Thesis for the Degree of Doctor of Philosophy

Practical Heterogeneous Unification for Dependent Type Checking

Víctor López Juan

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden 2021

Practical Heterogeneous Unification for Dependent Type Checking
Víctor López Juan

ISBN 978-91-7905-583-7
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5050
ISSN 0346-718X
Technical report 207D

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Tel. +46 (0)31-772 1000
Fax +46 (0)31-772 3663

This thesis has been prepared using LaTeX
Printed by Chalmers Print Service
Gothenburg, Sweden 2021

Practical Heterogeneous Unification for Dependent Type Checking

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

Abstract:
Dependent types can specify in detail which inputs to a program are allowed, and how the properties of its output depend on the inputs. A program called the type checker assesses whether a program has a given type, thus detecting situations where the implementation of a program potentially differs from its intended behaviour. When using dependent types, the inputs to a program often occur in the types of other inputs or in the type of the output. The user may omit some of these redundant inputs when calling the program, expecting the type checker to infer those subterms automatically.

Some type checkers restrict the inference of missing subterms to those cases where there is a provably unique solution. This makes the process more predictable, but also limits the situations in which the omitted terms can be inferred; specially when considering that whether a unique solution exists is in general an undecidable problem. This restriction can be made less limiting by giving flexibility to the type checker regarding the order in which the missing subterms are inferred. The type checker can then use the information gained by filling in any one subterm in order to infer others, until the whole program has been type-checked. However, this flexibility may in some cases lead to ill-typed subterms being inferred, breaking internal invariants of the type checker and causing it to crash or loop. The type checker could mitigate this by consistently rechecking the type of each inferred subterm, but this might incur a performance penalty.

An approach by Gundry and McBride (2012) called twin types has the potential to afford the desired flexibility while preserving well-typedness invariants. However, this method had not yet been tested in a practical setting. In this thesis we streamline the method of twin types in order to ease its practical implementation, justify the correctness of our modifications, and then implement the result in an established dependently-typed language called Agda. We show that our implementation resolves certain existing bugs in Agda while still allowing a wide range of examples to be type-checked, and achieves this without heavily impacting performance.

Keywords: dependent types, type checking, unification.

This thesis expands on the work contained in following publications:

Víctor López Juan
A Practical Implementation of Twin Types
27th International Conference on Types for Proofs and Programs (TYPES 2021)
Extended abstract
doi:10.5281/zenodo.5575799

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

Víctor López Juan
Practical Unification for Dependent Type Checking
Chalmers University of Technology
Licentiate thesis (2020)
research.chalmers.se/publication/519011

Acknowledgements

The road to defending this thesis has had many up and downs. I wish to thank my supervisor, Nils Anders Danielsson, for supporting me throughout this journey and relentlessly challenging me to improve the quality of my work. A special thank you goes to Peter Dybjer and Ana Bove for their understanding during the hardest moments. I am grateful to my co-supervisor Andreas Abel and my examiner Thierry Coquand for helping me understand the context in which this work takes place, and directing my attention to problems whose study was fruitful. I am thankful to them and other members of the Agda community, including Ulf Norell, Andrea Vezzosi and Jesper Cockx, for their help in understanding the practical challenges of implementing dependent type checking with metavariables.

I thank the opponent, Claudio Sacerdoti Coen, and the committee members Edwin Brady, Moa Johansson, Conor McBride, and Magnus Myréen as the substitute, both for accepting this task, and for their encouraging and insightful comments which have helped polish the final manuscript. I extend my thanks to the committee members and other researchers who have given me feedback on previous publications and talks, and to those who have made the many scientific contributions upon which this work builds.

These years in Sweden would not have been as enriching without the classmates, colleagues and friends I have met along the way, nor without the friends and family that stayed in touch despite the distance. I want to extend my gratitude to Carlos, Fabian, Baldur, Pedro, Daniel, Irene, Inari, Simon, Herbert, Stavros, Manos, Stefan, Franz, Salvo, Carlos, Borja, Felipe, Ralph, Carol and the many of you whom I have forgotten to mention. Thank you for offering both an understanding ear and honest criticism; for cheering me on with a gleeful song or giving me a stern reality-check; for sharing with me your intellectual curiosity and your joie de vivre.

Henrik, tack för att stå ut med mycket mer än du hade anmält dig till. Och tack till din familj, för att få ett främmande land att kännas som hemma.

Y gracias a mis padres, por recibirme siempre en casa como si nunca me hubiera marchado.

Contents

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 §3.4). However, for many programs and proofs that arise in practice (e.g. those where some of the resulting constraints are in Miller's pattern fragment [68]), unique solutions can be found. In this work, we describe rules for performing such an inference, with a focus on its practical implementability.

As demonstrated by Mazzoli and Abel [60], an entire type-checking problem involving metavariables can be reduced to a set of dependently-typed, higher-order unification constraints of the form $\Gamma \vdash t : A \approx u : B$, where $\Gamma \vdash t : A$ (t has type A in context Γ) and $\Gamma \vdash u : B$ (u has type B in context Γ). 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 [60] only refine constraints 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 (for instance, see Example 3.20).

Gundry and McBride's twin types approach [44] can handle constraints where the types of the two sides are distinct, by allowing each variable in the context to take up to two different types. The original presentation of twin types requires annotating all references to variables in the context to indicate which of the two types they take. In my licentiate thesis [54] we implemented an approach where the left (or right) side of the constraint only refers to the left (respectively, right) type of the variables in the context. This means that the underlying term syntax and type theory remain essentially intact. This approach is described in Chapter 4.

By defining an approach to unification without additional elements in the term syntax, we can rely on common assumptions about dependently-typed terms (§2.15) when discussing 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.

In my licentiate thesis [54], we demonstrated the feasibility of our modified approach by implementing it in an existing prototype [61] and used it to type-check some examples. In this thesis we implement and evaluate the approach (with suitable extensions) in the Agda type checker [7]. Implementing our approach in Agda fixes a range of long-standing bugs, either outright or by removing current workarounds. We have tested our implementation on three large Agda projects, and obtained comparable CPU and memory usage while being able to infer almost all the implicit arguments that were inferred before.

1.1 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 solution where all the metavariables are instantiated.

The case for uniqueness requires more explanation. In our setting, many metavariables will occur in definitions, whether they are function bodies or theorem statements. Avoiding non-unique solutions limits the potential for ambiguity in what the defined 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 `repeat` produces a vector in which all the components have the same value. The first argument to `repeat`, which is implicit, determines the length of the resulting vector. Unless given explicitly, this argument is inferred from the context where the result is used. The result of the first usage of `repeat` is passed as an argument to `rotate90`. Therefore, the resulting vector must have type `Vec 2 Int`, and thus the omitted argument must be `2`. In the second usage of `repeat` (line 23), 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, both for usability and for ease of implementation.

From a usability perspective, avoiding non-unique solutions means that if the user has a specific solution for an implicit argument in mind, and the program type-checks, then he can be sure that his solution was also the solution that the type checker chose. The user does not need to have any up-to-date knowledge of the internal details of the type checker's algorithm.

Listing 1.1: Non-unique implicit argument

```
-- Type of natural numbers (0 : Nat, 1 : Nat, 2 : Nat, ...)
Nat : Type
-- Type of integers (..., -2 : Int, -1 : Int, 0 : Int, 1 : Int, ...)
Int : Type
-- Type of lists of fixed length ([-1,3,2] : Vec 3 Int, ...)
Vec : (n : Nat) → Type → Type

-- repeat {n = 5} 4 ≡ [4,4,4,4,4]
-- repeat {n = 0} 4 ≡ []
-- (repeat 4 : Vec 3 Int) ≡ [4,4,4]    (n is given implicitly)
repeat : {n : Nat} → Int → Vec n Int

-- rotate90 [1,2] ≡ [-2,1]
rotate90 : Vec 2 Int → Vec 2 Int

-- print [1,2,3]
-- > [1,2,3]
print : {n : Nat} → Vec n Int → IO ()

main : IO ()
main = do
  print (rotate90 (repeat 1))    -- > [-1,1]
  print (repeat 6)               -- (?)
```

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 unaffected by 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 pointed out by Andreas Abel (personal communication), making all instantiations final is also helpful in an interactive setting, where reverting instantiations of metavariables which have already been output might be confusing to the user.

Coq allows non-unique solutions in certain cases. We believe that this may be well-suited in the case of Coq, as a common development approach tends more towards programs with relatively simple types together with proofs that these programs fulfill the desired properties. The proof terms themselves are usually generated by means of tactics, and the user is more interested in their existence than in their specific form. Furthermore, as argued by Ziliani and Sozeau [104, §2], in many cases one of the non-unique solutions is intuitively "better", for instance because it follows directly from first-order unification. Allowing non-unique solutions thus results in more implicit arguments being solved and a less frustrating experience for the user.

Note that for the simplified example in Listing 1.1, Coq will actually ask the user to give the implicit argument explicitly, as there is no solution which is clearly better than the others. In §6.2 we discuss a more realistic example where Coq does produce a non-unique solution.

1.2 Higher-order unification for dependent type checking

Inference of implicit arguments in proof assistants such as Coq, Lean, Idris or Agda is done by replacing the implicit arguments with placeholders (i.e. metavariables), producing a series of unification constraints. A unification constraint consists of at least a pair of terms t and u in a typing context Γ ($\Gamma \vdash t \approx u$). A constraint is solved by assigning terms to the metavariables (that is, instantiating them) so that both sides of the constraint become equal ($\Gamma \vdash t \equiv u$). The unification is called higher-order when metavariables may occur in the head of a term, which means that redexes can be created when a metavariable is instantiated with a solution.

In 1975, Huet [48] 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. In 1973, Huet [47] had proved that higher-order unification is undecidable. This means that there cannot exist an algorithm which always finds a solution when there is one, and also always terminates in the absence of a solution.

With Huet's work [48] as a starting point, in 1989, Elliott [34] described an algorithm for the λ_Π calculus which only constructs *approximately well-typed* terms, although with the property that they do at least have η-long, β-head-normal forms. Pym [87] arrived 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, as the problem of higher-order unification is undecidable, the algorithm may not terminate.

Later on, in 1990, Elliott [35] extended his previous work into a pre-unification algorithm for calculi with both 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 [58] implemented ALF, a precursor to Agda. Until then, implementations of type theory (among them NuPRL [17], Petersson's system [83] and Coq [20, 30]) would have the user build proof-trees using tactics until eventually a complete term is produced [70, section 1.4], in the style of LCF [41]. The complete term (which may or may not have been 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. Type-checking of the incomplete terms in ALF was done by using a simplified version of the ideas of Elliott [34] and Pym [87]. The type checker would produce a list of constraints and the user would refine the metavariables until all constraints are satisfied. Once all constraints are satisfied, 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 all 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 [58, section 8.4.1] that all terms involved in the execution of the unification algorithm can be typed in the simply-typed λ-calculus. All the terms involved are 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, Muñoz [70, 71] put Magnusson's [58] approach on a formal footing, in such a way that all intermediate terms are well-typed. The focus of Muñoz's approach is on proof search. As in the work of Huet [48], Elliott [34], and Pym [87], the focus is on the completeness of the algorithm; it is still possible that the algorithm may not terminate when a solution does not exist.

When using higher-order unification to build a type checker for dependent types with implicit arguments, an algorithm that does not find all existing solutions but is instead guaranteed to always give an answer in finite time (even if it is more often a negative one) may provide for a more predictable user experience. In 1991, Miller [68] 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.

In 1998, Catarina Coquand [18] built a dependent type checker inspired by ALF's interactive editing, called Agda. Agda implements structured type-theory [19], in turn based on Martin-Löf type-theory [59]. In 2004, McBride and McKinna [66] began to further develop the ideas of interactive dependently-typed programming from Agda and ALF. These efforts culminated in the release of the Epigram [63] system. Agda was in turn further developed by Norell and Coquand [79, 73], who among many improvements introduced a metavariable solving mechanism inspired by the one in Epigram. The metavariable solving mechanism introduced by Norell used a restricted form of Miller's pattern unification. The result of Norell's work became the first release of Agda 2.

Metavariable solving involves checking the equality of terms. As Norell and Coquand explain [79], normalizing terms in order to check for equality may make the type checker loop if those terms are not well-typed, but 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. This technique, implemented in Agda 2, improves on the previous version of Agda by preventing the generation of ill-typed terms in certain cases. According to Norell and Coquand [79], the improvement with respect to Muñoz's [71] 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 implementing metavariable solving, 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 [88] presented a terminating algorithm for dynamic pattern unification, and Abel and Pientka [2] extended the dynamic pattern unification technique to handle Σ-types with 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 is the approach currently used for solving metavariables in Agda 2, replacing the simplified version of pattern unification that was used in Norell's original implementation.

1.3 The binder problem

A common problem in dependently-typed unification arises when unifying binders, such as dependent product types. Consider a constraint of the form $\Gamma \vdash (x : A) \to B \approx (x : A') \to B'$. This constraint may be solved by separately unifying the domains (A and A') and the codomains (B and B'). However, the types B and B' live in different contexts ($\Gamma, x : A \vdash B$ **type** and $\Gamma, x : A' \vdash B'$ **type**). It is not self-evident what the type of x should be in $\Gamma, x : ? \vdash B \approx B'$ **type**, and different approaches exist.

Sequential constraints

One solution to the binder problem is to enforce a strict ordering in which the constraints $\Gamma \vdash A \approx A'$ and $\Gamma, x : ? \vdash B \approx B'$ are solved. This is the approach taken by Coq [101] in the examples we have tested. By only unifying the domain once the codomain has been unified, one can take $x : A$ in the second constraint, and the binder problem is sidestepped. The Lean proof assistant seems to behave similarly to Coq. Constraints are also strictly ordered in the approach used by Mazzoli and Abel [60] for solving the constraints resulting from their encoding of dependent type checking into higher-order unification constraints.

Ziliani and Sozeau [105] argue that in the context of Coq, constraint postponement is not crucial, and may even worsen the performance of the algorithm and make it harder to debug. In our perception, a common development methodology in Coq tends to enforce the properties of a function by proving separate lemmas; rather than by encoding these properties in the type of the function by means of dependent product types. This means that the preconditions for the binder problem to arise occur less frequently than when using more "type-driven" approaches, such as the one encouraged by Idris [95]. Furthermore, as discussed in §1.1, Coq will sometimes infer non-unique solutions for implicit arguments. This makes more unification problems solvable, but also means that arbitrary reordering of constraints could lead to unpredictability in which of the non-unique solutions is chosen. The benefits of constraint reordering are thus not unambiguously clear in the case of Coq.

However, when type-checking programs with more complex types, and especially when restricting the unifier to unique solutions and no backtracking,

being able to reorder constraints freely is advantageous (see Example 3.20).

Blocked constants

Agda takes what amounts to a well-typed-modulo approach, where constraints are well-typed provided that some other constraints are solved. The existence of ill-typed terms may however cause issues, as described by Norell and Coquand [73, 79]. To mitigate them, the type of the variable is replaced by a blocked constant. That is, $\Gamma, x : p \vdash B \approx B'$, where p reduces to A ($p \rightsquigarrow A$) when $\Gamma \vdash A \equiv A'$. The resulting terms are still potentially not well-typed, which in some known cases causes the type checker to crash [80].

On the other hand, in Idris 2, the binder problem is solved by taking $x : A$, and replacing all occurrences of x in B' by a blocked constant p of type A'; i.e. $\Gamma, x : A \vdash B \approx B'[p/x]$ **type**, where $p \rightsquigarrow x$ when $\Gamma \vdash A \equiv A'$. As opposed to replacing the type of the variable, as done in Agda, blocking the variable does produce well-typed terms.

We tested the same approach as in Idris 2 in an older version of Agda without causing any breakage in the standard test suite or the standard library. However, this straightforward approach was shown to not be powerful enough to support one case of existing Agda code [5]. It is also not clear how this approach would address the (similar) spine problem.

1.4 The spine problem

Higher-order unification in Agda is type-directed, that is, unification constraints include the type of terms ($\Gamma \vdash t \approx u : A$), and they are used to guide the unification algorithm. In particular, this is relied upon for implementing η-equality for records (including the unit type), omitting the type of the bound variable in λ-expressions, and for advanced features supported by Agda such as cubical type theory [102].

In the context of type-directed unification there is an additional issue besides the binder problem, which we have dubbed the spine problem. This problem arises when unifying the elimination spines of two terms in head-normal form. Consider an irreducible constant (for instance, a data constructor) $c : (x : A) \to (y : B) \to C$. A constraint $\Gamma \vdash c\, t\, u \approx c\, t'\, u' : T$ is solved by unifying $\Gamma \vdash t \approx t' : A$, and then $\Gamma \vdash u \approx u' : B[?/x]$. However, it is not clear what the shared type of u and u' should be.

In the case of Agda, a common type is obtained by substituting a blocked constant p for x in B: $\Gamma \vdash u \approx u' : B[p/x]$ where $p \rightsquigarrow t$ when $\Gamma \vdash t \equiv t' : A$. This approach is not a complete solution. First, the constraint may still be ill-typed, as it is not necessarily the case that either $\Gamma \vdash u : B[p/x]$ or $\Gamma \vdash u' : B[p/x]$. Furthermore, the fact that p is blocked may hinder further reduction of $B[p/x]$. To mitigate the latter, the blocked constant p may be refined into a partially blocked constant if there are commonalities between t and t', in a process called anti-unification. It is unclear whether such a procedure is sound.

Unification in other proof assistants such as Coq or Idris 2 is not type-directed, so they do not need to compute a common type for u and u'. However, the unification rules may implicitly rely on the existence of such a common type. Coq ensures that u and u' have a common type by unifying t and

t' first. This is analogous to how the binder problem is addressed by solving the resulting constraints in a specific order. Idris 2 does not ensure that such a common type exists, which under certain circumstances may lead the type checker to attempt to normalize ill-typed terms (§6.4).

1.5 The twin type approach

Gundry and McBride [45, 44] propose assigning two types to each variable in the context, and annotating each occurrence of the variable in a term to distinguish which of the two types applies (§3.10). This addresses both the binder problem (§1.3) and the spine problem (§1.4).

As part of my licentiate thesis [53, 54], a streamlined variant of Gundry and McBride's approach was implemented in an existing prototype [61]. The prototype was then used to type-check some specific examples, which showed that the streamlined approach was sufficient to solve many of the unification problems that arise during dependent type checking. This thesis subsumes the work in the licentiate thesis and extends it by demonstrating that the approach can scale to an established proof assistant and a larger body of examples.

1.6 Design choices

We build on the existing unification algorithms by Abel and Pientka [2], Gundry and McBride [44] and Mazzoli and Abel [60]. Together with some modifications of our own, we obtain a type checker which can handle a range of examples. In this section we explain the design choices that have guided our work. These choices constitute only one of many possible, valid approaches to higher-order unification for dependent type checking.

Assuming Type : Type

We first note that the type theory which we use includes Type : Type as an axiom (§2.11, "SET"). We make this choice under the premise that the specification of a universe hierarchy is largely orthogonal to the treatment of unification. However, in order to justify the suitability of our approach, we need to use properties (e.g. that all well-typed terms have a normal form) that may not hold in our unstratified theory (§2.12). Our reasoning relies on these assumptions, but we try our best to use them in such a way that the inherent contradictions arising from them are not exploited. The reader should inspect the proofs carefully and convince themselves that the results would hold in a properly stratified version theory.

Dependent type theory with Bool

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 (e.g. whether it is a Π-type or a Σ-type) is not determined until a given metavariable is instantiated (§4.6).

Term representation

The syntactic representation of terms affects the performance of term normalization and other aspects of unification. Agda represents variables in terms using de Bruijn indices [27], which are numbers that indicate the number of binders between the occurrence of the variable and the binder it refers to. De Bruijn indices are also used in the core languages of Idris 2 [16] and Coq [100]. The theoretical development in this thesis also uses de Bruijn indices in order to stay close to the implementation. We sometimes denote them with names for the sake of readability (§2.7).

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 [44], we allow unexpanded definitions in terms. Note that the free variables of a term play a relevant role in the applicability of certain unification rules (e.g. Rule-Schema 2). Having unnormalized terms in the theory allows us to formally discuss the correctness of that rule even in the presence of such terms.

Closed metavariables

The types and values of our metavariables all live in the empty context. The permitted dependencies of a metavariable are modelled by giving the metavariable a (dependent) function type. As Ziliani and Sozeau [105] 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 may lead to unnecessary β-reductions when a solution is substituted for a metavariable. In Agda [7] this is mitigated by removing the leading λ-abstractions from the bodies of the metavariables.

A second issue with closed metavariables, which was observed by Ziliani and Sozeau [105], is that when replacing their occurrences in terms by their bodies, many unsightly lambdas will occur unless the resulting term is β-normalized. This is not a concern for us, as we only consider terms in β-normal form.

Solutions as closed metasubstitutions

In our correctness proofs we consider only those solutions in which all metavariables are instantiated. We follow Abel and Pientka's approach [2] and use grounding metasubstitutions, in which each metavariable is assigned a closed term.

For the purposes of determining whether a program is type-correct, solutions to all metavariables must be found, in which case a unique closed solution will also be a most general unifier. However, defining uniqueness of solutions in terms of most-general unifiers additionally implies the open-world assumption (§4.8.1), which in particular means 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.

Implementation into Agda

Ultimately implementing our approach into an established language such as Agda was a guiding aspiration in our development. We have implemented our approach into a branch of the Agda 2 type-checker. In order to support existing Agda code, our Agda implementation preserves many features not included in our theoretical development, including inductive-recursive data types. For these features, also including sized types [98], universe levels [99] and singleton types with η-equality (§4.9.1, §5.11), we have only adapted the original implementation to avoid immediate breakage, but without assessing whether the correctness guarantees translate to these features. We believe that the approach could be generalized to support Cubical Agda [97], but we have not verified this.

1.7 Our contributions

- A high-level description of rules for higher-order unification (§4.5) in a dependent type theory with uninterpreted constants, metavariables, dependent products, dependent sums, and a boolean type, previously published as part of my licentiate thesis [54]. These rules implement the ideas in Gundry and McBride's twin-type approach [44] without requiring changes to the underlying type theory. From the equality of the underlying type theory we derive a heterogeneous notion of definitional equality (§4.2) and a heterogeneous notion of context equality (§4.5.2). We use these notions to justify that, in a properly stratified version of the theory in which certain assumptions would hold, our versions of the unification rules could be applied without generating ill-typed terms or producing non-unique solutions.

- An implementation of the unification rules in the Agda programming language, demonstrating the feasibility of implementing the approach in an established type checker, and ways in which the soundness checks required by our rules may be optimized (§5.7).

- Benchmarks of the implementation against examples produced by users of Agda, demonstrating that the approach is flexible enough to deal with a large amount of existing code and incurs moderate computational overhead compared to the baseline Agda implementation (§6.4).

1.8 Structure of the thesis

In Chapter 2 we describe the dependently-typed language used in this thesis. We give the assumptions about this language that are needed for the justifications of correctness in subsequent chapters to be applicable.

In Chapter 3, we describe the role of unification in dependent type checking, with some additional details about the existing approaches to that problem and their shortcomings.

In Chapter 4, we describe a system of unification rules for the constructs described in Chapter 2, with the aim of addressing the shortcomings in the approaches described in Chapter 3.

In Chapter 5, we describe how the ideas from Chapter 4 can be implemented in an established dependently-typed language while preserving existing functionality.

In Chapter 6, we use the implementation described in Chapter 5 to assess the practicality of the approach. We evaluate both its ability to solve existing issues, and the ability to preserve existing functionality and performance. We test the implementation on a large amount of existing code, including the Agda standard library and the Agda test suite.

Chapter 2

A dependently-typed language

We work with a dependent type theory with dependent function and sum types, η-equality, and large elimination. The theory presented below contains the same constructs as Mazzoli and Abel's theory [60], with the addition Σ-types and their corresponding equality rules. These constructs allow us to address the issues with constraint solving and unification that we describe in Chapter 3.

Our theory is similar to the one used by Gundry and McBride [44], 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 follow Agda in restricting the allowed terms to those in β-normal form. Variables are represented by de Bruijn indices. The terms are not explicitly scoped, which has implications when defining renamings (§2.8). 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). Atoms are distinct from metavariables in that they cannot and do not need to be instantiated when solving constraints, and are distinct from variables in that they may occur even in closed terms. We leverage these characteristics in Chapter 3 and Chapter 4 to illustrate certain relevant unification problems.

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.

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

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.
- $\overrightarrow{\square}^n$ 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, §2.3 defines signatures, which we consider as a vector-like object. If $\Sigma = \alpha : \mathrm{Bool}, \mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}$, 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

As discussed in the introduction (§1.6), a metavariable is not associated with a context which determines the allowed free variables of its eventual body t and their types. Instead, the theory is presented in such a way that these dependencies may be modelled by giving the metavariable a (dependent) function type.

Definition 2.1 (Fresh declaration)**.** We say that $\mathbb{a}$ is fresh for Σ (or, that α is fresh for 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 appropriate type B), a well-typed term of the form $\mathbb{A}\,\alpha$ expands to the term $\mathbb{A}\,(\lambda x.\lambda y.y)$. In §2.14 we give a full account of computation in well-typed 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.

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

$$\frac{\Sigma\ \mathbf{sig} \qquad \mathbb{c} \text{ is fresh for } \Sigma \qquad \Sigma;\cdot \vdash A\ \mathbf{type}}{\Sigma, \mathbb{c} : A\ \mathbf{sig}}\ \textsc{atom-decl}$$

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

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

The typing relations $\Sigma;\cdot \vdash A\ \mathbf{type}$, and $\Sigma;\cdot \vdash t : A$ are defined in §2.5.

Remark 2.5 (Signature inversion). Let $\Sigma = \Sigma_1, \Sigma_2$, with $\Sigma\ \mathbf{sig}$. Then $\Sigma_1\ \mathbf{sig}$, and:

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

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

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.

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

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.

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

Figure 2.2: Metavariables occurring in a term.

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

Figure 2.3: Atoms occurring in a term.

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.

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

2.4 Contexts ($\Sigma \vdash \Gamma$ **ctx**)

A context can be used to define the types of the free variables of a term. In our setting, variables are numerical indices, so a context can be represented as 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 ($\Sigma; \Gamma \vdash A$ **type**) is explained in §2.5.

Remark 2.13 (Context inversion). Let $\Gamma = \Gamma_1, \Gamma_2$. If $\Sigma \vdash \Gamma_1, \Gamma_2$ **ctx**, then $\Sigma \vdash \Gamma_1$ **ctx**. Also, if $\Sigma \vdash \Gamma, A$ **ctx**, then $\Sigma \vdash \Gamma$ **ctx** and $\Sigma; \Gamma \vdash A$ **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: $\Gamma, x : A$). We do not include variable names in the underlying representation of a context. A variable name in a context indicates the de Bruijn index to which a later-occurring variable name refers. For instance, "$\cdot \vdash x : \mathbb{A}, y : \mathbb{B}\, x, z : \mathbb{C}\, x\, y$ **ctx**" denotes the judgment "$\cdot \vdash \mathbb{A}, \mathbb{B}\, 0, \mathbb{C}\, 1\, 0$ **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 ($\Sigma; \Gamma \vdash A$ **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.

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

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

What it means for a term A to be of type Set ($\Gamma \vdash A : \text{Set}$) is defined in §2.11. Correspondingly, what it means for two terms of type Set to be equal ($\Gamma \vdash A \equiv B : \text{Set}$) is defined in §2.13.

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 $\Sigma; \Gamma \vdash A : \text{Set}$ and $\Sigma; \Gamma \vdash A \ \textbf{type}$ are equivalent for any Σ, Γ and A; as are the judgments $\Sigma; \Gamma \vdash A \equiv B : \text{Set}$ and $\Sigma; \Gamma \vdash A \equiv B \ \textbf{type}$.

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. $\text{Set}_0, \text{Set}_1, \text{Set}_2, ...$). In order to facilitate a stratification effort, we keep distinct judgments for "A is a term of type Set" ($\Sigma; \Gamma \vdash A : \text{Set}$) and "$A$ is a type" ($\Sigma; \Gamma \vdash A \ \textbf{type}$).

2.6 Context equality ($\Sigma \vdash \Gamma \equiv \Gamma' \ \textbf{ctx}$)

Equality of types extends pointwise to whole contexts. The type equality and context equality judgments will play a role when defining whether a given constraint is solved (§4.1, Definition 4.9).

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

$$\frac{}{\Sigma \vdash \cdot \equiv \cdot} \ \text{CTX-EMPTY-EQ}$$

$$\frac{\Sigma \vdash \Gamma_1 \equiv \Gamma_2 \ \textbf{ctx} \qquad \Sigma; \Gamma_1 \vdash A_1 \equiv A_2 \ \textbf{type}}{\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2 \ \textbf{ctx}} \ \text{CTX-VAR-EQ}$$

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

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

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,

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

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

Figure 2.4: Free variables in a term.

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 $x^{(+1)} = 1 + |\Gamma_2|$.

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 we use $\Pi\overrightarrow{(x : A)}^n B$ to denote the term $\Pi(x_1 : A_1)\Pi(x_2 : A_2)...\Pi(x_n : A_n)B$. When $n = 0$, it is interpreted as an absence of binders: $\lambda\vec{x}^0.t \stackrel{\text{def}}{\equiv} t$ and $\Pi\overrightarrow{(x : A)}^0 B \stackrel{\text{def}}{\equiv} B$.

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 $x \in \text{FV}(t)$, x refers to the de Bruijn index of variable x in the context in which t is typed. The expressions $\text{FV}(t) \subseteq \{\vec{x}\}$ 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: $(+\text{n})$)**.** For $n \in \mathbb{N}$, the renaming $[0... \mapsto n...]$ maps variable x to variable $x + n$:

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

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

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

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

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:

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

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.

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

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

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

Figure 2.5: Applying a renaming ρ to a term.

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 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 $\mathrm{FV}(t) \subseteq \rho_1^{-1}(B)$, we have $(t^{\rho_1})^{\rho_2} = t^{(\rho_1\rho_2)}$.

We have defined composition in terms of an inverse image construction. As proposed by McBride [62] and others [8, 12], if terms were explicitly scoped (by indicating the maximum allowed index for the free variables in a given term), one could instead consider renamings as functions between finite sets of variables, with the usual function composition. This yields the **FinSet** category [72]. Renamings can then only be applied to terms whose scope matches the domain of the renaming, and the result is a term whose scope is

the codomain of the renaming. A disadvantage of this approach is that several instances of renamings such as $(+1)$ need to be considered, depending on the scopes of the terms and the domains and codomains of the renamings involved.

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 [9, 10].

Note that terms such as "if $\mathbb{A}$ true $\mathbb{a}$ $\mathbb{b}$" and "if $\mathbb{A}$ false $\mathbb{a}$ $\mathbb{b}$" are not considered β-redexes, but are instead subject to δ-reduction (§2.14).

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 in β-normal form. That is, we need to perform a *hereditary* elimination.

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

(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).

Definition 2.32 (Hereditary elimination: $t\ @\ e \Downarrow r$)**.** Hereditary elimination is a relation $t\ @\ e \Downarrow r$. The full definition is given in 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 $t[u/x]$.

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

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 $t \mathbin{@} e_1 \ldots e_n \Downarrow r$ as a shorthand for $\exists t_1, \ldots, t_{n-1}.(t \mathbin{@} e_1 \Downarrow t_1) \wedge (t_1 \mathbin{@} e_2 \Downarrow t_2) \wedge \ldots \wedge (t_{n-1} \mathbin{@} e_n \Downarrow r)$. 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 $t \mathbin{@} e_1 \ldots e_n \Downarrow r$ holds for some r, we denote such an r by $t \mathbin{@} e_1 \ldots e_n$.

Definition 2.34 (Iterated hereditary substitution: $t[\vec{u}/\vec{x}] \Downarrow r$)**.** Let $\vec{x}^n = (m+n-1), (m+n-2), \ldots, (m+1), m$, 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 $\exists t_1, \ldots, t_{n-1}.(t[u_1^{(+(n-1))}/x_1] \Downarrow t_1) \wedge (t_1[u_2^{(+(n-2))}/x_2] \Downarrow t_2) \wedge \ldots \wedge (t_{n-1}[u_n^{(+0)}/x_n] \Downarrow r)$. 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 $t[\vec{u}/(n-1), \ldots, 0] \Downarrow r$.

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. See the proof of Remark 2.35 in the licentiate thesis [54]. □

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. See the proof of Remark 2.36 in the licentiate thesis [54]. □

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 $(t^\rho \mathbin{@} e^\rho) \Downarrow u^\rho$.*

Proof. Using Remark 2.30 (properties of renamings). See the proof of Lemma 2.39 in the licentiate thesis [54]. □

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 ($\Sigma; \Gamma \vdash h \Rightarrow A$)

Neutral terms consist of a head (e.g. a variable, a metavariable, an atom) followed by zero or more eliminators (§2.1, §2.2). As defined by the typing rules in §2.11, which eliminators may be applied to a head follows from its type. Although we do not provide a bidirectional presentation [84, 32], it is the case that the type of a head can be inferred from the signature and the context, and this property extends to all neutral terms (§2.17).

