \Gamma_2 \vdash u \approx t^{\text{res}} : A \ddagger B$

```
// We define an auxiliary recursive function PruneTerm
// Let t_0 be the value of t before calling PruneTerm, t_1 be the
   value of t when PruneTerm returns, and v' the value returned
   by PruneTerm
// Precondition: t_0[[v]]^n
// Postcondition: t_1 is the result of replacing the rigid
   occurrence of v in t_0 by v'
x⃗ := FV(u);
Function PruneTerm(n,v)
    if FV(v) − n ⊈ x⃗ then
        v := v' such that Σ ⊢ v ↘ v';
        // In practice, t is not updated by PruneTerm; here we do
           it to clarify which rules are involved
        By Rule schema 8 (term conversion), t := t', where t' is the
          result of replacing the rigid occurrence of v in t by v';
        if v = β e⃗ then
            if by Rule schema 17 (metavariable pruning) applied to
              t'[[v]]^n, Σ; C ⇝ Σ'; C then
                Σ := Σ';
                Let v' be such that Σ ⊢ v ↘ v';
                By Rule schema 8, t := t', where t' is the result of
                  replacing the rigid occurrence of v in t by v';
                return v';
        else if v = ΠU_1U_2 then
            return ΠPruneTerm(n,U_1)PruneTerm(n+1,U_2);
        // We write the remaining cases more compactly
        else if v = h e⃗ and v is strongly neutral then
            return h PruneTerm(n,e_1) ... PruneTerm(n,e_1);
        else if v = ΣU_1U_2 then
            return ΣPruneTerm(U_1)PruneTerm(n+1,U_2);
        else if v = λ.v_0 then
            return λ.PruneTerm(n+1,v_0);
        else if v = ⟨v_1, v_2⟩ then
            return ⟨PruneTerm(n,v_1), PruneTerm(n,v_2)⟩;
    return v
Σ^res := Σ;
t^res := PruneTerm(0,t);
// At this point, t^res = t
return
```

Algorithm 7: INTERSECT

input : Current signature Σ
Constraint $\mathcal{C} \stackrel{\text{def}}{\equiv} \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}^{N} \approx \alpha\, \vec{g}^{N} : A \ddagger B$
output : Updated signature Σ'
Constraints $\overrightarrow{\mathcal{D}}$

$n := N$;
for *i from N down to* 1 **do**
  if *by Rule schema 16 (generalized metavariable intersection),*
  $\Sigma; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}_{1,\ldots,i-1}\, f_i\, \vec{f}_{i+1,\ldots,n} \approx \alpha\, \vec{g}_{1,\ldots,i-1}\, g_i\, \vec{g}_{i+1,\ldots,n} : A \ddagger B \rightsquigarrow$
  $\Sigma'; \Gamma \ddagger \Gamma' \vdash \beta\, \overrightarrow{f'}^{n-1} \approx \beta\, \overrightarrow{g'}^{n-1} : A \ddagger B$ **then**
    $\Sigma := \Sigma'$; $\alpha := \beta$; $n := n - 1$; $\vec{f} := \overrightarrow{f'}$; $\vec{g} := \overrightarrow{g'}$;
$\Sigma' := \Sigma$;
if $\vec{f} = \vec{g}$ **then** $\overrightarrow{\mathcal{D}} := \square$;
else $\overrightarrow{\mathcal{D}} := \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f} \approx \alpha\, \vec{g} : A \ddagger B$;

Algorithm 8: Weak head normal form of a term, η-contracted (EtaContractWhnf)

// Removes redundant λ-abstractions and pair constructors with the aim of improving performance
input : Current signature Σ
Term t
output : Term in weak head normal form t'

repeat
  stop := false;
  $t' := t'$ such that $\Sigma \vdash t \searrow t'$;
  if $t' = \lambda.f\, 0$ *and* $0 \notin \text{FV}(f)$ **then**
    $t' := f^{(-1)}$
  else if $t' = \langle f\,.\pi_1, f\,.\pi_2 \rangle$ **then**
    $t' := f$
  else
    stop := true
until stop = *true*;

Algorithm 9: Type directed top-level η-expansion (EtaExpand)

input : Signature, context, term and type: $\Sigma; \Gamma \vdash t : A$
output : Term and type $t' : A'$

$A' := A'$ such that $\Sigma \vdash A \searrow A'$;
if $t \neq \lambda.u$ *and* $A' = \Pi U V$ **then**
  $t' := \lambda.(t^{(+1)}\ @\ 0)$;
  return
else if $t \neq \langle u, v \rangle$ *and* $A' = \Sigma U V$ **then**
  $t' := \langle t\ @\ .\pi_1, t\ @\ .\pi_2 \rangle$;
  return
else
  $t' := t$;
  return

Algorithm 10: Application of η-expansion to some neutral terms (EtaExpandDefHeaded)

input : Term t
output : Term in weak head-normal form t'

if $t = \mathrm{if}\, \vec{e}^{\,n}$ *and* $n < 4$ **then**
  $t' := \lambda^{4-n}.(t^{(+(4-n))}\ (4-n-1) \ldots 0)$
else
  $t' := t$

5.2 Constraint book-keeping

Algorithm 2 (SOLVE) checks whether a constraint is UNBLOCKED before performing the (potentially expensive) REFINE operation on this constraint. The function UNBLOCKED is left unspecified in Section 5.2.1 in order to avoid cluttering the pseudocode. In this section we give an overview of how the UNBLOCKED function is implemented in Tog$^+$.

5.2.1 Constraint unblocking (UNBLOCKED)

Each constraint $\mathcal{C}$ in the Algorithm 2 (SOLVE) is internally associated with an unblocker $\mathcal{B}$. The unblocker for each constraint describes the conditions under which the algorithm should attempt to make progress on that particular constraint.

In the original Tog [37], unblockers are a disjunction of metavariables $(\alpha_1 \vee \alpha_2 \vee ... \vee \alpha_n)$. We generalize unblockers to include not only metavariables, but also constraints, and not only disjunction, but also conjunctions. The aim is to restrict the conditions under which constraints may unblock, hopefully increasing performance. We also introduce a "never" unblocker ($\bot$), for constraints which have been deemed unsolvable, and an "always" unblocker ($\top$), for completeness.

Definition 5.1 (Unblocker)**.** An unblocker $\mathcal{B}$ is defined inductively as follows:

$\mathcal{B}$	::=	${}^*\alpha, {}^*\beta, ...$			metavariable
	|	${}^*\mathcal{C}$			internal constraint
	|	$\mathcal{B}_1 \wedge \mathcal{B}_2$	|	$\mathcal{B}_1 \vee \mathcal{B}_2$	conjunction and disjunction
	|	$\top$	|	$\bot$	always and never

Initially, every constraint has the $\top$ unblocker ("always"), which means that the constraint is unblocked. Each constraint returned by REFINE is annotated with a corresponding unblocker. This unblocker is computed by the same functions that generate the new constraints, by, among other things, noting which metavariables are preventing some term from normalizing further (yielding metavariable unblockers) and which types need to be equal in order for Rule schema 2 (metavariable instantiation) to be applicable (yielding internal constraint unblockers).

Definition 5.2 (Unblocking of constraints: UNBLOCKED$(\Sigma; \mathcal{C})$)**.** We say that a constraint $\mathcal{C}$ is unblocked in Σ (written UNBLOCKED$(\Sigma; \mathcal{C})$), if, for the unblocker $\mathcal{B}$ associated with $\mathcal{C}$, we have $\Sigma \vdash \mathcal{B}$ **unblocks**.

The relation $\Sigma \vdash \mathcal{B}$ **unblocks** is defined recursively as follows:

$\Sigma \vdash {}^*\alpha$ **unblocks**	when	α is instantiated in Σ
$\Sigma \vdash {}^*\mathcal{C}$ **unblocks**	when	$\Sigma \not\approx \mathcal{C}$
$\Sigma \vdash \mathcal{B}_1 \wedge \mathcal{B}_2$ **unblocks**	when	$\Sigma \vdash \mathcal{B}_1$ **unblocks** and $\Sigma \vdash \mathcal{B}_2$ **unblocks**
$\Sigma \vdash \mathcal{B}_1 \vee \mathcal{B}_2$ **unblocks**	when	$\Sigma \vdash \mathcal{B}_1$ **unblocks** or $\Sigma \vdash \mathcal{B}_2$ **unblocks**
$\Sigma \vdash \top$ **unblocks**		

Checking whether a metavariable is instantiated is a simple table lookup, but checking whether $\Sigma \vdash {}^{*}\mathcal{C}$ **unblocks** is more involved: it is equivalent to solving the constraint itself. In the following section we explain the how this check is implemented in Tog^{+}.

5.2.2 Ordering of rule application

As described in Algorithm 2, when turning a basic constraint (Definition 3.7) into internal constraints (Definition 4.2), the constraint that unifies the types is output before the constraint that unifies the terms.

All generated constraints are initially unblocked; and unblocked constraints are attempted in the order in which they are generated. This way, the algorithm attempts to unify the sides of a twin type before the constraint containing it is attempted, hopefully minimizing the occurrence of twin types with differing sides.

We discuss two issues related to the order in which rules are applied below. We consider further optimizing the order in which rules are applied outside the scope of this work.

Effect on the applicability of syntactic equality: One may fear that the applicability of syntactic equality is sensitive to which other rules have been applied to a constraint, and that there may exist problems that become unsolvable if syntactic equality is not applied at the right time. However, in our tests, Rule schema 1 (syntactic equality) is not necessary for solving any of the examples; it just affects performance. For all the other rules, we see no obvious way in which the preconditions for one rule would cease to hold by the application of another rule.

Therefore, we believe that the order in which constraints are solved may affect performance, but should not critically affect the ability of the algorithm to find a solution.

Use of Rule schema 19 (metavariable η-expansion): Applying Rule schema 19 increases the number of metavariables, and, as shown in Section 4.5.11, also may indirectly increase the number of constraints. Thus rule should not be applied indiscriminately to all metavariables of the appropriate type, but only in the presence of constraints of the form of those in Example 4.50 or Example 4.51, and only if other approaches to solve these constraints, such as Rule schema 2 (metavariable instantiation) have proven unfruitful.

5.2.3 Constraint satisfaction

In order to efficiently assess whether $\Sigma \mid\!\approx\, \mathcal{C}$ (or, equivalently, whether $\Sigma \vdash {}^{*}\mathcal{C}$ **unblocks**), the implementation keeps a tree of constraints, which has a node for each constraint that has ever been considered by Solve. Initially, the elaborator constraints are inserted as roots, with the corresponding internal constraints as children. When a call to Refine on $\mathcal{C}$ by Solve returns constraints $\overrightarrow{\mathcal{D}^{\text{res}}}$, these are added as children of $\mathcal{C}$. If $\overrightarrow{\mathcal{D}^{\text{res}}} = \square$, then $\mathcal{C}$ is

marked as *solved*. When all the children of a constraint are marked as solved, their parent is also marked as solved.

Each constraint in the tree has an identifier. Internal constraint unblockers do not contain the constraints themselves, but rather the identifier of the node in the tree of constraints.

By the correctness (and therefore soundness) of the rules, if $\Sigma; \mathcal{C} \rightsquigarrow^\star \Sigma'; \square$, then for any extension $\Sigma'' \sqsupseteq \Sigma'$, we have $\Sigma'' \models\!\!\approx \mathcal{C}$. Furthermore, the signature produced by a sound rule is always an extension of the previous signature, and signature extension is a transitive relation (Remark 2.152). Therefore, the implementation can (partially) determine whether, for the current signature Σ, we have $\Sigma \models\!\!\approx \mathcal{C}$ (and thus, whether $\Sigma \vdash {}^*\mathcal{C}$ **unblocks**) by checking whether $\mathcal{C}$ is marked as solved in the tree.

The constraint tree built for assessing constraint satisfaction has two further use cases:

Heterogeneous context equality: The constraint tree is also used to avoid redundant conversion checks when deciding heterogeneous context equality (Definition 4.37). For instance, checking the heterogeneous context equality when assessing the applicability of Rule schema 2 (metavariable instantiation) in Algorithm 4 is equivalent to checking whether certain constraints are satisfied.

Error reporting: When the unification algorithm cannot solve all the constraints (that is, when all the remaining constraints are blocked), the user is informed of which constraints could not be solved and why they are being blocked.

By finding the elaborator constraints in the constraint tree that are ancestors to the unsolved constraints, one could even report the position in the code from which this constraint originates, and even the typing rule or some other reason why the constraint was introduced. With this information, the user could decide whether to change the term, its type, or give more of the implicit arguments explicitly.

Although collection of this information can be enabled in the implementation, we have not implemented a way to display it in a user-friendly way.

5.3 Language extensions

The language implemented by Tog (and Tog$^+$) extends the one described in Chapter 2 in the following ways:

Implicit arguments: Function arguments can be marked as implicit, by surrounding them with curly braces (Listing 5.1). Such arguments may be omitted by the user, in which case they are internally replaced by a metavariable and inferred.

Listing 5.1: Implicit arguments

```
module Implicit where
```

```
id : {A : Set} -> A -> A
id x = x

postulate B : Set
postulate b : B

example_id : B
example_id = id b
```

The step of identifying the points in the code where implicit arguments have been omitted is not straightforward. We preserve the approach used by the original Tog implementation as explained by Mazzoli and Abel [36]. In this approach, only the top level function definitions have implicit arguments, and these may only occur before non-implicit arguments.

Inductive-recursive data types: Inductive-recursive datatypes can be defined. These datatypes may have several parameters (Listing 5.2). From the point of view of unification, datatype constructors (e.g. `Nat` and `Vec`) and data constructors (e.g. `zero`, `suc`, `nil` and `cons`) behave in the same way as postulates: they are equal only to themselves, and injective with respect to the judgmental equality (Section 2.12). Recursive definitions (e.g. `map`) have a judgmental equality analogous to the rules DELTA-IF-TRUE and DELTA-IF-FALSE.