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

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

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

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

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

Remark. Using the binder syntax described in §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).

$$\frac{\Gamma\ \mathbf{ctx}}{\Gamma \vdash \mathrm{Bool} : \mathrm{Set}}\ \textsc{bool} \quad \stackrel{\text{def}}{=} \quad \frac{\Sigma \vdash \Gamma\ \mathbf{ctx}}{\Sigma; \Gamma \vdash \mathrm{Bool} : \mathrm{Set}}\ \textsc{bool}$$

Even though the premises and the conclusion of a rule often share the same context Γ, there is a handful of rules involving binders in which the context is extended with new variables. For consistency, we have opted for an explicit presentation in which the whole context is threaded through the rules. Alternatively, a less explicit but also less cluttered presentation could be achieved by omitting the unchanged parts of the context [65].

Type constructors

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

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

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

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

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

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

2.12 The Set : Set judgment and normalization

Our theory includes $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$ as an axiom (SET). As described first by Girard [39] and then in a more succinct form by Hurkens [49], terms in a theory where $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$ are not necessarily normalizing.

Ultimately, the unification rules described in Chapter 4 are meant to be used with a properly stratified theory where all well-typed terms are normalizing. With the goal of simplifying the exposition, we consider proper stratification as a separate concern, and include the Set : Set axiom as a typing rule. In §2.15 we give postulates about types and terms in order to establish the existence of certain normal forms. These postulates may not hold in the unstratified theory defined in this chapter, but we have tried our best to not use them in ways that would reduce to *ex falso quodlibet*. We thus expect that our results would hold in a properly stratified version of the theory that we present here.

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

The judgmental equality (or definitional equality, as, in an intensional type theory such as this one, the two notions coincide) 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 $\Gamma \vdash A \equiv B : \mathrm{Set}$.

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

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

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

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

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

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

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

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

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".

Remark. The fact that, in the APP-EQ rule, f and g denote neutral terms precludes the possibility of β-redexes, which are disallowed by the syntax (§2.1).

η-conversion

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

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

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

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

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

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

Remark. Because the term syntax disallows the possibility of intermediate β-redexes (§2.1), the rules DELTA-META, DELTA-IF-TRUE and DELTA-IF-FALSE include an eliminator spine $\vec{e}$ in the LHS of the equality. This way they can be applied to the entire neutral term that contains the δ-redex.

Other rules

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

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

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

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

In §2.13 we have defined when two terms are equal at a given type. In this section we define a set of reduction steps which can be applied to a well-typed term in order to compute a judgmentally equal but arguably simpler form thereof. Some useful properties of this reduction relation and its connection with the term equality are introduced in §2.15.8.

Definition 2.41 ($\delta\eta$-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 on page 32, 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 implementation of the unification algorithm.

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

Remark. The $\delta\eta$-normalization relation is defined in such a way that the δ-rules $\textsc{meta}$, $\textsc{if}_1$ and $\textsc{if}_2$ only apply to the whole neutral term, thus precluding the creation of β-redexes, which are disallowed by the syntax.

Definition 2.42 (Iterated $\delta\eta$-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 the relation $\Sigma; \Gamma \vdash \_ \longrightarrow_{\delta\eta} \_ : A$.

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

2.15 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 our unification rules.

Postulates: 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 192.

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 $\Sigma; \Gamma \vdash t : T$.

$$
\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}
$$

2.15.1 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 so far.

Definition 2.44 (Judgment: $\Sigma; \Gamma \vdash J$)**.** A judgment J has any of the following forms: Δ **ctx**, $\Delta \vdash A$ **type**, $\Delta \vdash A \equiv B$ **type**, $\Delta_1 \equiv \Delta_2$ **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:

- $J = \Delta$ **ctx**, and $\Sigma \vdash \Gamma, \Delta$ **ctx**.
- $J = \Delta_1 \equiv \Delta_2$ **ctx** and $\Sigma \vdash \Gamma, \Delta_1 \equiv \Gamma, \Delta_2$ **ctx**.
- $J = \Delta \vdash A$ **type** and $\Sigma; \Gamma, \Delta \vdash A$ **type**.
- $J = \Delta \vdash A \equiv B$ **type** and $\Sigma; \Gamma, \Delta \vdash A \equiv B$ **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:

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

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

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

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 define hereditary substitution for judgments of the form $(\Delta \vdash t : B)$ explicitly; the remaining cases follow analogously to Definition 2.47.

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

2.15.2 Substitution and elimination

The following properties concern the behaviour of hereditary substitution and elimination. We will use them in Chapter 4 to justify the correctness of the term manipulations performed by our unification rules. The motivation for assuming these properties without proof is discussed in §2.12.

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 $(t\ @\ v)\Downarrow$, $B[v]\Downarrow$, and $\Sigma; \Gamma \vdash t\ @\ v : B[v]$.

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

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 $\Delta[u_1/x]\Downarrow$, $\Delta[u_2/x]\Downarrow$, $t_1[u_1^{(+|\Delta|)}/x]\Downarrow$, $t_2[u_2^{(+|\Delta|)}/x]\Downarrow$, $B[u_1^{(+|\Delta|)}/x]\Downarrow$, $B[u_2^{(+|\Delta|)}/x]\Downarrow$, $\Sigma \vdash \Gamma, \Delta[u_1/x], B[u_1^{(+|\Delta|)}/x] \equiv \Gamma, \Delta[u_2/x], B[u_2^{(+|\Delta|)}/x]$ **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 $x \notin \textsc{fv}(J)$, 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 $|\Delta| \notin \textsc{fv}(V, \Xi \vdash t : A)$.

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 $\Sigma \vdash \Gamma, \Delta^{\mathrm{a}}, \Xi^{\mathrm{a}} \equiv \Gamma, \Delta^{\mathrm{b}}, \Xi^{\mathrm{b}}$ **ctx**, $\Sigma \vdash \Gamma, \Delta^{\mathrm{a}}, \Xi^{\mathrm{a}} \vdash A^{\mathrm{a}} \equiv A^{\mathrm{b}}$ **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 $(t \mathbin{@} .\pi_1)\Downarrow$, $(u \mathbin{@} .\pi_1)\Downarrow$, $\Sigma; \Gamma \vdash t \mathbin{@} .\pi_1 \equiv u \mathbin{@} .\pi_1 : A$, 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, using Definition 2.18 and Remark 2.28. See the proof of Lemma 2.51 in the licentiate thesis [54]. □

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 $(u \mathbin{@} \vec{t})[v] = (u[v] \mathbin{@} t_1[v] \ldots t_n[v])$.

2.15.3 Typing and equality

In this section we introduce some properties which will be particularly useful when justifying the correctness of unification rules involving dependent products, dependent sums, and terms of these types.

Lemma 2.52 (Π inversion)**.** *If* $\Sigma; \Gamma \vdash \Pi AB : T$*, then* $\Sigma; \Gamma \vdash T \equiv \mathrm{Set}$ **type**$,$ $\Sigma; \Gamma \vdash A : \mathrm{Set}$ *and* $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$*. 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. See the proof of Lemma 2.52 in the licentiate thesis [54]. □

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 $\Sigma; \Gamma \vdash \Pi AB \equiv \Pi A'B' : \mathrm{Set}$, then $\Sigma; \Gamma \vdash A \equiv A' : \mathrm{Set}$ and $\Sigma; \Gamma, A \vdash B \equiv B' : \mathrm{Set}$.

Lemma 2.53 (Σ inversion)**.** *If* $\Sigma; \Gamma \vdash \Sigma AB : T$*, then* $\Sigma; \Gamma \vdash T \equiv \mathrm{Set}$ **type** $\Sigma; \Gamma \vdash A : \mathrm{Set}$ *and* $\Sigma; \Gamma, A \vdash B : \mathrm{Set}$*. 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 $\Sigma;\Gamma \vdash \Sigma AB \equiv \Sigma A'B' : \mathrm{Set}$, then $\Sigma;\Gamma \vdash A \equiv A' : \mathrm{Set}$ and $\Sigma;\Gamma, A \vdash B \equiv B' : \mathrm{Set}$.

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 we have $\Sigma;\Gamma \vdash t \equiv v : A$.*

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 $\Sigma;\Gamma \vdash A \equiv B :$ **type** and $\Sigma;\Gamma \vdash B \equiv C$ **type**, then $\Sigma;\Gamma \vdash A \equiv C$ **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. See the proof of Lemma 2.56 in the licentiate thesis [54]. □

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

Proof. See the proof of Lemma 2.57 in the licentiate thesis [54]. □

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

Proof. See the proof of Lemma 2.59 in the licentiate thesis [54]. □

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

Proof. See the proof of Lemma 2.60 in the licentiate thesis [54]. □

2.15.4 Contexts

When solving constraints, some steps involve replacing some types in a context by equal ones, or introducing additional variables into a context. In this section we introduce some properties that help show that the judgments that are valid in the original context are also valid in the new one.

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

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

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

Proof. Using Remark 2.61, Lemma 2.39. See the proof of Lemma 2.62 in the licentiate thesis [54]. □

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. Using Lemma 2.62. See the proof of Lemma 2.63 in the licentiate thesis [54]. □

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* $\textsc{fv}(t) \subseteq \{0, ..., |\Gamma| - 1\}$*.*

Proof. See the proof of Lemma 2.65 in the licentiate thesis [54]. □

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

2.15.5 Signatures

When solving constraints, some steps involve replacing some types in a signature by equal ones, or introducing additional declarations into the signature. In this section we introduce some properties that help show that the judgments that hold in the original signature also hold in the new one.

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 $\Sigma = \Sigma_1, \mathbb{c} : A, \Sigma_2$, there are Σ_1' and Σ_2' such that $\Sigma' = \Sigma_1', \mathbb{c} : A, \Sigma_2'$.

(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 $\Sigma \vdash \Gamma$ **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 $\Sigma \vdash \Gamma$* **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. Using Lemma 2.52, Lemma 2.53, Lemma 2.69, Lemma 2.62, Lemma 2.62, Postulate 4, Postulate 1, Remark 2.13, Remark 2.36, Remark 2.5, Remark 2.15. See the proof of Lemma 2.70 in the licentiate thesis [54]. □

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 $x \notin \text{FV}(J)$, then $\Sigma; \Gamma \vdash J(-1)$.

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

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

Lemma 2.72 (No extraneous constants)**.** *If $\Sigma \vdash J$, then* $\text{CONSTS}(J) \subseteq \text{DECLS}(\Sigma)$.

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

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 $\text{CONSTS}(J) \subseteq \text{DECLS}(\Sigma)$. □

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 $\Sigma \vdash \Gamma$ **ctx**, we have:

- If $\mathbb{0} : A \in \Sigma$, then $\Sigma; \Gamma \vdash A$ **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:

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

Note that because the terms involved in the first premise of the APP-EQ rule must be neutral, the DELTA-META_0 rule cannot be used in combination with APP-EQ to create β-redexes, which are disallowed by the syntax (§2.1).

Proof. Using Lemma 2.70, Remark 2.73, and Definition 2.33. See the proof of Remark 2.74 in the licentiate thesis [54]. □

2.15.6 Typing of neutral elements

Although the presentation of the type theory that we use is not bidirectional, it is still the case that the type of a neutral element is determined by the signature and the context it is typed in. In this section we introduce some properties which will help us exploit this fact in order to justify the correctness of the rules for solving constraints involving neutral terms.

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. Using Lemma 2.70, Postulate 4, Postulate 10, Postulate 11. See the proof of Lemma 2.75 in the licentiate thesis [54]. □

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. Using Lemma 2.75. See the proof of Corollary 2.77 in the licentiate thesis [54]. □

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. Using Lemma 2.75, Postulate 13, Remark 2.30, Remark 2.28, Lemma 2.63. See the proof of Lemma 2.78 in the licentiate thesis [54]. □

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. Using Definition 2.33, Lemma 2.75, Postulate 7, Lemma 2.56, Lemma 2.75, Postulate 6. See the proof of Lemma 2.79 in the licentiate thesis [54]. □

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

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

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

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. □

2.15.7 Inversion of typing and equality rules

In this section we introduce some properties that allow us to solve constraints involving λ-abstractions and pairs. We first make a remark concerning weakening and substitution that will help us prove the first property.

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. Using Lemma 2.70, Lemma 2.52, Lemma 2.62, Remark 2.81, Postulate 2. See the proof of Lemma 2.82 in the licentiate thesis [54]. □

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. Using Lemma 2.62 and Postulate 6. See the proof of Lemma 2.83 in the licentiate thesis [54]. □

Lemma 2.84 ($\langle,\rangle$-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. Using Postulate 3. See the proof of Lemma 2.84 in the licentiate thesis [54]. □

Lemma 2.85 (Injectivity of $\langle,\rangle$)**.** *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. Using Definition 2.32 and Postulate 7. See the proof of Lemma 2.85 in the licentiate thesis [54]. □

2.15.8 Term reduction

In this section we introduce some properties of the reduction relation that we defined in §2.14. These properties imply the existence of certain normal forms of terms. These normal forms can be used to justify the uniqueness of the solutions obtained from our unification rules.

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 the converse property holds; namely, 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. Using Lemma 2.54, Postulate 14. See the proof of Remark 2.89 in the licentiate thesis [54]. □

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. By induction on n, using Lemma 2.57, Postulate 10, Lemma 2.90. See the proof of Remark 2.91 in the licentiate thesis [54]. □

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 $\Sigma;\Gamma \vdash T \equiv \Sigma A B$ **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. □

2.16 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. This reduction computes the weak head-normal form of a term (WHNF).

WHNF is used for defining type application (Definition 2.104). It may also be used by a constraint solving algorithm to determine which unification rules are applicable to a given constraint.

Definition 2.95 (Weak head normal form: $\Sigma \vdash t \searrow u$)**.** The relation $\Sigma \vdash t \searrow u$ (read as "u is the weak head normal form of t in signature Σ") is inductively defined in Figure 2.8 on page 44.

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. Using Postulate 2, Postulate 3, Lemma 2.86, Lemma 2.56, Postulate 8, Remark 2.97, Lemma 2.86. See the proof of Lemma 2.98 in the licentiate thesis [54]. □

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

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

Figure 2.8: Inductive definition of the weak head normal form relation ($\Sigma \vdash \_ \searrow \_$). The signature Σ is considered to be given implicitly.

- $u = \mathrm{Bool}$.
- $u = \Sigma AB$ *for some* A, B.
- $u = \Pi AB$ *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* $\vec{e}$.
- $u = \mathbb{o}\,\vec{e}$ *for some atom* $\mathbb{o}$, *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*, ⟨_, _⟩)**.** 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*, the recursor if 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. Using Lemma 2.98, Postulate 14, Definition 2.41, Lemma 2.99. See the proof of Lemma 2.101 in the licentiate thesis [54]. □

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. □

2.17 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:

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

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 $\hat{@}$ 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. Using Definition 2.103, Lemma 2.101, Lemma 2.98, Postulate 10, Postulate 1, Postulate 4. See the proof of Lemma 107 in the licentiate thesis [54]. □

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. Using Lemma 2.107, Lemma 2.98, Lemma 2.106, Lemma 2.75, Postulate 10. See the proof of Lemma 2.111 in the licentiate thesis [54]. □

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,* $\Sigma;\Gamma \vdash A \equiv \Sigma UV\,\mathbf{type}$*, 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 $T \mathbin{\hat{@}} \vec{t} \Downarrow U$) 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* $\Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow U$ *if and only if* $\Sigma;\Gamma \vdash T\ \hat{@}^{R}\ \vec{t} \Downarrow U$*.*

Proof. See the proof of Lemma 2.115 in the licentiate thesis [54]. □

Lemma 2.116 (Free variables in type application)**.** *If* $\Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow A$*, 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 $\Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow A$. We proceed by induction on the derivation of $\Sigma;\Gamma \vdash T \mathbin{\hat{@}} \vec{t} \Downarrow A$, 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 $(t \mathbin{@} \vec{e}) \Downarrow u$*, then* $(t^{\rho} \mathbin{@} \vec{e}^{\rho}) \Downarrow u^{\rho}$*.*

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

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

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* $\textsc{fv}(t) \subseteq \{\vec{x}\}$*. Then,* $\Sigma; \cdot \vdash \lambda \vec{y}^n.t[\vec{x} \mapsto \vec{y}] : A$ *and* $(\lambda \vec{y}^n.t[\vec{x} \mapsto \vec{y}])\ @\ \vec{x} \Downarrow t$*.*

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

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

(iv) $\textsc{fv}(A') \subseteq \{\overrightarrow{y'}\}$ and

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

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 $\vec{x}$, using Lemma 2.119, Lemma 2.62, Lemma 2.98, Lemma 2.40, Lemma 2.78, Lemma 2.62 Postulate 13, Remark 2.29, Remark 2.102 Remark 2.30, Remark 2.28. We then use the property and Lemma 2.109, Lemma 2.115, Lemma 2.65, Remark 2.28, Lemma 2.116, Lemma 2.40, Remark 2.29 to prove the main result. See the proof of Lemma 2.120 in the licentiate thesis [54]. □

2.18 Metasubstitutions (Θ)

In our theory, unlike in other type-systems such as Damas-Hindley-Milner [46, 69, 23, 22], metavariables may occur in both types and terms. Uninstantiated metavariables can thus prevent otherwise well-typed terms from having desirable computational properties (e.g. all closed terms of type Bool normalizing to either true or false), and desirable mathematical properties (e.g. an empty type becoming trivially inhabitable). These issues can sometimes be avoided if the metavariables only occur in types [96], but making this distinction precise is not our focus.

We reserve the term metasubstitutions for signatures which instantiate all the metavariables. Metasubstitutions thus represent complete solutions; partial steps towards a solution are represented by the more general notion of a signature. Further implications of only considering complete solutions to a problem are discussed in §4.8.

Definition 2.121. Metasubstitution: (Θ)

$$\begin{array}{rcll} \Theta & ::= & \cdot & \text{empty metasubstitution} \\ & | & \Theta, \mathbb{a} : A & \text{atom} \\ & | & \Theta, \alpha := t : A & \text{metavariable instantiation} \end{array}$$

For a metasubstitution to be well-formed, we impose the additional condition that, even though the metasubstitution may contain atom declarations (e.g. $\mathbb{a} : A$) and also metavariable instantiations (e.g. $\alpha := t : A$), the terms and types in those declarations (i.e. t and A) do not contain any metavariables themselves. This condition will later on allow us to easily remove certain metavariables from a signature without compromising its well-formedness (Definition 2.132).

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:

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

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

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

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 $\mathbb{a} : A \in \Theta$, $\textsc{metas}(A) = \emptyset$; and, for all $\alpha := t : A \in \Theta$, $\textsc{metas}(t) \cup \textsc{metas}(A) = \emptyset$; 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 Θ **wf**, Σ **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 $D = \mathbb{a} : A$, $D = \alpha : A$ 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:

- $D = \mathbb{a} : A$, and $\Theta; \cdot \vdash \mathbb{a} : A$.

- $D = \alpha : A$, 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. See the proof of Remark 2.128 in the licentiate thesis [54]. □

Remark 2.129 (Alternative characterization of compatibility of a metasubstitution with a declaration). A well-formed metasubstitution Θ **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 $\Theta; \cdot \vdash B \equiv A$ **type**.

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

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

Proof. See the proof of Remark 2.129 in the licentiate thesis [54]. □

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. Using Lemma 2.62, Lemma 2.79. See the proof of Lemma 2.130 in the licentiate thesis [54]. □

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. Using Lemma 2.130, Remark 2.128, Lemma 2.69. See the proof of Remark 2.131 in the licentiate thesis [54]. □

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.

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

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

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

Proof. Using Remark 2.9. See the proof of Remark 2.133 in the licentiate thesis [54]. □

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. Using Postulate 12, 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 Θ **wf**, $\Theta \subseteq \Theta'$.
Let J be a judgment. If $\Theta' \vdash J$ and $\text{CONSTS}(J) \subseteq \text{DECLS}(\Theta)$, then $\Theta \vdash J$.

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

2.19 Closing metasubstitution (CLOSE(Σ))

We aim to build solutions to unification problems by modifying an initial signature stepwise. If this process is successful, the end result will be a signature Σ in which all the original constraints hold and all the metavariables are instantiated. In this section we define how to obtain a well-formed metasubstitution Θ from such a signature Σ.

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 (Figure 2.9).

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

Figure 2.9: Inductive definition of the meta-free normal form of a term.

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. Using Postulate 8. See the proof of Lemma 2.141 in the licentiate thesis [54]. □

Remark 2.142 (Metavariable-free term)*.* Let Σ **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 $\Sigma \vdash t \mathrel{\hat{\searrow}} t'$, $\Sigma \vdash A \mathrel{\hat{\searrow}} A'$, $\textsc{metas}(A') = \textsc{metas}(t') = \emptyset$, $\Sigma; \cdot \vdash A' \equiv A$ **type**, $\Sigma; \cdot \vdash t' \equiv t : A'$. Also, by Remark 2.15 (there is only set), if $\Sigma; \cdot \vdash A$ **type**, there is A' such that $\Sigma \vdash A \mathrel{\hat{\searrow}} A'$, $\Sigma; \cdot \vdash A$ **type**, and $\Sigma; \cdot \vdash A' \equiv A$ **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:

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

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

Proof. By induction on Σ, using Lemma 2.130, Remark 2.142, Remark 2.128, Lemma 2.69, Remark 2.142. See the proof of Lemma 2.144 in the licentiate thesis [54]. □

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

Definition 2.145 (Equality of metasubstitutions: $\Theta \equiv \Theta'$)**.** We say that two 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, metasubstitution equality is the transitive closure of the following relation:

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

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 derivation 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 $(\Theta^0, \Theta^1)\,\mathbf{wf}$, $(\Theta^0, \Theta)\,\mathbf{wf}$, $\Theta^0, \Theta^1 \vDash \Sigma$ 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, using Lemma 2.130, Lemma 2.72, Postulate 12, Lemma 2.69, Lemma 2.146, Lemma 2.147, Lemma 2.144. See the proof of Lemma 2.148 in the licentiate thesis [54].

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, $\Theta_1 \equiv \Theta_2$. □

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* $(\Theta^1)_\Sigma\,\mathbf{wf}$*,* $(\Theta^2)_\Sigma\,\mathbf{wf}$ *and* $(\Theta^1)_\Sigma \equiv (\Theta^2)_\Sigma$*.*

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

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

During constraint solving, 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

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

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 (Declarations in a signature extension). 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 (declarations in a signature extension), $\textsc{support}(\Sigma) \subseteq \textsc{support}(\Sigma')$. 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. See the proof of Lemma 2.155 in the licentiate thesis [54]. □

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 for Σ_2). Then $\Sigma_1, \Sigma_2 \sqsubseteq \Sigma_1', \Sigma_2$ (in particular, Σ_1', Σ_2 **sig**).*

Proof. Using Lemma 2.155 (preservation of judgments under signature extensions). See the proof of Corollary 2.156 in the licentiate thesis [54]. □

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. Using Remark 2.134, Remark 2.135, Remark 2.9, Remark 2.153, Lemma 2.69, Lemma 2.72, Postulate 12. See the proof of Lemma 2.157 in the licentiate thesis [54]. □

2.22 Non-reducible terms

A number of our unification rules involve 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 (i.e. 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:

- $x\,\vec{e}$.
- $\mathbb{0}\,\vec{e}$.
- $\mathrm{if}\,\vec{e}^{\,n}$, 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 $f\,t$ 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. Using Lemma 2.56 and Lemma 2.75. See the proof of Remark 2.160 in the licentiate thesis [54]. □

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 $\Sigma;\Gamma \vdash f_0 \longrightarrow_{\delta\eta} t : T$:

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

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

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

However, if t is 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 are necessarily 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. We proceed by induction on the structure of the derivation, using Remark 2.161. See the proof of Remark 2.162 in the licentiate thesis [54]. □

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

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 $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,...,i-1} \equiv h^2\,\overrightarrow{e^2}_{1,...,i-1} : \Pi UV$ (first case) or $\Sigma;\Gamma \vdash h^1\,\overrightarrow{e^1}_{1,...,i-1} \equiv h^2\,\overrightarrow{e^1}_{1,...,i-1} : \Sigma UV$ (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. We use Lemma 2.62, Lemma 2.86, Lemma 2.56, Lemma 2.75, Postulate 13, Postulate 10, Remark 2.160, Remark 2.89, Remark 2.162, Remark 2.91, Remark 2.92. See the proof of Lemma 2.163 in the licentiate thesis [54]. □

Definition 2.164 (Irreducible terms)**.** A strongly neutral term is irreducible if it is in one of the following forms:

- $x\,\vec{e}$.
- $\mathfrak{o}\,\vec{e}$.
- 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 $f\,\vec{e}$ for any elimination spine $\vec{e}$.

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 $\Sigma;\Gamma \vdash \lambda\vec{z}^n.f\,\vec{z}^n \longrightarrow^{\star}_{\delta\eta} \lambda\vec{z}^n.v : \Pi\vec{U}^n V$, and $\Sigma;\Gamma \vdash \lambda\vec{z}^n.v \not\longrightarrow_{\delta\eta} : \Pi\vec{U}^n V$.*

Proof. Using Postulate 15 (existence of a unique full normal form). See the proof of Lemma 2.166 in the licentiate thesis [54]. □

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:*

$$
\begin{array}{rcll}
t^{\text{nf}}, u^{\text{nf}}, v^{\text{nf}}, A^{\text{nf}}, B^{\text{nf}} & ::= & \text{Set} & \\
& | & \text{Bool} & \\
& | & \Pi A^{\text{nf}} B^{\text{nf}} & \\
& | & \Sigma A^{\text{nf}} B^{\text{nf}} & \\
& | & c & \\
& | & \lambda.t^{\text{nf}} & \\
& | & \langle t^{\text{nf}}, u^{\text{nf}} \rangle & \\
& | & h\,\overrightarrow{e^{\text{nf}}} & \textit{if } h\,\overrightarrow{e^{\text{nf}}} \textit{ is strongly neutral}
\end{array}
$$

$$
e^{\text{nf}} \quad ::= \quad t^{\text{nf}} \quad | \quad .\pi_1 \quad | \quad .\pi_2
$$

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. □

2.23 Rigidly occurring terms ($t[\![u]\!]$)

A subterm occurs rigidly in a term only if the occurrence cannot be made to "disappear" from the term by normalizing it (c.f. Definition 2.41, δη-normalization step). This fact can be used to justify transformations that will allow a unification algorithm to solve certain constraints. For instance, the fact that a metavariable occurs rigidly in a term can imply that certain arguments of that metavariable may be ignored (§4.5.9).

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:

$$
\begin{array}{rlll}
(\text{R-ID}) & u[\![u]\!]^0 & & \\
(\text{R-}\Pi_1) & (\Pi AB)[\![u]\!]^n & \textbf{if} & A[\![u]\!]^n \\
(\text{R-}\Pi_2) & (\Pi AB)[\![u]\!]^{1+n} & \textbf{if} & B[\![u]\!]^n \\
(\text{R-}\Sigma_1) & (\Sigma AB)[\![u]\!]^n & \textbf{if} & A[\![u]\!]^n \\
(\text{R-}\Sigma_2) & (\Sigma AB)[\![u]\!]^{1+n} & \textbf{if} & B[\![u]\!]^n \\
(\text{R-}\lambda) & (\lambda.t)[\![u]\!]^{1+n} & \textbf{if} & t[\![u]\!]^n \\
(\text{R-}\langle,\rangle_1) & (\langle t_1, t_2 \rangle)[\![u]\!]^n & \textbf{if} & t_1[\![u]\!]^n \\
(\text{R-}\langle,\rangle_2) & (\langle t_1, t_2 \rangle)[\![u]\!]^n & \textbf{if} & t_2[\![u]\!]^n \\
(\text{R-IRRED}) & (f\,\vec{e})[\![f]\!]^0 & \textbf{if} & f \text{ is an irreducible term} \\
(\text{R-STRONG}) & (h\,\vec{e})[\![u]\!]^n & \textbf{if} & h\,\vec{e} \text{ is strongly neutral} \\
& & \textbf{and} & \text{there is } t \in \vec{e} \text{ such that } t[\![u]\!]^n
\end{array}
$$

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 in t 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:

$$(\text{TR-ID}) \quad \Sigma;\Gamma \vdash u[\![\cdot \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash U \equiv T \textbf{ type}$$

$$(\text{TR-}\Pi_1) \quad \Sigma;\Gamma \vdash (\Pi AB)[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash A[\![\Delta \vdash u : U]\!] : \text{Set}$$

$$(\text{TR-}\Pi_2) \quad \Sigma;\Gamma \vdash (\Pi AB)[\![A',\Delta \vdash u : U]\!]' : T \quad \textbf{if} \quad \Sigma;\Gamma \vdash A' \equiv A \textbf{ type}$$
$$\textbf{and } \Sigma;\Gamma,A \vdash B[\![\Delta \vdash u : U]\!] : \text{Set}$$

$$(\text{TR-}\Sigma_1) \quad \Sigma;\Gamma \vdash (\Sigma AB)[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash A[\![\Delta \vdash u : U]\!] : \text{Set}$$

$$(\text{TR-}\Sigma_2) \quad \Sigma;\Gamma \vdash (\Sigma AB)[\![A',\Delta \vdash u : U]\!]' : T \quad \textbf{if} \quad \Sigma;\Gamma \vdash A' \equiv A \textbf{ type}$$
$$\textbf{and } \Sigma;\Gamma,A \vdash B[\![\Delta \vdash u : U]\!] : \text{Set}$$

$$(\text{TR-}\lambda) \quad \Sigma;\Gamma \vdash (\lambda.t)[\![A,\Delta \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash \Pi AB \equiv T \textbf{ type}$$
$$\textbf{and } \Sigma;\Gamma,A \vdash t[\![\Delta \vdash u : U]\!] : B$$

$$(\text{TR-}\langle,\rangle_1) \quad \Sigma;\Gamma \vdash (\langle t_1,t_2\rangle)[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash \Sigma AB \equiv T \textbf{ type}$$
$$\textbf{and } \Sigma;\Gamma \vdash t_1[\![\Delta \vdash u : U]\!] : A$$

$$(\text{TR-}\langle,\rangle_2) \quad \Sigma;\Gamma \vdash (\langle t_1,t_2\rangle)[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash \Sigma AB \equiv T \textbf{ type}$$
$$\textbf{and } \Sigma;\Gamma \vdash t_2[\![\Delta \vdash u : U]\!] : B[t_1]$$

$$(\text{TR-IRRED}) \quad \Sigma;\Gamma \vdash (f\,\vec{e})[\![\cdot \vdash f : U]\!]' : T \qquad \textbf{if} \quad f \text{ is an irreducible term}$$

$$(\text{TR-STRONG}) \quad \Sigma;\Gamma \vdash (h\,\overrightarrow{e_1}\,t\,\overrightarrow{e_2})[\![\Delta \vdash u : U]\!]' : T \qquad \textbf{if} \quad \Sigma;\Gamma \vdash h\,\overrightarrow{e_1} : \Pi AB$$
$$\textbf{and } \Sigma;\Gamma \vdash t[\![\Delta \vdash u : U]\!] : A$$
$$\textbf{and } h\,\overrightarrow{e_1}\,t\,\overrightarrow{e_2} \text{ is strongly neutral}$$

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 $\mathrm{FV}(u) - |\Delta| \subseteq \mathrm{FV}(t)$.

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 $\mathrm{FV}(v) - |\Delta| \subseteq \mathrm{FV}(r)$.*

Proof. Using Lemma 2.56, Lemma 2.57, Lemma 2.62, Remark 2.94, Lemma 2.86. See the proof of Lemma 2.172 in the licentiate thesis [54]. □

Corollary 2.173 (Preservation of head variable)**.** *If $\Sigma;\Gamma \vdash x\,\vec{e} \equiv t : U$, then $x \in \mathrm{FV}(t)$.*

Proof. Using Postulate 14, Remark 2.43, Lemma 2.172, Definition 2.41. See the proof of Corollary 2.173 in the licentiate thesis [54]. □

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. Using Lemma 2.167, Lemma 2.174, Lemma 2.170, Postulate 14, Lemma 2.172, Remark 2.28, Corollary 2.173, Remark 2.43. See the proof of Lemma 2.175 in the licentiate thesis [54]. □

Lemma 2.176 (Injectivity of normal forms with respect to irreducibles)**.** *Suppose that $\Theta;\Gamma,\overrightarrow{x:U}^n \vdash v : V$, and $\Theta;\Gamma,\vec{U} \vdash v \not\longrightarrow_{\delta\eta} : V$.*

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 $\Theta;\Gamma,\vec{U} \vdash v \not\longrightarrow_{\delta\eta} : V$, 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, 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 $h_i^{(+|\Delta|)} = {h'}_i^{(+|\Delta|)}$.

We proceed by induction on u_{nf}, using Remark 2.36, Remark 2.165, Lemma 2.163, Lemma 2.163, Lemma 2.83, Lemma 2.85, Postulate 10. See the proof of Lemma 2.176 in the licentiate thesis [54]. □

2.24 Out of scope features

The theoretical description is a subset of the constructs present in a practical implementation. Our focus is on the metavariable solving aspect of dependent type checking, with the goal is to defining unification rules which will only produce well-typed terms when applied. We have thus side-lined many equally important, but mostly orthogonal aspects of dependent type checking, some of which we will now briefly comment on.

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).

Generalized records with η

We have included 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. However, record types with no fields (i.e. the *unit type with* η*-equality*) cannot be modelled in a straightforward manner in our theory. In fact, η-equality for a record type with no fields can be particularly challenging to handle rigorously. For instance, whether the judgment $\Sigma; \Gamma, x : A \to B \vdash x\, t \equiv x\, u : B$ entails $\Sigma; \Gamma, x : A \to B \vdash t \equiv u : A$ depends on whether B is a singleton type, a fact which may in turn depend on the bodies of metavariables occurring in B. Solving constraints involving singleton types thus involves a non-trivial amount of bookkeeping. In the implementation we adopt a pragmatic approach, where we aim to preserve the existing functionality in Agda regarding singleton types but without providing any correctness guarantees (§4.9.1).

2.25 Closing remarks

In this chapter we have defined a term syntax and typing rules in the style of Martin-Löf Type Theory. In subsequent chapters we will use this theory and the properties that we have stated in order to describe sound higher-order unification rules. These unification rules can be used by a dependent type checking algorithm to infer those subterms which the user has omitted.

Chapter 3

Unification for dependent type checking

We are interested in the problem of type checking a term t where some of the subterms are missing. In this chapter we connect this problem to solving a set of higher-order unification constraints, and discuss how the resulting constraints may be solved.

The specific details of how the problem of reduction from type checking to unification constraints are outside the scope of this chapter. We will just refer to the approach described by Mazzoli and Abel [60], in which the type checking problem is entirely reduced to solving a set of higher-order unification constraints. This approach is implemented in the type checker prototype Tog [61], on which our previous prototype implementation Tog$^+$ [56] is based. Proof assistants such as Agda use a combination of direct application of the typing rules and solving unification constraints which may in some cases be more efficient [54, §5.5.2]. In both cases, the type checker produces a number of higher-order unification constraints that must be solved for the term to have the desired type. In this process the omitted subterms are also inferred.

We begin with the definition of a type checking problem with omitted subterms. The omitted subterms are represented using metavariables. More 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.

Definition 3.1 (Term with holes)**.** Consider a signature Σ and context Γ, such that $\Sigma \vdash \Gamma$ **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 $\textsc{metas}(t) \subseteq \textsc{support}(\Sigma) \cup H$.

$$
\begin{array}{lll}
\textsc{holes}(\Sigma, n, \mathrm{Set}) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, \mathrm{Bool}) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, \Pi A B) \Downarrow (H_1 \cup H_2) & \textbf{if} & \textsc{holes}(\Sigma, n, A) \Downarrow H_1 \\
 & \textbf{and} & \textsc{holes}(\Sigma, n+1, B) \Downarrow H_2 \\
 & \textbf{and} & H_1 \cap H_2 = \emptyset \\
\textsc{holes}(\Sigma, n, \Sigma A B) \Downarrow (H_1 \cup H_2) & \textbf{if} & \textsc{holes}(\Sigma, n, A) \Downarrow H_1 \\
 & \textbf{and} & \textsc{holes}(\Sigma, n+1, B) \Downarrow H_2 \\
 & \textbf{and} & H_1 \cap H_2 = \emptyset \\
\textsc{holes}(\Sigma, n, c) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, \lambda.t) \Downarrow H & \textbf{if} & \textsc{holes}(\Sigma, n+1, t) \Downarrow H \\
\textsc{holes}(\Sigma, n, \langle t_1, t_2 \rangle) \Downarrow (H_1 \cup H_2) & \textbf{if} & \textsc{holes}(\Sigma, n, t_1) \Downarrow H_1 \\
 & \textbf{and} & \textsc{holes}(\Sigma, n, t_2) \Downarrow H_2 \\
 & \textbf{and} & H_1 \cap H_2 = \emptyset \\
\textsc{holes}(\Sigma, n, \alpha\ (n-1)\ (n-2)\ ...\ 0) \Downarrow \{\alpha\} & \textbf{if} & \alpha \notin \textsc{decls}(\Sigma) \\
\textsc{holes}(\Sigma, n, h\ \vec{e}^m) \Downarrow (\bigcup_{i=1}^{m} H_i) & \multicolumn{2}{l}{\textbf{if } \forall i \in \{1, ..., m\}.\textsc{holes}(\Sigma, n, e_i) \Downarrow H_i} \\
 & \multicolumn{2}{l}{\textbf{and } \forall i, j.i \neq j.H_i \cap H_j = \emptyset} \\
 & \multicolumn{2}{l}{\textbf{and } (h = \alpha, \alpha \in \textsc{decls}(\Sigma)} \\
 & \multicolumn{2}{l}{\quad \textbf{or } h = x \textbf{ or } h = \mathbb{0} \textbf{ or } h = \mathrm{if})} \\
 & & \\
\textsc{holes}(\Sigma, n, .\pi_1) \Downarrow \emptyset & & \\
\textsc{holes}(\Sigma, n, .\pi_2) \Downarrow \emptyset & &
\end{array}
$$

Figure 3.1: Recursive definition of the set of holes in a term

Remark. An alternative to Definition 3.1 would be to extend the syntax of terms with a dedicated placeholder for omitted terms. For simplicity we choose to have a unified term syntax for terms with and without omitted subterms.

Definition 3.2 (Well-formed type checking problem: $\Sigma; \Gamma \vdash^? t : A$)**.** Consider a signature Σ and context Γ, such that $\Sigma \vdash \Gamma$ **ctx**, and a type A such that $\Sigma; \Gamma \vdash A$ **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 Θ **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 $\Theta \equiv \Theta'$.

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. Then:

(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 4.35 (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 $\lceil \Theta_1 \not\models \Theta_2 \rfloor$. 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 discuss how the solutions to the type checking problem are connected to the solutions for the resulting higher-order unification problem. We start by defining a notion of basic constraint, originally defined by Mazzoli and Abel [60] under the name "heterogeneous constraints".

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 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 $\Sigma; \Gamma_i \vdash t_i : A_i \cong u_i : B_i$?

A type checking problem is reduced to a unification problem by an elaboration algorithm:

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 $\Sigma \subseteq \Sigma'$, $\textsc{AtomDecls}(\Sigma) = \textsc{AtomDecls}(\Sigma')$, $\textsc{support}(\Sigma') \supseteq \textsc{metas}(t)$, Σ' **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 $\Theta_{\Sigma \cup t}$ **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 $\tilde{\Theta} \vDash \vec{\mathcal{C}}$.

3.2 Approaches to elaboration

There are several approaches to elaboration, that is, for reducing a type checking problem to set of unification constraints. In the approach by Norell and Coquand [73, 79] the constraints are homogeneous, that is, both sides have the same type.

Definition 3.13 (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 $\Sigma; \Gamma \vdash u : A$.

A homogeneous constraint $\Sigma; \Gamma \vdash t \approx u : A$ can be interpreted as a heterogeneous constraint $\Sigma; \Gamma \vdash t : A \approx u : A$. A signature Σ' extending Σ solves the constraint $\Sigma; \Gamma \vdash t \approx u : A$ if and only if $\Sigma'; \Gamma \vdash t \equiv u : A$ holds.

Blocked constants, originally defined as "guarded constants" [73, §3], are used to replace subterms which are not yet known to be of the appropriate type. They can be understood as an extension of the term syntax described in Chapter 2.

Definition 3.14 (Blocked constant)**.** A blocked constant has the form $\Sigma \vdash p : A \longrightarrow_\delta t$ when $\vec{\mathcal{S}}$, where $\vec{\mathcal{S}}$ is a set of well-formed homogeneous constraints, and $p : A \in \Sigma$.

Blocked constants reduce to the subterm they are replacing only once the inferred type and the appropriate type have been unified. That is, given a blocked constant $\Sigma \vdash p : A \longrightarrow_\delta t$ when $\vec{\mathcal{S}}$, and a signature Σ' such that Σ' extends Σ and Σ' solves $\vec{\mathcal{S}}$, we have that $\Sigma'; \cdot \vdash t : A$.

If we examine the elaboration algorithm as originally described [73, §3.3], there are three cases where blocked constants are used. In all of them, there exists a type B such that $\Sigma; \cdot \vdash t : B$, and Σ' solving $\vec{\mathcal{S}}$ is equivalent with $\Sigma'; \cdot \vdash A \equiv B$ **type**. Therefore, we can understand a blocked constant as the combination of a metavariable α and a basic constraint $\Sigma; \cdot \vdash \alpha : A \approx t : B$.

A more recent algorithm proposed by Mazzoli and Abel [60] elaborates a type checking problem (Definition 3.2) to Problem 3.9 (unification of dependently-typed terms). There is no longer a distinction between subterms omitted by the user, and those subterms which cannot be immediately type-checked: all 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.

The specifics of the elaboration algorithm used are outside of the scope of this work. The unification approach described in Chapter 4 (unifying without order) does not depend on the particular details of the elaboration, as long as it is correct.

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 omitted subterms in the original type checking problem.

Finding a unique solution to the basic constraints is an instance of higher-order unification. In the following sections we review how this problem has been approached both historically and in the context of dependent types.

3.3 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 [43] and Robinson [90].

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 $\mathbb{F}\, x$). ◀

3.4 (Un)decidability of higher order unification

Deciding whether a solution to a higher-order unification problem exists is undecidable, as shown by Huet [47]. Huet shows the undecidability of higher-order unification by encoding the (undecidable) Post correspondence problem [85] 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 $\{\mathtt{x}, \mathtt{y}\}$, 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 $\mathtt{xyx}$ is $\widetilde{\mathtt{xyx}} = \lambda z.(x\, (y\, (x\, z)))$.

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.

We observe 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, $\Theta \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \alpha := \lambda w_1 ... \lambda w_n \lambda z.\, z : (\mathbb{A} \to \mathbb{A}) \to ..._n \to (\mathbb{A} \to \mathbb{A}) \to \mathbb{A} \to \mathbb{A}$. This solution is unique if and only if the matching Post correspondence problem has no solution.

3.5 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 [68]. Problems in this fragment always have a unique, most-general solution (see Lemma 4.35, Miller's pattern condition). Paraphrasing Gundry and McBride [44], the behaviour of a metavariable is fully characterized by its application to distinct variables.

3.6 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 [67] observed, by postponing certain constraints, we can solve problems which are not strictly in the pattern fragment.

Furthermore, by using *pruning* (see §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 [88] in the context of dependent types.

3.7 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 [31] 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 4.35, 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 [2] extensively elaborate on this insight to extend dynamic pattern unification for a theory containing both dependent function (Π) and record (Σ) types.

3.8 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 §3.7 (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. We described these situations in the introduction as the binder problem (§1.3) and the spine problem (§1.4).

Reed [88] and Abel and Pientka [2] use a formulation of typing modulo constraints. This formulation relies on the fact that in their setting, metavariables in types can only appear as parameters to atomic type families, so unsolved constraints do not jeopardize the correctness of normalization.

In section 3.4 of Norell's thesis [73], 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 [73] 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 [80].

[1] In Agda and Coq, non-terminating recursive definitions are disallowed by a termination checker.

In the elaboration algorithm described in §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 [73]: 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.

3.9 Strictly ordered, homogeneous constraints

In the introduction we described the binder problem, which we repeat here:

Problem 3.18 (Binder problem)**.** Consider a basic constraint which unifies two Π-types: $\Pi(x : A)B$ and $\Pi(x : A')B'$. In order to obtain constraints that 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'.

The approach suggested by Mazzoli and Abel [60] sidesteps the binder problem by having both sides of the constraints have the same type. Such constraints are called *homogeneous*.

Definition 3.19 (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 [3].

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 3.20 (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. ◀

This strict ordering is also used when unifying binders and elimination spines. problems, with analogous limitations. In §6.6.1 we give examples where the limitations of sequential solving arise in instances of the binder problem (Example 6.1) and the spine problem (Example 6.2).

3.10 Twin types

In the introduction we describe the blocked constant approach (§1.3) used for enabling out-of-order solving of constraints. However, this approach is not sufficient to provide constraint-reordering while ensuring well-typedness when unifying elimination spines (§1.4).

Gundry and McBride [44] propose considering, as internal constraints, constraints with a twin context: that is, a context where each variable has two possible types.

$$\begin{array}{rcll} \Gamma & ::= & \cdot & \text{empty twin context} \\ & | & \Gamma, x : A & \text{variable of simple type} \\ & | & \Gamma, \widehat{x} : A_1 \ddagger A_2 & \text{variable of twin type} \end{array}$$

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 Problem 3.18 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 $\grave{x}$):

$$\begin{aligned}\Sigma; \Gamma \vdash \Pi(x : A)B : \mathrm{Set} \cong \Pi(x : A')B' : \mathrm{Set} \;\rightsquigarrow\\ \Sigma; \Gamma \vdash A : \mathrm{Set} \cong u : A' : \mathrm{Set} \;\wedge\\ \Sigma; \Gamma, x : A \ddagger A' \vdash B[x \mapsto \acute{x}] : \mathrm{Set} \cong B'[x \mapsto \grave{x}] : \mathrm{Set}\end{aligned}$$

This approach addresses the binder and spine problem without the limitations mentioned in §3.9. However, unlike the other approaches to unification, the twin variable approach has not been implemented into a type checker with a significant user community, and it is therefore not known how it performs in practice.

Furthermore, the twin type approach as stated requires significant changes to the syntax of the terms, which would increase the implementation effort. In order to facilitate our implementing job (and the eventual reuse of the code), we advocate for an approach which leaves the term syntax intact, and where the changes are limited to the representation of the unification constraints themselves and their contexts.

In Chapter 4 we build a set of unification rules based on a simplified variant of twin types. In Chapters 5 and 6 we demonstrate the implementability of our approach, and its suitability to type check a wide range of examples.

Chapter 4

Unifying without order

In this chapter we summarize the unification rules on which the implementation is based. We list their required preconditions and describe the key lemmas required to justify their correctness.

Our goal is to specify a set of unification rules that can be easily implemented for an existing dependent type checker, such as Agda. The full lemmas and formal proofs, including a completeness argument, can be found in my licentiate thesis [54]. The preconditions of the Rule-Schema 2 have been changed slightly for the implementation, therefore those proofs are restated here.

Our approach builds on Gundry and McBride's [44]. We both simplify and extend their unification rules to make it easier for us to implement them into the Agda type-checker.

There are three main differences with respect to Gundry and McBride's [44] approach:

- In Gundry and McBride's [44] 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 §4.1 (two-sided internal constraints) for more details.

  These twin variable annotations would eventually need to be removed when a metavariable is instantiated, potentially 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 my licentiate thesis [54, §5.6.1].

- All rules can be applied to terms which are not in δ-normal form. Excessive normalization may affect both readability [1] and performance [15]. 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 binder problem (§1.3) and the spine problem (§1.4) without the limitations from enforcing a strict ordering of constraints (§3.9), we use Gundry and McBride's [44] 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 [44], we do not extend the syntax of terms with twin variables (see §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.

$$\mathcal{C}, \mathcal{D}, \mathcal{E} \quad ::= \quad \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B \quad \text{internal constraint}$$

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 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 metasubstitution, and $\mathcal{C} = \Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A ‡ B$ be an internal constraint.

We say that Θ is a solution to $\mathcal{C}$ (written $\Theta \vDash \mathcal{C}$) if and only if $\Theta; \mathcal{C}$ **wf**, $\Theta \vdash \Gamma_1, A \equiv \Gamma_2, B$ **ctx** and $\Theta; \Gamma_1 \vdash t \equiv u : A$ (or, equivalently, $\Theta \vdash \Gamma_1 \equiv \Gamma_2$ **ctx**, $\Theta; \Gamma_1 \vdash A \equiv B$ **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, $\mathcal{C} = \Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A ‡ B$. Then there is a judgment J, $J = (\Gamma_1, A \equiv \Gamma_2, B$ **ctx**$) \wedge (\Gamma_1 \vdash t \equiv u : A)$ such that $\textsc{consts}(J) = \textsc{consts}(\mathcal{C})$ and, for any well-formed metasubstitution Θ **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 [44] 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 whose types are not necessarily equal.

Definition 4.12 (Heterogeneous equality: $\Sigma; \Gamma ‡ \Delta \vdash t \equiv u : A ‡ B$)**.** Two terms $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Delta \vdash u : B$ are heterogeneously equal (written $\Sigma; \Gamma ‡ \Delta \vdash t \equiv u : A ‡ B$), 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 $\Sigma; \Gamma ‡ \Delta \vdash t \equiv\!\{v\}\!\equiv u : A ‡ B$. We call v the *interpolant*, taking inspiration from the tangentially related concept in logic [21].

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 4.36 (heterogeneously equal contexts modulo variables) and Lemma 4.37 (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 [44], 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 of heterogeneous equality)**.** *Assume that* $\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2$ **ctx**. *Then, we have* $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \equiv u : A_1 \ddagger A_2$ *if and only if* $\Sigma; \Gamma_1 \vdash t \equiv u : A_1$.

Proof. Using Lemma 2.63, Postulate 14, Remark 2.43, and Lemma 2.86. See the proof of Lemma 4.13 in the licentiate thesis [54]. □

Example 4.14 shows that the heterogeneous equality is strictly stronger than the judgmental equality of the underlying theory:

Example 4.14 (Heterogeneous equality)**.**

$$\begin{aligned}
&\mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \alpha : \mathbb{A} \to \mathrm{Set};\\
&x : \mathbb{A} \to \mathbb{A} \ddagger (\alpha\, \mathbb{a}), z : (\alpha\, \mathbb{a}) \ddagger \mathbb{A} \to \mathbb{A}\\
&\vdash\\
&\langle x, \lambda y.z\, y\rangle \equiv\{\langle x, z\rangle\}\equiv \langle \lambda y.x\, y, z\rangle :\\
&(\alpha\, \mathbb{a} \times (\mathbb{A} \to \mathbb{A})) \ddagger ((\mathbb{A} \to \mathbb{A}) \times \alpha\, \mathbb{a})
\end{aligned}$$

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: $\Sigma \mid\!\approx\, \mathcal{C}$, $\Sigma \mid\!\approx\, \vec{\mathcal{C}}$)**.** 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 $\Sigma \mid\!\approx\, \mathcal{C}$), 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 $\Sigma \mid\!\approx\, \vec{\mathcal{C}}$), if, Σ is well-formed and for each $\mathcal{C} \in \vec{\mathcal{C}}$, $\Sigma \mid\!\approx\, \mathcal{C}$.

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, $\Theta \mid\!\approx\, \vec{\mathcal{C}}$.

When we can go in the other direction (that is, $\Theta \mid\!\approx\, \vec{\mathcal{C}}$ 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. $\Theta \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2$ **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. Using Lemma 2.69. See the proof of Lemma 4.20 in the licentiate thesis [54]. □

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. See the proof of Lemma 4.21 in the licentiate thesis [54]. □

In general terms, solving a unification problem can be done 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 explain how a type checking problem can be correctly reduced to a set of internal constraints. This elaboration is done in such a way that, for each heterogeneous constraint in the resulting unification problem, there are other constraints which make that constraint judgmentally homogeneous. This invariant will be preserved by the unification rules.

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 $\Gamma ‡ \Gamma \vdash A \approx B : \mathrm{Set} ‡ \mathrm{Set}$ and $\Gamma ‡ \Gamma \vdash u \approx v : A ‡ B$. 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 $\Theta \vdash \Gamma$ **ctx**. Also, by the SET rule and Remark 2.15 (there is only set), $\Theta; \Gamma \vdash \mathrm{Set}$ **type**, which gives $\Theta \vdash \Gamma, \mathrm{Set}$ **ctx**. By reflexivity of context equality $\Theta \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set}$ **ctx**. Because $\Theta \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set}$ **ctx** and $\Theta; \Gamma \ddagger \Gamma \vdash A \equiv B : \mathrm{Set} \ddagger \mathrm{Set}$, by Lemma 4.13 (homogenization of heterogeneous equality), $\Theta \vDash \mathcal{C}$.

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 $\Theta \vDash \vec{\mathcal{C}}$, in particular, $\Theta \vDash \vec{\mathcal{C}}'$. This means $\Theta; \Gamma \vdash A \equiv A_0 : \mathrm{Set}$ and $\Theta; \Gamma \vdash B \equiv A_0 : \mathrm{Set}$ for some term A_0. By transitivity of equality, $\Theta; \Gamma \vdash A \equiv B : \mathrm{Set}$. By Remark 2.15, $\Theta; \Gamma \vdash A \equiv B$ **type**. By reflexivity of context equality, $\Theta \vdash \Gamma \equiv \Gamma$ **ctx**. By Definition 2.16, $\Theta \vdash \Gamma, A \equiv \Gamma, B$ **ctx**. By Lemma 4.13 (homogenization of heterogeneous equality), $\Theta \vDash \mathcal{C}$.

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 elaboration 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 $\Theta \vdash \Gamma, A \equiv \Gamma, B$ **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), $\Sigma' \vdash \Gamma$ **ctx**. As in the proof of Lemma 4.23 (well-formedness of elaboration into internal constraints), $\Sigma' \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set}$ **ctx**. Because $\tilde{\Theta} \vDash \Sigma'$, we have $\tilde{\Theta} \vdash \Gamma, \mathrm{Set} \equiv \Gamma, \mathrm{Set}$ **ctx**.

  Also, by Definition 3.12 (correctness of an elaboration algorithm), $\tilde{\Theta}; \Gamma \vdash A \equiv B$ **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), $\tilde{\Theta}; \Gamma \vdash A \equiv B$ **type** and $\tilde{\Theta}; \Gamma \vdash t \equiv u : A$.

  By Lemma 2.70 (piecewise well-formedness of typing judgments), we have $\tilde{\Theta} \vdash \Gamma$ **ctx**. By reflexivity, this means $\tilde{\Theta} \vdash \Gamma \equiv \Gamma$ **ctx**. By Definition 2.16 (equality of contexts), $\tilde{\Theta} \vdash \Gamma, A \equiv \Gamma, B$ **ctx**; that is, $\tilde{\Theta} \vDash \mathcal{C}$.

Therefore, $\tilde{\Theta} \vDash \Sigma'; \vec{\mathcal{C}}$. □

4.4 Correctness of reduction rules

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 §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 $\Sigma'' \not\approx \vec{\mathcal{D}}$ then $\Sigma'' \not\approx \vec{\mathcal{C}}$.

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 ($\not\approx$), 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 ($\Sigma'' \not\approx \vec{\mathcal{C}}$). 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, $\vec{\mathcal{E}} = \overrightarrow{\mathcal{E}_1} \wedge \vec{\mathcal{C}} \wedge \overrightarrow{\mathcal{E}_2}$, $\vec{\mathcal{E}}' = \overrightarrow{\mathcal{E}_1} \wedge \overrightarrow{\mathcal{D}} \wedge \overrightarrow{\mathcal{E}_2}$, 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 $\Sigma'' \not\approx \vec{\mathcal{E}}'$ then $\Sigma'' \not\approx \vec{\mathcal{E}}$.*

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 $\Sigma;\vec{\mathcal{E}} \rightsquigarrow^\star \Sigma';\vec{\mathcal{E}}''$, using Lemma 2.155, Remark 2.152, Remark 2.137, and Remark 2.154. See the proof of Lemma 4.29 in the licentiate thesis [54]. □

In order to solve a problem $\Sigma;\vec{\mathcal{C}}$, rules are applied iteratively, stopping if it produces a signature Σ' such that $\Sigma;\vec{\mathcal{C}} \rightsquigarrow^\star \Sigma';\Box$. 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. Using Lemma 4.29, Corollary 2.149, Lemma 2.157, Lemma 4.21, Remark 2.134, Remark 4.8, Remark 4.6, Remark 2.153, Lemma 2.130, Remark 2.9, Remark 2.135, Remark 2.138, Remark 4.10 and Lemma 2.150. See the proof of Theorem 4.31 in the licentiate thesis [54]. □

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. According to Theorem 4.31 (correctness of unification), in order to show the correctness of a unification algorithm based on these rules 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 4.32 (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 4.32 are given in RuleSchema 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 4.33 (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, using Lemma 2.62, Postulate 2, Lemma 2.39, Remark 2.30. See the proof of Lemma 4.34 in the licentiate thesis [54]. □

Lemma 4.34 (General η-equality for pairs)**.** *Assume* $\Sigma; \Gamma \vdash u : \Sigma AB$*. Then* $(u\ @\ .\pi_1)\Downarrow$*,* $(u\ @\ .\pi_2)\Downarrow$ *and* $\Sigma; \Gamma \vdash u \equiv \langle u\ @\ .\pi_1, u\ @\ .\pi_2 \rangle : \Sigma AB$*.*

Proof. Using Postulate 3 (typing of hereditary projection). See the proof of Lemma 4.35 in the licentiate thesis [54]. □

The following lemma, derived from Miller's pattern condition [68], shows that a term of a function type can be characterized by the result of applying said term to distinct variables.

Lemma 4.35 (Miller's pattern condition)**.** *Let u, v be such that $\Sigma;\cdot \vdash u : \Pi\vec{A}^n.B$, $\Sigma;\cdot \vdash v : \Pi\vec{A}^n.B$ (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 $\Sigma;\cdot \vdash u \equiv v : \Pi\vec{A}^n.B$.*

Proof. Using Lemma 2.62, Lemma 2.65, Postulate 13, Remark 2.29, Lemma 2.78, Lemma 2.39, and Lemma 4.33. See the proof of Lemma 4.36 in the licentiate thesis [54]. □

The term u in Problem 4.32 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 $\Sigma \vdash \Gamma_1, B_1 \equiv \Gamma_2, B_2$ **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 4.36 (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.

$$\frac{}{\Sigma \vdash \cdot \equiv\{\cdot\}\equiv_{\emptyset,\emptyset} \cdot}\ \textsc{empty}$$

$$\frac{0 \notin X_1 \cup X_2 \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1-1,X_2-1} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, \mathrm{Set}\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{unused}$$

$$\frac{0 \in X_2 - X_1 \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1-1,(X_2-1)\cup \mathrm{FV}(A_2)} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, A_2\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used-r}$$

$$\frac{0 \in X_1 - X_2 \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{(X_1-1)\cup \mathrm{FV}(A_1),X_2-1} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, A_1\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used-l}$$

$$\frac{\begin{array}{r} a \stackrel{\text{def}}{=} \mathrm{FV}(A_1) \cap \mathrm{FV}(A_2) \\ X_1' \stackrel{\text{def}}{=} (X_1 - 1) \cup a \\ X_2' \stackrel{\text{def}}{=} (X_2 - 1) \cup a \\ \Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash A_1 \equiv\{A\}\equiv A_2 : \mathrm{Set} \ddagger \mathrm{Set} \qquad \Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{X_1',X_2'} \Gamma_2 \end{array}}{\Sigma \vdash \Gamma_1, A_1 \equiv\{\Gamma, A\}\equiv_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used}$$

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. Using such a placeholder instead of removing the variable altogether simplifies proofs by avoiding having to strengthen the terms typed in the context.

Lemma 4.37 (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 $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\mathrm{FV}(t)\cup b, \mathrm{FV}(u)\cup b} \Gamma_2$, where $b = \mathrm{FV}(B_1) \cap \mathrm{FV}(B_2)$.

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 $\Sigma; \Gamma, \Delta \vdash u : B$. (⋆)

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:

$$\frac{0 \notin X_1 \cup X_2 \qquad \Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{X_1 - 1, X_2 - 1} \Gamma_2'}{\Sigma \vdash \Gamma_1', A_1 \equiv\{\Gamma', \mathrm{Set}\}\equiv_{X_1, X_2} \Gamma_2', A_2}\ \text{UNUSED}$$

  From the premises of the rule, $0 \notin X_1$. By assumption, $\mathrm{FV}(\Delta \vdash t : B) \subseteq X_1$, which means $0 \notin \mathrm{FV}(\Delta \vdash t : B)$.

  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, $\mathrm{FV}(\mathrm{Set}, \Delta \vdash t : B) = \mathrm{FV}(\Delta \vdash t : B) - 1 \subseteq X_1 - 1$.

  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 $\mathrm{FV}(\Delta \vdash u : B) \subseteq X_2$, then $\Gamma, \Delta \vdash u : B$.

- Rule USED-L:

$$\frac{0 \in X_1 - X_2 \qquad \Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{(X_1 - 1) \cup \mathrm{FV}(A_1), X_2 - 1} \Gamma_2'}{\Sigma \vdash \Gamma_1', A_1 \equiv\{\Gamma', A_1\}\equiv_{X_1, X_2} \Gamma_2', A_2}\ \text{USED-L}$$

Assume $\Gamma_1, \Delta \vdash t : B$, i.e. $\Gamma_1', A_1, \Delta \vdash t : B$.

Because $\mathrm{FV}(\Delta \vdash t : B) \subseteq X_1$, by Definition 2.45 (free variables of a scoped and typed term), $\mathrm{FV}(A_1, \Delta \vdash t : B) = \mathrm{FV}(A_1) \cup (\mathrm{FV}(\Delta \vdash t : B) - 1) \subseteq (X_1 - 1) \cup \mathrm{FV}(A_1)$.

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, $\mathrm{FV}(\Delta \vdash u : B) \subseteq X_2$. By the premises of the rule, $0 \notin X_2$; therefore, $0 \notin \mathrm{FV}(\Delta \vdash u : B)$. By Lemma 2.71 (variables of irrelevant type), $\Gamma_2', \mathrm{Set}, \Delta \vdash u : B$.

Also from $\mathrm{FV}(\Delta \vdash u : B) \subseteq X_2$ we deduce $\mathrm{FV}(\mathrm{Set}, \Delta \vdash u : B) = \mathrm{FV}(\mathrm{Set}) \cup (\mathrm{FV}(\Delta \vdash u : B) - 1) = \mathrm{FV}(\Delta \vdash u : B) - 1 \subseteq X_2 - 1$. 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:

$$\frac{\begin{array}{c} a \stackrel{\text{def}}{=} \mathrm{FV}(A_1) \cap \mathrm{FV}(A_2) \\ X_1' \stackrel{\text{def}}{=} (X_1 - 1) \cup a \\ X_2' \stackrel{\text{def}}{=} (X_2 - 1) \cup a \\ \Sigma; \Gamma_1' \ddagger \Gamma_2' \vdash A_1 \equiv\{A\}\equiv A_2 : \mathrm{Set} \ddagger \mathrm{Set} \qquad \Sigma \vdash \Gamma_1' \equiv\{\Gamma'\}\equiv_{X_1', X_2'} \Gamma_2' \end{array}}{\Sigma \vdash \Gamma_1', A_1 \equiv\{\Gamma', A\}\equiv_{X_1, X_2} \Gamma_2', A_2} \text{ USED}$$

Assume that $\Sigma; \Gamma_1', A_1, \Delta \vdash t : B$ with $\mathrm{FV}(\Delta \vdash t : B) \subseteq X_1$.

From the first premise of the rule, we have $\Sigma; \Gamma_1' \vdash A_1 \equiv A : \mathrm{Set}$ and $\mathrm{FV}(A) \subseteq \mathrm{FV}(A_1) \cap \mathrm{FV}(A_2) = a$.

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), $\mathrm{FV}(A, \Delta \vdash t : B) = \mathrm{FV}(A) \cup (\mathrm{FV}(\Delta \vdash t : B) - 1) \subseteq a \cup (X_1 - 1)$.

By the induction hypothesis, $\Sigma; \Gamma', A, \Delta \vdash t : B$, i.e. $\Sigma; \Gamma, \Delta \vdash t : B$.

From the assumptions $\mathrm{FV}(\Delta \vdash u : B) \subseteq X_2$ 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, $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash B_1 \equiv\{B\}\equiv B_2 : \mathrm{Set}$.

Assume $\Sigma; \Gamma_1 \vdash t : B_1$. From the hypothesis, we have $\Sigma; \Gamma_1 \vdash B_1 \equiv B : \mathrm{Set}$, with $\mathrm{FV}(B) \subseteq \mathrm{FV}(B_1) \cap \mathrm{FV}(B_2) = b$. By the CONV-EQ rule, $\Sigma; \Gamma_1 \vdash t : B$.

By Definition 2.45 (free variables of a scoped and typed term), $\mathrm{FV}(\cdot \vdash t : B) = \mathrm{FV}(t) \cup \mathrm{FV}(B) \subseteq \mathrm{FV}(t) \cup b$,

Also by hypothesis, $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\text{FV}(t)\cup b,\text{FV}(u)\cup b} \Gamma_2$. 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

$$\Sigma = \Sigma_1, \alpha : A, \Sigma_2$$

$$t' = t[\vec{x} \mapsto \vec{y}]$$

$$\text{all variables in } \vec{x} \text{ are pair-wise distinct} \tag{1}$$

$$\text{FV}(t) \subseteq \vec{x} \tag{2a}$$

$$\text{CONSTS}(t) \subseteq \text{DECLS}(\Sigma_1) \tag{2b}$$

$$\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash B_1 \equiv\{B\}\equiv B_2 : \text{Set} \ddagger \text{Set} \tag{3a}$$

$$\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\{x_i \mid i=1,\ldots,n\}\cup b,\text{FV}(t)\cup b} \Gamma_2 \tag{3b}$$

$$b \stackrel{\text{def}}{=} \text{FV}(B_1) \cap \text{FV}(B_2)$$

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 4.37 (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 $\Sigma; \Gamma \vdash B \equiv B' : \text{Set}$. 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, $\text{CONSTS}(t') \subseteq \text{DECLS}(\Sigma_1)$. Because $\Sigma_1; \cdot \vdash A$ **type**, by Lemma 2.72 (no extraneous constants), $\text{CONSTS}(A) \subseteq \text{DECLS}(\Sigma_1)$. 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 $\Sigma'' \models\!\!\approx \mathcal{C}$.

(i) By rule 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, $\mathrm{FV}(t) \subseteq \vec{x} = \mathrm{FV}(\alpha\,\vec{x})$. Also, trivially, $\mathrm{FV}(t) \subseteq \mathrm{FV}(t)$.