In the implementation, the constructors do not take the type parameters as arguments. This is analogous to the way that λ-abstractions and pairs are implemented. Note that this implies that the constructor (and thus the term) may have several different types. Constructor-headed terms are treated as strongly neutral; the type parameters (in the example in Listing 5.2, `(A : Set)` and `(n : Nat)`) are inferred from the type of the constraint.

Listing 5.2: Inductive data types

```
module Vec where

data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat

data Vec (A : Set) (n : Nat) : Set
data Vec A n where
  nil  : n == zero -> Vec A n
  cons : {m : Nat} (p : n == suc m) (x : A) (xs : Vec A m) -> Vec A n

map : {A : Set} {B : Set} {n : Nat} -> (A -> B) -> Vec n A -> Vec n B
map _ (nil eq)       = nil eq
map f (cons eq x xs) = cons eq (f x) (map f xs)
```

Identity type with the J axiom: The implementation includes an identity type and the J axiom. In Listing 5.3, it is shown that the built-in equality type respects substitutivity (also known as indiscernibility of identicals or Leibniz's law), and is a symmetric, transitive and congruent relation.

Listing 5.3: Properties of the identity type

```
module IdentityType where

subst : {A : Set} {x y : A} (P : A -> Set) ->
        x == y -> P x -> P y
subst P = J (\ x y _ -> P x -> P y) (\ x p -> p)

sym : {A : Set} {x : A} {y : A} -> x == y -> y == x
sym {A} {x} {y} p = subst (\ y -> y == x) p refl

trans : {A : Set} {x : A} {y : A} {z : A} ->
         x == y -> y == z -> x == z
trans {A} {x} p q = subst (\ y -> x == y) q p

cong : {A B : Set} {x : A} {y : A} (f : A -> B) ->
        x == y -> f x == f y
cong f p = subst (\ z -> f x == f z) p refl
```

Record types with η-equality: The Σ-type in the language described in the theory is generalised to dependent records with an arbitrary number of fields. These records, as does the Σ-type in the calculus, exhibit η-equality. Listing 5.4 shows the particular case of how the Σ-type may be defined as a record type.

Listing 5.4: Σ-type implemented in Tog

```
module Record where

record Sigma (A : Set)(B : A -> Set) : Set
record Sigma A B where
  constructor pair
  field
    fst : A
    snd : B fst

eta : (A : Set) (B : A -> Set) (x : Sigma A B) ->
        x == pair (fst x) (snd x)
eta A B x = refl
```

As explained in Section 4.7.1, η-equality for records with no fields as implemented in Agda leads to non-unique solutions, as we reported [7]. We implement η-equality for such records in Tog$^+$, but restrict the unification rules with the aim of preserving completeness. In particular, Rule schema 14 (strongly neutral terms) is not applied until at least one of the sides of the constraint is known not to have singleton type; and rule R-STRONG is disallowed when assessing rigid occurrences in Rule schema 17 (metavariable pruning).

Simplification of metavariable intersection: Furthermore, for simplicity, we implement the simpler metavariable intersection rule (Rule schema 15)

instead of the more general Rule schema 16. We do not expect any issues regarding unit types, as we expect that those arguments of a metavariable that are of unit type can always be killed.

5.4 Benchmarking methodology

In the coming sections we benchmark the CPU and memory usage of the implementation in order to justify certain design choices. We compare with a reference implementation, Agda, to put our figures in context.

The benchmarks are run on an Intel i7-8550U CPU, 32GiB RAM, running 64bit Linux. Agda and Tog are compiled using GHC 8.4.4 (`-O2` enabled), and the Stackage 12.26 LTS release. CPU usage is measured as the total execution time of the process with respect to the wall clock, while memory usage is measured as the maximum amount allocated from the system by the GHC runtime system (where 1 MB = 10^6 bytes). The specific versions and command line arguments of the programs used are detailed in Table 5.1.

To reduce the variability due to concurrent processes running on the same machine, we take the median of at least 10 measurements. In all plots, we report a 95% confidence interval for the median. This interval is calculated using the non-parametric bootstrap technique, which uses sampling with replacement from the data itself (in our case, 1000 samples) in order to approximate the actual distribution of a statistic (in this case, the median).

5.4.1 Comparing Tog with Agda

Tog code can be run essentially unchanged in Agda. Because of the lack of universe stratification in Tog/Tog$^+$, Agda may need to be given the `--type-in-type` option when type-checking a Tog/Tog$^+$ file. Additionally, if the Tog program being type-checked by Agda uses Tog built-in identity type, this type must be loaded into Agda by importing the module in Listing 5.5.

Listing 5.6 shows an example of how to adapt a Tog program so that it can run in both Tog and Agda:

1. `{-# OPTIONS --type-in-type #-}` disables universe stratification checks (see Remark 2.15).
2. `{-@AGDA-}` makes Tog/Tog$^+$ignore the following line.
3. `open import Prelude` includes the definitions in Listing 5.5. A file named `Prelude.agda` with the contents of Listing 5.5 must be in Agda's import path.
4. `open Sigma` makes the constructors and fields of the `Sigma` record type available in the global scope, in the same way as they are by default in Tog.

Implementation	Commit	Command line arguments
Agda	87a3b458	`--type-in-type --without-K --no-termination-check --ignore-interfaces --no-positivity-check`
Tog$^+$	c13fdd5e	`--quiet --synEquality 2 --physicalEquality --solver W --noCheckElaborated --termType HC4 --fastElaborate`
Tog$^+$ without hash consing	—"—	`--quiet --synEquality 2 --physicalEquality --termType S --solver W --noCheckElaborated --fastElaborate`
Tog$^+$ without fast elaboration	—"—	`--quiet --synEquality 2 --physicalEquality --solver W --noCheckElaborated --termType HC4`
Tog$^+$ without syntactic equality	—"—	`--quiet --synEquality 0 --solver W --noCheckElaborated --termType HC4 --fastElaborate`

Table 5.1: Command line options for the benchmarks in Chapter 5

Listing 5.5: Tog Prelude

```
{-# OPTIONS --type-in-type #-}
{-# OPTIONS --without-K #-}
module Prelude where

data _==_ {A : Set}(x : A) : A -> Set where
  refl : x == x

J : {A : Set} {x y : A}
    (P : (x : A) (y : A) -> x == y -> Set) ->
    ((z : A) -> P z z refl) -> (p : x == y) -> P x y p
J P h refl = h _
```

Listing 5.6: Adapted version of Listing 5.4

```
{-# OPTIONS --type-in-type #-}
module Record where

{-@AGDA-}
open import Prelude

record Sigma (A : Set)(B : A -> Set) : Set
record Sigma A B where
  constructor pair
  field
    fst : A
    snd : B fst

{-@AGDA-}
open Sigma

eta : (A : Set) (B : A -> Set) (x : Sigma A B) ->
         x == pair (fst x) (snd x)
eta A B x = refl
```

In order to make the benchmark results more comparable between Tog+ and Agda, we have disabled some features of Agda, such as the aforementioned universe levels (`--type-in-type` flag) termination checking (`--no-termination-check` flag) and positivity checking (`--no-positivity-check` flag), in order to not make Agda unfairly slower. Another reason to disable universe levels, together with the K axiom (`--without-K` flag), is to make the underlying type theory in Agda closer to the one in Tog and Tog+.

Noe that our benchmarks are done with a small set of programs, which where also used to guide the development of the prototype itself. Furthermore, the Agda implementation does more bookkeeping work than our prototype; for instance, it keeps track of more information associated with the type-checked terms and definitions, and it also serializes the type-checked definitions together with syntax highlighting information and writes the result to disk. On the other hand, Agda has been developed over the years by a big community, with many optimizations to deal with specific use cases. Because of these differences, we will only make claims regarding the technical feasibility of our approach; but we will not make claims about the relative performance of the two tools beyond our case study.

5.5 From Tog to Tog+

Tog+ extends Tog with an implementation of the unification algorithm described in Section 5.1. The implementation is modified with respect to Section 5.1 in order to improve performance and to accommodate the language extensions described in Section 5.3.

Apart from the unification algorithm, we modify the term representation and the elaboration procedure to increase the performance of type checking.

5.5.1 Term representation using hash-consing

If represented as their syntax trees, the size of the terms involved in type checking can be exponentially large compared to size of the original program. In Tog$^+$, in order to increase the amount of sharing in the representation, and thus reduce the memory footprint, we use hash-consing on terms.

In hash consing, introduced by Deutsch [17] in the context of Lisp, each encountered term is represented at most once in memory, and assigned a unique identifier. This has the potential of reducing the memory footprint of a collection of terms with a large number of identical subterms. It also makes it possible to compare terms for syntactic equality in constant time.

In the rest of this section we assess the performance of our hash consing implementation on two examples which are typically hard for dependent type checkers. These are families of examples of increasing size, and involve implicit arguments which, in their fully expanded form, are exponential in the size of the program, and contain many duplicated subterms. We thus expect that increasing sharing via hash consing will reduce the memory footprint, and that memoizing operations on terms will reduce the execution time. Both of these examples were originally written for Agda and then in turn adapted to the original Tog implementation.

Repeated application of the identity function: One example is repeated application of the identity function (Listing 5.7), suggested by Shao et al. [54].

When all implicit arguments are inferred, the size (without sharing) of the term that gets passed as the first implicit argument to `id` increases exponentially in the number of occurrences of `id`. We use hash-consing to reduce the memory usage of terms, and the processing footprint of term reduction and comparison.

Using hash-consing results in a great decrease in both CPU time (Figure 5.1) and maximum heap size (Figure 5.2). In the case of Agda, the example is fast because of an optimization by Norell [46] which inlines simple functions like `id`. An analogous example which precludes this optimization (Listing 5.8) brings the performance of Agda closer to that of Tog$^+$ without hash consing (c.f. Figure 5.3 and Figure 5.4).

Projection functions: Another example where there is redundancy in the term representation is given in Listing 5.9, which was presented as a particularly slow example by Mazzoli [35].

Listing 5.7: Example with $n = 20$ applications of `id`

```
module Ids20 where

id : {A : Set} -> A -> A
id x = x

id20 : {A : Set} -> A -> A
id20 = (id id id id id id id id id id
        id id id id id id id id id id)
```

Listing 5.8: Example with $n = 20$ applications of `id`*, while preventing inlining.*

```
module IdsBinder20 where

slow20 : {A : Set} -> ((A : Set) -> A -> A) -> A -> A
slow20 id = ((id _) (id _) (id _) (id _) (id _)
             (id _) (id _) (id _) (id _) (id _)
             (id _) (id _) (id _) (id _) (id _)
             (id _) (id _) (id _) (id _) (id _))
```