By (v), (vi), and (vii), $\Sigma''; \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\,\vec{x}^n \equiv\!\{t\}\!\equiv t : B_1 \ddagger B_2$.

By Definition 4.17 (constraint satisfaction), $\Sigma'' \models\!\!\approx \mathcal{C}$.

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, $\Theta \vDash \Sigma$, $\Theta \vdash \Gamma_1, B_1 \equiv \Gamma_2, B_2$ **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 $\mathrm{DECLS}(\Theta) = \mathrm{DECLS}(\Sigma')$, 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 $\Sigma \vdash \Gamma_1$ **ctx** and $\Theta \vDash \Sigma$, we have $\Theta \vdash \Gamma_1$ **ctx**. Because $\alpha := u : B \in \Theta$, by the rule 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 $\alpha := u : B \in \Theta$, 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 4.35 (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 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)**.**

$$\begin{aligned} \Sigma; \Gamma \ddagger \Gamma' \vdash \Pi A B \approx \Pi A' B' : \mathrm{Set} \ddagger \mathrm{Set} \rightsquigarrow \\ \Sigma; \Gamma \ddagger \Gamma' \vdash A \approx A' : \mathrm{Set} \ddagger \mathrm{Set} \wedge \\ \Gamma \ddagger \Gamma', A \ddagger A' \vdash B \approx B' : \mathrm{Set} \ddagger \mathrm{Set} \end{aligned}$$

Proof of correctness. Using Lemma 2.52 and Postulate 10. See the proof of correctness for Rule-Schema 3 in the licentiate thesis [54]. □

Rule-Schema 4 (Injectivity of Σ)**.**

$$\begin{aligned} \Sigma; \Gamma \ddagger \Gamma' \vdash \Sigma A B \approx \Sigma A' B' : \mathrm{Set} \ddagger \mathrm{Set} \rightsquigarrow \\ \Sigma; \Gamma \ddagger \Gamma' \vdash A \approx A' : \mathrm{Set} \ddagger \mathrm{Set} \ \wedge \ \Gamma \ddagger \Gamma', A \ddagger A' \vdash B \approx B' : \mathrm{Set} \ddagger \mathrm{Set} \end{aligned}$$

Proof. We follow the same reasoning as in the proof for Rule-Schema 3, using Lemma 2.53 (Σ inversion) and Postulate 11 (injectivity of Σ). □

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. Using Remark 4.16, Lemma 2.64 and Lemma 2.63. See the proof of correctness for Rule-Schema 7 in the licentiate thesis [54]. □

Remark 4.38 (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 $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$, we get a rule $\Sigma; \vec{\mathcal{C}'} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}'}$; where for each $\mathcal{C}_i = \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A_1 \ddagger A_2$ we have $\mathcal{C}'_i \stackrel{\text{def}}{=} \Gamma_2 \ddagger \Gamma_1 \vdash u \approx t : A_2 \ddagger A_1$, and for each $\mathcal{D}_i = \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A_1 \ddagger A_2$ we have $\mathcal{D}'_i \stackrel{\text{def}}{=} \Gamma_2 \ddagger \Gamma_1 \vdash u \approx t : A_2 \ddagger A_1$.

4.5.5 Term conversion

Consider the following problem:

$$\Sigma; \cdot \ddagger \cdot \vdash \mathbb{o} \approx \lambda x.\mathbb{o}\, x : (\mathbb{A} \to \mathbb{A}) \ddagger (\mathbb{A} \to \mathbb{A}) \qquad (\star)$$
$$\Sigma = \mathbb{A} : \mathrm{Set}, \mathbb{o} : \mathbb{A} \to \mathbb{A}$$

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 $\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A'$, 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 δη-reduct) implies $\mathrm{FV}(t') \subseteq \mathrm{FV}(t)$.

Rule-Schema 8 (Term conversion)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A' \rightsquigarrow \Sigma; \Gamma \ddagger \Gamma' \vdash t' \approx u : A \ddagger A'$$
$$\textbf{where} \quad \Sigma; \Gamma \vdash t \equiv t' : A$$
$$\textsc{fv}(t) \supseteq \textsc{fv}(t')$$

Proof of correctness. Using Lemma 2.70 (piecewise well-formedness of typing judgments). See the proof of correctness for Rule-Schema 9 in the licentiate thesis [54]. □

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)**.**

$$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger A' \rightsquigarrow \Sigma; \Gamma_0 \ddagger \Gamma' \vdash t \approx u : A_0 \ddagger A'$$
$$\textbf{where} \quad \Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0 \ \textbf{ctx}$$

Proof of correctness. Using Lemma 2.63 (preservation of judgments by type conversion). See the proof of correctness for Rule-Schema 9 in the licentiate thesis [54]. □

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)**.**

$$\Sigma; \square \rightsquigarrow \Sigma'; \square$$
$$\textbf{where} \quad \Sigma \sqsubseteq \Sigma' \ \textbf{and} \ \Sigma' \sqsubseteq \Sigma \ \textbf{ctx}$$

Proof of correctness. Using Remark 2.153 and Lemma 2.157. See the proof of correctness for Rule-Schema 10 in the licentiate thesis [54]. □

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. Using Lemma 2.82 (λ inversion), Lemma 2.83 (injectivity of λ) and Postulate 10 (injectivity of Π). See the proof of correctness for Rule-Schema 11 in the licentiate thesis [54]. □

To unify a pair it suffices to unify each component individually.

Rule-Schema 12 (Pairs)**.**

$$\begin{aligned}\Sigma; \Gamma \ddagger \Gamma' \vdash \langle t_1, t_2 \rangle \approx \langle u_1, u_2 \rangle : \Sigma A B \ddagger \Sigma A' B' \rightsquigarrow \\ \Sigma; \Gamma \ddagger \Gamma' \vdash t_1 \approx u_1 : A \ddagger A' \ \wedge\ \Gamma \ddagger \Gamma' \vdash t_2 \approx u_2 : B[t_1] \ddagger B'[u_1] \\ \textbf{where} \quad B[t_1]\Downarrow \textbf{ and } B'[u_1]\Downarrow\end{aligned}$$

Proof of correctness. Using Lemma 2.84, Remark 2.133, Postulate 11, Lemma 2.85, Postulate 4 Remark 2.15. See the proof of correctness for Rule-Schema 12 in the licentiate thesis [54]. □

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 4.39 (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 4.40 (No solutions)**.** Consider the problem:

$$\Sigma, \alpha : \mathrm{Bool}, \beta : \mathrm{Bool}; \cdot \vdash \mathrm{if}\,(\lambda.\mathbb{A})\,\alpha\,\mathbb{b}\,\mathbb{c} \equiv \mathrm{if}\,(\lambda.\mathbb{A})\,\beta\,\mathbb{c}\,\mathbb{b} : \mathbb{A}$$

This problem has a solution, namely $\Theta = \Sigma, \alpha := \mathrm{true} : \mathrm{Bool}, \beta := \mathrm{false} : \mathrm{Bool}$.

Analogously to Example 4.39 (strong neutral unification), we can solve the problem by solving the following problem instead:

$$\begin{aligned}\Sigma, \alpha : \mathrm{Bool}, \beta : \mathrm{Bool}; \cdot &\vdash \lambda.\mathbb{A} \approx \lambda.\mathbb{A} : \mathrm{Bool} \to \mathbb{A}\\ \cdot &\vdash \alpha \approx \beta : \mathrm{Bool}\\ \cdot &\vdash \mathbb{c} \approx \mathbb{b} : \mathbb{A}\\ \cdot &\vdash \mathbb{b} \approx \mathbb{c} : \mathbb{A}\end{aligned}$$

However, Θ is no longer a solution of the resulting problem, as this would imply $\Theta; \cdot \vdash \mathbb{c} \equiv \mathbb{b} : \mathbb{A}$. ◀

Even when a solution is found, the solution might not be unique:

Example 4.41 (Non-unique solutions)**.** Consider the problem:

$$\begin{aligned}\Sigma, \alpha : \mathrm{Bool}, \beta : \mathrm{Bool}; \cdot &\vdash \beta \approx \mathrm{true} : \mathrm{Bool}\\ \cdot &\vdash \mathrm{if}\,(\lambda.\mathbb{A})\,\alpha\,\mathbb{c}\,\mathbb{c} \equiv \mathrm{if}\,(\lambda.\mathbb{A})\beta\,\mathbb{c}\,\mathbb{c} : \mathbb{A}\end{aligned}$$

This problem has two solutions, namely $\Theta_1 = \Sigma, \alpha := \mathrm{true} : \mathrm{Bool}, \beta := \mathrm{true} : \mathrm{Bool}$, $\Theta_2 = \Sigma, \alpha := \mathrm{false} : \mathrm{Bool}, \beta := \mathrm{true} : \mathrm{Bool}$.

Analogously to Example 4.39 (strong neutral unification), we can solve the problem by solving the following problem instead:

$$\begin{aligned}\Sigma, \alpha : \mathrm{Bool}, \beta : \mathrm{Bool}; \cdot &\vdash \beta \approx \mathrm{true} : \mathrm{Bool}\\ \cdot &\vdash \lambda.\mathbb{A} \approx \lambda.\mathbb{A} : \mathrm{Bool} \to \mathbb{A}\\ \cdot &\vdash \alpha \approx \beta : \mathrm{Bool}\\ \cdot &\vdash \mathbb{c} \approx \mathbb{c} : \mathbb{A}\\ \cdot &\vdash \mathbb{c} \approx \mathbb{c} : \mathbb{A}\end{aligned}$$

By applying Rule-Schema 1 (syntactic equality) and Rule-Schema 2 (meta-variable instantiation), we have:

$$\Sigma, \beta := \mathrm{true} : \mathrm{Bool}, \alpha := \beta : \mathrm{Bool}; \square$$

And $\textsc{close}(\Sigma, \beta := \mathrm{true} : \mathrm{Bool}, \alpha := \beta : \mathrm{Bool}) \Downarrow \Theta_1$. 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 4.39, without losing solutions, as in Example 4.40, or sacrificing uniqueness (Example 4.41).

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. Using Lemma 2.106, Lemma 2.98, Lemma 2.111, Lemma 2.155, Corollary 2.77, Corollary 2.76, Lemma 2.113, Remark 2.36, Remark 2.159, Remark 2.17, Lemma 2.163, Lemma 2.75, Postulate 10, Lemma 2.70 and Lemma 2.163. See the proof of correctness for Rule-Schema 14 in the licentiate thesis [54]. □

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 4.42 (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:

$$\begin{aligned}\Sigma, \gamma : \mathbb{A}, \alpha := \lambda x.\gamma : \mathrm{Bool} \to \mathbb{A}, \beta : \mathbb{A}; x : \mathrm{Bool} \ddagger \mathrm{Bool} \vdash \beta \approx \mathbb{F}\,(\alpha\, x) : \mathbb{A} \ddagger \mathbb{A} \\ \cdot \vdash \alpha\, \mathrm{true} \approx \mathbb{o} : \mathbb{A} \ddagger \mathbb{A}\end{aligned}$$

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 4.43 (Bad pruning)**.** The approach in Example 4.42 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 4.42, while avoiding cases such as Example 4.43.

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 4.44 (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 $\beta \notin \textsc{decls}(\Sigma)$; and

(iv) $\Sigma_1; \cdot \vdash T \equiv \Pi \vec{A}^{n-1} \Pi B U^{(+1)}$ **type** for some B.

Lemma 4.45 (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. Using Postulate 13, Remark 2.5, Lemma 2.70, Lemma 2.52, Corollary 2.156, Lemma 2.69 and Lemma 2.62. See the proof of Lemma 4.46 in the licentiate thesis [54]. □

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 4.46 (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. Using Lemma 2.75, Lemma 2.70, Lemma 2.69, Corollary 2.58, Postulate 13, Remark 2.142, Lemma 2.130, Remark 2.137, Lemma 4.33, Lemma 2.59, Lemma 2.62 and Lemma 2.155. See the proof of Lemma 4.47 in the licentiate thesis [54]. □

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 4.47 (Intersection)**.** *Assume that* $\Theta; \Gamma \vdash \alpha\, \vec{f} \equiv \alpha\, \overrightarrow{f'} : A$*, 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* $f \in \overrightarrow{f'}$*,* 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* $0 \notin \mathrm{FV}(v)$*.*

Proof. Using Lemma 2.70, Lemma 2.112, Lemma 4.33, Lemma 2.166, Lemma 2.86, Lemma 2.62, Remark 2.30, Remark 2.35, Lemma 2.79 and Lemma 2.75. See the proof of Lemma 4.48 in the licentiate thesis [54]. □

Rule-Schema 15 (Metavariable intersection)**.**

$$\begin{aligned}
&\Sigma; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}^{\,n}\, x\, \vec{g}^{\,m} \approx \alpha\, \overrightarrow{f'}^{\,n}\, y\, \overrightarrow{g'}^{\,m} : A \ddagger A' \\
&\rightsquigarrow\ \Sigma'; \Gamma \ddagger \Gamma' \vdash \beta\, \vec{f}\, \vec{g} \approx \beta\, \overrightarrow{f'}\, \overrightarrow{g'} : A \ddagger A' \\
&\quad \textbf{where} \\
&x \neq y && (1) \\
&\text{all terms in } \vec{f},\ \overrightarrow{f'},\ \vec{g},\ \overrightarrow{g'} \text{ are irreducible} && (2) \\
&\Sigma \vdash \textsc{kill}(\alpha, n+1) \mapsto \Sigma', \text{ where } \Sigma = \Sigma_1, \alpha : U, \Sigma_2 \\
&\qquad \text{and } \Sigma' = \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2 && (3)
\end{aligned}$$

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. Using Lemma 4.45, Lemma 2.155, Lemma 2.70, Lemma 2.69, Lemma 4.47, Lemma 4.46, Remark 2.134 and Lemma 2.63. See the proof of correctness for Rule-Schema 15 in the licentiate thesis [54]. □

The proof above (via Lemma 4.47), 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)**.**

$$\begin{aligned}
&\Sigma_1, \alpha : U, \Sigma_2; \Gamma \ddagger \Gamma' \vdash \alpha\, \vec{f}^{\,n} \approx \alpha\, \vec{g}^{\,n} : A \ddagger A' \ \rightsquigarrow \\
&\Sigma_1, \beta : T, \alpha := u : U, \Sigma_2; \Gamma \ddagger \Gamma' \vdash \beta\, \vec{f}_{1,\ldots,i-1}\, \vec{f}_{i+1,\ldots,n} \approx \beta\, \vec{g}_{1,\ldots,i-1}\, \vec{g}_{i+1,\ldots,n} : A \ddagger A' \\
&\qquad \textbf{where} \\
&\qquad i \in \{1, \ldots, n\} \\
&\qquad f_i = h\, \vec{e},\ g_i = h'\, \vec{e'},\ h \neq h' \\
&\qquad \text{all terms in } \vec{f} \text{ and } \vec{g} \text{ are irreducible} \\
&\qquad \Sigma_1, \alpha : U, \Sigma_2 \vdash \textsc{kill}(\alpha, i) \mapsto \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2
\end{aligned}$$

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 4.48 (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. Using Lemma 2.170, Lemma 2.112, Lemma 2.166, Lemma 2.86, Remark 2.88, Remark 2.43, Lemma 2.172, Postulate 6, Lemma 2.75 and Lemma 2.175. See the proof of Lemma 4.49 in the licentiate thesis [54]. □

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

$t[\![\alpha\ \overrightarrow{f_1}^n\ (y\ \vec{e})\ \overrightarrow{f_2}^m]\!]^k$ for $k \in \mathbb{N}$ (1)

$y \notin \mathrm{FV}(v)$ (2)

every $f \in \overrightarrow{f_1}$ or $f \in \overrightarrow{f_2}$ is irreducible (3)

$\Sigma \vdash \mathrm{KILL}(\alpha, n+1) \mapsto \Sigma'$, where $\Sigma = \Sigma_1, \alpha : U, \Sigma_2$
and $\Sigma' = \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2$ (4)

Proof of correctness. Using Lemma 4.45, Lemma 2.155, Remark 4.7, Lemma 4.48, Lemma 4.46, Remark 2.134 and Lemma 2.69. See the proof of correctness for Rule-Schema 17 in the licentiate thesis [54]. □

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 [2] 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

$$\beta \text{ is fresh for } \Sigma$$
$$\Sigma_1; \cdot \vdash T \equiv \Pi \vec{A}^n \Pi(\Sigma U V) B \textbf{ type}$$
$$B((+2)+1)[\langle 1, 0\rangle/0] \Downarrow B'$$
$$T_\beta = \Pi \vec{A}^n \Pi U \Pi V B'$$
$$t_\alpha = \lambda \vec{x}^n\, y.\beta\, \vec{x}^n\, (y\, .\pi_1)\, (y\, .\pi_2)$$

Proof of correctness. Using Lemma 2.70, Lemma 2.52, Lemma 2.53, Lemma 2.62, Postulate 1, Lemma 2.69, Lemma 2.40, Postulate 5, Postulate 4, Definition 2.151, Corollary 2.156, Lemma 2.130, Remark 2.129, Remark 2.73, Lemma 2.62, Postulate 2, Lemma 2.141, Definition 2.122, Remark 2.131, Remark 2.15, Postulate 9, Postulate 6 and Lemma 4.33. See the proof of correctness for Rule-Schema 18 in the licentiate thesis [54]. □

Remark. Rule-Schema 18 allows a unification algorithm to solve some constraints where metavariables are applied to pairs, i.e. constraints of forms such as $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, \langle x, y\rangle \approx t : A_1 \ddagger A_2$. Cases where metavariables are applied to λ-abstractions, such as constraints of the form $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash \alpha\, (\lambda x.y\, x) \approx t : A_1 \ddagger A_2$, may sometimes be tackled by using Rule-Schema 8 (term conversion) to η-contract the argument, thus hopefully bringing the constraint into the pattern fragment. We expect that in our setting, due to the complete absence of subtyping, η-contraction is type-preserving, and all well-typed terms are judgmentally equal to their η-contracted form, when it exists.

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 [2] show how these eliminators can be removed.

Example 4.49. Consider the following problem:

$$\mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \mathbb{b} : \mathbb{A}, \alpha : \mathbb{A} \to \mathbb{A} \to \mathbb{A} \times \mathbb{A};$$
$$x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha\, x\, y\, .\pi_1 \approx x : \mathbb{A} \ddagger \mathbb{A},$$
$$x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha\, x\, y\, .\pi_2 \approx y : \mathbb{A} \ddagger \mathbb{A}$$

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:

$$\begin{array}{l}
\mathbb{A} : \mathrm{Set}, \mathtt{a} : \mathbb{A}, \mathtt{b} : \mathbb{A}, \\
\alpha_1 : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \\
\alpha_2 : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \\
\alpha := \lambda x.\lambda y.\langle \alpha_1\ x\ y, \alpha_2\ x\ y \rangle : \mathbb{A} \to \mathbb{A} \to \mathbb{A} \times \mathbb{A}; \\
x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha_1\ x\ y \approx x : \mathbb{A} \ddagger \mathbb{A}, \\
x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{A} \ddagger \mathbb{A} \vdash \alpha_2\ x\ y \approx y : \mathbb{A} \ddagger \mathbb{A}
\end{array}$$

Then Rule-Schema 2 can be applied both constraints.

◀

Example 4.50. Consider the following unification problem:

$$\begin{array}{l}
\mathbb{A} : \mathrm{Set}, \mathtt{a} : \mathbb{A}, \\
\mathbb{B} : \mathbb{A} \to \mathrm{Set}, \mathtt{b} : \mathbb{B}\ \mathtt{a}, \\
\alpha : \mathbb{A} \to \mathbb{B}\ \mathtt{a} \to \Sigma \mathbb{A}(\mathbb{B}\ \mathtt{a}); \\
\cdot \vdash \alpha\ \mathtt{a}\ \mathtt{b}\ .\pi_1 \approx \mathtt{a} : \mathbb{A} \ddagger \mathbb{A} \wedge \\
x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{B}\ \mathtt{a} \ddagger \mathbb{B}\ \mathtt{a} \vdash \alpha\ x\ y \approx \langle x, y \rangle : \Sigma \mathbb{A}(\mathbb{B}\ \mathtt{a}) \ddagger \Sigma \mathbb{A}(\mathbb{B}\ (\alpha\ \mathtt{a}\ \mathtt{b}\ .\pi_1))
\end{array}$$

In order to instantiate α by applying Rule-Schema 2 (metavariable instantiation), we need to have $\Sigma; \Gamma \ddagger \Gamma' \vdash \Sigma \mathbb{A}(\mathbb{B}\ \mathtt{a}) \equiv \Sigma \mathbb{A}(\mathbb{B}\ (\alpha\ \mathtt{a}\ \mathtt{b}\ .\pi_1)) : \mathrm{Set} \ddagger \mathrm{Set}$.

By η-expanding α as in the previous example and then applying Rule-Schema 12 (pairs), we obtain the following problem:

$$\begin{array}{l}
\mathbb{A} : \mathrm{Set}, \mathtt{a} : \mathbb{A}, \\
\mathbb{B} : \mathbb{A} \to \mathrm{Set}, \mathtt{b} : \mathbb{B}\ \mathtt{a}, \\
\alpha_1 : \mathbb{A} \to \mathbb{B}\ \mathtt{a} \to \mathbb{A}, \\
\alpha_2 : \mathbb{A} \to \mathbb{B}\ \mathtt{a} \to \mathbb{B}\ \mathtt{a}, \\
\alpha := \lambda x.\lambda y.\langle \alpha_1\ x\ y, \alpha_2\ x\ y \rangle : \mathbb{A} \to \mathbb{B}\ \mathtt{a} \to \Sigma \mathbb{A}(\mathbb{B}\ \mathtt{a}); \\
\cdot \vdash \alpha_1\ \mathtt{a}\ \mathtt{b} \approx \mathtt{a} : \mathbb{A} \wedge \\
x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{B}\ \mathtt{a} \ddagger \mathbb{B}\ \mathtt{a} \vdash \alpha_1\ x\ y \approx x : \mathbb{A} \ddagger \mathbb{A} \wedge \\
x : \mathbb{A} \ddagger \mathbb{A}, y : \mathbb{B}\ \mathtt{a} \ddagger \mathbb{B}\ \mathtt{a} \vdash \alpha_2\ x\ y \approx y : \mathbb{B}\ \mathtt{a} \ddagger \mathbb{B}\ (\alpha_1\ \mathtt{a}\ \mathtt{b})
\end{array}$$

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 4.49 and 4.50 may be generalized as the following rule:

Rule-Schema 19 (Metavariable η-expansion)**.**

$$\Sigma_1, \alpha : U, \Sigma_2; \Box \rightsquigarrow$$
$$\Sigma_1, \alpha_1 : \Pi \vec{T}.A, \alpha_2 : \Pi \vec{T}.B[\alpha_1\ \vec{x}], \alpha := \lambda \vec{x}^n.\langle \alpha_1\ \vec{x}, \alpha_2\ \vec{x} \rangle : U, \Sigma_2; \Box$$
where
$$\Sigma = \Sigma_1, \alpha : U, \Sigma_2$$
$$\Sigma_1; \cdot \vdash U \equiv \Pi \vec{T}^n.\Sigma AB \textbf{ type}$$
$$\alpha_1 \text{ and } \alpha_2 \text{ fresh for } \Sigma,\ \alpha_1 \neq \alpha_2$$

Remark. Rule-Schema 19 may be applied to any metavariable, regardless of whether it occurs in a constraint or not.

Proof of correctness. Using Lemma 2.70 (piecewise well-formedness of typing judgments), Lemma 2.52 (Π inversion), Lemma 2.53 (Σ inversion), Remark 2.15 (there is only set), Postulate 1 (typing of hereditary substitution), Lemma 2.69 (signature weakening), Corollary 2.156 (horizontal composition of extensions), Lemma 2.130 (alternative characterization of a compatible metasubstitution), Remark 2.129 (alternative characterization of compatibility of a metasubstitution with a declaration), Lemma 2.62 (context weakening), Postulate 2 (typing of hereditary application), Postulate 3 (typing of hereditary projection), Lemma 2.141, Remark 2.131, Postulate 7 (congruence of hereditary projection), Remark 2.137 (metasubstitution weakening), Postulate 4 (congruence of hereditary substitution), Postulate 6 (congruence of hereditary application), Lemma 2.75 (uniqueness of typing for neutrals), Lemma 4.34 (general η-equality for pairs) and Lemma 4.33 (general η-equality for Π-types). See the proof of correctness for Rule-Schema 19 in the licentiate thesis [54]. □

4.5.12 Context variable currying

In the same way that one can remove Σ-types from metavariable arguments, Abel and Pientka [2] show how one can remove Σ-types from constraint contexts. This will help us instantiate metavariables whose arguments contain projections.

Example 4.51. Consider the following problem:

$$\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B};$$
$$x : (\mathbb{A} \times (\mathbb{A} \to \mathbb{B})) \ddagger (\mathbb{A} \times (\mathbb{A} \to \mathbb{B})) \vdash \alpha\ (x\ .\pi_1)\ (x\ .\pi_2) \approx (x\ .\pi_2)\ (x\ .\pi_1) : \mathbb{B} \ddagger \mathbb{B}$$

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:

$$\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B};$$
$$x_1 : \mathbb{A} \ddagger \mathbb{A}, x_2 : (\mathbb{A} \to \mathbb{B}) \ddagger (\mathbb{A} \to \mathbb{B}) \vdash \alpha\ x_1\ x_2 \approx x_2\ x_1 : \mathbb{B} \ddagger \mathbb{B}$$

By applying rule Rule-Schema 2, we have

$$\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha := \lambda x.\lambda y.(y\ x) : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B}; \Box$$

◀

The technique in Example 4.51 can be generalized as the following rule:

Rule-Schema 20 (Context variable currying)**.**

$$\Sigma; \Gamma_1 \ddagger \Gamma_2, x : \Sigma U_1 V_1 \ddagger \Sigma U_2 V_2, \Delta_1 \ddagger \Delta_2 \vdash t_1 \approx t_2 : A_1 \ddagger A_2 \rightsquigarrow$$
$$\Sigma; \Gamma_1 \ddagger \Gamma_2, x_1 : U_1 \ddagger U_2, x_2 : V_1 \ddagger V_2, \Delta_1' \ddagger \Delta_2' \vdash t_1' \approx t_2' : A_1' \ddagger A_2' \quad \textbf{where}$$
$$(\Delta_1 \vdash t_1 : A_1)^{((+2)+1)}[\langle 1, 0\rangle/0] \Downarrow (\Delta_1' \vdash t_1' : A_1')$$
$$(\Delta_2 \vdash t_2 : A_2)^{((+2)+1)}[\langle 1, 0\rangle/0] \Downarrow (\Delta_2' \vdash t_2' : A_2')$$

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 4.52 (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 4.53 (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. Using Lemma 2.70, Lemma 2.53, Lemma 2.62, Postulate 1, Remark 2.13, Lemma 2.62, Lemma 2.40, Postulate 4, Remark 2.49, Postulate 5, Lemma 2.63, Lemma 2.51, Remark 2.28, Lemma 4.52, Lemma 4.53, Postulate 11 and Remark 2.133. See the proof of correctness for Rule-Schema 20 in the licentiate thesis [54]. □

Remark 4.54. 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 Example of constraint solving

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 give a concrete example of constraints where flexibility in the order in which constraints are solved help find a solution. This example was used in previous case study [54,

§5.6] to motivate our prototype for unification with twin types, Tog$^+$ [56]. We will use this example as a starting point when comparing our approach to that of other systems (§6.6).

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.

$$\begin{array}{lll}
\mathtt{U} & \stackrel{\text{def}}{=} \text{Bool} \times \text{Bool} & : \text{Set} \\
\mathtt{set} & \stackrel{\text{def}}{=} \langle \text{true}, \text{false} \rangle & : \mathtt{U} \\
\mathtt{el}\ \ (b : \text{Bool}) & \stackrel{\text{def}}{=} \langle \text{false}, b \rangle & : \mathtt{U} \\
\mathtt{El}\ \ (u : \mathtt{U}) & \stackrel{\text{def}}{=} \text{if}\ (\lambda.\text{Bool})\ (u\ .\pi_1)\ \text{true}\ (u\ .\pi_2) & : \text{Bool}
\end{array}$$

These shorthands mimic some sort of inductive data type (`U`) with two constructors (`set`, `el`) and an inductively-defined function of type $\mathtt{U} \to \text{Bool}$ (i.e. `El`). Although we implement our approach in a type-checker which supports inductive datatypes, we use these abbreviations in this section so that we can keep the discussion within the syntax of our language as defined in §2.1.

With these constants declared, we can now introduce Example 4.55.

Example 4.55 (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.

$$\mathbb{F} : \text{Bool} \to \text{Set}, \mathbb{P} : (X : \text{Set}) \to (x : X) \to \text{Set}, \alpha : \text{Bool} \to \mathtt{U}\ \mathbf{;}$$
$$x : \text{Bool} \vdash \mathbb{P}\ (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U})\ (\lambda y.\mathtt{set}) : \text{Set} \cong \mathbb{P}\ (\mathbb{F}\ \text{true} \to \mathtt{U})\ (\lambda y.(\alpha\, x)) : \text{Set}$$

◀

By Definition 4.22, the constraint in Example 4.55 decomposes into two internal constraints, yielding the problem $\Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)}$, where:

$$\begin{array}{l}
\Sigma^{(0)} \stackrel{\text{def}}{=} \mathbb{F} : \text{Bool} \to \text{Set}, \mathbb{P} : (X : \text{Set}) \to (x : X) \to \text{Set}, \alpha : \text{Bool} \to \mathtt{U} \\
\mathcal{C}_1^{(0)} \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool} \vdash \text{Set} \approx \text{Set} : \text{Set} \ddagger \text{Set} \\
\mathcal{C}_2^{(0)} \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool} \vdash \mathbb{P}\ (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U})\ (\lambda y.\mathtt{set}) \approx \\
\qquad\qquad \mathbb{P}\ (\mathbb{F}\ \text{true} \to \mathtt{U})\ (\lambda y.(\alpha\, x)) : \text{Set} \ddagger \text{Set}
\end{array}$$

The constraints can be refined 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:

$$\begin{array}{l}
\mathcal{C}_1^{(1)} \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool} \vdash \mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U} \approx \mathbb{F}\ \text{true} \to \mathtt{U} : \text{Set} \ddagger \text{Set} \\
\mathcal{C}_2^{(1)} \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool} \vdash (\lambda y.\mathtt{set}) \approx (\lambda y.(\alpha\, x)) : \\
\qquad\qquad (\mathbb{F}\ (\mathtt{El}\ (\alpha\, x)) \to \mathtt{U}) \ddagger (\mathbb{F}\ \text{true} \to \mathtt{U})
\end{array}$$

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:

$$\mathcal{C}_1^{(2)} \stackrel{\text{def}}{=} x : \text{Bool}‡\text{Bool} \vdash \mathbb{F}\ (\texttt{El}\ (\alpha\ x)) \approx \mathbb{F}\ \text{true} : \text{Set}‡\text{Set}$$
$$\mathcal{C}_2^{(2)} \stackrel{\text{def}}{=} x : \text{Bool}‡\text{Bool}, y : \mathbb{F}\ (\texttt{El}\ (\alpha\ x))‡\mathbb{F}\ \text{true} \vdash \texttt{U} \approx \texttt{U} : \text{Set}‡\text{Set}$$

   By Rule-Schema 11 (λ-abstraction), $\Sigma^{(0)}; \mathcal{C}_2^{(1)} \rightsquigarrow \Sigma^{(0)}; \mathcal{C}_3^{(2)}$, where:

$$\mathcal{C}_3^{(2)} \stackrel{\text{def}}{=} x : \text{Bool}, y : (\mathbb{F}\ (\texttt{El}\ (\alpha\ x)))‡(\mathbb{F}\ \text{true}) \vdash \texttt{set} \approx \alpha\ x : \texttt{U}‡\texttt{U}$$

   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:

$$\mathcal{C}_1^{(3)} \stackrel{\text{def}}{=} x : \text{Bool}‡\text{Bool} \vdash \texttt{El}\ (\alpha\ x) \approx \text{true} : \text{Bool}‡\text{Bool}$$

   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:

$$\mathcal{C}_1^{(4)} \stackrel{\text{def}}{=} x : \text{Bool}‡\text{Bool} \vdash \text{true} \approx \text{true} : \text{Bool}‡\text{Bool}$$

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:

$$\Theta = \mathbb{F} : \text{Bool} \to \text{Set}, \mathbb{P} : (X : \text{Set}) \to (x : X) \to \text{Set}, \alpha := \lambda x.\texttt{set} : \text{Bool} \to \texttt{U}$$

4.7 Limitations of the rule toolkit

Here we describe a set of constraints which cannot be solved by unification using twin types, despite being in the pattern fragment, and which arise from Agda code written by Danielsson [24]. The fact that these constraints cannot be solved means that the user needs to supply some additional information (i.e. give some implicit arguments explicitly) in order for the example to be recognized as well-typed. We discuss how common this problem is and possible wais to mitigate it in §6.5.

For clarity, we use the letters A, P and p as variable names, even though they are not nominated as such. We also de-duplicate twin types where the two sides are identical: e.g. Set denotes the twin type Set‡Set.

Consider the following signature:

$$
\begin{aligned}
\Sigma_0 \stackrel{\text{def}}{=}\ & \mathbb{op} : (A : \mathrm{Set}) \to (P : \mathrm{Set}) \to ((x : A) \to P\,x) \to ((x : A) \to P\,x) \\
& \mathbb{eq} : (A : \mathrm{Set}) \to A \to A \to \mathrm{Set} \\
& \mathbb{refl} : (A : \mathrm{Set}) \to (x : A) \to \mathbb{eq}\,A\,x\,x \\
& \mathbb{elim} : (A : \mathrm{Set}) \to (x : A) \to (y : A) \to \\
& \qquad (P : (x' : A) \to (y' : A) \to \mathbb{eq}\,A\,x'\,y' \to \mathrm{Set}) \to \\
& \qquad ((x'' : A) \to P\,x''\,x''\,(\mathbb{refl}\,A\,x'')) \to (z : \mathbb{eq}\,A\,x\,y) \to P\,x\,y\,z
\end{aligned}
$$

$$
\begin{aligned}
\Sigma \stackrel{\text{def}}{=} \Sigma_0, & \alpha : (A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\,x \to P\,x \to P\,x \\
& \beta : (A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\,x \to P\,x \to P\,x \\
& \gamma : (A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\,x \to \mathrm{Set} \\
& \delta : (A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\,x \to \gamma\,A\,x\,P\,p \\
& \varepsilon : (A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\,x \to \\
& \qquad (x' : \gamma\,A\,x\,P\,p) \to (y' : \gamma\,A\,x\,P\,p) \to \mathbb{eq}\,(\gamma\,A\,x\,P\,p)\,x'\,y' \to \mathrm{Set} \\
& \varphi : (A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\,x \to \\
& \qquad (x'' : (\gamma\,A\,x\,P\,p)) \to P\,x''\,x''\,(\mathbb{refl}\,(\gamma\,A\,x\,P\,p)\,x'')
\end{aligned}
$$

Let $\Gamma \stackrel{\text{def}}{=} A : \mathrm{Set}, x : A, P : A \to \mathrm{Set}, p : P\,x$, and $\vec{y} \stackrel{\text{def}}{=} A\,x\,P\,p$ in Γ (that is, $\vec{y} \stackrel{\text{def}}{=} 3\,2\,1\,0$).

Consider the following constraints:

$$
\begin{aligned}
\mathcal{C}_1 &\stackrel{\text{def}}{=} \Gamma \vdash \varepsilon\,\vec{y}\,(\delta\,\vec{y})\,(\delta\,\vec{y})\,(\mathbb{refl}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})) \approx Px \to Px : \mathrm{Set} \\
\mathcal{C}_2 &\stackrel{\text{def}}{=} \Gamma \vdash \alpha\,\vec{y}\,p \approx \mathbb{elim}\,A\,x\,x\,(\lambda y.\lambda z.\lambda.(P\,y \to P\,z))\,(\lambda.\lambda z.z)\,(\mathbb{refl}\,A\,x)\,p : P\,x \\
\mathcal{C}_3 &\stackrel{\text{def}}{=} \Gamma \vdash \beta\,\vec{y}\,p \approx p : P\,x \\
\mathcal{C}_4 &\stackrel{\text{def}}{=} \Gamma \vdash \mathbb{elim}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})\,(\delta\,\vec{y})\,(\varepsilon\,\vec{y})\,(\varphi\,\vec{y})\,(\mathbb{refl}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})) \approx \alpha\,\vec{y} : \\
&\qquad (\varepsilon\,\vec{y}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})\,(\delta\,\vec{y})\,(\mathbb{refl}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})))‡(P\,x \to P\,x) \\
\mathcal{C}_5 &\stackrel{\text{def}}{=} \Gamma \vdash \varphi\,\vec{y}\,(\delta\,\vec{y}) \approx \beta\,\vec{y} : \\
&\qquad (\varepsilon\,\vec{y}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})\,(\delta\,\vec{y})(\mathbb{refl}\,(\gamma\,\vec{y})\,(\delta\,\vec{y})))‡(P\,x \to P\,x)
\end{aligned}
$$

As they stand, there is no clear way to make progress using only the rules in §4.5. The arguments to the metavariables ε, α and β in, respectively, $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ are not applied to distinct variables. In $\mathcal{C}_1$, some of the arguments are not even variables; while in $\mathcal{C}_2$ and $\mathcal{C}_3$, all of the duplicated variables occur in the other side of the constraint, so Rule-Schema 17 (metavariable pruning) cannot help. In the case of $\mathcal{C}_4$ and $\mathcal{C}_5$, the two sides of the constraint have different types, thus Rule-Schema 2 (metavariable instantiation) cannot be applied. Ignoring the precondition (3a) would result in the following ill-typed

signature (where the $''$ mark means that the type of that metavariable remains unchanged):

$$\begin{aligned}\Sigma' \stackrel{\text{def}}{\equiv} \Sigma_0, \beta : '', \gamma : '', \delta : '', \varepsilon : '', \varphi : '',\\ \alpha := \lambda \vec{y}.\mathtt{elim}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\varepsilon\ \vec{y})\ (\varphi\ \vec{x})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})) : ''\end{aligned}$$

We do not have Σ' **sig**. If this were the case, from the definition of well-formed signature, we would need to have, in particular:

$$\begin{aligned}&\beta : '', \gamma : '', \delta : '', \varepsilon : '', \varphi : ''; \cdot \vdash\\ &\lambda \vec{y}.\mathtt{elim}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\varepsilon\ \vec{y})\ (\varphi\ \vec{x})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})) :\\ &(A : \mathrm{Set}) \to (x : A) \to (P : A \to \mathrm{Set}) \to P\ x \to P\ x \to P\ x\end{aligned}$$

By Corollary 2.58 and Postulate 10, this would entail:

$$\begin{aligned}&\beta : '', \gamma : '', \delta : '', \varepsilon : '', \varphi : ''; A : \mathrm{Set}, x : A, P : A \to \mathrm{Set}, z : P\ x \vdash\\ &\mathtt{elim}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\varepsilon\ \vec{y})\ (\varphi\ \vec{x})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})) : P\ x \to P\ x\end{aligned}$$

By the typing rules, we have:

$$\begin{aligned}&\beta : '', \gamma : '', \delta : '', \varepsilon : '', \varphi : ''; A : \mathrm{Set}, x : A, P : A \to \mathrm{Set}, z : P\ x \vdash\\ &\mathtt{elim}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\varepsilon\ \vec{y})\ (\varphi\ \vec{x})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})) :\\ &(\varepsilon\ \vec{y}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})))\end{aligned}$$

By Lemma 2.75, we have that $P\ x \to P\ x$ and $\varepsilon\ \vec{y}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y}))$ would need to be equal as types. However, these types cannot become equal until the metavariable ε is instantiated.

In this specific case, the (at this point) ill-typed instantiation is what it takes to solve the constraints. Applying Rule-Schema 2 to $\mathcal{C}_4$ (ignoring precondition (3a)) and then simplifying results (Rule-Schema 8) we obtain the following:

$$\begin{aligned}&\mathcal{C}_1 \stackrel{\text{def}}{\equiv} ''\\ &\mathcal{C}'_2 \stackrel{\text{def}}{\equiv} \Gamma \vdash \mathtt{elim}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\varepsilon\ \vec{y})\ (\varphi\ \vec{x})\ (\mathtt{refl}\ (\gamma\ \vec{y})\ (\delta\ \vec{y}))\ \vec{y}\ p \approx\\ &\qquad\qquad \mathtt{elim}\ A\ x\ x\ (\lambda y.\lambda z.\lambda.(P\ y \to P\ z))\ (\lambda.\lambda z.z)\ (\mathtt{refl}\ A\ x)\ p : P\ x\\ &\mathcal{C}_3 \stackrel{\text{def}}{\equiv} ''\\ &\mathcal{C}_5 \stackrel{\text{def}}{\equiv} ''\end{aligned}$$

Then, (i) applying Rule-Schema 14 (strongly neutral terms) to $\mathcal{C}'_2$, then (ii) using Rule-Schema 2 on the first, second and fourth constraints (and Rule-Schema 1 on the remainder), and finally (iii) simplifying the constraints (Rule-Schema 8 and Rule-Schema 9), we obtain the problem $\Sigma''; \mathcal{C}'_1, \mathcal{C}_3, \mathcal{C}'_5$, where:

$$
\begin{aligned}
\Sigma'' \stackrel{\text{def}}{=} \Sigma_0,\\
&\beta : '', \gamma := \lambda\vec{y}.A : '', \delta := \lambda\vec{y}.x,\\
&\varepsilon := (\lambda\vec{y}.\lambda z_1.\lambda z_2.\lambda.(P\ z_1 \to P\ z_2)) : '',\\
&\varphi := \lambda\vec{y}.\lambda.\lambda z.z : '',\\
&\alpha := \lambda\vec{y}.\mathbb{elim}\ (\gamma\ \vec{y})\ (\delta\ \vec{y})\ (\delta\ \vec{y})\ (\varepsilon\ \vec{y})\ (\varphi\ \vec{x})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}'_1 &\stackrel{\text{def}}{=} \Gamma \vdash P\ x \to P\ x \approx P\ x \to P\ x : ''\\
\mathcal{C}_3 &\stackrel{\text{def}}{=} ''\\
\mathcal{C}'_5 &\stackrel{\text{def}}{=} \Gamma \vdash \lambda x.x \approx \beta\ \vec{y} : (P\ x \to P\ x) \ddagger (P\ x \to P\ x)
\end{aligned}
$$

By applying Rule-Schema 2 (metavariable instantiation) we obtain the body of β. Constraints $\mathcal{C}'_1$ and $\mathcal{C}_3$ become trivially solvable after simplification, by applying Rule-Schema 1. By reordering and normalizing the signature (Rule-Schema 10), we obtain a well-formed, closed signature:

$$
\begin{aligned}
\Sigma''' \stackrel{\text{def}}{=} \Sigma_0,\\
&\alpha := \lambda\vec{y}.\mathbb{elim}\ A\ x\ x\ (\lambda z_1.\lambda z_2.\lambda.(P\ z_1 \to P\ z_2))\ (\lambda.\lambda z.z) : '',\\
&\beta := \lambda\vec{y}.\lambda x.x : '',\\
&\gamma := \lambda\vec{y}.A : '',\\
&\delta := \lambda\vec{y}.x : '',\\
&\varepsilon := \lambda\vec{y}.\lambda z_1.\lambda z_2.\lambda.(P\ z_1 \to P\ z_2) : '',\\
&\varphi := \lambda\vec{y}.\lambda.\lambda z.z : '',
\end{aligned}
$$

However, finding this closed signature involved ignoring one of the preconditions of Rule-Schema 2. It is not straightforward how this "leaps of faith" can be formalized in the theoretical framework we use.

4.8 Beyond correctness

In §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 §4.8.1.

On the other hand, the correctness theorem only partially specifies conditions under which a solution exists. As explained in §3.4, whether a solution exists is undecidable in general. However, we can also partially characterize those conditions under which the original problem is unsolvable (§4.8.2).

4.8.1 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 [50]. 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 §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. □

4.8.2 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 $\Sigma; \vec{\mathcal{C}}$ is unsolvable if there does not exist Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$.

Correct rules preserve problem unsolvability. This means that a unification algorithm may use the rules in §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 $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^n \Sigma'; \overrightarrow{\mathcal{D}}$, and there is no solution Θ' such that $\Theta' \vDash \Sigma'; \overrightarrow{\mathcal{D}}$, then there is no solution Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$.*

Proof. Proceed by induction on n.

- Case 0: Proceed by contradiction; assume there exists Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$. Then $\Theta' = \Theta$ fulfills $\Theta' \vDash \Sigma'; \overrightarrow{\mathcal{D}}$, which is a contradiction. Therefore, there is no Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$.
- Case $1 + n$: Then we have $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^n \Sigma''; \vec{\mathcal{E}} \rightsquigarrow \Sigma'; \overrightarrow{\mathcal{D}}$. 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 $\Sigma; \vec{\mathcal{C}}$ has a solution is undecidable (§3.4). 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:*

$$
\begin{aligned}
T_1 &\stackrel{\text{def}}{=} \{c\} \\
T_2 &\stackrel{\text{def}}{=} \{f \mid f \textit{ is strongly neutral}\} \\
T_3 &\stackrel{\text{def}}{=} \{\Pi AB\} \\
T_4 &\stackrel{\text{def}}{=} \{\Sigma AB\} \\
T_5 &\stackrel{\text{def}}{=} \{\text{Bool}\} \\
T_6 &\stackrel{\text{def}}{=} \{\text{Set}\}
\end{aligned}
$$

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.

$$\cdot \;;\; \cdot \vdash \text{true} \approx \text{false} : \text{Bool}$$

◀

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.

$$\alpha : \text{Bool} \to \text{Bool} \;;\; \cdot \vdash \alpha\, \text{true} \approx \text{true} : \text{Bool} \wedge \cdot \vdash \alpha\, \text{true} \approx \text{false} : \text{Bool}$$

◀

4.9 Extensibility and limitations

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.

4.9.1 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 $\alpha : \Pi\vec{A}^n.\mathrm{Unit}$, one can instantiate it directly as $\alpha := \lambda\vec{x}^n.\langle\rangle : \Pi\vec{A}^n.\mathrm{Unit}$. 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 $\cdot; x : \mathrm{Bool} \to \mathrm{Unit} \vdash x\,\mathrm{true} \equiv x\,\mathrm{false} : \mathrm{Unit}$ without $\mathrm{true} = \mathrm{false}$, 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 [4], 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 keep the existing support of singleton types in Agda so that the existing Agda code keeps working (§5.11).

4.10 Concluding remarks

In this chapter we have described a set of rules for solving unification constraints and justified their correctness. The rules can be applied to the constraints in any order. In the following chapters we demonstrate how to implement the rules in an existing proof assistant, and evaluate their impact on type-checking performance.

Chapter 5

Implementation

The approach in Chapter 4 was previously implemented and tested on a type-checker prototype [56]. Tests on this prototype showed that it could address the limitations mentioned in §3.9, while exhibiting comparable performance to Agda [6] in a specific case study based on a paper by McBride [64]. A description of the implementation and an evaluation of its performance can be found in my licentiate thesis [54].

While encouraging, the prototype only implements a small part of the functionality included in Agda. It furthermore relies on a technique called hash-consing, pioneered by Ershov [36] and introduced by Deutsch [29] and Goto [42] in the context of Lisp, which is not widely adopted by dependently-typed proof assistants.

In order to assess whether our approach can scale to a large system, we modify Agda in line with the method described in Chapter 4, without using far-reaching optimizations such as hash-consing. We call the resulting implementation Agda.ε.

The unification rules that Agda uses and the specific order in which they are applied are the result of years of work by the Agda implementors; describing them in detail is outside the scope of this work. In this chapter we restrict ourselves to explaining the key differences between Agda.ε and the baseline Agda implementation.

Agda is implemented in Haskell. We use pseudocode with a strong Haskell flavour, and assume familiarity with certain Haskell extensions such as DataKinds [93] and TypeApplications [94]. These are used as a limited form of dependently-typed programming in order to keep track of which side of the constraint (and thus, which side of the context) a term lives in. This usage is inspired by the dependently-typed programming technique described by Eisenberg and Weirich [33].

For the sake of clarity, we may display type declarations and function definitions which are less general than the ones found in the code. We will also omit certain fields and data constructors if they refer to functionality that is outside of the scope of this work. Those code fragments which would expose too many details of the Agda implementation are given with the aid of pseudocode.

The main goals of this chapter are (i) describing the experimental setup for

the experimental results obtained in Chapter 6, and (ii) providing a starting point for practitioners wanting to implement the techniques in this thesis in their own dependent type-checker. Showing how to implement a full dependent type-checker or how to achieve an optimal implementation of our approach is outside of the scope of this work.

5.1 Twin types

The homogeneous constraints in Agda are of the form $\Gamma \vdash t \approx u : A$. In Agda.$\varepsilon$, following Definition 4.2, constraints are of the form $\Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A_1 ‡ A_2$, where $\Gamma_1 \vdash t : A_1$ and $\Gamma_2 \vdash u : A_2$.

Many operations on terms, such as reduction, occur in a specific side of the context. To reduce bugs we keep track of which side of the constraint each terms lives in. We do this by means of a type level data-type ContextSide, and a new-type wrapper with a phantom parameter of kind ContextSide:

```
1  data ContextSide = LHS      -- ^ Left side of the context
2                   | RHS      -- ^ Right side of the context
3                   | Single   -- ^ One side of the context, when both sides
4                              -- are equivalent.
5                   | Both     -- ^ Both sides of the context
6
7  newtype OnSide (side :: ContextSide) t = OnSide { onSide :: t }
```

A ProblemId identifies a set of constraints. We can represent a set of ProblemId as an ISet ProblemId. ISet is a type-safe wrapper around a representation of sets of integers based on big-endian patricia trees [52, 82].

```
1  newtype ISet a = ISet { runISet :: IntSet }
2  type ProblemId = Nat
```

According to the implementor of the IntSet data structure, and compared to the generic Data.Set implementation, IntSet performs "particularly well on binary operations such as union and intersection", which are the ones we make most use of in our implementation. These operations are asymptotically linearithmic on the size of the sets. However, the IntSet implementation also uses bitmaps on the leaves, which makes the memory and CPU footprint much smaller when the problem identifiers are numerically close together.

Types in Agda consist of a Term and a Sort. The Term data type represents the syntax of a term, of which the syntax described in Figure 2.1 is a simplified version. The Sort includes additional information which is not handled in our theoretical development, such as the universe level in which a type lives. The representation of types (Type), the syntax of terms (Term) and the handling of the Sort associated with a type all remain unchanged with respect to Agda:

```
1  data Type = El Sort Term
```

We represent a pair of types $A_1 ‡ A_2$ using the TwinT datatype, which is a specialization of TwinT':

```
1  data TwinT' a =
2      SingleT { unSingleT :: OnSide 'Both a }
3    | TwinT { twinPid :: ISet ProblemId
```

```
4           , twinLHS :: OnSide 'LHS a
5           , twinRHS :: OnSide 'RHS a
6           }
7
8 type TwinT = TwinT' Type
```

The SingleT constructor optimizes for the case where both sides of the twin type are syntactically identical. Certain operations, such as applying a substitution, depend only on the syntax of the types. By using the SingleT constructor we can avoid redundant work in those cases. The twinLHS and twinRHS fields represent the left and right hand sides respectively. The twinPid field identifies a set of constraints, which, when all of them are solved, will make both sides of the twin type equal.

For example, given a problem with identifier i consisting of the constraint $\Gamma_1 \vdash A_1 \approx A_2 : \text{Set}‡\text{Set}$, we can construct the twin type TwinT $\{i\}$ A_1 A_2. We will usually leave the problem set implicit, but we may add it as a subindex to the double-dagger operator; e.g. $A_1‡_{\{i\}}A_2$. The type Set‡Set can be represented as either SingleT Set or TwinT $\emptyset$ Set Set.

5.2 Heterogeneous contexts

A heterogeneous context ($\Gamma_1‡\Gamma_2$) is represented as a value of the following Haskell type:

```
1 data Context_ = Empty
2               | Entry (ISet ProblemId) TwinT Context_
```

The empty context $\cdot$ is represented as Empty. A context "$\Gamma_1‡\Gamma_2, A_1‡A_2$" is represented as Entry P $A_1‡A_2$ $\Gamma_1‡\Gamma_2$.

For each context we consider the problem set associated with it. The problem set of an empty context (Empty) is $\emptyset$. For a non-empty context (Entry P $A_1‡A_2$ $\Gamma_1‡\Gamma_2$), the associated problem identifier set is stored as the first field of the Entry constructor (P), and defined as the union of the problem identifier set of $A_1‡A_2$ ($\emptyset$ in the case of the SingleT constructor), and the problem identifier set of $\Gamma_1‡\Gamma_2$.

Computing the problem identifier set associated with a context is a quick way to assess if the two sides of the context are equal (see Listing 5.2). This is useful to determine whether Rule-Schema 2 (metavariable instantiation) is applicable (see §5.7).

The common parts of the computation of P are shared for performance. For example, consider the context $\Gamma \stackrel{\text{def}}{=} A_1‡A'_1, A_2‡A'_2, ..., A_n‡A'_n$, Let p_i be is the problem set of entry $A_i‡A'_i$, and $P_i = p_1 \cup ... \cup p_i$ the problem set of the context $\Gamma^{(i)} \stackrel{\text{def}}{=} A_1‡A'_1, A_2‡A'_2, ..., A_i‡A'_i$. In particular, P_n is the problem set of the whole context $\Gamma^{(n)} = \Gamma$, and $P_0 = \emptyset$.

Each P_i is computed as the union of p_i and P_{i-1}, and is stored into the corresponding Entry. Thus, the computation of $p_1 \cup ... \cup p_i$ is shared among P_i, P_{i+1}, ..., P_n. This means that if the unification algorithm needs to repeatedly check the problem set of a context as it is extended with new entries, it will only perform a number of set operations ($\cup$) linear in the final size of the context, not quadratic. This helps us keep performance closer to the original Agda, as demonstrated in the evaluation section (§6.4.3).

Once the problem identifier set is computed of a context, it can be used in two ways. One can query the environment of the unification algorithm and check that all the constraints associated with each problem identifier in the set have been solved, which would mean that both sides of the context are equal. Alternatively, one may just check whether the set is empty. This is a much faster operation which only takes constant time, but may lead to more false negatives. We use these two alternatives at different places in the code, depending on how costly and frequently traversed we expect the code path followed in the negative case to be.

5.3 Context switching

In Agda, the context of a constraint is not explicitly included in its syntax. Instead, constraint solving is performed in a monad which, among other things, exposes the context in which the current constraint lives. For the purposes of this explanation, we will mark a monad where the context is available using the typeclass constraint MonadContext m:

```
1 getContext_ :: MonadContext m ⇒ m Context_
```

The type-checking monad in Agda was originally structured so that the same computation environment is used for solving constraints and manipulating terms. This environment contains among other values an associated context in which both constraints and terms live and which affects how they are manipulated. Thus, the type-checking monad in Agda exposes the context Γ of the current constraint or term.

In our development, terms also live in a single-sided context (Γ), but constraints may live in a twin context ($\Gamma_1 ‡ \Gamma_2$). Enforcing this distinction in Agda.ε at the type level by having different data types for single-sided and twin contexts could reduce bugs. However, implementing this distinction in Agda.ε would have required an overhaul of the type-checking monad. In order to minimize the amount of required modifications to the code, we keep the same data type for constraints and for terms, and store single-sided contexts as heterogeneous contexts with identical left and right sides. Thus, a single-sided context Γ is represented equivalently to the heterogeneous context $\Gamma ‡_\emptyset \Gamma$. We use the SingleT as the constructor of each twin type in the context to avoid duplication. If a function attempts to access the type of a variable with twin type while expecting it to be single sided a runtime exception is thrown. If that were to happen, that would be considered a bug. The position in the code at which the access attempt happend is included in the thrown exception in order to facilitate debugging.

A monad fulfilling MonadContext is also required to be reader monad, which means that one can locally replace the context by another one. Using this capability, we define a switchSide function which can locally switch the context to the left side (switchSide @'LHS) or to the right side (switchSide @'RHS). It can alternatively switch to an unspecified side of the context (switchSide @'Single), but it will check that the problem identifier set of the context is empty before doing so, and will throw an error otherwise. We do not allow switching to the "Both" side, as that would be a no-op, and thus has no good reason for existing other than as a programming error in the code:

```
switchSide :: forall (side :: ContextSide) (m :: ⋆ → ⋆).
                (side ≠ Both, MonadContext m) ⇒ m a → m a
switchSide m = do
  Γ1 ‡p Γ2 ← getContext_
  case side of
    'LHS    → In context Γ1 do m
    'RHS    → In context Γ2 do m
    'Single → if p == ∅ then
                 In context Γ1 do m
              else
                 Throw error
```

An example of an operation that is implemented in Agda in a way that depends on the context of the therm is reduction. More specifically, reduction is implemented as a typeclass with a method reduce. The reduction happens inside a monad which exposes, among other information, the context where the term lives:

```
class Reduce t where
    reduce' :: MonadContext m ⇒ t → m t
```

Reduction for terms annotated with OnSide is implemented as a typeclass instance, using the switchSide function to automatically switch to the appropriate context:

```
instance Reduce a ⇒ Reduce (OnSide side a) where
  reduce' (OnSide a) = OnSide <$> switchSide @side (reduce' a)
```

The Reduce typeclass has instances for types containing terms in some way or another: Term, Type,.... When the Agda.ε implementation needs to reduce a twin type, each side of the twin type must be reduced in each corresponding side of the context. Because these types are wrapped in OnSide constructors which specify the side of the context in which they live, the typeclass mechanism can handle the context-switching automatically:

```
instance Reduce a ⇒ Reduce (TwinT' a) where
  reduce' (TwinT{twinLHS,twinRHS,..}) = do
    twinLHS ← reduce' twinLHS
    twinRHS ← reduce' twinRHS
    return TwinT{twinLHS,twinRHS,..}

  reduce' (SingleT (OnSide a)) = do
    noPids ← problem identifiers of getContext_
    -- If the problem id set of the context is empty,
    if null noPids then
    -- ... then both sides of the context are interchangable,
    -- ... and we can perform the reduction in any of the two sides
    -- ... of the context.
      SingleT . OnSide @'Both <$> (switchSide @'Single (reduce' a))
    else
    -- Otherwise, the result of the reduction may be potentially
    -- different, so we need to perform it in each of the two
    -- sides separately.
      reduce' TwinT { twinPid  = ∅
                    , twinLHS = OnSide @'LHS a
```

```
21                    , twinRHS = OnSide @'RHS a
22                    }
```

In some cases, a twin type in a context may have been represented as a single-sided type (i.e. with the SingleT constructor), but the context it lives in has two distinct sides (i.e. the problem identifier set of the current context is not empty). In those cases, the operation must be applied independently in each of the two sides of the context.

Other operations on terms, such as applying a substitution, depend only on the syntax of the term and not on the types of the variables in the context. Therefore, these operations can be performed without regard to the side annotations, and without the need to switch contexts.

5.4 Constraint representation

Using the data types defined in §5.1, we can extend the syntax of constraints according to our approach:

```
1 data Constraint
2   = ValueCmp_ TwinT (OnSide 'LHS Term) (OnSide 'RHS Term)
3   | ElimCmp_ TwinT (TwinT' Term) (OnSide 'LHS [Elim]) (OnSide 'RHS [Elim])
```

A constraint of the form $\Gamma_1 ‡ \Gamma_2 \vdash t \approx u : A_1 ‡ A_2$ can be represented as ValueCmp_ ($A_1 ‡ A_2$) (OnSide @LHS t) (OnSide @RHS u). The context $\Gamma_1 ‡ \Gamma_2$ is carried implicitly in the environment (see §5.2).

In Agda, Rule-Schema 14 (strongly neutral terms) does not need to be applied in one go, but can instead be applied step-wise. To minimize alterations to the code, we preserve this behaviour. The ElimCmp_ constructor is used for this purpose, as described in §5.10.

Constraints are solved within a specific monad. For simplicity, we will indicate that constraints are solved in a monad m fulfilling MonadConstraint m. Such a monad m gives access to the context of the current constraint being solved (that is, MonadConstraint m entails MonadContext m). It also gives information about which of the problem identifiers associated with a twin type or a twin context correspond to constraints which have already been solved.

5.5 Constraint blocking

Constraints in Agda are generated from the typing rules as the definitions written by the user are parsed. The Agda type-checker attempts to solve these constraints as soon as they are created. There are two reasons for this. First, solving the constraints as they are created minimizes the amount of unsolved constraints at a given time, and thus the amount of memory required to hold them. Furthermore, metavariables instantiated by solving a constraint can enable later terms to be type-checked without generating new constraints.

Sometimes a constraint cannot be solved immediately. Indeed, one of the motivations for our approach to higher-order unification is to allow postponement of constraints when no unification rules apply to them, allowing other

constraints to be attempted instead (§3.6). Subsequent constraints may instantiate metavariables thus making an unification rule applicable to the postponed constraint.

At certain points during the unification algorithm, Agda iterates sequentially through the current list of unsolved constraints and *retries* them with the hope that enough additional information has been obtained to make some of these postponed constraints solvable. However, retrying a constraint only to postpone it again is wasteful. In order to avoid retrying constraints on which no progress can be made, an *unblocker* is computed whenever a constraint is postponed, and stored together with the postponed constraint. The unblocker for each constraint describes the conditions under which the algorithm should attempt to make progress on that particular constraint. This may be a set of metavariables that are preventing some term from normalizing further, or a set of constraints that when solved will make a metavariable instantiation well-typed (§5.7).

In the original Tog [61] and in earlier versions of Agda, unblockers are a disjunction of metavariables ($\alpha_1 \vee \alpha_2 \vee ... \vee \alpha_n$). Such an unblocker allows the constraint to be retried as soon as any of the metavariables in the unblocker has been instantiated. Norell [77] extended unblockers in Agda (which are perhaps unintuitively called Blockers in the actual implementation) following the design in our prototype Tog$^+$ [56], in order to unblock on not only disjunctions (UnblockOnAny) but also conjunctions (UnblockOnAll) of metavariable instantiations (UnblockOnMeta). This enables more fine-grained control of constraint retries. Later on, Norell [78] added an unblocker for problem identifiers (UnblockOnProblem), as had also been done in Tog$^+$.

```
1 data Unblocker = UnblockOnAll {u_1, ..., u_n}  -- notation: u_1 ∧ ... ∧ u_n
2                | UnblockOnAny {u_1, ..., u_n}  -- notation: u_1 ∨ ... ∨ u_n
3                | UnblockOnMeta α       -- notation: α, where α is a metavariable
4                | UnblockOnProblem p    -- notation: p, where p is a problem
```

The values of the Unblocker data type form a bounded lattice. The top element of the lattice of unblockers (i.e. UnblockOnAll $\emptyset$) is denoted $\top$, and the bottom element (i.e. UnblockOnAny $\emptyset$). is denoted $\bot$.

When building new unblockers from existing ones, we use smart constructors so that $\top$ and $\bot$ are immediately identifiable by their syntax. These smart constructors are not monadic, so for instance they cannot take into account that a metavariable α has already been solved, and automatically turn UnblockOnMeta α into $\top$. The unblockers are instead updated by the unifier when a new metavariable is instantiated. For instance, the unification algorithm will not retry a constraint tagged with the unblocker "$\alpha \wedge p$". The unblocker may be updated to α when all the constraints in problem p are solved, and then to $\top$ once α is instantiated. The associated constraint will then be eligible to be retried.

We have preserved the workings of unblockers in Agda when implementing Agda.ε, and rely on the existing functions in the Agda implementation to update the unblockers as needed. The complexity of the process of updating an unblocker may be up to linear in its size. Although there is potential for optimization, acceptable performance may be achieved with the existing data structures, as demonstrated in Chapter 6.

Unblocker semantics

There is a discrepancy between the way unblockers were implemented in Agda, and the way we use them in the extensions that we made for Agda.ε. In Agda, that a failed equality check may return $\top$ to signal that the test should be retried again at some unspecified point, while we use $\top$ to signal that the test succeeded. Due to the lack of clarity about when constraints blocked on "$\top$" should be retried, it was not clear to us what allowing $\top$ as a potential unblocker for a failed equality check would mean. We instead added code to convert between the two approaches where applicable, and have inspected the existing code to ensure that our usage in Agda.ε of unblockers in general (and of $\top$ in particular) is consistent with the invariants that we aim to preserve.

5.6 Constraint prioritization

In the Agda type-checker, there are no clever heuristics in the implementation about the order in which to retry constraints when there are multiple constraints that may be retried. Rather, constraints are reattempted in the order in which they were added to the list of pending constraints.

For Agda.ε, we have added infrastructure to the type-checker to exert a certain degree of control over the order in which constraints are retried. This infrastructure is integrated with the unblocker mechanism described in §5.5. First, we add a new type of unblocker, which postpones the constraint until a given amount of additional "effort" is allowed. Effort is represented by an element of a well-ordered set, for instance the natural numbers ($\mathbb{N}^+ := \{1, ..., n\}$).

```
1 data Unblocker = ...
2                | UnblockOnEffort ℕ⁺
```

Given an unblocker, unblocksOnEffort computes an element in the extended lattice $\mathbb{N}_\infty := \mathbb{N}^+ \cup \{0, \infty\}$, intuitively amounting to the level of additional effort required to unblock that particular constraint:

```
1 unblocksOnEffort :: Unblocker → ℕ ∪ {0, ∞}
2 unblocksOnEffort (UnblockOnEffort e) = e
3 unblocksOnEffort UnblockOnMeta{}  = ∞
4 unblocksOnEffort UnblockOnProblem{} = ∞
5 unblocksOnEffort (UnblockOnAll {u_1, ..., u_n}) =
6     maximum ({0} ∪ {unblocksOnEffort u_1, ...,unblocksOnEffort u_n})
7 unblocksOnEffort (UnblockOnAny {u_1, ..., u_n}) =
8     minimum ({∞} ∪ {unblocksOnEffort u_1, ...,unblocksOnEffort u_n})
```

For instance, the unblocker 10 requires effort 10, as does $10 \vee \alpha$. The unblocker $\top$ requires effort 0; the unblocker $1 \wedge \alpha$ requires effort ∞ (as the metavariable α is not instantiated).