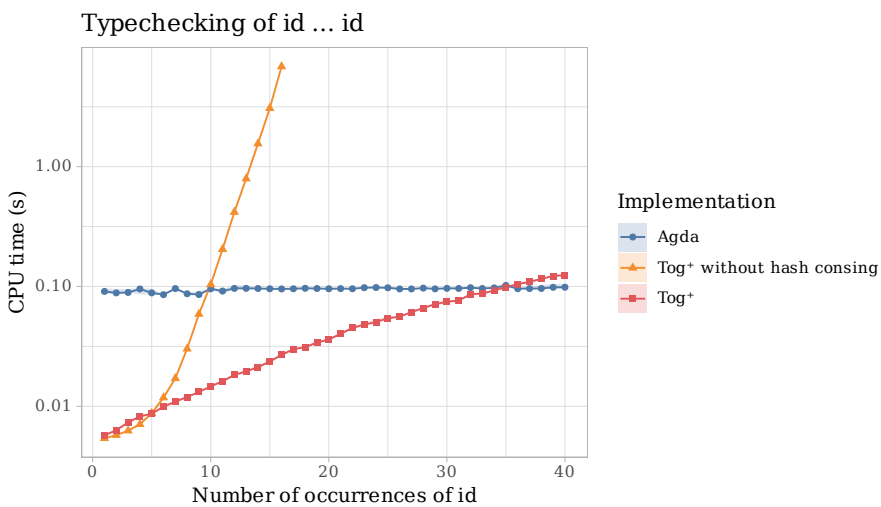

Figure 5.1: CPU usage of `id`

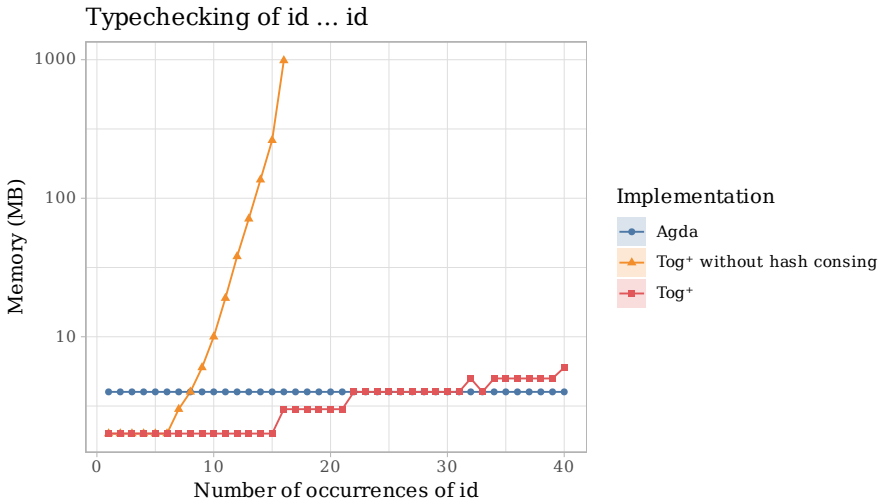

Figure 5.2: Memory usage of `id`

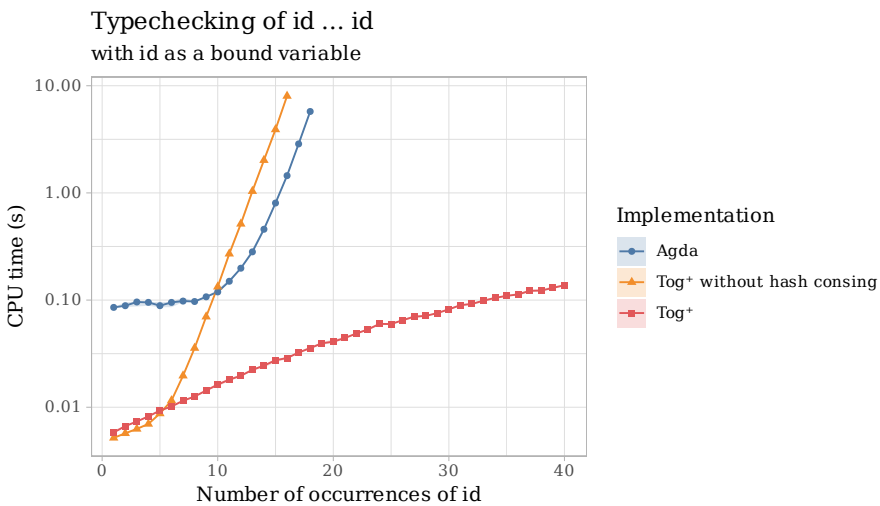

Figure 5.3: CPU usage of `id` *when the inlining optimization is prevented*

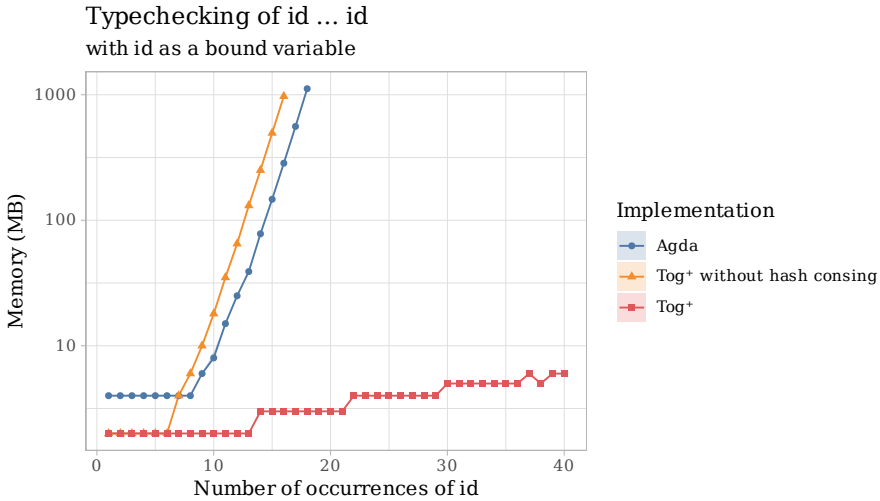

Figure 5.4: Memory usage of `id` *when the inlining optimization is prevented*

Listing 5.9: Example of projections from data type for $n = 7$

```
module Data where

data Sigma (A : Set)(B : A -> Set) : Set
data Sigma A B where
  pair : (x : A) -> B x -> Sigma A B

fst : {A : _} {B : _} -> Sigma A B -> A
fst (pair x y) = x

snd : {A : _} {B : _} (p : Sigma A B) -> B (fst p)
snd (pair x y) = y

data Unit : Set
data Unit where
  tt : Unit

Cat : Set
Cat =
  Sigma Set                                                    (\ Obj  ->
  Sigma (Obj -> Obj -> Set)                                    (\ Hom  ->
  Sigma ((X : _) -> Hom X X)                                   (\ id   ->
  Sigma ((X Y Z : _) -> Hom Y Z -> Hom X Y -> Hom X Z)         (\ comp ->
  Sigma ((X Y : _)(f : Hom X Y) -> comp _ _ _ (id Y) f == f) (\ idl ->
  Sigma ((X Y : _)(f : Hom X Y) -> comp _ _ _ f (id X) == f) (\ idr ->
  Sigma ((W X Y Z : _)
         (f : Hom W X)(g : Hom X Y)(h : Hom Y Z) ->
```

```
          comp _ _ _ (comp _ _ _ h g) f ==
          comp _ _ _ h (comp _ _ _ g f))                    (\ assoc ->
  Unit)))))))

Obj : (C : Cat) -> Set
Obj C = fst C

Hom : (C : Cat) -> Obj C -> Obj C -> Set
Hom C = fst (snd C)

id : (C : Cat) -> (X : _) -> Hom C X X
id C = fst (snd (snd C))

comp : (C : Cat) -> (X Y Z : _) -> Hom C Y Z -> Hom C X Y -> Hom C X Z
comp C = fst (snd (snd (snd C)))

idl : (C : Cat) -> (X Y : _)(f : Hom C X Y) ->
      comp C _ _ _ (id C Y) f == f
idl C = fst (snd (snd (snd (snd C))))

idr : (C : Cat) -> (X Y : _)(f : Hom C X Y) ->
      comp C _ _ _ f (id C X) == f
idr C = fst (snd (snd (snd (snd (snd C)))))

assoc : (C : Cat) ->
        (W X Y Z : _) (f : Hom C W X)(g : Hom C X Y)(h : Hom C Y Z) ->
        comp C _ _ _ (comp C _ _ _ h g) f ==
        comp C _ _ _ h (comp C _ _ _ g f)
assoc C = fst (snd (snd (snd (snd (snd (snd C))))))
```

We compare the performance of the type-checker on examples of different sizes, from 0 to 7. The size of the example corresponds to the maximum number of nested applications of `fst` and `snd` in the file. Listing 5.10 shows the test file for size 3.

Listing 5.10: Example of projections from data type for $n = 3$. For succintness, we have omitted the code (`{- ... -}`) corresponding to the definitions of `Sigma`, `fst`, `snd` and `Unit` in Listing 5.9.

```
{-# OPTIONS --type-in-type #-}
module Data3 where

{- ... -}

Cat : Set
Cat =
  Sigma Set                     (\ Obj ->
  Sigma (Obj -> Obj -> Set)     (\ Hom ->
  Sigma ((X : _) -> Hom X X)    (\ id  ->
  Unit)))
```

```
Obj : (C : Cat) -> Set
Obj C = fst C

Hom : (C : Cat) -> Obj C -> Obj C -> Set
Hom C = fst (snd C)

id : (C : Cat) -> (X : _) -> Hom C X X
id C = fst (snd (snd C))
```

Both of the functions `fst` and `snd` take the types `A` and `B` as implicit arguments. However, the type of the function `snd` also has an application of the function `fst`. This means that the type checker needs to infer and manipulate implicit arguments whose size (without sharing) grows exponentially with the number of nested applications of these functions (Listing 5.11).

Listing 5.11: Listing 5.10 with partially expanded implicit arguments ($n = 3$). If the all of the implicit arguments (`{_}`) were to be filled, each definition body would be several times bigger than as defined below.

```
{- ... -}

Obj : (C : Cat) -> Set
-- Obj C = fst C
Obj C = fst {Set}
            {\ obj -> Sigma (obj -> obj -> Set)
                        (\hom -> Sigma ((X : obj) -> hom X X)
                                       (\ id -> Unit))}
            C

Hom : (C : Cat) -> Obj C -> Obj C -> Set
-- Hom C = fst (snd C)
Hom C = fst {fst {_} {_} C -> fst {_} {_} C -> Set}
            {\ hom -> Sigma ((X : fst {_} {_} C) -> hom X X)
                            (\ id -> Unit)}
            (snd {Set}
                 {\ obj -> Sigma (obj -> obj -> Set)
                   (\ hom -> Sigma ((X : obj) -> hom X X)
                                   (\ id -> Unit))}
                 C)

id : (C : Cat) -> (X : _) -> Hom C X X
-- id C = fst (snd (snd C))
id C = fst {(X : fst {_} {_} C) -> fst {_} {_}
                                       (snd {_} {_} C) X X}
           {\ id -> Unit}
           (snd {_} {_} (snd {_} {_} C))
```

When type-checking examples of different sizes, we observe that hash-consing reduces both the CPU time (Figure 5.5) and the amount of memory

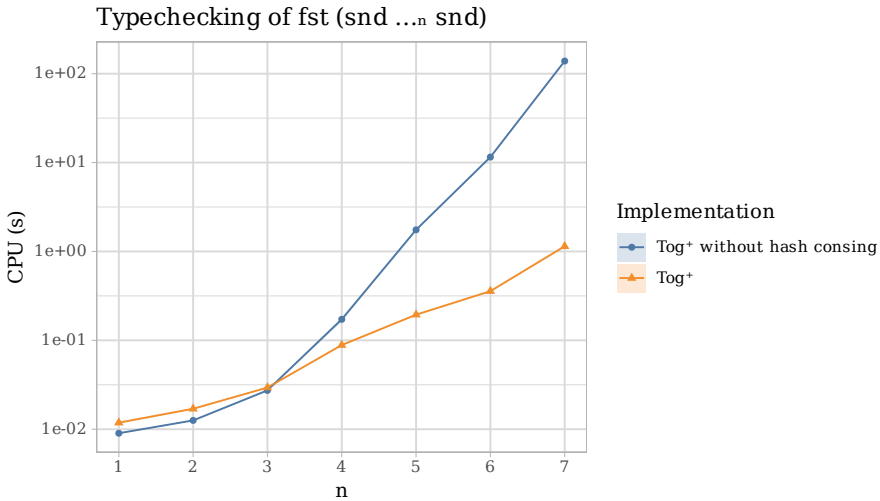

Figure 5.5: CPU usage of the `Data` *example.*

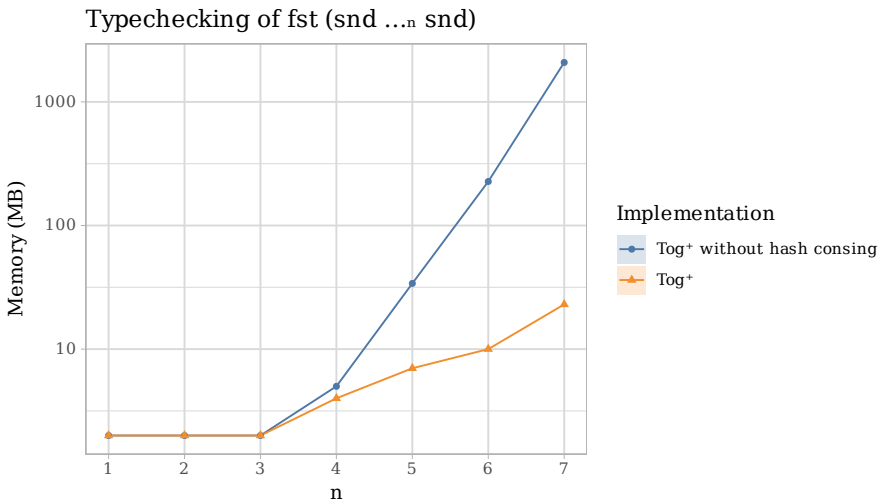

Figure 5.6: Memory usage of the `Data` *example*

required (Figure 5.6) by orders of magnitude. Further reductions could be achieved with a specialized optimization for projection-like functions, which is done for Agda as explained by Abel [1], and implemented by Norell [45].

Hash consing also allows for performing syntactic equality checks in constant time, instead of linear in the size of the terms involved. Performing these checks result in significant reductions in execution time for large examples (see Section 5.6.1).

5.5.2 Elaboration

Mazzoli and Abel [36] show that dependent type checking can be reduced to unification. This means in particular that the type-checker does not manipulate ill-typed terms. Instead, there is an elaboration step before typechecking which translates the syntax written by the user into well-typed terms (possibly involving new metavariables), and a set of unification constraints. In this approach, metavariables fulfill two functions: they stand for implicit arguments that the user omitted; and they also replace subterms which, if present, would lead to possibly untyped terms.

However, the introduction of metavariables and constraints by the elaboration procedure leads to additional constraints that must be solved, and metavariables that must be instantiated. This is particularly inefficient in the case of neutral terms where the head is of known type (see Remark 3.14). At the suggestion of Andreas Abel, we add an additional case to the elaboration rule for application so that, when a constant which is known to have a Π-type is applied to a term, the domain and codomain of the Π-type are extracted directly by the elaboration instead of the unifier.

Listing 5.12: Example of applications for $n = 7$

```
module App7 where

data Bool : Set where
  true  : Bool
  false : Bool

first : Bool -> Bool -> Bool -> Bool -> Bool -> Bool -> Bool -> Bool
first x1 x2 x3 x4 x5 x6 x7  = x1
res1 : Bool
res1 = first true true true true true true true
res2 : Bool
res2 = first true true true true true true true
res3 : Bool
res3 = first true true true true true true true
res4 : Bool
res4 = first true true true true true true true
res5 : Bool
res5 = first true true true true true true true
res6 : Bool
res6 = first true true true true true true true
res7 : Bool
```

```
res7 = first true true true true true true true
res8 : Bool
res8 = first true true true true true true true
res9 : Bool
res9 = first true true true true true true true
res10 : Bool
res10 = first true true true true true true true
```

To measure the impact of this optimization we compare the performance of elaboration in examples with 10 function applications of the same length n, for varying n. The case for $n = 7$ is shown in Listing 5.12.

Figure 5.7 and Figure 5.8 show how enabling the shortcut in the elaboration algorithm (fast elaboration) eliminates a big part of the cost of type checking. We apply similar shortcuts when elaborating λ-abstractions, `refl`, and the application of data constructors, type constructors and the `J` axiom.