The amount of effort allowed to be spent solving the current constraint is carried around in the environment of the constraint solving monad. It can be read and also set locally.

```
1 getEffortLevel  :: MonadConstraint m ⇒ m ℕ
```

The effort level is initially set to 0. We implement a strive function, which the implementor may call at any point during the unification algorithm (see §5.7) to check the whether the currently-allowed effort level is high enough:

```
data Strive = Doable
            | ExtraEffort ℕ⁺

strive : MonadEffort m ⇒ ℕ → m Strive
strive e = do
  e' ← getEffortLevel
  if e ≤ e' then
    return Doable
  else
    -- Return the increase in effort required to
    -- reach the desired effort level
    return (ExtraEffort (e' − e))
```

The Agda implementor can use the result of the function to determine whether to continue the execution (Doable constructor), or postpone the constraint until additional effort is allowed (ExtraEffort constructor).

The type-checker keeps a list of postponed (therefore, unsolved) constraints. Each unsolved constraint can be understood as a 4-tuple with the following components:

- Γ :: Context_: The context of the constraint.
- u :: Unblocker: An unblocker, which determines whether the constraint can be solved.
- e :: $\mathbb{N}$: The effort level in the constraint environment when the constraint was postponed. If the unblocker u is $\top$, this is also the effort level that will be present in the environment when the constraint is attempted again.

The effort level is initially set to 0. The unblockers are updated as metavariables are instantiated and unification problems are solved.

If, at any point, all constraints are blocked (that is, their blocker is different from $\top$), Agda will throw a type error reporting the constraints that could not be solved. In Agda.ε, we define the function tryOneConstraintHarder which may be called whenever the unification algorithm runs out of unblocked constraints to solve, but before a type error is reported. This function checks if any of the blocked constraints could be unblocked by increasing the effort level. If that is the case, this constraint is put back with an AlwaysUnblock unblocker, (i.e. $\top$), and the effort level in its environment is increased by the required amount (δ):

```
-- Returns True if a constraint has been unblocked, False otherwise
tryOneConstraintHarder :: MonadConstraint m ⇒ m Bool
tryOneConstraintHarder = do
  pcs ← Get the list of unsolved constraints
  pcs₁ ← [(e, pc) | pc ← pcs
                  , let e = unblocksOnEffort (unblocker of pc)]
```

```
7   -- · Ignore constraints that require infinity effort (∞)
8   -- (this means they cannot be unblocked by only increasing effort),
9   -- · Ignore constraints that require no extra effort (0)
10  -- (otherwise we could produce an infinite loop in Agda)
11  let (pcs2, pcs3) = partition (λ (e,_) →(e < ∞) && (e > 0)) pcs
12  case sortBy (compare `on` fst) (pcs2) of
13    [] → return False
14    (δ,pc):pcs4 → do
15      -- Increase the effort by the required δ, and unblock the constraint
16      let pc1 = pc{e = e pc + δ, u = ⊤}
17      Set the list of unsolved constraints to (pc1:map snd pcs4)
18      return True
```

When the unification code processes the unblocked constraint, and reaches again the same point in the code where strive was called, the effort level in the environment will now be high enough that Doable will be returned instead, and the constraint solving may continue instead.

The complexity of computing the effort level required to unblock a constraint is up to linear in the size of the unblocker. This could warrant further optimization if very large unblockers were to occur, but we expect the tryOneConstraintHarder operation to be run infrequently. Additionally, we are looking for a constraint to attempt using a potentially expensive operation, so the cost of computing the effort may be dwarfed by the cost of the expensive operation itself.

Currently, only one of the effort levels (10) is meaningfully used in Agda.ε; thus the full ordered set of effort levels could be summarized as $0 < 10 < \infty$. This also means that the distinction between the absolute effort required to solve the constraint e and the increase in effort required to solve a constraint is purely academic, as the only increase that happens in practice is from 0 to 10. We have implemented it using the full set of natural numbers for increased flexibility, but the set of possible effort values could be reduced to the aforementioned three (0,10,∞) without any reduction in the implemented functionality.

5.7 Metavariable instantiation

Agda does not make use of a small core language without metavariables. Instead, terms may contain metavariables, and the subterms those metavariables stand for (their bodies) are globally defined. Metavariables are replaced by their bodies as needed to apply the unification rules. One of the key ways in which the well-typedness of a constraint may be broken is by assigning a term of the wrong type to a metavariable. As shown in the theoretical development (§4.5), the key point where well-typedness of constraints is enforced is when metavariables are instantiated. More specifically, applying Rule-Schema 2 (metavariable instantiation) to a constraint (e.g. $\Gamma_1 \ddagger \Gamma_2 \vdash \alpha\,\vec{x} \approx v : B_1 \ddagger B_2$) requires the conditions $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash B_1 \equiv \{B\} \equiv B_2 : \mathrm{Set} \ddagger \mathrm{Set}$ (3a) and $\Sigma \vdash \Gamma_1 \equiv \{\Gamma\} \equiv_{\{x_i | i=1,\ldots,n\} \cup b, \mathrm{FV}(t) \cup b} \Gamma_2$, where $b = \mathrm{FV}(B_1) \cap \mathrm{FV}(B_2)$ (3b) to hold. As shown in the correctness proof of Rule-Schema 2, those preconditions are sufficient for the metavariable to be instantiated to a term of the correct type. We implement the checks for preconditions (3a) and (3b) in Agda.ε

in accordance to Definition 4.12 (heterogeneous equality) and Definition 4.36 (heterogeneously equal contexts modulo variables), respectively.

The check for precondition (3a) is implemented by the function checkTwinEqual given in Listing 5.1. If checkTwinEqual $B_1 ‡ B_2$ returns $\top$, it means that $\Sigma; \Gamma_1 ‡ \Gamma_2 \vdash B_1 \equiv \{B\} \equiv B_2 : \mathrm{Set} ‡ \mathrm{Set}$ for some B. As per Definition 4.12, we call B the interpolant. In Agda.ε, as in our previous prototype Tog$^+$, solving all the constraints associated to a twin is sufficient for both sides of a twin type becoming heterogeneously equal (albeit not necessary). Thus, equality of both sides of a twin type is checked in two ways: a quick one which may lead to false negatives, based on the constraints associated with the twin (see also §5.1); and a slow but definitive one, based on checking convertibility between both sides. Checking convertibility is computationally intensive, so the constraint prioritization machinery (§5.6) is used to postpone this until no other constraints can be tackled. The hope is that solving other constraints can produce enough information for the type-checker to verify that the metavariable can be safely instantiated, or that the metavariable itself is instantiated somewhere else. In both cases the expensive convertibility check can be dispensed with.

In order to check (3b) in Agda.ε, we need to know the intersection of the free variables of $\mathrm{FV}(B_1)$ and $\mathrm{FV}(B_2)$ (see Rule-Schema 2). Thus, we need to compute these two sets in order to check (3b). The more variables in $\mathrm{FV}(B_1) \cap \mathrm{FV}(B_2)$, the stricter that precondition (3b) becomes. However, if metavariables occurring in the types B_1 or B_2 are instantiated, these types could potentially be normalized into into an equivalent types with potentially fewer free variables. This means that when computing $\mathrm{FV}(B_1) \cap \mathrm{FV}(B_2)$, we want to know not only which variables the sets contain, but also which metavariables would need to be instantiated in order for there to exist a normalized form of B_1 or B_2 which does not contain that variable. Thus, the set of free variables is represented as a mapping between the de Bruijn index of a variable $(0, 1, 2, ...)$ and an unblocker which represents the conditions under which the variable could potentially be normalized away. For example, if $B_1 = B_2 = \alpha\, x$, the variable x would occur with unblocker α, as instantiating the metavariable α to a suitable function (e.g. $\alpha := \lambda z.\mathrm{true}$) would mean that there exists a normalized version of the term in which x is not free (i.e. x has been "normalized away"). A variable with unblocker $\bot$ represents what is commonly known as a *rigid occurrence*, which is a free variable that will persist in the term regardless of which metavariables are instantiated. The unblocker $\top$ represents a variable which does not occur in the term.

The type of sets of free variables with associated unblockers is VarSetBlocked. In the actual implementation, VarSetBlocked is a type of finite maps, where variables which do not occur in the term are simply not present in the map. Here, for ease of presentation, we consider VarSetBlocked as a type of functions, with all non-free variables mapping to $\top$.

```
1 type VarSetBlocked :: ℕ → Unblocker
```

The free variables of a term t (i.e. $\mathrm{FV}(t)$) or a type A (i.e. $\mathrm{FV}(A)$) with their corresponding unblockers are computed as the overloaded function calls "freeVarsBlocked t" and "freeVarsBlocked A" respectively:

```
1 freeVarsBlocked :: Term → VarSetBlocked
```

Listing 5.1: Check whether two sides of a twin type are equal. This operation is done in a monad m in which we can check whether a problem is solved, and also check terms and types for convertibility in the current context and signature.

```
1  checkTwinEqual :: (MonadConstraint m) ⇒ TwinT → m Unblocker
2  checkTwinEqual (SingleT t) = return AlwaysUnblock
3  checkTwinEqual TwinT{twinPid,twinLHS,twinRHS} = do
4    -- Construct an unblocker which unblocks when all the unsolved problems in twinPid
5    -- are solved (a problem might have become solved after the twin type was
6    -- created)
7    pids ← {p ∈ twinPid | there are unsolved constraints associated with p}
8    if null pids then
9      -- If all problems are solved, then the unblocker will be AlwaysUnblock,
10     -- and we return that.
11     return ⊤
12   else
13    -- (Perhaps) attempt to check if the
14    -- two sides of the twin type are equal.
15    bs ← ( strive  10 >>= \case
16          -- Block the constraint until the allowed effort  is  increased by δ
17          ExtraEffort  δ → return (UnblockOnEffort δ)
18          -- The following check returns an unblocker:
19          --  · ⊤ if the two types are convertible
20          --  · ⊥ if the two types are not convertible, and will never be convertible
21          --  regardless  of any future metavariable instantiations.
22          --  · An unblocker signifying when the comparison should be re-attempted
23          --  (for instance,  a metavariable which may make one of the sides reduce
24          --  and become equal to the given type).
25          Doable             → Check whether twinLHS and twinRHS are convertible
26    -- Unblock when either all problems are solved, or the convertibility  check
27    -- should be attempted again.
28    return (((⋀_{p∈pids} UnblockOnProblem p) ∨ bs)
```

```
freeVarsBlocked :: Type → VarSetBlocked
```

The expression freeVarsInterpolant $B_1 \ddagger B_2$ computes the set $\text{FV}(B_1) \cap \text{FV}(B_2)$:

```
freeVarsInterpolant (SingleT a) = freeVarsBlocked a
freeVarsInterpolant :: TwinT → VarSetBlocked
freeVarsInterpolant (SingleT a) = freeVarsBlocked a
freeVarsInterpolant (TwinT{twinLHS,twinRHS}) =
  (x ↦ freeVarsBlocked twinLHS(x) ∨ freeVarsBlocked twinRHS(x)
```

Given two sets of blocked variables, we can check whether the context is indeed heterogeneously equal modulo these variables by using the function checkContextEqual (Listing 5.2). Thus, using the functions checkTwinEqual (Listing 5.1) and checkContextEqual (Listing 5.2), we can put together the check for the preconditions of metavariable instantiation. This check in the metavariable assignment routine for a constraint $\Gamma \vdash \alpha\, \vec{x} \approx v : B_1 \ddagger B_2$ is described in Listing 5.3. In the call to checkContextEqual, we take fvL $\stackrel{\text{def}}{=} \text{FV}(\vec{x}) \cup b$ and fvR $\stackrel{\text{def}}{=} \text{FV}(v) \cup b$, where $b = \text{FV}(B_1) \cap \text{FV}(B_2)$, as per (3b). Due to how VarSetBlocked is defined, the union of free variable sets corresponds, perhaps unintuitively, to the pointwise *conjunction* of the unblockers for each variable.

Also note how, in Listing 5.3, when computing the free variable sets of B_1, B_2 and v, we first apply instantiateFull to them in order to expand any already solved metavariables, and thus potentially reduce the set of free variables in the term. One may obtain even smaller sets of free variables by further normalizing the types and terms involved. This potentially costly computation might allow more constraints to be solved, but we have not found the need to do this in our tests. If the need should arise, one could use the constraint prioritization mechanism described in §5.6 so that the additional normalization is only performed as a last resort.

5.8 Fast twin simplification

Applying an operation (such as a substitution, or a full instantiation of all the metavariables it contains) to a two-sided twin type (i.e. one using the constructor TwinT) involves up to double the amount of computation than if the two sides are known to be syntactically identical (i.e. one using the constructor SingleT).

To increase the chances of the term being of the latter form, we can check whether the associated constraint has been solved, which implies that the two sides of the twin are heterogeneously equal. However, just because the two sides are heterogeneously equal, it does not mean that we can replace the twin by any of its sides. We could replace them by the interpolant, but computing this term explicitly would both increase the implementation effort, and also be potentially slow. However, if the two sides of *all* the twins in the context are heterogeneously equal, then the heterogeneous equality becomes a homogeneous one (Lemma 4.13). In that case, we can replace the twin context and type by any one of their sides. Here we choose the left-side (Listing 5.4), but there is no theoretical reason why we could not have chosen the right side instead.

Listing 5.2: Checking of heterogeneous equality of the local context

```
checkContextEqual :: (MonadConstraint m) ⇒
                     VarSetBlocked → VarSetBlocked → m Unblocker
checkContextEqual fvL fvR = do
  -- Check if all the problem identifiers from the context
  -- correspond to solved constraints
  ctxIsHomogeneous ← All problems in the problem id set of getContext_ are solved
  if ctxIsHomogeneous then
    -- If all the problems associated with the context are solved,
    -- then both sides of the context are equal
    return ⊤
  else
    -- Otherwise, we need to do more work, and check for convertibility
    go fvL fvR
  where
    go fvL fvR = do
    -- Check if the variable sets are empty, in which case we
    -- don't need to check anything.
      if null fvL or null fvR then
        return ⊤
      else
        ctx ← getContext_
        case ctx of
          ·          → return ⊤
          Γ, A₁‡A₂ → In context Γ $ do
            -- If the variable 0 is present in both fvL and fvR ...
            if u₁ ≝ fvL 0 ≠ ⊤ and u₂ ≝ fvR 0 ≠ ⊤ then
              checkTwinEqual a >>= \case
                  ⊤ → do
                    -- Expand all metavariables in the type
                    tA ← instantiateFull (A₁‡A₂)
                    let fvA  = freeVarsInterpolant tA
                    -- Continue checking the other variable,
                    -- plus those which must occur in any interpolant of A₁ and A₂.
                    go (x ↦ (u₁ ∨ fvA(x)) ∧ fvL(x + 1))
                       (x ↦ (u₂ ∨ fvA(x)) ∧ fvR(x + 1))
                  u₀ → return (u₀ ∨ u₁ ∨ u₂)
            -- If the variable 0 is present only in fvL ...
            else if u₁ ≝ fvL 0 ≠ ⊤ then
                 fvA ← freeVarsBlocked <$> instantiateFull A₁
                 go (x ↦ (u₁ ∨ fvA (x)) ∧ fvL (x + 1)) (x ↦ fvR (x + 1))
            -- The following case is impossible, because in all calls to
            -- checkContextEqual we have that fvL ⊇ fvR;
            -- and this invariant is preserved by all recursive calls to go.
            -- We have checked that this branch was not triggered in any of
            -- the examples during our evaluation.
            else if u₂ ≝ fvR 0 ≠ ⊤ then
              Throw exception
            else
              -- If the variable 0 does not occur in any of the sets,
              -- we consider it unused and continue with the rest of the context
              go (x ↦ fvL (x + 1)) (x ↦ fvR (x + 1))
```

Listing 5.3: Implementation in Agda.ε of the check for the preconditions of Rule-Schema 2 (metavariable instantiation)

```
...
checkTwinEqual B₁‡B₂ >>= \case
  ⊤ → do
  -- If both sides of the twin type are equal, obtain the free
  -- variables occurring in the type of the constraint, and in the terms
  -- on each side of the constraint.
    fvTarget ← freeVarsInterpolant <$> (instantiateFull B₁‡B₂)
    fvArgs   ← freeVarsBlocked x⃗
    fvV      ← freeVarsBlocked <$> (instantiateFull v)
    checkContextEqual
      (x ↦ fvArgs(x) ∧ fvTarget(x))
      (x ↦ fvV(x) ∧ fvTarget(x))
      >>= \case
        -- If both checks pass, we allow the metavariable instantiation
        -- routine to continue.
        ⊤ → return ()
  -- If either the type equality or the context equality checks
  -- failed, we block the constraint until they might succeed.
        u   → block current constraint on unblocker u
  u → block current constraint on unblocker u
-- Do some further checks and instantiate the metavariable
...
```

5.9 Unification of binders

When applying Rule-Schema 3 (injectivity of Π) for solving a constraint $\Gamma_1‡\Gamma_2 \vdash \Pi A_1 B_1 \approx \Pi A_2 B_2 : \mathrm{Set}‡\mathrm{Set}$, we compare the domains and the codomains. Listing 5.5 shows how this rule is applied in Agda.ε. First, a constraint is introduced to unify A_1 and A_2. The constraint is associated with a newly-created problem identifier, which is then used in the creation of a corresponding twin type. The context is extended by this twin type, and the codomains B_1 and B_2 are compared in the extended context.

Note that the domain and codomain of a Π-type in Agda (as well as the context) contain additional information, such as modalities (e.g. irrelevance) or suggestions of variable names for pretty-printing. These values are handled in the same way as they were in Agda, respectively by checking whether the modalities coincide for both sides of the Π-type, or by arbitrarily choosing one of the variable name suggestions.

5.10 Unification of spines

As explained in §5.4, the rule for unifying strongly neutral terms in Agda does not need to be applied to the whole elimination spine, but may instead be applied to only a prefix of the eliminators. We preserve this behaviour in Agda.ε, and adapt it to handle twin types.

Listing 5.4: Simplifying a twin type or term based on whether the constraints associated with it and with the types in the context have been solved. The monad m gives access to the set of solved problems.

```
simplifyTwin :: (MonadConstraint m) ⇒ a → (a → m b) → m b
simplifyTwin b κ = do
  case b of
    -- Single types cannot be simplified further
    SingleT{} → κ b
    TwinT{twinPid,twinLHS=OnSide @'LHS twinLHS} → do₁
      pids ← Get unsolved problems in twinPid
      if null pids₁ then
        getContext_ >>= \case
        -- If the context is empty, and both sides of the twin are equal,
        -- then we can replace the twin by its left side
          Empty                → SingleT (OnSide @'Both twinLHS)
          Entry ctxPids ty ctx → do
            pids₂ ← Get unsolved problems in ctxPids
            -- We update the context entry so that it contains only
            -- those problems which are actually unsolved.
            In context (Entry pids₂ ty ctx) $ do
              -- If all problems in the twin and in the context are solved,
              -- to simplify the twin.
              κ (if ISet.null pids₂ then
                    (SingleT (OnSide @'Both twinLHS))
              -- Otherwise, we return the twin as-is. We update the twinPid
              -- to avoid rechecking the solved problems again.
                 else
                    b{twinPid=pids₁})
      else
        κ b{twinPid=pids₁}
```

Listing 5.5: Unifying two Π-types. This code is used in order to solve a constraint of the form $\Gamma_1 ‡ \Gamma_2 \vdash \Pi A_1 B_1 \approx \Pi A_2 B_2 : \mathrm{Set} ‡ \mathrm{Set}$

```
...
-- The domain of the Π-type contains additional information besides
-- the type, such as modalities.
(Pi A₁ B₁, Pi A₂ B₂) → do
    -- Let ctx be the context of the current constraint
    ctx ← getContext_
    pid ← Attempt to solve the constraint ctx ⊢ A₁ ≈ A₂ : Set‡Set and
          get the associated problem id
    -- Add the new variable to the context, keeping the domain information
    -- (modalities, etc...) of the left-hand side.
    let ctx' = ctx,(TwinT { twinPid  = {pid}
                          , twinLHS  = OnSide @'LHS A₁
                          , twinRHS  = OnSide @'RHS A₂
                          })
    Solve the constraint ctx' ⊢ B₁ ≈ B₂ : Set‡Set
 ...
```

Constraints unifying strongly neutral terms (Definition 2.158) may be represented using the constructor ElimCmp_. In this text, we use the notation ElimCmp_ $(A_1 \ddagger_{p_1} A_2)$ $(f_1 \ddagger_{p_2} f_2)$ $(t_1\ \overrightarrow{e_1})$ $(t_2\ \overrightarrow{e_2})$ to represent the following constraint:

```
ElimCmp_
    (TwinT{twinLHS=A₁,twinRHS=A₂,twinPid=p₁})
    (TwinT{twinLHS=f₁,twinRHS=f₂,twinPid=p₂})
    (OnSide @'LHS t₁ e₁⃗)
    (OnSide @'RHS t₂ e₂⃗)
```

Solving the constraint $\Gamma_1 \ddagger \Gamma_2 \vdash h\ \overrightarrow{e_1} \approx h\ \overrightarrow{e_2} : T_1 \ddagger T_2$, where both sides are strongly-neutral terms and $\Gamma_1 \ddagger \Gamma_2 \vdash h \Rightarrow A_1 \ddagger A_2$, is equivalent to solving the constraint ElimCmp_ $A_1 \ddagger A_2$ $(h \ddagger h)$ $(\overrightarrow{e_1})$ $(\overrightarrow{e_2})$. Constraints represented by the constructor ElimCmp_ are solved by iteratively applying the steps shown in Listing 5.6. Here we only consider Σ-types; other record types are supported analogously.

Listing 5.6: Solving elimination constraints. This code applies one step of the Rule-Schema 14 in order to solve constraints unifying two strongly neutral terms.

```
...
ElimCmp_ (ΠA₁B₁‡_{p₁} ΠA₂B₂) (f₁‡_{p₂} f₂) (·) (·) → do
  Done

ElimCmp_ (ΠA₁B₁‡_{p₁} ΠA₂B₂) (f₁‡_{p₂} f₂) (t₁ e₁⃗) (t₂ e₂⃗) → do
  i ← Attempt constraint Γ ⊢ t₁ ≈ t₂ : A₁‡_{p₁} A₂ and get the problem id
  Solve the constraint ElimCmp_ (B₁[t₁]‡_{p₁∪{i}} B₂[t₂]) ((f₁ t₁)‡_{p₂∪{i}} (f₂ t₂)) e₁⃗ e₂⃗

ElimCmp_ ΣA₁B₁‡_{p₁} ΣA₂B₂ (f₁‡_{p₂} f₂) (.π₁ e₁⃗) (.π₁ e₂⃗) → do
  Solve the constraint ElimCmp_ (A₁‡_{p₁} A₂) ((f₁ .π₁)‡_{p₂} (f₁ .π₁)) (e₁⃗) (e₂⃗)

ElimCmp_ ΣA₁B₁‡_{p₁} ΣA₂B₂ (f₁‡_{p₂} f₂) (.π₂ e₁⃗) (.π₂ e₂⃗) → do
  Solve ElimCmp_ (B₁[f₁ .π₁]‡_{p₁∪p₂} B₂[f₂ .π₁]) ((f₁ .π₂)‡_{p₂} (f₁ .π₂)) (e₁⃗) (e₂⃗)
...
```

Each step of the unification of the strongly neutral terms introduces new twins. As explained in §5.1, the subindices of the dagger operator represent problem identifiers associated with the twin type or term. The twin types arising from processing elimination constraints are an example where solving the constraints associated to the problem identifiers associated to a twin type is a sufficient condition, but perhaps not a necessary one, for the two sides of the twin type to become equal.

The case where one side is an application and the other is a projection is not covered in Listing 5.6. For instance, consider the constraint $\Gamma_1 \ddagger \Gamma_2 \vdash h\ \overrightarrow{e_1}\ t\ \overrightarrow{e_2} \approx h\ \overrightarrow{e'_1}\ .\pi_1\ \overrightarrow{e'_2} : T_1$, with $\overrightarrow{e_1}$ and $\overrightarrow{e'_1}$ having the same length, and ditto for $\overrightarrow{e_2}$ and $\overrightarrow{e'_2}$. In Agda.ε, both sides of the constraint are well-typed, which means that the type of f_1 must reduce to a function type, while the type of f_2 must reduce to a record type. In both Agda and Agda.ε, unification of the

spines is performed from left-to-right. We thus conjecture that the unification algorithm will have detected a discrepancy between two earlier arguments (and thus aborted further progress on the constraint) before attempting to unify the subsequent eliminators. We have thus disregarded this case from the implementation. If this case were to be reached, an exception tagged with the source code location is thrown, thus exposing the conjecture as the cause of the error.

5.11 Singleton types with η-equality

Agda implements an η-rule for singleton types (e.g. empty records). This is supported by a function which checks whether a record type is a singleton, which we refer to as "isSingletonRecord". As explained in §4.9.1, this rule is not covered in our theoretical development, and may in some cases break the completeness of the unification algorithm. However, we did not wish to break existing Agda code using this functionality, so we extended the existing support for singleton types in order to support twin types (Listing 5.7).

Listing 5.7: Checking for singleton types

```
1 isSingletonRecord :: (MonadConstraint m) ⇒ Type → m Bool
2
3 isSingletonRecord_ :: (MonadConstraint m) ⇒ TwinT → m Bool
4 isSingletonRecord_ A1‡A2 = do
5   x ← switchSide @'LHS (isSingletonRecord A1)
6   if x then
7     return True
8   else
9     switchSide @'RHS (isSingletonRecord A2)
```

A key insight here is that, whenever we have introduced a twin type in the implementation, there are corresponding constraints such that, when they are solved, both sides of the constraint will become equal. This follows Definition 4.19 (essentially homogeneous problem) and Lemma 4.23 in the theoretical development. In the implementation, this fact is witnessed by every twin type having an associated set of problem identifiers.

In particular, if any of the sides of the type of the constraint is a singleton type, the other will be a singleton type once all of the constraints resulting from type-checking the program are solved. Therefore, it suffices to check whether any of the sides of the twin type is a singleton type.

The implementation of this shortcut has not led to any issues in our tests. We consider a theoretical proof of its correctness to be outside the scope of this work.

5.12 Other changes

In this chapter we have detailed the main implementation changes required to solve the binder problem and spine problem using twin types. The remainder

of the changes to the unification algorithm involve analyzing on which side of the context each encountered term lived, and adapting the code to switch to the appropriate side before processing the term. This was done on a case-by-case basis with the aid of the context-side annotations (§5.1) and the type-class mechanism (§5.3).

Introducing twin types affects other parts of the unification algorithm which have to preserve the additional information in the constraints, including optimizations such as argument omission in projection-like functions and the detection of head-injective functions. Finally, parts of the user interface also needed to be adapted in order to display the additional information carried by the twin types and clarify the type errors arising from them. This is a cross-cutting concern which affected all tasks.

5.13 Beyond Agda

In this chapter we have shown the main changes that needed to be performed in an existing type-checker in order to implement our approach to higher-order unification, using Agda as a concrete example.

Our development work was guided by the type-level annotations (by means of the OnSide wrapper) that we added to indicate on which side terms reside. This allowed us to leverage the Haskell type-checker and the existing type-class machinery to ensure that each side of a constraint or of a twin type is manipulated in the right context.

The inclusion of a set of problem identifiers with every twin type allows for a straightforward evaluation of whether both sides of a context or a twin type are equal. This can be used to minimize the need for redundant term comparisons, for instance when assessing whether the conditions required for instantiating a metavariable or simplifying a twin type are fulfilled.

Finally, enforcing the preconditions for metavariable instantiation may in some cases require checking types for convertibility, which can entail additional, expensive reductions. We show how to prioritize constraints so that these more expensive checks are only performed as a last resort.

We expect that the techniques that we have used for the implementation in Agda will also be helpful when implementing the approach into other proof assistants.

Chapter 6

Evaluation

In this chapter we evaluate whether our approach is a practical alternative for dependent type checking with implicit arguments. The evaluation is based on the implementation described in Chapter 5.

6.1 Implementation effort

We aim to produce a scheme for heterogeneous unification that can be implemented with relatively low impact and low effort into an existing dependent type checker. This means that the amount of person-hours required and the amount of lines of code affected should be minimized.

Methodology: We first evaluate the programmer effort of the implementation itself. This includes the amount of time that was required, and the amount of lines of code that needed to be added, modified, or, in some cases, deleted. The time data is self-reported and given with a granularity of weeks. The number of lines added, modified and deleted is obtained by using the `--word-diff` flag of the `git-diff` tool [86].

Programmer time: The implementation of the basic functionality for heterogeneous unification into Agda was done over the course of 18 weeks at a rate of approximately 25 hours of development work per week (Table 6.1).

The infrastructure changes noted in Table 6.1 amount to those described in §5.1 together with the addition of new data types to supplement existing functionality, and the implementation corresponding type class instances that generalize existing functionality to those types. The "Metavariable instantiation" item corresponds to the implementation of the checks described in §5.7. Unification of binders corresponds to the changes described in §5.9, and the unification of spines to those described in §5.10. The comparatively long time required to implement the unification of spines was due to the fact that enabling this feature introduced many new twin types into the constraints, which among other issues exposed the need for the more powerful twin equality check described in Listing 5.1 and the prioritization machinery that regulates the use of this check (§5.6). The line "Type-directed equality on terms" in Table 6.1

Task	Time (weeks)
Infrastructure (§5.1)	3
Unification of binders (§5.9)	2
Metavariable instantiation (§5.7)	1
Type-directed equality on terms (§5.12)	2
Context currying (Rule-Schema 20)	1
Unification of elimination spines (§5.10)	4
Head-injectivity analysis (§5.12)	1½
Projection-like functions (§5.12)	1½
Fixing of major bugs, refactoring and testing	2
Total	18

Table 6.1: Effort required to implement the functionality required in Chapter 5. Each week corresponds to approximately 25 grad-student-hours.

covers our inspection and modification of the unification algorithm to ensure that operations on the involved terms are performed in the right context. The effort required for some of these modifications is accounted separately under the items "Head-injectivity analysis" and "Projection-like functions".

As these changes entail a relatively major change in Agda's conversion checker algorithm, additional testing and bug fixing will almost certainly be required before the implementation can be merged into the main branch. This time is not included in Table 6.1

Code changes: Another goal was to minimize the amount of changes required to the type checker itself. Table 6.2 summarizes the amount of changes required to the Agda code. Our approach did not require changes to the term syntax; therefore the majority of the additions and changes are concentrated in the constraint solver itself (TypeChecking.Conversion), the data structures representing the context and the constraints (TypeChecking.Monad.Base), specialized functions and instances to manipulate the aforementioned data structures (TypeChecking.Heterogeneous), and the code required to check the additional prerequisites for metavariable instantiation (TypeChecking.MetaVars). The lines deleted from TypeChecking.Conversion correspond to the removal of the code related to anti-unification, which is part of Agda's partial solution to the spine problem. We expect that this removal will prevent future bugs related to the implementation of anti-unification and its interaction with other features of Agda.

6.2 Functional testing

To assess the viability of our approach we need to check whether Agda.ε is sufficiently powerful to type-check the programs that Agda users currently write.

Agda includes a test suite in order to help Agda developers avoid regressions when making changes to the Agda codebase. This test suite at the point of our implementation consists of 1568 test cases of programs which successfully

Haskell module	Lines	Added	Modified	Deleted	% edited
TypeChecking.Conversion	2175	409	197	91	28%
TypeChecking.Monad.Base	4316	494	24	0	12%
TypeChecking.Heterogeneous	369	369	0	0	100%
TypeChecking.MetaVars	1546	105	36	0	9.1%
TypeChecking.Reduce	1397	71	34	0	7.5%
TypeChecking.Monad.Context	474	53	40	0	20%
Syntax.Internal.Blockers	288	69	8	0	27%
TypeChecking.Constraints	332	44	19	0	19%
TypeChecking.MetaVars.VarSetBlocked	63	63	0	0	100%
Utils.IntSet.Typed	60	60	0	0	100%
TypeChecking.Injectivity	432	19	37	0	13%
TypeChecking.Monad.Constraints	222	28	13	0	18%
TypeChecking.Pretty	419	36	3	0	9.3%
Interaction.JSONTop	429	33	4	0	8.6%
TypeChecking.Records	816	6	28	2	4.2%
Interaction.BasicOps	1194	14	15	0	2.4%
Utils.Dependent	29	29	0	0	100%
TypeChecking.Conversion.Pure	150	14	12	1	17%
Syntax.Translation.InternalToAbstract	1283	24	2	0	2%
Syntax.Internal	1183	17	6	0	1.9%
Other modules	22848	135	110	10	1.1%
Total	40025	2092	588	104	6.7%

Table 6.2: List of modules in Agda.ε, in descending total number of lines added or changed. For each module, the line count for the corresponding file in Agda.ε (after our changes) is shown, followed by the number of lines of code added, deleted and changed in each module with respect to the original Agda implementation.

type-check (Succeed), 1263 test cases of programs which are not expected to type-check (Fail), and 415 test cases for interactive theorem proving, among others.

Our approach introduces additional checks before metavariables are solved. This results in the test cases for Bugs.Issue3027 and Bugs.Issue3027b are no longer known bugs; they become failing test cases, as was desired. These additional checks also mean that certain tests do not pass any longer. This happens either because of the use of features outside of the scope of our development, such as sized types or cubical type theory; changes in the specific way in which certain non-well-typed programs fail to type-check or, in a handful of cases, due to inherent limitations in our approach, as discussed in §6.5.

6.3 Addressed bugs in Agda

Our development was motivated by certain long-standing issues in the Agda implementation (see §1.3, §1.4). In some cases, temporary workarounds had been put in place to fix these internal errors. These workarounds, although they prevent the type checker from crashing for a given class of examples, rely on disabling internal checks for desirable invariants and are thus not definitive solutions. We aim to avoid this kind of errors by preventing the creation ill-typed terms that cause internal errors. Here we comment on some reports of such internal errors, and how Agda.ε addresses them.

Issue #1467: In this issue [25], overly-optimistic constraint elimination leads to inconsistent constraints. The issue was fixed by re-checking whether there are any pending demonstrably unsolvable constraints (which would be indicative of a type error) before the internal error is triggered. In Agda.ε this additional check is no longer necessary for this particular case, which we consider indicative of a more solid approach.

Issue #2709: The issue is triggered by the instance argument machinery. It is a brittle test case; small changes in the input can cause the error not to trigger any longer. A reduced version by Ulf Norell [74] triggers the error in Agda, but that same test file does not trigger the error any longer in Agda.ε. We consider this as evidence of our approach being more principled.

Issue #3027: This bug [80] originally inspired our implementation. This is a long-standing issue in Agda where a metavariable is instantiated to a term which is not of the appropriate type, causing an erroneous reduction further down the line. Agda.ε prevents the instantiation, thus fixing the internal error.

Issue #3870: This issue [75] is another instance where instance search results in ill-typed terms, which in Agda may result in ill-typed constraints. The issue was temporarily solved by silencing the internal error when it occurred in the context of an instance search [76]. This is a workaround which may hide important bugs in the future. Agda.ε removes this work-around without triggering the reported error.

6.4 Performance evaluation

Our second concern when assessing the practicality of our approach is whether Agda.ε has comparable performance to the Agda implementation it is based on. In this section we answer this question by measuring the resource usage of the implementation when type-checking a selection of benchmarks and existing Agda projects.

6.4.1 Methodology

We perform measurements on a Thinkpad T480 laptop with 32GB RAM and an Intel(R) Core(TM) i7-8550U CPU. We measure the total amount of CPU time taken by Agda to type-check the example, and the maximum resident memory of the corresponding Agda process. Due to several factors, including intermitent thermal throttling of the processor and concurrent system processes, these measurements can be quite noisy. To obtain more precision, we average measurements over $n = 40$ executions.

In order to average the measurements, we use the geometric mean. The geometric mean has been used in certain SPEC benchmark suites [57, §9.1.3] due to its algebraic properties and its ability to de-emphasize unusually large data points, compared to the arithmetic mean. For each time and memory measurement, we report a 95% confidence interval around each estimated mean. Due to the use of the geometric mean, the resulting intervals are not symmetric; for readability, we report the largest symmetric interval that contains the calculated interval.

In order to compare the performance of Agda.ε with the Agda baseline, we calculate, for each test case, a 95% confidence interval for the average ratio of the resource usage between Agda.ε (numerator) and Agda (denominator). We then render these in terms of a percentage increase; i.e. $100 \cdot (r - 1)$, where r is the calculated ratio. The confidence intervals for the percentages are color-coded depending on whether they contain only positive values, both positive and negative values, or only negative values.

Appendix A contains more details on the statistical model used to calculate the confidence intervals and how it compares to the empirical distribution of the measurements.

6.4.2 Project benchmarks

We tested Agda.ε on three large software developments. This selection of developments is meant to cover a range of Agda features that are used in the code that Agda users may write.

Agda standard library A collection of "tools needed to write both programs and proofs easily", written by Danielsson et al. [26].

Agda prelude An "alternative to the Agda standard library that focuses more on programming and type checking time performance. [...] Makes heavy use of instance arguments.". It is written mainly by Norell [81].

Table 6.3: CPU time for type-checking existing projects developed with Agda.

	Mean CPU (s)		
Project	**Agda.ε**	**Δ%**	**Agda (baseline)**
agda-prelude	110 ± 1.2	+5.1% … +1.8%	106 ± 1.2
agda-stdlib	531 ± 7.8	+0.64% … -2.8%	537 ± 5.3
HoTT-Intro	223 ± 2.2	+5.9% … +3.1%	214 ± 2.1

Table 6.4: Resident memory when type-checking existing projects developed with Agda.

	Mean Memory (MB)		
Project	**Agda.ε**	**Δ%**	**Agda (baseline)**
agda-prelude	683 ± 15	+6% … -0.92%	666 ± 17
agda-stdlib	1870 ± 13	-3.3% … -4.8%	1950 ± 8.2
HoTT-Intro	2100 ± 37	-2.4% … -6.9%	2210 ± 36

HoTT-Intro An in-progress "Agda formalization of the Introduction to Homotopy Type Theory book by Egbert Rijke", written mainly by the book's author [89].

The performance of Agda and Agda.ε when type-checking the three projects is shown in Table 6.3 and Table 6.4. Confidence intervals for the relative increase in CPU and memory usage are plotted in Figure 6.5. We observe that in all three cases, the performance of Agda.ε is comparable to that of the Agda baseline.

6.4.3 Context singleness check

As explained in Chapter 5, some operations such as metavariable instantiation (§5.7) or simplifying twin types (§5.8) require checking whether the two sides of the context are heterogeneously equal. This operation can be linear in the size of the context. Thus, if this operation is repeated at every level of a term with a large amount of nested binders, this can result in a quadratic number of such checks. A similar issue had already been triggered in Agda during the implementation of an unrelated feature [13].

To mitigate this issue, we include the set of problem identifiers associated with a context in the context itself in a way in which its computation is shared by all further extensions of the context (§5.2). In Figure 6.6 we see how

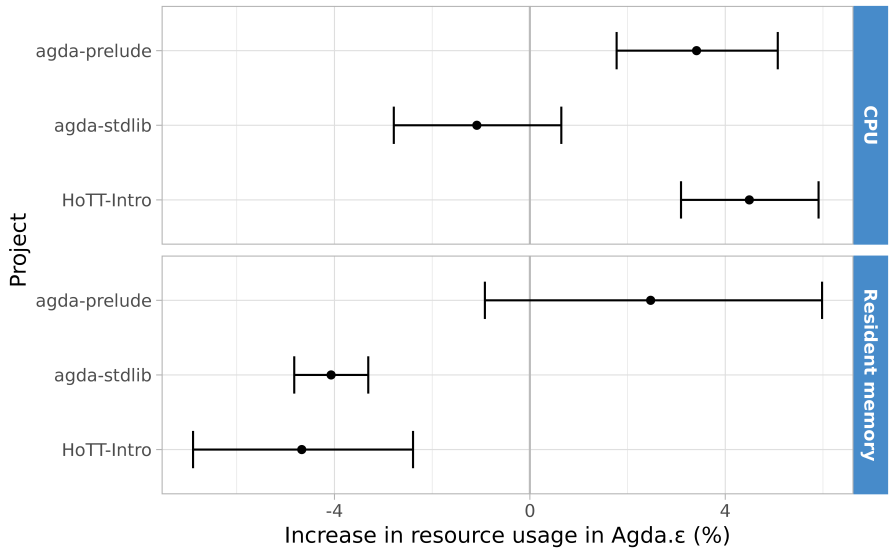


Figure 6.5: Increase in resource usage when type-checking selected projects with Agda.ε compared to the Agda baseline (95% CI).

type-checking the test case for the aforementioned issue is much faster after optimizing the context representation.

6.4.4 Other benchmarks

We have also compared the performance of Agda.ε and Agda on a selection of examples from the Agda benchmark suite. These benchmarks, while not representative of all the ways that Agda is used in practice, reflect specific situations in which Agda formerly performed particularly bad, either by the developers or by the Agda users. Studying these benchmarks helped us for example discover the issues that lead us to the optimizations for the fast context singleness check (§5.7).

The results of these benchmarks are collected in Table 6.7 and Table 6.8. Confidence intervals for the relative increase in CPU and memory usage for each benchmark are plotted in Figure 6.9. We observe that, for this selection of benchmarks, the performance of Agda.ε is comparable to that of the original Agda, with the upper-bounds of the confidence intervals for the increase in resource usage always staying below +10%.

6.5 Limitations and future work

In this chapter we have show how Agda.ε can be used to type-check a range of existing code while retaining a similar performance to Agda. However, there are still issues to solve in Agda.ε before the changes can be integrated into Agda proper.

Table 6.7: CPU usage when type-checking examples from the benchmark suite included with the Agda implementation.

	Mean CPU (s)		
Benchmark	**Agda.ε**	**Δ%**	**Agda (baseline)**
categories/Categories	0.441 ± 0.012	+4.5% … -2.4%	0.436 ± 0.01
categories/Language	0.553 ± 0.012	+6.9% … +0.22%	0.534 ± 0.013
categories/Language3	0.373 ± 0.0089	+4.4% … -2.3%	0.37 ± 0.0092
categories/Primitive	0.271 ± 0.0057	+7% … +0.55%	0.261 ± 0.0062
examples/CwF	2.65 ± 0.065	+5% … -1.9%	2.61 ± 0.064
examples/FunctorComposition	0.396 ± 0.0074	+5.2% … -1.1%	0.388 ± 0.0099
examples/Issue4044	0.587 ± 0.016	+9.7% … +2%	0.555 ± 0.014
misc/Coverage	0.407 ± 0.0082	+4.5% … -2%	0.403 ± 0.011
misc/Ids	5.92 ± 0.15	+0.55% … -5.5%	6.07 ± 0.12
misc/LateMetaVariableInstantiation	0.126 ± 0.0026	+2.7% … -3.1%	0.127 ± 0.0028
misc/SlowOccurrences	1.14 ± 0.027	+4.7% … -1.7%	1.12 ± 0.025
monad/Monad	37.9 ± 0.85	+2.5% … -3.4%	38.1 ± 0.81
proj/Data	0.265 ± 0.0061	+9% … +2.6%	0.251 ± 0.0052
proj/Nested	0.26 ± 0.0052	+6.4% … -0.044%	0.252 ± 0.0063
proj/Record	0.273 ± 0.0054	+6.8% … +0.61%	0.263 ± 0.0062
std-lib/Any	119 ± 2.9	+4.6% … -2.2%	117 ± 2.8
Syntacticosmos/Syntacticosmos	3.88 ± 0.098	+5.8% … -0.8%	3.79 ± 0.082

Table 6.8: Resident memory when type-checking examples from the benchmark suite included with the Agda implementation.

	Mean Memory (MB)		
Benchmark	**Agda.ε**	**Δ%**	**Agda (baseline)**
categories/Categories	105 ± 0.018	+1.7% … +1.6%	103 ± 0.074
categories/Language	108 ± 0.018	+2.5% … +2.5%	106 ± 0.018
categories/Language3	96.4 ± 0.22	+2.9% … +2.1%	94.1 ± 0.29
categories/Primitive	90.7 ± 0.1	+3.6% … +3.4%	87.7 ± 0.029
examples/CwF	234 ± 0.86	+3.7% … +2.6%	227 ± 0.85
examples/FunctorComposition	103 ± 0.09	+3.5% … +2.4%	99.7 ± 0.49
examples/Issue4044	119 ± 0.19	+2.1% … +1.7%	117 ± 0.11
misc/Coverage	99.2 ± 0.039	+3.6% … +3.5%	95.8 ± 0.04
misc/Ids	1220 ± 8	+0.87% … -0.74%	1220 ± 5.9
misc/LateMetaVariableInstantiation	83.1 ± 0.056	+3.3% … +3.2%	80.5 ± 0.024
misc/SlowOccurrences	188 ± 0.068	+2.1% … +1.9%	184 ± 0.11
monad/Monad	335 ± 3.3	+1.9% … -0.73%	333 ± 3.1
proj/Data	90 ± 0.071	+3.3% … +3.1%	87.2 ± 0.058
proj/Nested	89.5 ± 0.044	+3.5% … +3.3%	86.6 ± 0.038
proj/Record	90 ± 0.058	+3.4% … +3.2%	87.1 ± 0.088
std-lib/Any	813 ± 2.8	+0.17% … -0.9%	816 ± 3.5
Syntacticosmos/Syntacticosmos	120 ± 0.33	+3.6% … +2.7%	116 ± 0.35

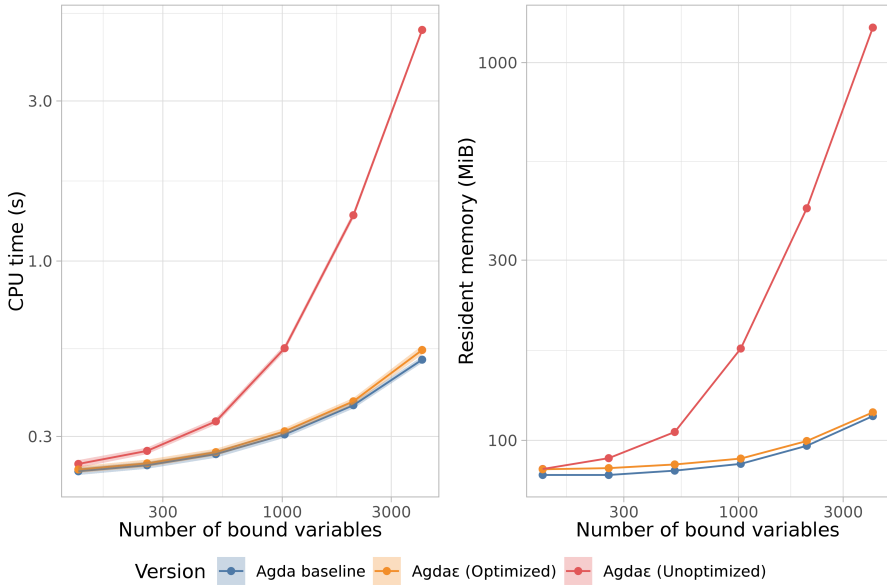


Figure 6.6: Impact of context-singleness optimization on CPU time and resident memory (logarithmic scale on both axes).

Unsolvable problems

The implementation of our approach entails the introduction of more stringent checks before metavariables are instantiated; i.e before implicit arguments are inferred. This means that certain examples which used to type-check earlier do not type-check any longer. These examples can be made to type-check again by giving some additional information to the type checker, i.e. giving some implicit arguments explicitly. Among the benchmarks evaluated in §6.4.2 and §6.4.4, we found four cases in the standard library where arguments that are inferred by Agda had to be given explicitly for Agda.ε to type-check the file [55]. As for the Agda test-suite (§6.2), four successful cases and the two failure cases needed to be modified in order to add extra information.

Other cases where additional arguments need to be provided have been discovered in user projects, such as the one reported by Danielsson [24]. The constraints that prevent this example from type-checking are described in more detail in §4.7.

It is not obvious how to fix these issues cleanly, as they inherently require instantiating metavariables with terms that are not yet of the right type at the point where the instantiation needs to occur, and the constraints that need to be solved for the instantiation to be well-typed depend in turn on the same instantiation in order to become solvable. If the techniques implemented in Agda.ε are adopted by the main Agda implementation, the users will need to modify their existing code to give the arguments that Agda.ε fails to infer explicitly. If the Agda developers want to preserve the status quo and avoid the need to give the arguments explicitly in these cases, the user may be given the option to specify a maximum number of allowed non-well-typed instantiations

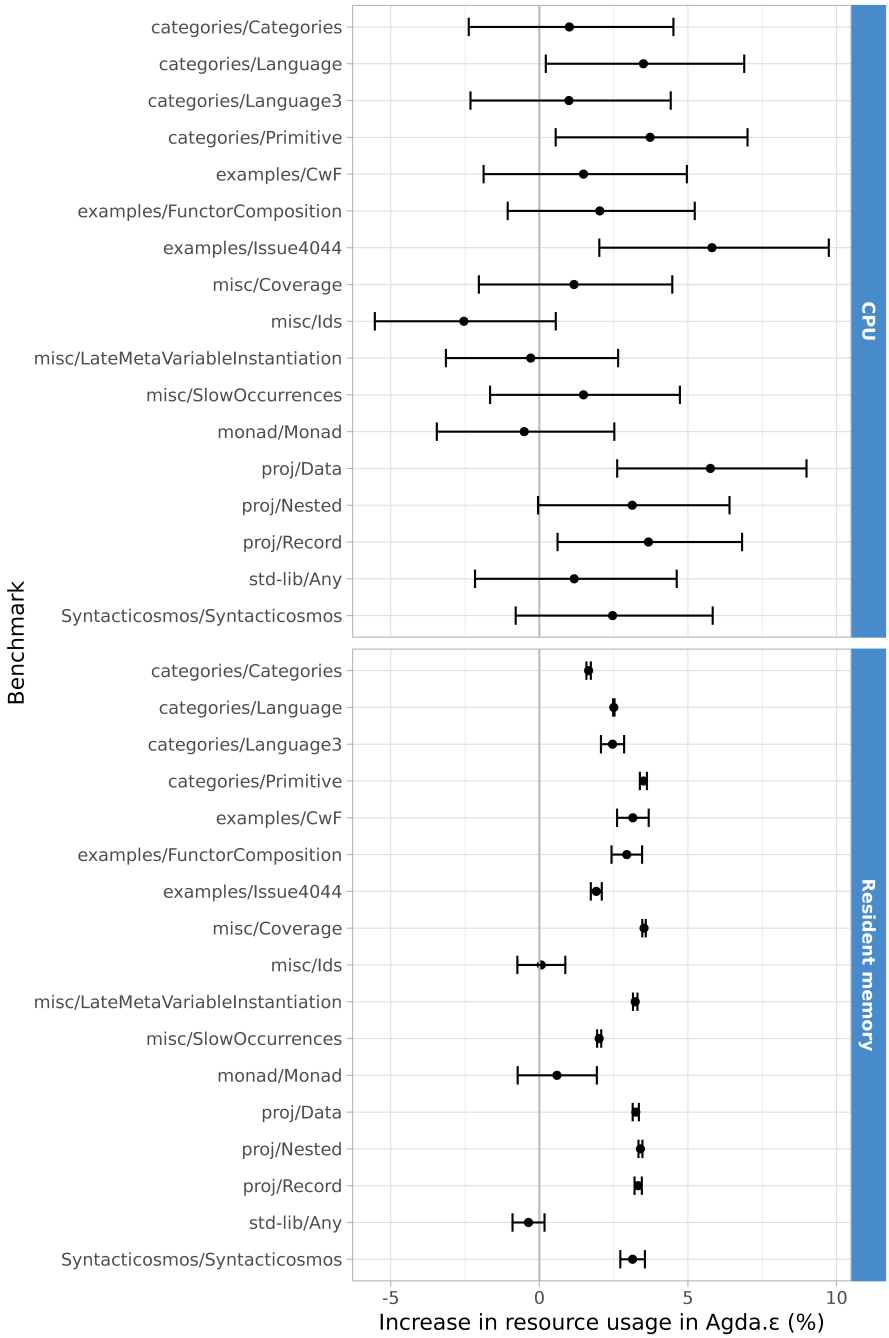


Figure 6.9: Increase in resource usage when type-checking selected benchmarks with Agda.ε compared to the Agda baseline (95% CI).

as a per-module option, in the vein of the approach taken by certain parser generators for resolving shift-reduce conflicts [38, §5.2]. The postponement machinery described in 5.6 can be used in order to avoid performing these potentially ill-typed metavariable instantiations until no other constraints can be solved.

Cubical type theory

Agda supports cubical type theory, a theory which gives computational meaning to univalence and higher inductive types [102]. Due to enabling out-of-order, type-directed unification, our theory is well-suited to support cubical Agda. The implementation of our approach in Agda.ε does not support the Cubical Agda functionality. However, after a discussion with the original implementor, we believe that our approach can also be adapted to this part of the proof assistant.

Testing of TypeTopology project

Running our implementation on Escardo's TypeTopology project [37] results the type checker taking a long time to type-check a specific definition, which forced us to abort it. Similar situations have arisen with this project in the past, as it relies on the Agda type checker keeping certain terms unnormalized in order to be able to be type-checked in a reasonable amount of time. An unsolved meta can cause further normalization resulting much longer type-checking times. Further analysis is required to determine what the exact cause of the slow-down is.

Testing of Agda categories library

We have attempted to evaluate the Agda categories library [14]. However, we were unable to find a version that ran with the specific combination of versions of Agda and the standard library that we have used for our implementation.

Universe levels

An issue has been reported in Agda [92] apparently arising from inconsistencies in the inference of universe levels. This issue is similar to #3027 (§6.3), in that it generates an ill-typed constraint. However, in this case the ill-typed constraint is about mismatched levels for two types. Level constraints exist separate from the constraints on terms, and types can be compared regardless of which level they are at, so this is not addressed by our solution.

Further optimization

In the licentiate thesis [54, §5.6] we analyzed a series of examples based on a paper by McBride [64]. The code we used was an adapted version of the `categories/Language` Agda benchmark. When analyzed in our modified version of Agda, the adapted benchmark type-checks much more slowly in Agda.ε than the adapted benchmark does in Agda, and than the original `categories/Language` benchmark does in either Agda.ε or Agda (which performs adequately in both). We presume that Agda.ε might be solving certain

constraints in a different order than Agda, thus preventing certain optimizations from applying, resulting in more term normalization and/or larger terms. Further analysis is needed to ascertain what the exact cause is.

Other features

There are some other features of Agda which are not yet fully supported in Agda.ε. This includes sized types, subtyping, and universe levels. The first two are somewhat experimental in character. As for universe levels, they are introduced in Agda to ensure that the theory is properly stratified, so that certain inconsistencies can be avoided [39]. Although they were not considered in our theoretical development §2.12, we have kept the existing Agda infrastructure for them in Agda.ε, and have not observed any issues with them arising from our changes. Full support may require strengthening the preconditions of type equality (3a) and context equality (3b) in Rule-Schema 2 in order to take into account not only whether the types in the context and in the constraint match, but also whether the constraints associated to their universe levels have been solved.

6.6 Comparison with other proof assistants

Regarding proof assistants based on dependent types, there is a good amount of variation in the features that the underlying type theory supports and in how proofs and programs are written. This makes it difficult to perform one-to-one comparisons of performance on meaningfully large examples. It is however possible to compare the capabilities of existing proof assistants by observing how they behave on certain specially-chosen unification constraints.

These constraints, which unify two types, are realized as a (i) term whose type is inferred, and (ii) a type against which the inferred type is compared. We expect that instantiating metavariables so that the program type-checks is equivalent to solving the constraint that the example is derived from.

For easier reading, we will use the term syntax from Chapter 2 when describing the constraints involved.

6.6.1 Coq, Matita and Lean

Coq is a proof assistant which has been used for large projects in formalization of mathematics [40] and in formal verification of software [51]. We have evaluated Coq 8.13.2, released on May 2021.

The main issue that motivated our approach is the binder problem (§1.3). Here we implement a version thereof into Coq, distilled from the more complex unification problem in Example 4.55.

Example 6.1 (Binder problem)**.** An instance of the binder problem distilled from Example 4.55.

We define the following shorthands:

$$\begin{array}{lll} \texttt{U} & \stackrel{\text{def}}{=} \text{Bool} \times \text{Bool} & : \text{Set} \\ \texttt{set} & \stackrel{\text{def}}{=} \langle \text{true}, \text{false} \rangle & : \texttt{U} \\ \texttt{El}\ \ (u : \texttt{U}) & \stackrel{\text{def}}{=} \text{if}\,(\lambda.\text{Bool})\,(u\,.\pi_1)\,\text{true}\,(u\,.\pi_2) & : \text{Bool} \end{array}$$

The problem is then as follows:

$$\mathbb{F} : \text{Bool} \to \text{Set}, \mathbb{G} : \texttt{U} \to \text{Set}, \alpha : \text{Bool} \to \texttt{U}\ \texttt{;}$$
$$x : \text{Bool} \vdash \mathbb{F}\,(\texttt{El}\,(\alpha\,x)) \to \mathbb{G}\,(\alpha\,x) : \text{Set} \cong \mathbb{F}\,\text{true} \to \mathbb{G}\,\texttt{set} : \text{Set}$$

◀

The Coq implementation is detailed in Listing 6.10. The constraints generated by Coq might not be identical to the ones in our examples, but our tests suggest they are analogous to them.

Listing 6.10: Rendering of Example 6.1 in Coq, together with the output from the type checker. We use the symbol `_` *to induce Coq to create a metavariable, which we name* `alpha`*.*

```
(* 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  => true
    | false => (snd u)
  end.

(* Atoms *)
Axiom F : bool -> Set.
Axiom G : U -> Set.

(* Metavariables and constraints *)
Definition c1 :
      let alpha : bool -> U := _ in
       forall (x: bool),
       (F (El (alpha x)) -> G (alpha x)) ->
       (F true           -> G set') :=
     fun x y => y.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
File "Pi.v", line 21, characters 16-17:
Error:
In environment
x : bool
y : F (El (?u x)) -> G (?u x)
The term "y" has type "F (El (?u x)) -> G (?u x)"
while it is expected to have type "F true -> G set'".

[Exit code: 0]
```

In this example, the definition of the function `c1` forces the type of the second argument (`F (El (alpha x)) −> G set'`) and the type of the result of the function (`F true −> G set'`) to coincide. As explained in §1.3, Coq unifies function types by first unifying the domains, and then unifying the codomains. We presume that Coq's algorithm has no strategy to unify the domains in this case, as they cannot be further simplified and the resulting constraint does not fall into the pattern fragment (§3.5). Because the domains cannot be unified, Coq gives up on unifying the two Π-types altogether and reports them as unequal (Listing 6.10, Output).

Listing 6.11: Example in Listing 6.10 modified so that the codomains are equal.

```
(* 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  => true
    | false => (snd u)
  end.

(* Atoms *)
Axiom F : bool −> Set.
Axiom G : U −> Set.

(* Metavariables and constraints *)
Definition c1 :
      let alpha : bool −> U := _ in
       forall (x: bool),
        (F true  −> G (alpha x)) −>
        (F true  −> G set') :=
     fun x y => y.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
[Exit code: 0]
```

In order to confirm our hypothesis, we attempt to simplify the example so that the domains are equal (Listing 6.11). In this case, Coq can use the information in the codomain to find a solution (Listing 6.11, Output). Giving the solution to the metavariable directly (`let alpha : bool −> U := fun x =>set'`), also allows the code to type-check (Listing 6.12).

Listing 6.12: Example in Listing 6.10 modified so that the solution to the metavariable is given.

```
(* 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   => true
    | false  => (snd u)
  end.

(* Atoms *)
Axiom F : bool -> Set.
Axiom G : U -> Set.

(* Metavariables and constraints *)
Definition c1 :
      let alpha : bool -> U := fun x => set' in
       forall (x: bool),
       (F (El (alpha x)) -> G (alpha x)) ->
       (F true           -> G set') :=
     fun x y => y.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
[Exit code: 0]
```

Spine problem

We have observed in our tests that the spine problem (§1.3) in Coq is addressed in the same vein as the binder problem, by unifying the arguments from left to right. This ensures that all of the resulting constraints are homogeneous and that the resulting metavariable instantiations are well-typed.

An example of the shortcomings of this approach is that the order of the arguments can determine whether an example can be type-checked or not. Consider the example in Example 6.2, also distilled from Example 4.55. We implement this in Listing 6.13. Similarly to Listing 6.10, Coq attempts to unify the type of the last argument of the function and the type of its result. This type is in both cases a family of types with two arguments. Coq cannot unify the first two arguments, and thus gives a type error.

Example 6.2 (Spine problem)**.** An instance of the binder problem distilled from Example 4.55.

We define the following shorthands:

$$
\begin{array}{lll}
\texttt{U} & \stackrel{\text{def}}{=} \text{Bool} \times \text{Bool} & : \text{Set} \\
\texttt{set} & \stackrel{\text{def}}{=} \langle \text{true}, \text{false} \rangle & : \texttt{U} \\
\texttt{El}\ \ (u : \texttt{U}) & \stackrel{\text{def}}{=} \text{if}\,(\lambda.\text{Bool})\,(u\,.\pi_1)\,\text{true}\,(u\,.\pi_2) & : \text{Bool}
\end{array}
$$

The problem is then as follows:

$$
\begin{array}{l}
\mathbb{P} : \text{Bool} \to \texttt{U} \to \text{Set}, \alpha : \text{Bool} \to \texttt{U}\ \texttt{;} \\
x : \text{Bool} \vdash \mathbb{P}\,(\texttt{El}\,(\alpha\,x))\,(\alpha\,x) : \text{Set} \cong \mathbb{P}\,\text{true}\,\texttt{set} : \text{Set}
\end{array}
$$

◀

Listing 6.13: Rendering of Example 6.2 in Coq, together with the output from the type checker.

```
(* Atoms *)
Axiom F : bool -> 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  => true
    | false => (snd u)
  end.

Axiom Pred : bool -> U -> Set.

(* Metavariables and constraints *)
Definition c1 :
       let alpha : bool -> U := _ in
        forall (x: bool),
        (Pred (El (alpha x)) (alpha x)) ->
        (Pred  true          set') :=
      fun x y => y.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
File "SpineMini.v", line 24, characters 16-17:
Error:
In environment
x : bool
y : Pred (El (?u x)) (?u x)
The term "y" has type "Pred (El (?u x)) (?u x)"
while it is expected to have type "Pred true set'".

[Exit code: 0]
```

Switching the order of the two arguments of the Pred type family allows Coq to type-check the example (Listing 6.14). The same happens if the solution to the metavariable is given directly (Listing 6.15).

Listing 6.14: Rendering of Example 6.2 in Coq, with the order of arguments to Pred swapped, together with the output from the type checker.

```
(* Atoms *)
Axiom F : bool -> 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   => true
    | false  => (snd u)
  end.

Axiom Pred : U -> bool -> Set.

(* Metavariables and constraints *)
Definition c1 :
       let alpha : bool -> U := _ in
        forall (x: bool),
        (Pred (alpha x) (El (alpha x)) ) ->
        (Pred set'      true       ) :=
      fun x y => y.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
[Exit code: 0]
```

Listing 6.15: Rendering of Example 6.2 in Coq, with the solutoin to the metavariable given, together with the output from the type checker.

```
(* Atoms *)
Axiom F : bool -> 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   => true
    | false  => (snd u)
  end.

Axiom Pred : bool -> U -> Set.

(* Metavariables and constraints *)
Definition c1 :
       let alpha : bool -> U := fun z => set' in
        forall (x: bool),
        (Pred (El (alpha x))  (alpha x)) ->
        (Pred true             set') :=
      fun x y => y.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
[Exit code: 0]
```

First-order unification

In the introduction we describe a generic example of a program in which an implicit argument had more than one possible solution (Listing 1.1). Although Coq allows non-unique solutions in some cases, it does reject that example. This is expected as there is no indication that a specific value for the implicit argument is more likely to be intended by the user.

In other cases, Coq uses first-order unification to infer the solution that the user most-likely intended. Applying first-order unification matches the terms syntactically, thus prioritizing solutions that can be obtained without normalizing terms. However, there exist examples where the intentions of the user are ambiguous, and which type-check nevertheless.

Listing 6.16 shows an example of first-order unification in action. This listing defines a datatype for vectors (lists indexed by their length), and two functions: `append`, which concatenates two vectors; and `repeat`, which, similarly to the example in the introduction, produces a vector of a given length with copies of the same value.

Using these two functions, we define `mkVec`, which creates a vector with m copies of the value `true` followed by n copies of the value `false`. For example, `mkVec 1 2` produces the list [true; false; false].

Calling `mkVec _ _` means that Coq should infer the arguments itself, based on the expected type of the expression. The result of the call to `mkVec` depends on these inferred arguments, so we can use `Eval` to observe which values were inferred. By looking at the Coq type checker output (Listing 6.16, Output), we can see that Coq uses first-order unification between the inferred type of `t (_ + _)` and the expected types (`t (0 + 2)` and `t (2 + 0)`) to disambiguate between them.

In the case of definition `list3`, there are at least two possible evaluations: [false; false; false; false] or [true; true; true; true]. The user could arguably have any of those in mind, as each of the expected types for the arguments of `append` suggest a different possibility. Predicting which of the two possible results will be produced would require some level of knowledge of the workings of Coq's unification algorithm.

Matita

We have evaluated Example 6.1 and Example 6.2 in Matita 0.99.3 [11] and obtained similar behaviour as with the analogous programs in Coq. The code used for the evaluation and the corresponding type checker output are detailed in Appendix B.2.

Lean

We have also evaluated Example 6.1 and Example 6.2 in Lean 3.28.0 [28] and obtained similar behaviour as with the analogous programs in Coq. The code used and the corresponing type checker output are detailed in Appendix B.3.