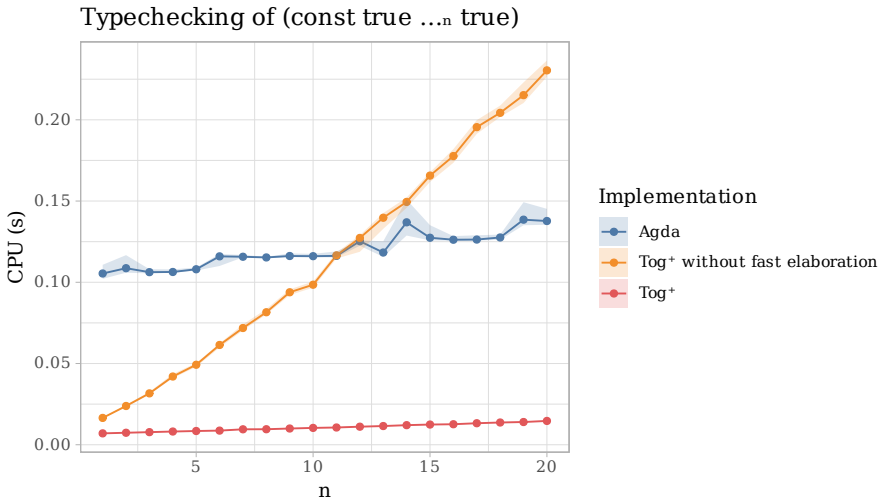

Figure 5.7: CPU usage of the `App` *example.*

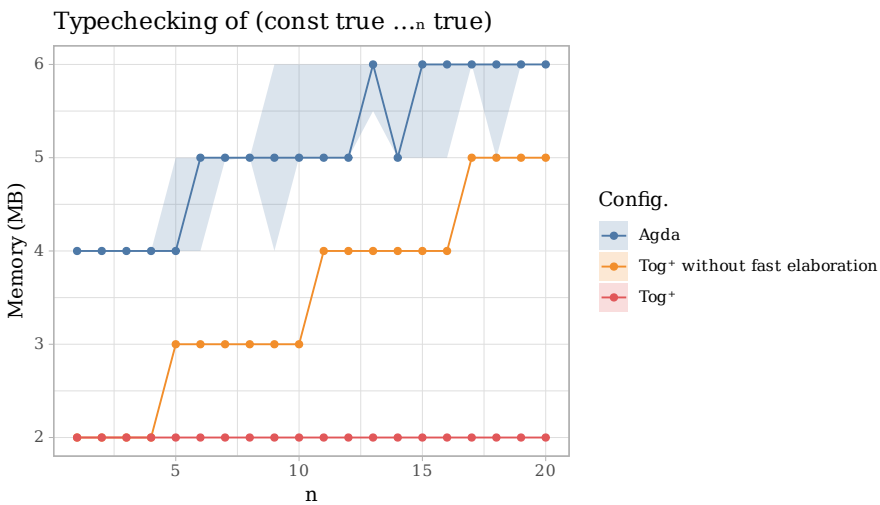

Figure 5.8: Memory usage of the `App` *example*

5.6 Case study: Type Theory in Type Theory

To assess the feasibility of our algorithm, we use it to typecheck a large example, and compare its performance with Agda.

Listing A.1 in appendix A shows a small definition of a dependently typed language, using the technique by McBride [39], and implemented for Tog by Nils Anders Danielsson.

The embedded dependently typed language is then used to define progressively more complicated structures, such as a pointed set (Listing 5.15), a reflexive graph (Listing 5.17), part of the definition of precategories (Listing 5.18), and part of the definition of a setoid (Listing 5.19).

These examples give rise to constraints such as the one in Listing 5.13 or 5.14. As explained in Section 5.7, many proof assistants cannot handle them, but Agda can. This makes these examples a good way of benchmarking the implicit argument inference capabilities of our approach, and provides some indication of whether this approach could replace the current one used in Agda.

Listing 5.13: Constraint from the pointed set example (Listing 5.15). The constraint can be solved by unifying two terms before their types have been fully unified.

```
_443 : Unit -> U
ctx: []
lhs:
  Sigma (Unit -> U) (\u ->
    Sigma (_==_ (Sigma Unit (\g -> Set) -> U)
                (\g -> u (fst g)) (\_ -> set))
          (\_ -> (A : Set) -> A))
  : Set
rhs:
  Sigma (Unit -> U) (\u ->
    Sigma (_==_ (Sigma Unit (\g -> El (_443 g)) -> U)
                 (\g -> u (fst g)) (\g -> _443 (fst g)))
          (\_ -> (A : Set) -> A))
  : Set
```

Listing 5.14: Constraint from the multigraph example (Listing 5.16). The constraint can be solved by currying a context variable with a two-sided type.

```
_1733 : (Sigma Unit (\g -> Set)) -> U
ctx: []
lhs: _==_ (Sigma (Sigma Unit (\g -> Set)) (\g -> snd g)
           -> U)
       (\g -> el (snd (fst g)))
       (\_ -> set) : Set
rhs: _==_ (Sigma (Sigma Unit (\g -> Set)) (\g -> El (_1733 g))
          -> U)
       (\g -> _1733 (fst g))
       (\g -> set) : Set
```

Listing 5.15: Definition of a pointed set

```
pointU : U
pointU =
  sigma set (\ obj -> (el obj))

point : Type empty (\ _ -> pointU)
point =
     -- Objects.
   sigma' set'
     -- Point.
   (el' (var zero))
```

Listing 5.16: Definition of a multigraph

```
graphU : U
graphU =
  sigma set (\ obj ->
  (fun (el obj) (fun (el obj) set)))

graph : Type empty (\ _ -> graphU)
graph =
     -- Vertices
   sigma' set'
     -- Edges
  (pi' (el' (var zero)) (pi' (el' (var (suc zero))) set'))
```

Listing 5.17: Definition of a fully-reflexive multigraph

```
refl-graphU : U
refl-graphU =
  sigma set (\ obj ->
  sigma (fun (el obj) (fun (el obj) set)) (\ hom ->
  (pi (el obj) (\ x -> el (hom x x)))))

refl-graph : Type empty (\ _ -> refl-graphU)
refl-graph =
     -- Objects.
   sigma' set'
     -- Morphisms.
  (sigma' (pi' (el' (var zero)) (pi' (el' (var (suc zero))) set'))
     -- Identity.
  (pi' (el' (var (suc zero)))
               (el' (app (app (var (suc zero)) (var zero)) (var zero)))))
```

Listing 5.18: Part of the definition of a precategory

```
raw-categoryU : U
raw-categoryU =
  sigma set (\ obj ->
  sigma (fun (el obj) (fun (el obj) set)) (\ hom ->
  times
    (pi (el obj) (\ x -> el (hom x x)))
    (pi (el obj) (\ x -> pi (el obj) (\ y -> pi (el obj) (\ z ->
       fun (el (hom x y)) (fun (el (hom y z)) (el (hom x z))))))))))

raw-category : Type empty (\ _ -> raw-categoryU)
raw-category =
     -- Objects.
   sigma' set'
     -- Morphisms.
  (sigma' (pi' (el' (var zero)) (pi' (el' (var (suc zero))) set'))
     -- Identity.
  (sigma' (pi' (el' (var (suc zero)))
               (el' (app (app (var (suc zero)) (var zero)) (var zero))))
     -- Composition.
     (pi' (el' (var (suc (suc zero))))                              -- X.
     (pi' (el' (var (suc (suc (suc zero)))))                        -- Y.
     (pi' (el' (var (suc (suc (suc (suc zero))))))                  -- Z.
     (pi' (el' (app (app (var (suc (suc (suc (suc zero)))))
                         (var (suc (suc zero))))
                    (var (suc zero))))                          -- Hom X Y.
     (pi' (el' (app (app (var (suc (suc (suc (suc (suc zero))))))
                         (var (suc (suc zero))))
                    (var (suc zero))))                          -- Hom Y Z.
          (el' (app (app (var (suc (suc (suc (suc (suc (suc zero)))))))
                         (var (suc (suc (suc (suc zero))))))
                    (var (suc (suc zero))))))))                 -- Hom X Z.
```

Listing 5.19: Part of the definition of a setoid

```
setoidU : U
setoidU =
  sigma set (\ a ->
  sigma (fun (el a) (fun (el a) set)) (\ rel ->
  times (pi (el a) (\ x -> el (rel x x))) (
  times (pi (el a) (\ x -> pi (el a) (\ y ->
           fun (el (rel x y)) (el (rel y x))))) (
        (pi (el a) (\ x -> pi (el a) (\ y -> pi (el a) (\ z ->
           fun (el (rel x y)) (fun (el (rel y z)) (el (rel x z)))))))
        ))))

setoid : Type empty (\ _ -> setoidU)
setoid =
          -- The set.
```

```
 sigma' set'
        -- The relation.
(sigma' (pi' (el' (var zero))
             (pi' (el' (var (suc zero))) set'))            -- _ ≈ _.
        -- Reflexivity.
(sigma' (pi' (el' (var (suc zero)))
             (el' (app (app (var (suc zero)) (var zero))
                                             (var zero))))
        -- Symmetry.
(sigma' (pi' (el' (var (suc (suc zero))))                  -- x.
        (pi' (el' (var (suc (suc (suc zero)))))            -- y.
        (pi' (el' (app (app (var (suc (suc (suc zero))))
                            (var (suc zero)))
                       (var zero)))                        -- x ≈ y.
             (el' (app (app (var (suc (suc (suc (suc zero)))))
                            (var (suc zero)))
                       (var (suc (suc zero))))))))       -- y ≈ x.
        -- Transitivity.
        (pi' (el' (var (suc (suc (suc zero)))))            -- x.
        (pi' (el' (var (suc (suc (suc (suc zero))))))      -- y.
        (pi' (el' (var (suc (suc (suc (suc (suc zero)))))))-- z.
        (pi' (el' (app (app (var (suc (suc (suc (suc (suc zero))))))
                            (var (suc (suc zero))))
                       (var (suc zero))))                  -- x ≈ y.
        (pi' (el' (app (app (var (suc (suc (suc
                                 (suc (suc (suc zero)))))))
                            (var (suc (suc zero))))
                       (var (suc zero))))                  -- y ≈ z.
             (el' (app (app (var (suc (suc (suc
                                 (suc (suc (suc (suc zero))))))))
                            (var (suc (suc (suc (suc zero))))))
                       (var (suc (suc zero))))))))))     -- x ≈ z.
```

These examples cannot be type-checked using the original Tog implementation due to the strict ordering of constraints (Section 3.9). However, our implementation Tog$^+$is able to type-check these examples, and do so with a usage of CPU time (Figure 5.9) and memory (Figure 5.10) which is comparable to the Agda implementation.

Even though the figures show Tog$^+$to be apparently more efficient than Agda, one should consider that Agda does additional processing on the type-checked terms (such as writing out syntax highlighting information, or storing associated line ranges in code). We expect this additional work to be responsible for a constant factor increase in resource usage by Agda with respect to Tog$^+$.

5.6.1 Impact of syntactic equality

With this example we can also measure the impact of the use of Rule schema 1 (syntactic equality). In Figure 5.11 we see how enabling Rule schema 1 has a

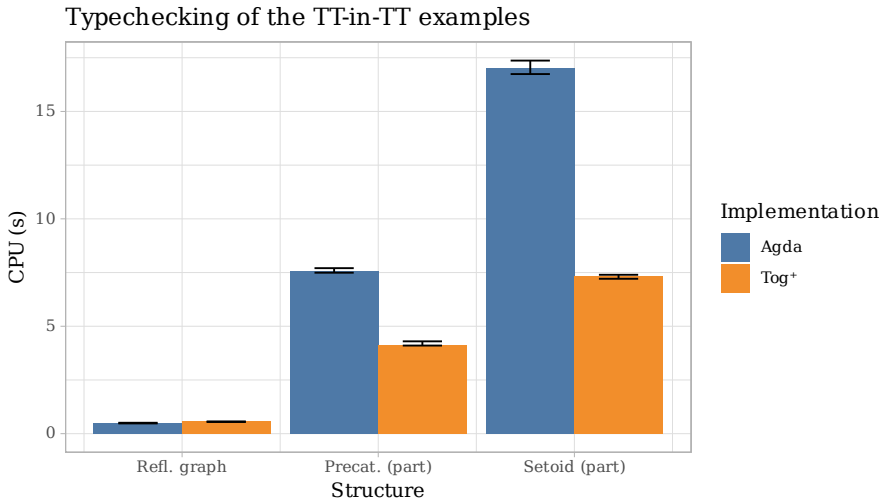

Figure 5.9: CPU usage of the TT-in-TT examples, Agda vs Tog. We show the results for the reflexive graph (Listings A.1+5.17), a part of the definition of a precategory (Listings A.1+5.18), and a part of the definition of a setoid (Listings A.1+5.19).

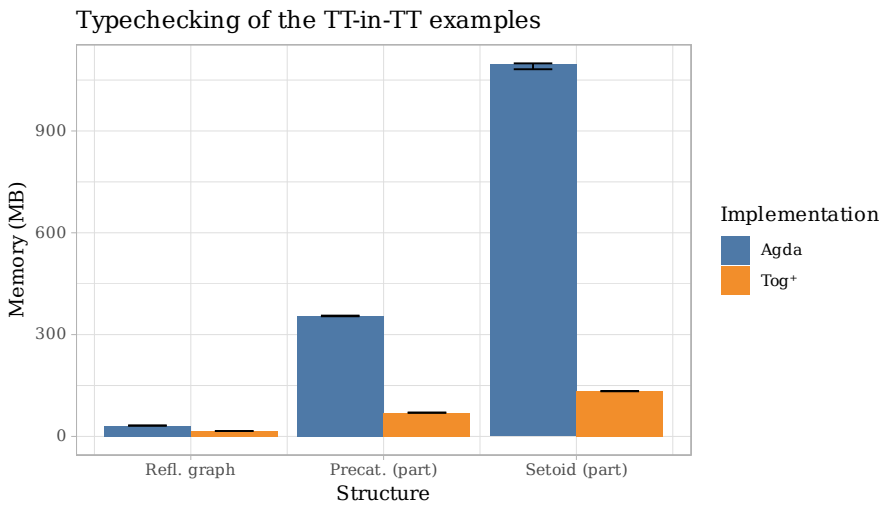

Figure 5.10: Memory usage of the TT-in-TT examples in Figure 5.9.

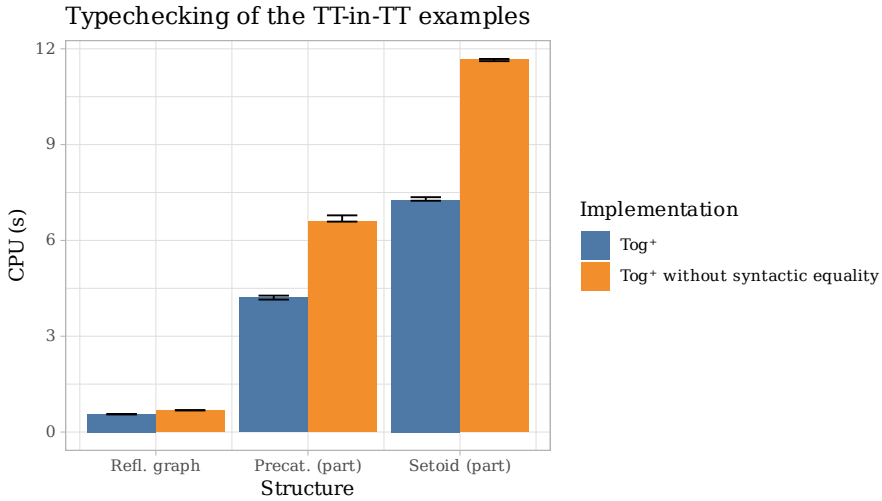

Figure 5.11: CPU usage of the TT-in-TT examples in Figure 5.9, with and without syntactic equality.

results in a consistent and significant lowered CPU usage of Tog$^+$. The impact on memory usage is less clear, as shown in Figure 5.12.

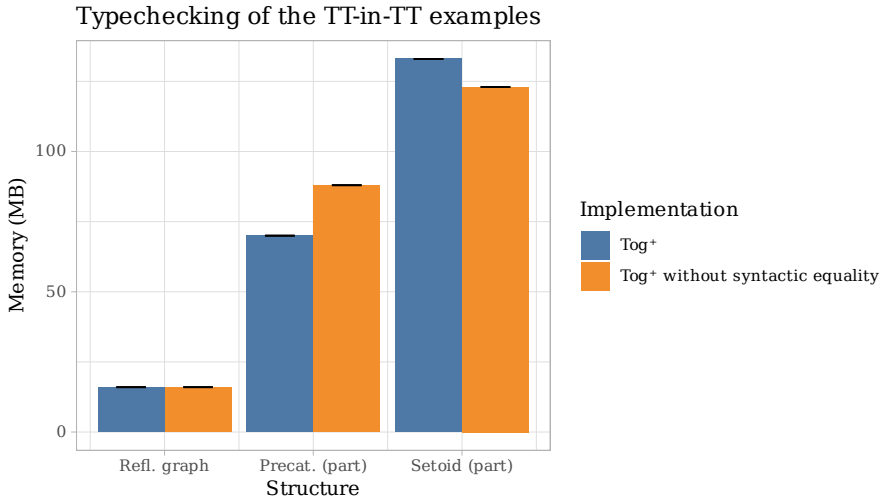

Figure 5.12: Memory usage of the TT-in-TT examples in Figure 5.9, with and without syntactic equality.

5.7 Related systems

One of our goals for this approach to unification is to allow flexibility in the order in which constraints are solved, while preserving the well-typedness of constraints at every step of the algorithm. In this section, we explain the constraints that make the example in Section 5.6 particularly interesting for us, and use it as a starting point to compare the power of Tog$^+$'s unification algorithm with that of other systems.

For the sake of readability, we define the following shorthands. The type annotations are given for the sake of clarity; the definitions themselves are purely metasyntactic.

$$
\begin{array}{lll}
\texttt{U} & \stackrel{\text{def}}{=} \text{Bool} \times \text{Bool} & : \text{Set} \\
\texttt{set} & \stackrel{\text{def}}{=} \langle \text{true}, \text{false} \rangle & : \texttt{U} \\
\texttt{el}\ \ (b : \text{Bool}) & \stackrel{\text{def}}{=} \langle \text{false}, b \rangle & : \texttt{U} \\
\texttt{El}\ \ (u : \texttt{U}) & \stackrel{\text{def}}{=} \text{if}\ (\lambda.\text{Bool})\ (u\ .\pi_1)\ \text{true}\ (u\ .\pi_2) & : \text{Bool}
\end{array}
$$

These shorthands mimic some sort of inductive data type (`U`) with two constructors (`set`, `el`) and an inductively-defined function of type $\texttt{U} \to \text{Bool}$ (i.e. `El`). Although our implementation does support inductive datatypes (Section 5.3), we use these abbreviations in this section so that we can keep the discussion within the syntax of our language as defined in Section 2.1.

We can now introduce Example 5.3. It is an example where finding a solution relies on having flexibility for the order in which constraints are solved. This example is a simplification of a problem which arises when typechecking the case study in Section 5.6 (see Listing 5.13).

Example 5.3 (Cross-dependent constraint)**.** In the problem below, a metavariable α occurs in a term $(\lambda y.(\alpha\, x))$, which has to be unified with another term $(\lambda y.\mathtt{set})$, whose type $(\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U})$ contains the same metavariable α.

$$\mathbb{F} : \mathrm{Bool} \to \mathrm{Set}, \mathbb{P} : (X : \mathrm{Set}) \to (x : X) \to \mathrm{Set}, \alpha : \mathrm{Bool} \to \mathtt{U}\,;$$
$$x : \mathrm{Bool} \vdash \mathbb{P}\ (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U})\ (\lambda y.\mathtt{set}) : \mathrm{Set} \cong \mathbb{P}\ (\mathbb{F}\ \mathrm{true} \to \mathtt{U})\ (\lambda y.(\alpha\, x)) : \mathrm{Set}$$

◀

By Definition 4.22, the constraint in Example 5.3 decomposes into two internal constraints, yielding the problem $\Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)}$, where:

$$\Sigma^{(0)} \stackrel{\text{def}}{=} \mathbb{F} : \mathrm{Bool} \to \mathrm{Set}, \mathbb{P} : (X : \mathrm{Set}) \to (x : X) \to \mathrm{Set}, \alpha : \mathrm{Bool} \to \mathtt{U}$$
$$\mathcal{C}_1^{(0)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathrm{Set} \approx \mathrm{Set} : \mathrm{Set} \ddagger \mathrm{Set}$$
$$\mathcal{C}_2^{(0)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathbb{P}\ (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U})\ (\lambda y.\mathtt{set}) \approx \mathbb{P}\ (\mathbb{F}\ \mathrm{true} \to \mathtt{U})\ (\lambda y.(\alpha\, x)) : \mathrm{Set} \ddagger \mathrm{Set}$$

The constraints are refined using Algorithm 2, following the steps below:

1. By Rule schema 1 (syntactic equality), $\Sigma^{(0)}; \mathcal{C}_1^{(0)} \rightsquigarrow \Sigma^{(0)}; \square$. By Rule schema 14 (strongly neutral terms), $\Sigma^{(0)}; \mathcal{C}_2^{(0)} \rightsquigarrow \Sigma^{(0)}; \mathcal{C}_1^{(1)}, \mathcal{C}_2^{(1)}$, where:

$$\mathcal{C}_1^{(1)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U} \approx \mathbb{F}\ \mathrm{true} \to \mathtt{U} : \mathrm{Set} \ddagger \mathrm{Set}$$
$$\mathcal{C}_2^{(1)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash (\lambda y.\mathtt{set}) \approx (\lambda y.(\alpha\, x)) : (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U}) \ddagger (\mathbb{F}\ \mathrm{true} \to \mathtt{U})$$

   Therefore, $\Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)} \rightsquigarrow^{\star} \Sigma^{(0)}; \mathcal{C}_1^{(1)}, \mathcal{C}_2^{(1)}$.

2. By Rule schema 3 (injectivity of Π), $\Sigma^{(0)}; \mathcal{C}_1^{(1)} \rightsquigarrow \Sigma^{(0)}; \mathcal{C}_1^{(2)}, \mathcal{C}_2^{(2)}$, where:

$$\mathcal{C}_1^{(2)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \approx \mathbb{F}\ \mathrm{true} : \mathrm{Set} \ddagger \mathrm{Set}$$
$$\mathcal{C}_2^{(2)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool}, y : \mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \ddagger \mathbb{F}\ \mathrm{true} \vdash \mathtt{U} \approx \mathtt{U} : \mathrm{Set} \ddagger \mathrm{Set}$$

   By Rule schema 11 (λ-abstraction), $\Sigma^{(0)}; \mathcal{C}_2^{(1)} \rightsquigarrow \Sigma^{(0)}; \mathcal{C}_3^{(2)}$, where:

$$\mathcal{C}_3^{(2)} \stackrel{\text{def}}{=} x : \mathrm{Bool}, y : (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x))) \ddagger (\mathbb{F}\ \mathrm{true}) \vdash \mathtt{set} \approx \alpha\, x : \mathtt{U} \ddagger \mathtt{U}$$

   Therefore, $\Sigma^{(0)}; \mathcal{C}_1^{(1)}, \mathcal{C}_2^{(1)} \rightsquigarrow^{\star} \Sigma^{(0)}; \mathcal{C}_1^{(2)}, \mathcal{C}_2^{(2)}, \mathcal{C}_3^{(2)}$.

3. By Rule schema 14 (strongly neutral terms), $\Sigma^{(0)}; \mathcal{C}_1^{(2)} \rightsquigarrow \Sigma^{(0)}; \mathcal{C}_1^{(3)}$, where:

$$\mathcal{C}_1^{(3)} \stackrel{\text{def}}{=} x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \mathtt{El}\ (\alpha\, x) \approx \mathrm{true} : \mathrm{Bool} \ddagger \mathrm{Bool}$$

   By Rule schema 1 (syntactic equality), $\Sigma^{(0)}; \mathcal{C}_2^{(2)} \rightsquigarrow \Sigma^{(0)}; \square$.

By Rule schema 2 (metavariable instantiation): $\Sigma^{(0)}; \mathcal{C}_3^{(2)} \rightsquigarrow \Sigma^{(1)}; \square$, where:

$$\Sigma^{(1)} \stackrel{\text{def}}{=} \mathbb{F} : \text{Bool} \to \text{Set}, \mathbb{P} : (X : \text{Set}) \to (x : X) \to \text{Set},$$
$$\alpha := \lambda x.\texttt{set} : \text{Bool} \to \texttt{U}$$

Therefore, $\Sigma^{(0)}; \mathcal{C}_1^{(2)}, \mathcal{C}_2^{(2)}, \mathcal{C}_3^{(2)} \rightsquigarrow^\star \Sigma^{(1)}; \mathcal{C}_1^{(3)}$.

4. By Rule schema 8 (term conversion), $\Sigma^{(1)}; \mathcal{C}_1^{(3)} \rightsquigarrow \Sigma^{(1)}; \mathcal{C}_1^{(4)}$, where:

$$\mathcal{C}_1^{(4)} \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool} \vdash \text{true} \approx \text{true} : \text{Bool} \ddagger \text{Bool}$$

5. Finally, by Rule schema 1 (syntactic equality), $\Sigma^{(1)}; \mathcal{C}_1^{(4)} \rightsquigarrow \Sigma^{(1)}; \square$.

Because $\Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)} \rightsquigarrow^\star \Sigma^{(1)}; \square$ and $\Sigma^{(1)}$ is closed, by Theorem 4.31, there exists a unique solution Θ such that $\Theta \vDash \Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)}$, where:

$$\Theta = \mathbb{F} : \text{Bool} \to \text{Set}, \mathbb{P} : (X : \text{Set}) \to (x : X) \to \text{Set}, \alpha := \lambda x.\texttt{set} : \text{Bool} \to \texttt{U}$$

5.7.1 Comparison with Coq, Matita, Idris, Lean and Tog

We test examples equivalent to Example 5.3 on the dependently-typed proof assistants Coq 8.11.0, Matita 0.99.3, Idris 1.3.2, Lean 3.4.2, and also on the original Tog. All of these implementations get stuck on the corresponding example (see Listing 5.20, and, in appendix B, Listings B.1, B.2, B.3 and B.4). In all cases, there is a metavariable `alpha` with a variable x in scope. In all cases, the type checker is not able to infer a term for `alpha`, even though `set'` is the unique solution. The example indeed type checks in all the proof assistants once the solution for `alpha` is filled.

Listing 5.20: Minilang example in Coq:

```
(* Atoms *)
Axiom F : bool -> Set.
Axiom Pred : forall (X : Set) (x : X), Set.

(* Shorthands *)
Definition U    : Set := bool * bool.
Definition set' : U  := (true, true).
Definition el'  (b : bool) : U := (false, b).

Definition El (u : U) : bool :=
  match (fst u) with
    | true  => false
    | false => (snd u)
  end.

(* Metavariables and constraints *)
Definition c1 :
```

```
forall (x: bool),
 let alpha : U := _ in
  (Pred ((F (El alpha)) -> U) (fun y => set')) ->
  (Pred ((F true)       -> U) (fun y => alpha)) :=
fun x y => y.
```

Output:

```
File "Minilang.v", line 22, characters 16-17:
Error:
In environment
x : bool
y : Pred (F (El ?u) -> U) (fun _ : F (El ?u) => set')
The term "y" has type "Pred (F (El ?u) -> U) (fun _ : F (El ?u) => set')"
while it is expected to have type
 "Pred (F true -> U) (fun _ : F true => ?u)".
```

5.7.2 Comparison with Agda

Agda is able to deal successfully with an example analogous to the ones in Section 5.7.1. However, Agda's unifier may at some times be over-eager when comparing terms, which results in ill-typed terms in the course of unification, as reported by Norell [33].

Consider the well-formed unification problem below. Note that `F` and `f` could also be defined only in terms of if; we use pattern-matching syntax for the sake of readability.

$$\begin{aligned}&\mathtt{Nat} : \mathrm{Set},\\ &\mathtt{0} : \mathtt{Nat},\\ &\mathtt{1} : \mathtt{Nat},\\ &\mathtt{2} : \mathtt{Nat},\\ &\mathbb{D} : \mathtt{Nat} \to \mathrm{Set}\\ &\alpha : \mathtt{Nat} \to \mathrm{Set},\\ &\beta : \mathtt{Nat} \to \mathrm{Bool},\end{aligned}$$

```
F : Bool → Set
F false = Bool
F true  = Nat

f : (b : Bool) → F b → Nat
f false false = 0
f false true  = 1
f true  x     = 2
```

;

$$\begin{aligned}
&\cdot \vdash (x : \mathtt{Nat}) \to \alpha\, x : \mathrm{Set} \approx (x : \mathtt{F}\,(\beta\,0)) \to \mathbb{D}\,(f\,(\beta\,0)\,x) : \mathrm{Set} && (1)\\
&\cdot \vdash \beta : \mathtt{Nat} \to \mathrm{Bool} \quad \approx \lambda.\mathrm{false} : \mathtt{Nat} \to \mathrm{Bool} && (2)\\
&\cdot \vdash \alpha\,\mathtt{0} \quad \approx \mathbb{D}\,0 : \mathrm{Set} && (3)
\end{aligned}$$

Coq, Matita, Idris, Lean, Tog and Tog$^+$ correctly instantiate $\beta := \lambda.\mathrm{false}$, and then recognize that the resulting problem has no solution, because $\mathtt{Nat} \neq \mathtt{F}\,(\beta\,0)$.

However, Agda will first solve constraint (1), incorrectly instantiating α with an ill-typed term $(\lceil \alpha := \lambda x.\mathbb{D}\,(f\,(\beta\,0)\,x) : \mathtt{Nat} \to \mathrm{Set} \rfloor)$. Agda will then solve constraint (2), instantiating $\beta := \lambda.\mathrm{false} : \mathtt{Nat} \to \mathrm{Bool}$. Finally, Agda will attempt to solve constraint (3) by reducing the LHS to the ill-typed term $\lceil \mathbb{D}\,(f\,\mathrm{false}\,0) \rfloor$. This leads to a crash in the Agda type checker, which assumes that all intermediate terms are well-typed.

This behaviour can result in unexpected crashes even in well-typed programs, for example when using instance search: see the issues reported by Abel et al. [6], Abel et al. [8] and Norell [47].

5.7.3 Comparison with the twin variable approach

By implementing twin types in the style of Gundry and McBride [25, 27], Tog$^+$ is able to type check an example analogous to the ones in Section 5.7.1. We make certain changes to their unification rules to make them more amenable to an implementation in the context of a dependent type checker such as Agda.

Twin contexts, but no twin variables: We have a strict separation of variables on both sides of the context. That is, variables on the left hand side of the constraint only reference the left hand size of the context, and viceversa. Because it is always clear which side of a context a variable refers to, we can dispense with the twin variable annotations (Section 3.10) altogether. This reduces the space of possible terms, which is specially interesting in an implementation with hash consing as it reduces the memory footprint (Section 5.5.1).

Heterogeneous syntactic equality: Rule schema 1 (syntactic equality) can immediately solve constraints in which both the LHS and the RHS are syntactically identical. This rule mimics the one proposed by Gundry and McBride [25, cf. (4.1) in Figure 4.14].

When type checking our case study (Section 5.6), enabling Rule schema 1 (syntactic equality) results in a significant speed-up compared to recursively unifying the term using the remaining unification rules (Figure 5.11).

In our version of the rule, we only check whether the terms are equal, but completely ignore the types. The soundness of the rule is justified in terms of Definition 4.12 (heterogeneous equality).

By ignoring the types, we avoid the potential cost of comparing them for equality. Also, because variables do not have twin type annotations, whether the two sides of the context are equal does not need to be accounted for; in fact, we ignore the context completely. Furthermore, when using hash consing, the syntactic equality becomes a constant time operation. We expect that this simplification of the rule leads to an improvement in performance.

Avoiding type checking in metavariable instantiation: The preconditions for Rule schema 2 (metavariable instantiation) ensure that the body of α has the right type; that is, $\Sigma_1; \cdot \vdash \lambda\vec{y}.t' : A$. As suggested by Gundry and McBride [25, 27] and also as implemented in Tog, the implementation does not type check the metavariable body (as is done in the original implementation by Gundry [26]), but instead relies on conditions on the context and the types to establish that the body of α has the right type.

In Rule schema 2 the preconditions for well-typedness of the body of α are phrased in terms of Definition 4.37 (heterogeneously equal contexts modulo variables) for the context and Definition 4.12 (heterogeneous equality) for the type. However, the middle term of these heterogeneous equalities (e.g. v for

$\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \equiv\{v\}\equiv u : A_1 \ddagger A_2$) is not computed explicitly in the implementation (Tog$^+$). Instead, a unification constraint is associated with each twin type, and the twin types are recognized as heterogeneously equal when the corresponding constraint is solved.

Note that Definition 4.37 (heterogeneously equal contexts modulo variables) also performs a form of context strengthening, by ignoring those variables which are not used in the constraint. To make this strengthening sound, our Definition 4.12 imposes an additional constraint on the free variables of the middle term; namely, $\mathrm{FV}(v) \subseteq \mathrm{FV}(t) \cup \mathrm{FV}(u)$. In practice, this restriction does not affect the choice of rules or the implementation of the algorithm, as neither extracting subterms nor reducing terms according to Definition 2.41 ($\delta\eta$-normalization step) introduces new free variables.

Heterogeneous context currying: As opposed to the corresponding rule by Gundry and McBride [25, cf. (4.15) in Figure 4.14], our context currying rule (Rule schema 20) works for any context variable where both sides have the required form, even if the two sides of the twin type are otherwise not equal. Despite being more powerful in this sense, the implementation becomes more straightforward than if we also had to check whether both sides of the twin type can be unified.

Listing 5.14 shows an example where such heterogeneous currying is needed. Here is a distilled version of the resulting constraint:

Example 5.4 (Heterogeneous currying constraint)**.**

$$
\begin{aligned}
&\mathbb{F} : \mathrm{Bool} \to \mathrm{Set},\\
&\mathbb{P} : (X : \mathrm{Set}) \to (x : X) \to \mathrm{Set},\\
&\alpha : \mathrm{Bool} \to \mathtt{U}\\
&\mathbf{;}\\
&. \vdash\\
&\mathbb{P}\ (\Sigma(x : \mathrm{Bool})(\mathbb{F}\ x) \qquad\qquad \to \mathtt{U})\ (\lambda y.\mathtt{el}\ (y\ .\pi_1))\\
&\cong\\
&\mathbb{P}\ (\Sigma(x : \mathrm{Bool})(\mathbb{F}\ (\mathtt{El}\ (\alpha\ x))) \to \mathtt{U})\ (\lambda y.\alpha\ (y\ .\pi_1))\\
&: \mathrm{Set} \ddagger \mathrm{Set}
\end{aligned}
$$

The unique solution is $\Theta \stackrel{\text{def}}{=} \mathbb{F} : \mathrm{Bool} \to \mathrm{Set}, \mathbb{P} : (X : \mathrm{Set}) \to (x : X) \to \mathrm{Set}, \alpha := \lambda z.\mathtt{el}\ z : \mathrm{Bool} \to \mathtt{U}]$. ◀

The resulting example can be solved by using Rule schema 20 (context variable currying), but it is not obvious to us that it can be tackled with the rules proposed by Gundry and McBride [25, cf. Figure 4.14].

Otherwise, both Gundry and McBride's approach [25] and ours seem to be similar regarding the kind of constraints that they can solve.

Most general unifiers vs. closed metasubstitutions: Our correctness theorem is phrased in the style of Abel and Pientka [3]. Solutions are (closed) metasubstitutions Θ, in which each metavariable is assigned a closed term. By contrast, Gundry and McBride phrase the correctness in terms of producing a most general unifier.

For a program to be type-correct, closed solutions to all metavariables must be found, in which case the most general unifier will also be the unique closed solution.

If no closed solution exists, it is possible that the original problem may be reduced to a problem with no constraints (e.g. $\Sigma; \vec{\mathcal{C}} \leadsto^\star \Sigma'; \square$), where some of the metavariables in Σ' are uninstantiated.

One important difference between the most-general unifier approach and the closed metasubstitution approach is that the former automatically implies the open-world assumption (Section 4.6.1), which means in particular that extending the signature with additional constants does not invalidate the uniqueness of the obtained solutions. With closed metasubstitutions, whether this assumption holds or not depends on the specific rules used. As we explain in Remark 4.57, the assumption does hold for our choice of unification rules.

5.8 Future work

The goal of this line of research is a verified-correct implementation of dependent type checking with implicit arguments. Our work gets us closer to this goal, but leaves however some main questions unaddressed.

Postulates: We make a limited number of assumptions which correspond to our intuition about how terms in the type theory we work in should behave.

However, as of November 2018, and according to Krishnaswami [32], proving the correctness of hereditary substitution in a dependently-typed language with large elimination without resorting to a larger theory with explicit substitutions is an open question.

A proof of correctness of unification in the given type theory necessitates a proper stratification of Set that replaces the SET rule. The existence of hereditarily-substituted terms in the resulting, stratified theory must then be addressed.

Termination and coverage of unification: We only address the correctness of the individual rules that make up the algorithm. However, we do not assess whether the unification algorithm always terminates, and under which circumstances it can find a solution.

As Gundry and McBride explain in Gundry's thesis [25], the rule SET easily leads to non-termination. Once the theory is properly stratified the question of whether the algorithm terminates can be considered.

Mazzoli and Abel [36] show that unification problems arising from the elaboration of programs without metavariables always fall entirely in the pattern fragment, and thus, if the program is well-typed, then the resulting unification problem can always be solved by pattern unification. Because our unification algorithm is implemented using their algorithm as a starting point, we conjecture that this property also holds for our implementation.

Singleton types with η-equality: The case study in Section 5.6 makes use of singleton types with η-equality in order to avoid the need to write empty environments explicitly.

Agda uses the η-equality in a sound way, but ignores the impact it has on completeness when it comes to pruning. Thus, pruning may produce non-unique solutions [7]. However, due to lack of evidence of the practical impact of this issue, the behaviour is still in place in Agda.

In our implementation (Tog$^+$), we introduce a unit type with η-equality, but simultaneously weaken Rule schema 17 (metavariable pruning) and Rule schema 14 (strongly neutral terms) with the aim of preserving correctness.

A full solution would be able to preserve the full power of pruning while addressing the issues that η-equality for singleton types presents for its completeness. This would however require keeping track of the necessary information about the types of subterms, which would in turn require important changes in our implementation. We consider this out of the scope of this work.

Generalization of performance: The example we choose performs adequately with the hash-consing implementation. However, hash-consing has its own drawbacks. A successful implementation must either also use hash-consing and/or find a mechanism to deal with the duplicated effort due to the twin types.

We believe that by perfecting the book keeping associated with twin types explained in Section 5.2.3, it would be possible to minimize the presence of twin types in the constraints, and also reduce the amount of redundant computation.

Signature ordering and occurs check: By Definition 2.151 (signature extension) and Lemma 2.155 (preservation of judgments under signature extensions), reordering a signature does not affect which judgments hold under the signature (as long as the resulting signature is well-formed). Therefore, the ordering of the signature is of limited relevance, as long as an ordering which makes the signature well-formed exists.

When implementing the algorithm in Section 5.1, we do not reorder the signature as needed. Instead, the signature is kept as an unordered set.

$$\{\mathbb{A} : \mathrm{Set}, \alpha := \beta : \mathbb{A}, \beta := \mathbb{o} : \mathbb{A}, \mathbb{o} : \mathbb{A}\}$$

However, in this unordered signature, a cyclic dependency between metavariables could lead to an infinite sequence of reductions:

$$\Sigma \cong \{\mathbb{A} : \mathrm{Set}, \alpha := \beta : \mathbb{A}, \beta := \alpha : \mathbb{A}\}$$

$$\Sigma; \cdot \vdash \alpha \longrightarrow_{\delta\eta} \beta \longrightarrow_{\delta\eta} \alpha \longrightarrow_{\delta\eta} \beta \longrightarrow_{\delta\eta} \ldots : \mathbb{A}$$

The issue is that the body of α mentions β which in turn mentions α. To avoid this issue, before instantiating a metavariable, we perform an *occurs check* to ensure that instantiation will not introduce any cyclic dependencies in the signature between metavariable bodies.

Another problem in an unordered signature is that of metavariables or atoms occurring directly or indirectly in their own type. This leads to what is called "very dependent types", and allows perplexing declarations such as the following:

$$\{\alpha := \mathbb{D}\ \beta : \mathrm{Set}, \beta := \mathbb{d} : \alpha, \mathbb{D} : \alpha \to \mathrm{Set}, \mathbb{d} : \mathbb{D}\ \beta\}$$

However, no analogous check is performed for the occurrence of constants in their types in our implementation at the point of instantiation. This is neither done in Agda, as reported by Abel et al. [5] using the example above.

Unlike metavariables occurring in themselves, which can easily lead to non-termination, examples such as the one above do not seem to be an issue for Agda in practice. The lack of cycles in the declarations in the final signature can be ensured after unification is completed, which was done by Agda together with the termination check for recursive function definitions (until this check was disabled for practical reasons [5]).

5.9 Conclusions

This thesis demonstrates how the techniques by Mazzoli and Abel [36] and Gundry and McBride [27] can be leveraged and extended in order to type-check challenging dependently typed programs.

At the elaboration phase, we use the algorithm proposed by Mazzoli and Abel [36] with some minor modifications (Section 5.5.2).

For unification, we take the algorithm by Gundry and McBride [27], simplify it, and extend it with a shortcut syntactic equality (Rule schema 1) and context currying for twin variables (Rule schema 20).

In Section 4.5 we provide a detailed argument for the correctness of our unification rules, modulo some (admitedly incorrect) postulates. The correctness argument is done on the same term representation as is used in the implementation (β-normal terms, no explicit substitutions, variables as deBruijn indices). Despite its reliance on certain postulates about the type theory, and the difficulty of trusting large technical proofs that have not been machine-checked, we believe that the correctness argument is detailed enough to provide a good level of assurance of the soundness and completeness of the unification rules.

The evaluation section suggests that implementing a unification algorithm based on twin-types in a dependently-typed programming language such as Agda could be done while using the same underlying type theory, and result in an improved preservation of internal invariants over the current state of affairs.

There are still mismatches between the theoretical rules (Section 4.5) and the actual implementation (Section 5.1), which are recounted in Section 5.8. Many of these mismatches are also shared with the Agda implementation; we hope that this thesis will show the more salient obstacles which stand in the way of a completely invariant-preserving implementation.

Finally, our prototype uses hash consing as a quick-and-dirty alternative to the years of optimization work done by the Agda community. The fact that a straightforward optimization such as hash consing results in type-checking times comparable to the current Agda implementation suggests that the our unification technique could be implemented in an existing proof assistant without drastically increasing its resource usage. It also suggests that hash consing may in some cases be a potentially useful optimization.

Appendix A

Code for the TT-in-TT case study

Listing A.1: Definition of the dependently-typed language

```
module TTInTT where

-- We define a substitutitivity property using the
-- J axiom.
subst : {A : Set} {x y : A} (P : A -> Set) ->
        x == y -> P x -> P y
subst P = J (\ x y _ -> P x -> P y) (\ x p -> p)

-- We define the type with no elements as its Church encoding
Empty : Set
Empty = (A : Set) -> A

-- We define the unit type with η-equality.
-- All values of type Unit are definitionally equal to tt.
record Unit : Set
record Unit where
  constructor tt

-- We define a sum type as follows
data Either (A : Set) (B : Set) : Set
data Either A B where
  left  : A -> Either A B
  right : B -> Either A B

-- We define a Σ-type with η-equality, as in the
-- theoretical development.
record Sigma (A : Set) (B : A -> Set) : Set
record Sigma A B where
  constructor pair
  field
```

```
    fst : A
    snd : B fst

-- This is the uncurry operation for Σ-types, which
-- mimics the uncurry operation for pairs in Haskell
uncurry : {A : Set} {B : A -> Set} {C : Sigma A B -> Set} ->
          ((x : A) (y : B x) -> C (pair x y)) ->
          ((p : Sigma A B) -> C p)
uncurry f p = f (fst p) (snd p)

-- Non-depedent product types are a specific case of
-- Σ-types.
Times : Set -> Set -> Set
Times A B = Sigma A (\ _ -> B)

------------------------------------------------------------------------
-- A universe

-- We define a datatype to encode a fragment of the universe of types
-- in dependent type theory.  Note that this is an inductive-recursive
-- definition, because a function that eliminates from U (i.e. El) is
-- used in the definition of U itself.

data U : Set

El : U -> Set

data U where
  set   : U
  el    : Set -> U
  sigma : (a : U) -> (El a -> U) -> U
  pi    : (a : U) -> (El a -> U) -> U

-- The El function returns the Agda type corresponding to a
-- given code (i.e. a term of type U).

El set         = Set
El (el A)      = A
El (sigma a b) = Sigma (El a) (\ x -> El (b x))
El (pi a b)    = (x : El a) -> El (b x)

-- Abbreviations.

fun : U -> U -> U
fun a b = pi a (\ _ -> b)

times : U -> U -> U
times a b = sigma a (\ _ -> b)
```

```
-- We now define a language to describe a fragment of dependent type
-- theory with a universe of types U. We do this by defining syntax
-- for contexts, variables, types and terms.

------------------------------------------------------------------------
-- Contexts

data Ctxt : Set

-- Types.

Ty : Ctxt -> Set

-- Environments.

Env : Ctxt -> Set

-- A context is a list of types. Each type in the list is typed
-- in the preceding context G.

data Ctxt where
  empty : Ctxt
  snoc  : (G : Ctxt) -> Ty G -> Ctxt

-- A type gives a code for each possible assignment of Agda terms to the
-- variables in the context.

Ty G = Env G -> U

-- The environment associated to a context is an Agda type whose
-- elements are assignments of terms to each of the variables in the
-- context of the Agda type corresponding to the type of that variable.

Env empty      = Unit
Env (snoc G s) = Sigma (Env G) (\ g -> El (s g))

-- Variables are represented by their corresponding de Bruijn indices,
-- using the auxiliary functions zero and suc.
--
-- Var G T is the type of variables of type T in context G.
-- Such a variable is either:
--   - The last variable in the context, if, for an abstract
--     environment 'g', the code of that variable's type (s (fst g))
--     is equal to (t g).
--   - A variable of some type 'u' in the preceding context, if, for
--     an abstract environment 'g', the code of that variable's type
--     (u (fst g)) is equal to (t g).
--
-- In both cases, s and u are typed only in the initial fragment of
```

```
-- the context (all variables except the last), so we restrict the
-- environment g to its initial segment (fst g).
Var : (G : Ctxt) -> Ty G -> Set
Var empty      t = Empty
Var (snoc G s) t =
  Either ((\ g -> s (fst g)) == t)
         (Sigma _ (\ u -> Times ((\ g -> u (fst g)) == t) (Var G u)))

-- The variable (zero) has type s in any context where the last
-- variable has type s.
zero : {G : _} {s : _} ->
       Var (snoc G s) (\ g -> s (fst g))
zero = left refl

-- If x has type t in G, then (suc x) has type t in the extended
-- context (snoc G s).
suc : {G : _} {s : _} {t : _}
      (x : Var G t) ->
      Var (snoc G s) (\ g -> t (fst g))
suc x = right (pair _ (pair refl x))

-- A lookup function.
-- Given a variable and an environment, the term mapped to that
-- variable in the environment is returned.
lookup : (G : Ctxt) (s : Ty G) -> Var G s -> (g : Env G) -> El (s g)
lookup empty      _ absurd     _ = absurd _
lookup (snoc vs v) _ (left  eq) g = subst (\ f -> El (f g)) eq (snd g)
lookup (snoc vs v) t (right p)  g =
   subst (\ f -> El (f g)) (fst (snd p))
     (lookup _ _ (snd (snd p)) (fst g))

------------------------------------------------------------------------
-- A language

-- Syntax for types.

data Type (G : Ctxt) (s : Ty G) : Set

-- Terms.

data Term (G : Ctxt) (s : Ty G) : Set

-- The semantics of a term.
-- eval gives the Agda term corresponding to a term in the language.
eval : {G : _} {s : _} -> Term G s -> (g : Env G) -> El (s g)

-- When defining the Agda types for language types and language terms,
-- we use equality proofs to make up for the lack of support for
-- indices in Tog  datatypes.
```

```
data Type G s where
  set''   : s == (\ _ -> set) -> Type G s
  -- Terms of type 'set' represent codes of types. To obtain the
  -- corresponding type, we apply el'' to the term.
  el''    : (x : Term G (\ _ -> set)) ->
            (\ g -> el (eval {G} {\_ -> set} x g)) == s ->
            Type G s
  sigma'' : {t : _} {u : _} ->
            Type G t ->
            Type (snoc G t) u ->
            (\ g -> sigma (t g) (\ v -> u (pair g v))) == s ->
            Type G s
  pi''    : {t : _} {u : _} ->
            Type G t ->
            Type (snoc G t) u ->
            (\ g -> pi (t g) (\ v -> u (pair g v))) == s ->
            Type G s

data Term G s where
  var   : Var G s -> Term G s
  lam'' : {t : _} {u : _} ->
          Term (snoc G t) (uncurry u) ->
          (\ g -> pi (t g) (\ v -> u g v)) == s ->
          Term G s
  app'' : {t : _} {u : (g : Env G) -> El (t g) -> U} ->
          Term G (\ g -> pi (t g) (\ v -> u g v)) ->
          (t2 : Term G t) ->
          (\ g -> u g (eval t2 g)) == s ->
          Term G s
  -- For conciseness, we do not include the constructors and
  -- eliminators for the Σ-type

-- The interpretation of a variable is obtained from the environment
eval (var x)          g = lookup _ _ x g
-- The interpretation of a λ-abstraction is an Agda function, where
-- the bound variable is added to the environment.
eval (lam'' t eq)     g = subst (\ f -> El (f g)) eq
                          (\ v -> eval t (pair g v))
-- The interpretation of an application is the Agda function
-- application of the evaluation of the terms.
eval (app'' t1 t2 eq) g = subst (\ f -> El (f g)) eq
                          (eval t1 g (eval t2 g))

-- Abbreviations.
-- These abbreviations make up for the lack of support for indices in
-- Tog  datatypes.
set' : {G : Ctxt} -> Type G (\ _ -> set)
set' = set'' refl
```

```
el' : {G : Ctxt}
      (x : Term G (\ _ -> set)) ->
      Type G (\ g -> el (eval {G} {\_ -> set} x g))
el' x = el'' x refl

sigma' : {G : Ctxt} {t : Env G -> U} {u : Env (snoc G t) -> U} ->
         Type G t ->
         Type (snoc G t) u ->
         Type G (\ g -> sigma (t g) (\ v -> u (pair g v)))
sigma' s t = sigma'' s t refl

pi' : {G : _} {t : _} {u : _} ->
      Type G t ->
      Type (snoc G t) u ->
      Type G (\ g -> pi (t g) (\ v -> u (pair g v)))
pi' s t = pi'' s t refl

lam : {G : _} {t : _} {u : _} ->
      Term (snoc G t) (uncurry u) ->
      Term G (\ g -> pi (t g) (\ v -> u g v))
lam t = lam'' t refl

app : {G : _} {t : _} {u : (g : Env G) -> El (t g) -> U} ->
      Term G (\ g -> pi (t g) (\ v -> u g v)) ->
      (t2 : Term G t) ->
      Term G (\ g -> u g (eval t2 g))
app t1 t2 = app'' t1 t2 refl
```

Appendix B

Code and output of system comparisons

Listing B.1: Minilang example in Idris:

```
module Minilang

-- Atoms
postulate F : Bool -> Type
postulate P : (X : Type) -> (x : X) -> Type

-- Shorthands --
U : Type
U = (Bool, Bool)

Set'  : U
Set'  = (True, True)

El' : Bool -> U
El' b = (False, b)

El : U -> Bool
El u = if (fst u) then True else (snd u)
-----------------------------------------------

postulate c : {alpha : Bool -> U} ->
      let ty1   : Type = (x : Bool) -> (F (El (alpha x))) -> U
          ty2   : Type =      Bool  -> (F  True         ) -> U in
      (P ty1 (\x , y => Set'), P ty2 (\x , y => alpha x))

postulate AreEqual : {A : Type} -> (A , A) -> Type

-- Constraints
c1 : Type
c1 = AreEqual c
```

Output:

```
Minilang.idr:30:6-15:
   |
30 | c1 = AreEqual c
   |      ~~~~~~~~~~
When checking right hand side of c1 with expected type
        Type

When checking an application of Minilang.AreEqual:
  Type mismatch between
    (P ((x : Bool) -> F (El (alpha x)) -> U) (\x8, y => Set'),
     P (Bool -> F True -> U) (\x10, y11 => alpha x10)) (Type of c)
  and
    (P (Bool -> F True -> (Bool, Bool)) (\x10, y11 => alpha x10),
     P (Bool -> F True -> (Bool, Bool))
            (\x10, y11 => alpha x10)) (Expected type)

  Specifically:
          Type mismatch between
            P ((x : Bool) ->
               F (if fst (alpha x) then True else snd (alpha x)) ->
               (Bool, Bool))
               (\x8, y => (True, True))
          and
            P (Bool -> F True -> (Bool, Bool))
               (\x10, y11 => alpha x10)

[ExitFailure: 1]
```

Listing B.2: Minilang example in Lean:

```
constant F : bool -> Type
constant B : Type
constant P : Π (X : Type), X -> Type

-- Shorthands
def U    : Type := bool × bool
def set' : U    := (tt, tt)
def el'  (b : bool) : U := (ff, b)

def El (u : U) : bool :=
   match u.1 with
     | tt  := tt
     | ff  := u.2
   end

-- (* Metavariables and constraints *)
def c1 :
      let alpha (x : bool) : U := _ in
```

```
      (P (Π (x : bool), (F (El (alpha x))) -> U) (λ x y, set')) ->
      (P (Π (x : bool), (F true)           -> U) (λ x y, alpha x)) :=
      λ x, x
```

Output:

```
Minilang.lean:18:34:
error: don't know how to synthesize placeholder
context:
x : bool
  U
[ExitFailure: 1]
```

Listing B.3: Minilang example in Matita:

```
include "basics/bool.ma".
include "basics/types.ma".

(* Atoms *)
axiom F : bool -> Type[0].
axiom B : Type[0].
axiom P : (Π (X : Type[0]). Π (x : X). Type[0]).

(* Shorthands *)
definition U                : Type[0]   bool × bool.
definition mkSet            : U          true  ,  true .
definition mkEl  (b : bool) : U          false ,  b .

definition El (u : U) : bool   match (\fst u) with
                                 [ true  => true
                                 | false => (\snd u)
                                 ].

(* Metavariables and constraints *)
definition c1 :
     let alpha : bool → U   λx.? in
     (P (Π (x : bool). (F (El (alpha x))) → U) (λ x. λ y. mkSet)) →
     (P (Π (x : bool). (F true)           → U) (λ x. λ y. alpha x))
     λ x. x.
```

Output:

```
********** DISAMBIGUATION ERRORS: **********
***** Errors obtained during phases 4: *****
*Error at 721-722: The term
x
has type
(P ( x:bool.F (El ?59[...])→U) (λx:bool.λy:F (El ?59[...]).mkSet))
but is here used with type
(P ( x:bool.F true→U) (λx:bool.λy:F true.?59[...]))
```

```
***** Errors obtained during phases 3: *****
*Error at 721-722: The term
x
has type
(P ( x:bool.F (El ?58[...])→U) (λx:bool.λy:F (El ?58[...]).mkSet))
but is here used with type
(P ( x:bool.F true→U) (λx:bool.λy:F true.?58[...]))

***** Errors obtained during phases 2: *****
*Error at 721-722: The term
x
has type
(P ( x:bool.F (El ?57[...])→U) (λx:bool.λy:F (El ?57[...]).mkSet))
but is here used with type
(P ( x:bool.F true→U) (λx:bool.λy:F true.?57[...]))

***** Errors obtained during phases 1: *****
*Error at 721-722: The term
x
has type
(P ( x:bool.F (El ?56[...])→U) (λx:bool.λy:F (El ?56[...]).mkSet))
but is here used with type
(P ( x:bool.F true→U) (λx:bool.λy:F true.?56[...]))


Info:  Compilation failed
[ExitFailure: 1]
```

Listing B.4: Minilang example in Tog:

```
module Minilang where

record Sigma (A : Set) (B : A -> Set) : Set
record Sigma A B where
  constructor pair
  field
    fst : A
    snd : B fst

data Bool : Set where
  true : Bool
  false : Bool

if : (A : Bool -> Set) -> (b : Bool) -> A true -> A false -> A b
if _ true  x y = x
if _ false x y = y

-- Atoms
postulate F : Bool -> Set
postulate P : (X : Set) -> (x : X) -> Set
```

```
-- Shorthands --
U : Set
U = Sigma Bool (\_ -> Bool)

set'  : U
set'  = pair true true

El : U -> Bool
El u = if (\_ -> Bool) (fst u) true (snd u)
-----------------------------------------------

alpha : Bool -> U
alpha x = _

-- Constraints
c1 : (P ((x : Bool) -> F (El (alpha x)) -> U) (\x -> \y -> set')) ->
     (P ((_ : Bool) -> F true           -> U) (\x -> \y -> alpha x))
c1 x = x
```

Output:

```
------------------------------------------------------------------------
-- Checking declarations
------------------------------------------------------------------------
-- Solved Metas: 193
-- Unsolved Metas: 3
------------------------------------------------------------------------
_184 (38:60) : Minilang.Bool -> Minilang.Bool

_185 (38:60) : Minilang.Bool -> Minilang.Bool

_196 (39:8) :
  Minilang.P
    ((x : Minilang.Bool) ->
     Minilang.F
       (Minilang.if (\_ -> Minilang.Bool) (_184 0#x)
          Minilang.true
          (_185 0#x)) ->
     Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
    (\_ _ -> Minilang.pair Minilang.true Minilang.true) ->
  Minilang.P
    (Minilang.Bool ->
     Minilang.F Minilang.true ->
     Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
    (\x _ -> Minilang.pair (_184 1#x) (_185 1#x))

------------------------------------------------------------------------
-- Unsolved problems: 1
------------------------------------------------------------------------
```

```
** Waiting on [128, 127] [[_184]]
      [ x : Minilang.P
        ((x : Minilang.Bool) ->
         Minilang.F
           (Minilang.if (\_ -> Minilang.Bool) (_184 0#x)
              Minilang.true
              (_185 0#x)) ->
         Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
        (\_ _ -> Minilang.pair Minilang.true Minilang.true)
      ; _ : Minilang.Bool
      ] |-
      Minilang.if (\_ -> Minilang.Bool) (_184 0#x)
        Minilang.true
        (_185 0#x) = Minilang.true : Minilang.Bool
    >>
      UnifySpine
      ctx:
        [ x : Minilang.P
          ((x : Minilang.Bool) ->
           Minilang.F
             (Minilang.if (\_ -> Minilang.Bool) (_184 0#x)
                Minilang.true
                (_185 0#x)) ->
           Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
          (\_ _ -> Minilang.pair Minilang.true Minilang.true)
        ; _ : Minilang.Bool
        ]
      type: (Minilang.F Minilang.true -> Set) -> Set
      h: no head
      elims1:
        [ $ \_ -> Minilang.Sigma Minilang.Bool
            (\_ -> Minilang.Bool)
        ]
      elims2:
        [ $ \_ -> Minilang.Sigma Minilang.Bool
            (\_ -> Minilang.Bool)
        ]
  >>
    UnifySpine
    ctx:
      [ x : Minilang.P
        ((x : Minilang.Bool) ->
         Minilang.F
           (Minilang.if (\_ -> Minilang.Bool) (_184 0#x)
              Minilang.true
              (_185 0#x)) ->
         Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
        (\_ _ -> Minilang.pair Minilang.true Minilang.true)
      ]
```

```
    type:
      (Minilang.Bool ->
       Minilang.F Minilang.true ->
       Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool)) ->
      Set
    h:
      Minilang.P
        (Minilang.Bool ->
         Minilang.F Minilang.true ->
         Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
    elims1:
      [ $ \x _ -> Minilang.pair (_184 1#x) (_185 1#x) ]
    elims2:
      [ $ \_ _ -> Minilang.pair Minilang.true Minilang.true ]
>>
  [ x : Minilang.P
    ((x : Minilang.Bool) ->
     Minilang.F
       (Minilang.if (\_ -> Minilang.Bool) (_184 0#x)
          Minilang.true
          (_185 0#x)) ->
     Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
    (\_ _ -> Minilang.pair Minilang.true Minilang.true)
  ] |-
  _196 0#x
  =
  0#x
  :
  Minilang.P
    (Minilang.Bool ->
     Minilang.F Minilang.true ->
     Minilang.Sigma Minilang.Bool (\_ -> Minilang.Bool))
    (\x _ -> Minilang.pair (_184 1#x) (_185 1#x))
------------------------------------------------------------------------
Type error:
Error at 0:0 :
  UnsolvedMetas [_184, _185, _196]
[ExitFailure: 1]
```

Index

List of Definitions, Notations and Problems

List of Postulates

List of Theorems, Propositions, Lemmas and Remarks

List of Examples

List of Algorithms

List of Figures

List of Listings

List of Tables

Bibliography

[1] Andreas Abel. Documented the conditions of projection-likeness. Agda Source Code Repository, January 2017. URL https://github.com/agda/agda/commit/3b02558118dd1fb4b00f82248c2d3c2e595281fe#diff-8abf7ddf426bbf6ef075fd206f630f2c .

[2] Andreas Abel. Type checker normalizes too much. Agda Issue #4125, October 2019. URL https://github.com/agda/agda/issues/4125 .

[3] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Typed Lambda Calculi and Applications (TLCA 2011)*. 2011. doi:10.1007/978-3-642-21691-6_5 .

[4] Andreas Abel, Francesco Mazzoli, and Ulf Norell. Strange metavariable behaviour when the first argument comparison is stuck. Agda Issue #1258, August 2014. URL https://github.com/agda/agda/issues/1258 .

[5] Andreas Abel, Jesper Cockx, and Nils Anders Danielsson. Agda allows "very dependent" types. Agda Issue #1556, August 2015. URL https://github.com/agda/agda/issues/1556 .

[6] Andreas Abel, Nils Anders Danielsson, and Ulf Norell. Inconsistent constraints leading to violated invariants in conversion checking. Agda Issue #1467, March 2015. URL https://github.com/agda/agda/issues/1467 .

[7] Andreas Abel, Nils Anders Danielsson, and Víctor López Juan. Overzealous pruning (reprise). Agda Issue #2876, December 2017. URL https://github.com/agda/agda/issues/2876 .

[8] Andreas Abel, Martin Stone Davis, Ulf Norell, et al. (No longer an) Internal error at src/full/Agda/TypeChecking/Substitute.hs:98. Agda Issue #2709, August 2017. URL https://github.com/agda/agda/issues/2709 .

[9] Andreas Abel, Jesper Cockx, Nils Anders Danielsson, and Víctor López Juan. Regression related to fix of #3027. Agda Issue #4408, January 2020. URL https://github.com/agda/agda/issues/4408 .

[10] Robin Adams. *A modular hierarchy of logical frameworks*. PhD thesis, Faculty of Engineering and Physical Sciences, University of Manchester, 2004. URL https://repository.royalholloway.ac.uk/items/2fa04c91-c933-8da6-3bc4-9d300b20cc54/10/ .

[11] Robin Adams. Lambda-free logical frameworks. *CoRR*, abs/0804.1879, 2008. URL http://arxiv.org/abs/0804.1879 .

[12] Jesper Cockx and "palkarz". Type checker explosion. Agda Issue #3554, February 2019. URL https://github.com/agda/agda/issues/3554 .

[13] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3): 285–326, 1984.

[14] Catarina Coquand. The homepage of the Agda type checker, 1998. URL https://web.archive.org/web/20070909205649/http://www.cs.chalmers.se/~catarina/agda/ .

[15] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. Technical Report RR-0401, INRIA, May 1985. URL https://hal.inria.fr/inria-00076155 .

[16] Nils Anders Danielsson and Ulf Norell. Inconsistent constraints leading to violated invariants in conversion checking. Agda Issue #1467, March 2014. URL https://github.com/agda/agda/issues/1467 .

[17] L. Peter Deutsch. *An interactive program verifier*. Xerox, Palo Alto Research Center, 1973. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.696.5498 .

[18] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, and Christine Paulin-Mohring. The Coq proof assistant user's guide : version 5.6. Research Report RT-0134, INRIA, 1991. URL https://hal.inria.fr/inria-00070034 .

[19] Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206(1-2):1–50, 1998. doi:10.1016/S0304-3975(97)00141-2 .

[20] Conal M Elliott. Higher-order unification with dependent function types. In *International Conference on Rewriting Techniques and Applications*, pages 121–136. Springer, 1989. doi:10.1007/3-540-51081-8_104 .

[21] Conal M Elliott. *Extensions and applications of higher-order unification*. PhD thesis, Carnegie Mellon University, 1990. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.8369 .

[22] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. URL https://web.archive.org/web/20190604052524/https://www.cs.cmu.edu/~kw/scans/girard72thesis.pdf .

[23] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF – A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979. ISBN 978-3-540-09724-2. doi:10.1007/3-540-09724-4 .

[24] James R Guard. Automated logic for semi-automated mathematics. Technical report, Applied Logic Corp. Princeton, NJ, 1964. URL https://web.archive.org/web/20200509053351/https://apps.dtic.mil/dtic/tr/fulltext/u2/602710.pdf .

[25] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, Department of Computer and Information Sciences, University of Strathclyde, 2013. URL http://adam.gundry.co.uk/pub/thesis/ .

[26] Adam Gundry. Unification and type inference algorithms. Source code repository, February 2015. URL https://github.com/adamgundry/type-inference . Commit 4cee7626.

[27] Adam Gundry and Conor McBride. A tutorial implementation of dynamic pattern unification. Unpublished, 2012. URL http://adam.gundry.co.uk/pub/pattern-unify/ .

[28] Gérard P Huet. The undecidability of unification in third order logic. *Information and control*, 22(3):257–267, 1973. doi:10.1016/0304-3975(81)90040-2 .

[29] Gerard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975. doi:10.1016/0304-3975(75)90011-0 .

[30] Antonius JC Hurkens. A simplification of Girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995. doi:10.1007/BFb0014058 .

[31] C. Maria Keet. Open world assumption. In *Encyclopedia of Systems Biology*, pages 1567–1567. 2013. ISBN 978-1-4419-9863-7. doi:10.1007/978-1-4419-9863-7_734 .

[32] Neel Krishnaswami. Answer to "hereditary substitution with a universe hierarchy". Theoretical Computer Science Stack Exchange. URL https://cstheory.stackexchange.com/questions/41924/hereditary-substitution-with-a-universe-hierarchy/41928 .

[33] Víctor López Juan and Ulf Norell. Internal error in the presence of unsatisfiable constraints. Agda Issue #3027, April 2018. URL https://github.com/agda/agda/issues/3027 .

[34] Lena Magnusson. *The Implementation of ALF – a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Department of Computer Science, Chalmers University of Technology, 1994.

[35] Francesco Mazzoli. Data. Tog Source Repository, November 2014. URL https://github.com/bitonic/tog/blob/7047c561557328952dbbbe7a6944c470f10ac192/examples/slow/Data.agda .

[36] Francesco Mazzoli and Andreas Abel. Type checking through unification, 2016. arXiv:1609.09709v1 .

[37] Francesco Mazzoli, Nils Anders Danielsson, Ulf Norell, Andrea Vezzosi, Andreas Abel, et al. Tog - a prototypical implementation of dependent types, 2017. URL https://github.com/bitonic/tog .

[38] Conor McBride. Epigram, 2007. URL http://www.e-pig.org .

[39] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP'10, 2010. doi:10.1145/1863495.1863497 .

[40] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, 1992. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.2557 .

[41] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991. doi:10.1093/logcom/1.4.497 .

[42] César Muñoz. A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory. Research Report RR-3309, INRIA, 1997. URL https://hal.inria.fr/inria-00073380 .

[43] César Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theoretical Computer Science*, 266(1-2):407–440, 2001. doi:10.1016/S0304-3975(00)00196-1 .

[44] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2007. URL http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf .

[45] Ulf Norell. Detection of projection like functions. Agda Source Repository, September 2011. URL https://github.com/agda/agda/commit/3ce53cc6f4aec36aa0fb5f6697f49ec4ff747ecd .

[46] Ulf Norell. Auto-inline simple definitions. Agda Source Repository, April 2018. URL https://github.com/agda/agda/commit/2a3495289d5ee089abc9a7c041c5b05dbc472f37 .

[47] Ulf Norell. Internal error during instance search. Agda Issue #3870, June 2019. URL https://github.com/agda/agda/issues/3870 .

[48] Ulf Norell and Catarina Coquand. Type checking in the presence of meta-variables. Unpublished, 2007. URL http://www.cse.chalmers.se/~ulfn/papers/meta-variables.html .

[49] Kent Petersson. A programming system for type theory. Technical Report 9, Programming Methodology Group, University of Gothenburg and Chalmers University of Technology, 1984. ISSN 0282-2083.

[50] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946. doi:10.1090/S0002-9904-1946-08555-9 .

[51] David J. Pym. *Proofs, search and computation in general logic*. PhD thesis, 1990. URL `http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-125/` .

[52] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 49–56, 2009. doi:10.1145/1577824.1577832 .

[53] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965. doi:10.1145/321250.321253 .

[54] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ICFP '98*, 1998. doi:10.1145/289423.289460 .

[55] Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *ACM SIGPLAN Notices*, volume 50, pages 179–191. ACM, 2015. doi:10.1145/2784731.2784751 .

[56] Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *Journal of Functional Programming*, 27, 2017. doi:10.1017/S0956796817000028 .