Listing 6.16: Example of first-order unification in Coq

```
Inductive t : nat -> Type :=
  | nil : t 0
  |cons : forall (n:nat), bool -> t n -> t (S n).

Notation "[ x ; y ; .. ; z ]" :=
  (cons _ x (cons _ y .. (cons _ z nil) ..)).

Fixpoint append (m n : nat) (x : t m) (y : t n) : t (m + n) :=
  match x with
  | nil => y
  | cons _ b x' => cons _ b (append _ _ x' y)
  end.

Fixpoint repeat (n : nat) (b : bool) : t n :=
  match n with
    | O => nil
    | S m => cons m b (repeat m b)
  end.

Definition mkVec (m n : nat) : t (m + n) :=
  append _ _ (repeat _ true) (repeat _ false).

Definition list1 : t (0 + 2) := mkVec _ _.
Eval cbv in list1.

Definition list2 : t (2 + 0) := mkVec _ _.
Eval cbv in list2.

Definition list3 : _ :=
  let a := mkVec _ _ in
  append (0+2) (2+0) a a.
Eval cbv in list3.
```

Output:

```
The Coq Proof Assistant, version 8.13.2 (May 2021)
compiled on May 28 2021 15:28:51 with OCaml 4.11.1
     = [false; false]
     : t (0 + 2)
     = [true; true]
     : t (2 + 0)
     = [false; false; false; false]
     : t (0 + 2 + (2 + 0))
[Exit code: 0]
```

6.6.2 Idris 2

Idris is "a programming language designed to encourage type-driven development" [95]. The type checker in Idris 1 avoided the binder problem by enforcing a strict ordering of constraints, as described in the licentiate thesis [54, §5.7.1]. The solution to the binder problem in Idris 2 (v0.3.0, January release) involves replacing the variables of conflicting type with a guarded constant. That is, a constraint $\Gamma \vdash \Pi A B \approx \Pi A' B'$ is reduced to the constraints $\Gamma \vdash A \approx A'$ and $\Gamma, x : A \vdash B \approx B'[p/x]$, where p is a constant of type A' that reduces to x when the constraint $\Gamma \vdash A \approx A'$ is solved. This trick enables Idris 2 to solve the binder problem (Listing 6.17).

Listing 6.17: Rendering of Example 6.1 in Idris, together with the output from the type checker. We use the symbol `_` *to induce Coq to create a metavariable, which we name* `alpha`*.*

```
module Pi

-- 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)

-- Atoms
data F : Bool -> Type

data G : U -> Type

-- Metavariables and constraints
c1 : (c : (alpha : Bool -> U) ->
       ((x : Bool) -> F (El (alpha x)) -> G (alpha x))) ->
       ((x : Bool) -> F True          -> G Set')
c1 c = c _
```

Output:

```
Idris 2, version 0.3.0
1/1: Building Pi (Pi.idr)
[Exit code: 0]
```

However, this approach makes a somewhat arbitrary choice about the side of the constraint on which the variable remains unaltered, and the side on which the variable is replaced by the blocked constant. Consider Example 6.3, an instance of the binder problem inspired by an issue raised by Danielsson [5], in response to a similar approach to the one in Idris 2 being implemented into Agda.

Example 6.3 (Two sided constraint)**.**

$$
\begin{aligned}
&\mathbb{F} : \mathrm{Bool} \to \mathrm{Set},\\
&\mathbb{f} : \mathbb{F}\ \mathrm{true},\\
&\alpha : \mathbb{F}\ \mathrm{true} \to \mathrm{Bool} \to \mathrm{Bool}\\
&;\\
&x : \mathrm{Bool} \vdash (z : \mathbb{F}\ (\alpha\ \mathbb{f}\ \mathrm{true})) \to \mathbb{F}\ x\\
&\approx (z : \mathbb{F}\ \mathrm{true}) \to \mathbb{F}\ (\alpha\ z\ x) : \mathrm{Set}‡\mathrm{Set}
\end{aligned}
$$

Applying the rules Rule-Schema 3 (injectivity of Π), Rule-Schema 14 (strongly neutral terms) and Rule-Schema 2 (metavariable instantiation) gives the solution $\alpha := \lambda z.\lambda x.x$. ◀

This example can be translated into an Idris 2 program. We performed two translations with different handedness. The translation in Listing 6.18 type-checks, while the translation in Listing 6.19 fails to do so . Giving the solution explicitly (e.g. `AreEqual _ (c (\ z, x =>x))`) makes the latter program type-check (Listing 6.20). Note that the variable x is bound in two places, instead of directly in the context. Also, the atomic constant $\mathbb{f}$ is rendered as a data constructor `mkFT` instead of as a top-level constant, due to our inability to find a straightforward way to introduce postulates in Idris 2. We have not observed that any of these two discrepancies makes a difference in Idris ability to solve the constraints. It is possible that Idris will not construct exactly the constraints in Example 6.4, but the results of tests and the inspection of the Idris 2 source code makes us confident that analogous constraints are indeed created.

One may expand the example so that Idris 2 cannot instantiate the metavariables even if the two sides of the constraints are swapped (see Example 6.4). The corresponding Idris code and output is detailed in Appendix B.1.

Example 6.4 (Expanded two sided constraint)**.**

$$
\begin{aligned}
&\mathbb{F} : \mathrm{Bool} \to \mathrm{Set},\\
&\mathbb{f} : \mathbb{F}\ \mathrm{true},\\
&\mathbb{G} : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Set},\\
&\alpha : \mathbb{F}\ \mathrm{true} \to \mathrm{Bool} \to \mathrm{Bool}\\
&\beta : \mathbb{F}\ \mathrm{true} \to \mathrm{Bool} \to \mathrm{Bool}\\
&;\\
&x : \mathrm{Bool} \vdash (y : \mathbb{F}\ (\alpha\ \mathbb{f}\ \mathrm{true})) \to (z : \mathbb{F}\ \mathrm{true}) \to \mathbb{G}\ (\beta\ z\ x)\ x\\
&\approx (y : \mathbb{F}\ \mathrm{true}) \to (z : \mathbb{F}\ (\beta\ \mathbb{f}\ \mathrm{true})) \to \mathbb{G}\ x\ (\alpha\ y\ x) : \mathrm{Set}‡\mathrm{Set}
\end{aligned}
$$

◀

As a possible mitigation, Idris 2 could realize that the variable `z` is only used on one side, and introduce the blocked variable on the side that does not actually mention the variable. But this is more of a heuristic than a final fix.

Listing 6.18: Idris 2 rendition of Example 6.3 (Left version). The AreEqual *type constructor forces the two sides of the given pair type to be the same. We use the data constructor* MkFT *to represent the atomic constant* 𝟙 *due to the lack of a more straightforward way to define constants of arbitrary type in Idris 2.*

```
data F    : Bool -> Type where
  MkFT : F True

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c : (alpha : F True -> Bool -> Bool) ->
      Pair  ((x : Bool) -> (z : F True)              -> F (alpha z x))
            ((x : Bool) -> (z : F (alpha MkFT True)) -> F x)) -> Type
c1 c = AreEqual _ (c _)
```

Output:

```
Idris 2, version 0.3.0
1/1: Building BlockedConstantLs (BlockedConstantLs.idr)
[Exit code: 0]
```

Listing 6.19: Idris 2 rendition of Example 6.3 (Right version). Compare with Listing 6.18.

```
data F    : Bool -> Type where
  MkFT : F True

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c : (alpha : F True -> Bool -> Bool) ->
     Pair  ((x : Bool) -> (z : F (alpha MkFT True)) -> F x)
           ((x : Bool) -> (z : F True)              -> F (alpha z x))) -> Type
c1 c = AreEqual _ (c _)
```

Output:

```
Idris 2, version 0.3.0
1/1: Building BlockedConstantRs (BlockedConstantRs.idr)
Error: While processing right hand side of c1. Can't solve constraint between: ?_
       MkFT True and True.

BlockedConstantRs.idr:9:20--9:23
    |
  9 | c1 c = AreEqual _ (c _)
    |

[Exit code: 1]
```

Listing 6.20: Example in Listing 6.19 with the solution given explicitly.

```
data F    : Bool -> Type where
  MkFT : F True

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c : (alpha : F True -> Bool -> Bool) ->
     Pair   ((x : Bool) -> (z : F (alpha MkFT True)) -> F x)
            ((x : Bool) -> (z : F True)              -> F (alpha z x))) -> Type
c1 c = AreEqual _ (c (\ z, x => x))
```

Output:

```
Idris 2, version 0.3.0
1/1: Building BlockedConstantRs_solved (BlockedConstantRs_solved.idr)
[Exit code: 0]
```

Spine problem

The spine problem is not explicitly addressed in Idris 2. Arguments in elimination spines may be unified in any order; therefore, the constraints in Example 6.2 pose no issues (Listing 6.21).

Listing 6.21: Rendering of Example 6.2 in Idris, together with the output from the type checker.

```
module SpineMini

-- 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)

data Pred : bool -> U -> Type

-- Metavariables and constraints
c1 : (c : (alpha : Bool -> U) ->
        ((x : Bool) -> Pred (El (alpha x)) (alpha x))) ->
        ((x : Bool) -> Pred True           Set')
c1 c = c _
```

Output:

```
Idris 2, version 0.3.0
1/1: Building SpineMini (SpineMini.idr)
```

```
[Exit code: 0]
```

However, the fact that the spine problem is not addressed can result in the creation of ill-typed terms. We can demonstrate this by forcing the type checker to assign the ill-typed term $(\lambda x.x\,x)$ to a metavariable $\gamma : (\text{Bool} \to \text{Bool}) \to \text{Bool}$ (Example 6.2, Listing 6.22). With some additional arguments, one can force the type checker to reduce the term $\gamma\,\gamma$ to β-normal form, which results in an infinite loop (Listing 6.23).

Example 6.5 (Spine problem causing ill-typed instantiation)**.** Consider the following shorthands:

$$\begin{aligned} \mathsf{F}\ z &\stackrel{\text{def}}{=} \text{if}\,(\lambda.\text{Set})\,z\,\text{Bool}\,(\text{Bool} \to \text{Bool}) \\ \mathsf{f}\ z\ x\ y &\stackrel{\text{def}}{=} \text{if}\,(\lambda z.\mathsf{F}\ z)\,z\,x\,y \end{aligned}$$

Using them, we define the following constraints:

$$\begin{aligned} &\mathbb{P} : (y : \text{Bool}) \to \mathsf{F}\ y \to \text{Bool} \to ((\text{Bool} \to \text{Bool}) \to \text{Bool}) \to \text{Set}, \\ &\alpha : \text{Bool} \to \text{Bool}, \\ &\beta : (\text{Bool} \to \text{Bool}) \to \text{Bool}, \\ &\gamma : (\text{Bool} \to \text{Bool}) \to \text{Bool}, \\ &; x : \text{Bool}, z : \text{Bool} \to \text{Bool} \vdash \\ &\mathbb{P}\ \text{true}\ (\beta\,z)\ (\alpha\,x)\ \gamma \\ &\approx \mathbb{P}\ (\alpha\,\text{true})\ (\mathsf{f}\ (\alpha\,\text{true})\,\text{true}\,z)\,\text{false}\ (\lambda x.x\ (\beta\,x)) : \text{Set} \end{aligned}$$

Applying Rule-Schema 14, and then applying Rule-Schema 2 without checking the preconditions results in the ill-typed instantiation $\gamma := \lambda x.x\,x$. ◀

Listing 6.22: Example 6.5 translated into Idris. Note the presence of the ill-typed subterm (\x =>x x) in the terms leading to the type error.

```
F     : Bool -> Type
F True = Bool
F False = Bool -> Bool

mkF : (b : Bool) -> Bool -> (Bool -> Bool) -> F b
mkF True x y = x
mkF False x y = y

data P : (b : Bool) -> F b -> Bool ->
         ((Bool -> Bool) -> Bool) -> Type where

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c :
    (alpha : Bool -> Bool) ->
    (beta  : (Bool -> Bool) -> Bool) ->
    (gamma : (Bool -> Bool) -> Bool) ->
```

```
18
19      Pair  ((x : Bool) ->
20              (z : Bool -> Bool) ->
21              P True (beta z) (alpha x) gamma)
22
23            ((x : Bool) ->
24              (z : Bool -> Bool) ->
25              P (alpha True) (mkF (alpha True) True z) False (\x => x (beta x))))
26      -> Type
27
28  c1 c = AreEqual _ (c _ _ _)
```

Output:

```
1  Idris 2, version 0.3.0
2  1/1: Building SpineProblemLittleOmega (SpineProblemLittleOmega.idr)
3  Error: While processing right hand side of c1. When unifying (Bool -> (z : (Bool
       -> Bool)) -> P True z False (\x => x x), Bool -> (z : (Bool -> Bool))
       -> P False (mkF False True z) False (\x => x x)) and (Bool -> (z : (Bool
       -> Bool)) -> P False (mkF False True z) False (\x => x x), Bool -> (z : (
       Bool -> Bool)) -> P False (mkF False True z) False (\x => x x)).
4  Mismatch between: True and False.
5
6  SpineProblemLittleOmega.idr:28:20--28:27
7       |
8   28  | c1 c = AreEqual _ (c _ _ _)
9       |
10
11 [Exit code: 1]
```

Listing 6.23: Expanded version of Example 6.5 translated into Idris. The Idris 2 type checker times out (which our testing code reports as exit code 124), presumably because of an attempt to reduce (\x =>x x) (\x =>x x) to β-normal form.

```
1  F    : Bool -> Type
2  F True = Bool
3  F False = Bool -> Bool
4
5  mkF : (b : Bool) -> Bool -> (Bool -> Bool) -> F b
6  mkF True x y = x
7  mkF False x y = y
8
9  G    : Bool -> Type
10 G True = Bool -> Bool
11 G False = (Bool -> Bool) -> Bool
12
13 mkG : (b : Bool) -> (Bool -> Bool) -> ((Bool -> Bool) -> Bool) -> G b
14 mkG True x y = x
15 mkG False x y = y
16
17 data P : (b : Bool) -> F b -> G b -> Bool ->
18           ((Bool -> Bool) -> Bool) -> Bool -> Type where
19
```

```
data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c :
    (alpha : Bool -> Bool) ->
    (beta  : (Bool -> Bool) -> Bool) ->
    (gamma : (Bool -> Bool) -> Bool) ->
    (delta : ((Bool -> Bool) -> Bool) -> (Bool -> Bool)) ->

    Pair  ((x : Bool) ->
           (z : Bool -> Bool) ->
           (w : (Bool -> Bool) -> Bool) ->
           P True (beta z) (delta w) (alpha x) gamma True)

          ((x : Bool) ->
           (z : Bool -> Bool) ->
           (w : (Bool -> Bool) -> Bool) ->
           P (alpha True) (mkF (alpha True) True z)
             (mkG (alpha True) (\x => x) w)
             False (\x => x (beta x)) (gamma (delta gamma))))
    -> Type
c1 c = AreEqual _ (c _ _ _ _)
```

Output:

```
Idris 2, version 0.3.0
[Exit code: 124]
```

6.7 Concluding remarks

We developed test cases for binder problem and the spine problem in several existing dependently-typed languages. We have observed that existing proof assistants may, for our test cases, prove too strict in terms of which metavariables they can solve, or too lax regarding the invariants they preserve.

Our test cases are designed to stress type checkers in very specific ways, which means it is hard to know how often analogous examples occur in practice. However, as shown by the bugs that have been reported in Agda over the years (§6.3), even though these issues stay hidden for the majority of the time, they eventually manifest themselves when the language users push the capabilities of the type checker.

We have also demonstrated how the changes implemented in Chapter 5 can address the binder and spine problems in an existing type checker, while preserving its functionality and performance. We therefore believe that the techniques we evaluated can be of use for other implementations of dependent type checking with metavariables.

Chapter 7

Conclusion

The usability of a dependently-typed programming language benefits from the ability of users to omit redundant terms from their programs and proofs. These omitted terms may be inferred by solving an associated higher-order unification problem, where the omitted terms become metavariables, and their value is determined by solving constraints derived from the typing rules. Assigning a term to a metavariable may allow the type checker to further normalize the terms in which the metavariable occurs. If the assigned term is not well-typed, the type checker may crash or loop.

Ensuring the well-typedness of intermediate terms leads to two concrete problems, which we have dubbed "the binder problem" and "the spine problem". We have developed test-cases of these problems for a range existing proof assistants, and observed that these problems are often handled in a way that either unnecessarily restrict the terms that can be inferred, or allows for ill-typed terms to appear in certain circumstances (§6.6). The twin type approach by Gundry and McBride [45, 44] tackles these issues, but it has not been implemented into a fully-fledged proof assistant.

In Chapter 4 we present an approach to higher-order unification for dependent type checking using heterogeneous constraints, in the form of a streamlined version of twin types where each side of a constraint has its own separate context and type. Our approach improves on unification with twin types by requiring no annotations on variables, making the syntax and the typing rules more closely match those of Martin-Löf Type Theory [59]. Furthermore, the soundness of our approach is based on a heterogeneous notion of equality we define in §4.2, which dispenses with the need to check that both sides of a constraint have the same type before solving a constraint by reflexivity (§4.5.1). By contrast, our notion of completeness (i.e. uniqueness of solutions) is based on a homogeneous notion equality, which means that it does not rely on any specific property of the twin types, and thus can be justified by the same reasoning that one would apply in a homogeneous setting. We expect that the aforementioned improvements make the implementation of our approach into existing proof assistants more straightforward.

We have used our experience implementing our approach in Agda (Chapter 5) as a case study for evaluating what a developer can expect when implementing our approach into an existing dependently-typed programming lan-

guage. The implementation of our approach in Agda required a moderate amount of programmer time. Few changes to the codebase were made, and these changes were mostly limited to the parts dealing with unification constraints (§6.1). The fact that our approach to the binder and spine problems preserves typing invariants addresses some long-standing bugs in Agda without the need for workarounds (§6.3). In particular, we were able to remove the anti-unification functionality which had been implemented in Agda as part of a number of workarounds addressing the spine problem.

By allowing for constraints to be solved out-of-order, our approach largely preserves the existing ability of Agda to infer implicit arguments (§6.2). We only observed a small amount of cases where the user needs to give additional information about the implicit arguments (§6.5).

Finally, the use of a specific representation for the data structures containing the information that is used to enforce the well-typedness invariants (§5.7) helps us maintain a level of resource usage comparable to the existing Agda implementation. The performance of Agda.ε when type-checking certain large Agda projects (§6.4.2), and a selection of benchmarks which had in the past made Agda perform slowly (§6.4.4) was in both cases comparable to the base Agda implementation.

From these results, we conclude that our approach is a practical way to implement higher-order heterogeneous unification when type-checking a dependently-typed language.

Appendix A

Statistical modelling of the benchmark data

We aim to analyze the increase in resource usage of our implementation Agda.ε relative to the resource usage for the baseline Agda implementation. As explained in §6.4.1, our measurements are subject to error from different sources. It is therefore important that we model this error in order to draw justified conclusions from the data.

A.1 Statistical model

Let $y_i^{a,b}$ be a random variable representing, for run $i \in \{1, ..., n\}$ and version of Agda $a \in \{\text{Agda}.\varepsilon, \text{Agda}\}$, the usage of a resource (either CPU or memory) for a given test case (for instance, type-checking the standard library), altogether defining a benchmark b. We perform $n = 40$ runs of each combination of benchmark b and version of Agda a, in order to ascertain what the average resource usage for that combination is.

In §6.4.1, we explain that we use the geometric mean to average measurements. Thus, the average resource usage across the n runs will be $\left(\prod_{i=1}^{n} y_i^{a,b}\right)^{\frac{1}{n}}$. This can be rewritten as $\exp\left(\frac{1}{n}\sum_{i=1}^{n}\log\left(y_i^{a,b}\right)\right)$, where $\frac{1}{n}\sum_{i=1}^{n}\log\left(y_i^{a,b}\right)$ is the arithmetic mean of the logarithms of the resource usage. Because each of the runs correspond to running the same program on the same machine and with limited load from concurrent processes, we have no reason to expect large extreme values in the distribution of $\log\left(y_i^{a,b}\right)$. We thus assume that these variables are thin-tailed with a well-defined expected value $\mu_{a,b}$ and (finite) variance $\sigma_{a,b}^2$. In the limit ($n \longrightarrow \infty$), the mean $\frac{1}{n}\sum_{i=1}^{n}\log\left(y_i^{a,b}\right)$ tends to the expected value $\mathrm{E}\left[\log\left(y_i^{a,b}\right)\right] = \mu_{a,b}$.

Confidence intervals for average resource usage

In the absence of additional information about the distribution of $\log\left(y_i^{a,b}\right)$ ($i = 1, ..., n$), we assume that they are independent, normally distributed random variables with mean $\mu_{a,b}$ and variance $\sigma_{a,b}^2$ (i.e.

$\log\left(y_i^{a,b}\right) \sim \text{Normal}(\mu_{a,b}, \sigma_{a,b}^2)$). For each version of Agda a and test case b, both $\mu_{a,b}$ and $\sigma_{a,b}^2$ are unknown (but non-random) constants. The constant $\mu_{a,b}$ can be understood as the value that one would obtain if we were to perform the experiment an infinite number of times and take the arithmetic mean of the logarithms of the measurements. The value $\exp\left(\mu_{a,b}\right)$ can thus be understood as the result of taking the geometric mean of an infinite number of measurements. The distribution of the $\log\left(y_i^{a,b}\right)$ can be understood as arising from each measurement being subject to an independent, normally-distributed measurement error $\log\left(\varepsilon_i^{a,b}\right) \sim \text{Normal}(0, \sigma_{a,b}^2)$. This measurement error increases (or decreases) the measured value $y_i^{a,b}$ relative to the "true" value $\exp\left(\mu_{a,b}\right)$ by a *factor* of $\varepsilon_i^{a,b}$.

Under the normality assumption, we estimate $\mu_{a,b}$ by $\hat{\mu}_{a,b} \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} \log\left(y_i^{a,b}\right)$. If we define $\hat{\sigma}_{a,b}^2 \stackrel{\text{def}}{=} \frac{1}{n-1}\sum_{i=1}^{n}\left(\log\left(y_i^{a,b}\right) - \hat{\mu}_{a,b}\right)^2$, then the 95% confidence interval ($\alpha \stackrel{\text{def}}{=} 0.05$) for $\mu_{a,b}$ is $\hat{\mu}_{a,b} \pm \frac{\hat{\sigma}_{a,b}}{\sqrt{n}}\text{t}_{n-1,1-\alpha/2}$, where $\text{t}_{n-1,1-\alpha/2}$ denotes the $1-\alpha/2$ quantile of the t-distribution with $n-1$ degrees of freedom. The confidence interval is such that if we were to repeat the experiment many more times and compute the interval again, the true $\hat{\mu}_{a,b}$ would be contained in the computed interval 95% of the time. By the same token, the geometric mean $(\exp(\hat{\mu}_{a,b}))$ would be contained 95% of the time in the interval $\exp\left(\hat{\mu}_{a,b} \pm \frac{\hat{\sigma}_{a,b}}{\sqrt{n}} \cdot \text{t}_{n-1,1-\alpha/2}\right)$. For readability, we use the larger, less precise interval which is centered at $\exp\left(\hat{\mu}_{a,b}\right)$, namely $\exp\left(\hat{\mu}_{a,b}\right) \pm \exp\left(\hat{\mu}_{a,b}\right)\left(\exp\left(\frac{\hat{\sigma}_{a,b}}{\sqrt{n}} \cdot \text{t}_{n-1,1-\alpha/2}\right) - 1\right)$.

Confidence intervals for average ratios of resource usage

In order to compare the resource usage of Agda.ε and Agda for a given benchmark b, we observe the ratio $\frac{y_i^{\text{Agda}.\varepsilon,b}}{y_i^{\text{Agda},b}}$. By the same token as when modelling the average resource usage, we will study the logarithm of this ratio, i.e. $\log\left(y_i^{\text{Agda}.\varepsilon,b}\right) - \log\left(y_i^{\text{Agda},b}\right)$. The expected value of this ratio is $r_b \stackrel{\text{def}}{=} \text{E}\left[\log\left(y_i^{\text{Agda}.\varepsilon,b}\right) - \log\left(y_i^{\text{Agda},b}\right)\right] = \mu_{\text{Agda}.\varepsilon,b} - \mu_{\text{Agda},b}$, which we can estimate by $\hat{r}_b = \hat{\mu}_{\text{Agda}.\varepsilon,b} - \hat{\mu}_{\text{Agda},b}$. The quantity $\exp(r_b)$ can be interpreted as the geometric mean of the ratios $\frac{y_i^{\text{Agda}.\varepsilon,b}}{y_i^{\text{Agda},b}}$ $(i = 1, ..., n)$ as $n \longrightarrow \infty$, and it can be estimated by $\exp(\hat{r}_b)$.

In order to calculate confidence intervals for $\hat{r}_b$ we can use the Welch-Satterthwaite formula [103, 91]. If we define $\hat{\sigma}_b$ as:

$$\hat{\sigma}_b \stackrel{\text{def}}{=} \sqrt{\frac{\hat{\sigma}_{\text{Agda}.\varepsilon,b}^2}{n} + \frac{\hat{\sigma}_{\text{Agda},b}^2}{n}}$$

then $\frac{\hat{r}_b - r_b}{\hat{\sigma}_b}$ has a t-distribution with ν degrees of freedom, where ν is approximated by:

$$\nu \stackrel{\text{def}}{=} (n-1)\frac{\left(\hat{\sigma}_{\text{Agda}.\varepsilon,b}^2 + \hat{\sigma}_{\text{Agda},b}^2\right)^2}{\hat{\sigma}_{\text{Agda}.\varepsilon,b}^4 + \hat{\sigma}_{\text{Agda},b}^4}$$

Thus, $\hat{r}_b \pm \hat{\sigma}_b \cdot \mathrm{t}_{\nu,1-\alpha/2}$ ($\alpha \stackrel{\text{def}}{=} 0.05$) is a 95% confidence interval for r_b. The formula for ν is simpler than the one found in the literature due to the fact that in this particular case the sizes of the two populations are identical ($n = 40$).

For readability, we want to report the percentage increase (or decrease) in resource usage. We define $\Delta_b^{\%} = 100 \cdot (\exp(r_b) - 1)$. Then, $100 \cdot \left(\exp\left(\hat{r}_b \pm \hat{\sigma}_b \cdot \mathrm{t}_{\nu,1-\alpha/2}\right) - 1\right)$ is a 95% confidence interval for $\Delta_b^{\%}$. As for the other confidence intervals, this means that if we were to repeat the experiment many times with the same number of runs and calculate the confidence interval as we did now, the true $\Delta_b^{\%}$ (as derived from the true r_b) would be contained in the corresponding confidence interval 95% of the time.

A.2 Adequacy of the model

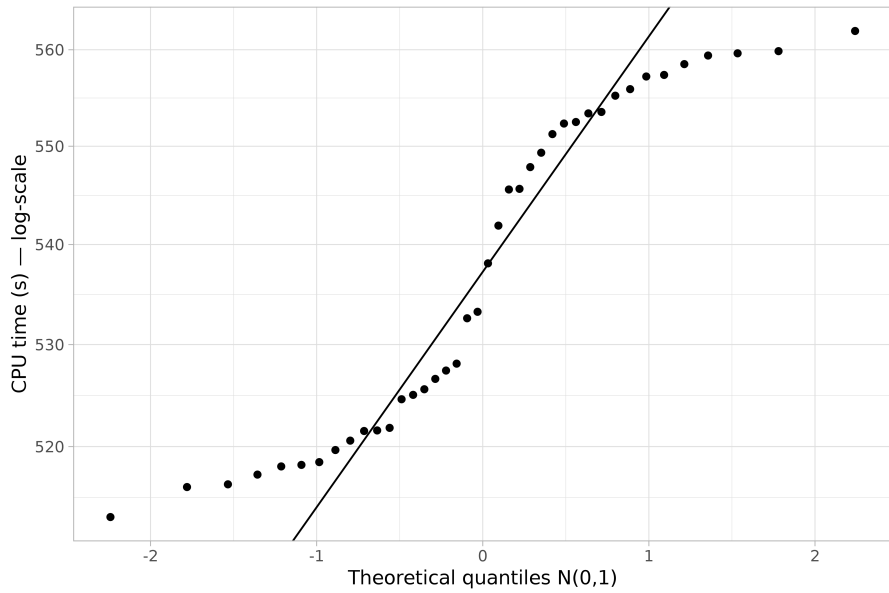
In §A.1 we assume that the logarithms of the resource usage measurements are normally distributed. To decide if the model is adequate we want to assess how this choice may affect our conclusions. For this analysis we will focus on the CPU and memory usage of type-checking the standard library with the baseline Agda implementation.

In Figure A.1 we compare the quantiles of the normal distribution against the empirical quantiles of the logarithms of CPU time measurements. Visual inspection shows that the distribution of the logarithms of times is symmetric, as would be expected from a normal distribution. Note that the steps on the y-axis are almost linearly spaced, which means that the logarithm transformation had a small effect on the final shape of the distribution. As for the overall shape, the time measurements are less disperse than one would expect from a normal distribution, as evidenced by the "S"-shape in the overall trend in the points. Lower dispersion means that the confidence intervals that we obtained in §A.1 may be too broad; that is, the estimates that we have obtained for the average resource usage may be more accurate than the confidence intervals would suggest. Note that our aim is not to maximize the statistical power of our study, but instead to obtain error bounds around our estimates so that the results can be interpreted with the appropriate caution. Overestimating the size of the confidence interval for display purposes is consistent with this goal.

Furthermore, we are mainly concerned with the average resource usage of our implementation, rather than modelling the distribution of an individual run. In the licentiate thesis [54, §5.4], in order to obtain error bounds for the average resource usage, we used the bootstrap. The bootstrap is a non-parametric method which does not assume a specific distribution for the data, while here we have assumed a normal distribution. In Figure A.2 we compare the interval derived from the theoretical formulation in §A.1 with the corresponding percentiles of the distribution obtained with the bootstrap technique. As expected, the empirical distribution obtained from the latter technique is less disperse than the theoretical one. However, the central interval containing 95% of the points in the empirical distribution is nearly identical to the 95% confidence interval in the theoretical formulation.

In the case of measurements of resident memory, the distribution is much more irregular, including several outliers (Figure A.3). Here we need to more

Figure A.1: Quantile-quantile plot of CPU time measurements when type-checking the Agda standard library using the Agda baseline, with log-scale on the y axis. A straight line is drawn through the points corresponding to the first and third quartiles of the empirical distribution. If the logarithms of time measurements followed normal distribution they would lie on the given straight line. An "S" shape is evidence that the points are closer to the mean than they would if they were normally distributed.



carefully assess whether the conclusions drawn from a log-normal model are valid. As in the case of the CPU time, we compare the distribution for the geometric mean of the memory usage obtained by a bootstrap method (Figure A.4) with the theoretical distribution based on the model in §A.1. Again, the distributions differ at the tails, but are nearly identical in the interval between the 2.5% and 97.5% percentiles.

To summarize, for both CPU time and resident memory measurements, the distributions for the individual measurements are far from normal. This means that the parameters that we infer do not necessarily say much about their distribution. However, we believe a priori that their distributions are thin-tailed, and we have observed that when a sufficient number of measurements is averaged, the 95% confidence intervals under the assumptions from the model in §A.1 are consistent with the bootstrapped empirical distribution that we sampled in this section. We conclude that using the bootstrap or any other technique to better approximate the distribution of individual measurements could unnecessarily complicate the analysis without meaningfully changing the conclusions. We therefore choose to use the model from §A.1 for the analysis in §6.4.

Figure A.2: Empirical distribution (10^6 bootstrap samples) of the geometric mean of the CPU time measurements in Figure A.1 compared with the estimated theoretical distribution. The line $y = x$ is drawn. The reported theoretical 95% CI (highlighted in orange) matches well with a corresponding range of points from the empirical, bootstrapped distribution (highlighted in red).

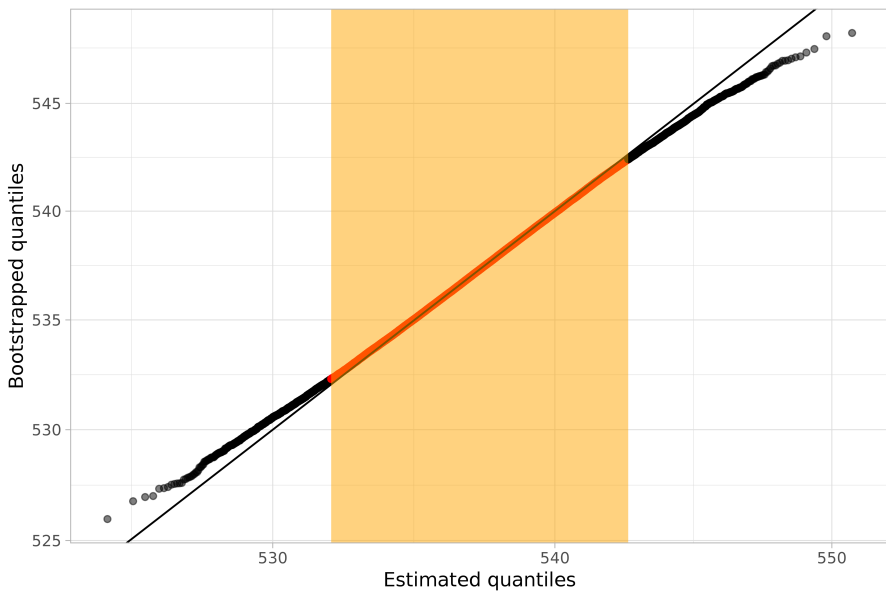


Figure A.3: Quantile-quantile plot of resident memory measurements when type-checking the Agda standard library using the Agda baseline, with log-scale on the y axis. A straight line is drawn through the points corresponding to the first and third quartiles of the empirical distribution. A number of measurements (marked in red) are outliers. These are defined as falling outside of the interval $[Q_1 - 1.5 \cdot IQR, Q_3 + 1.5 \cdot IQR]$*, with* Q_1 *and* Q_3 *being the first and third quartiles, and* $IQR = Q_3 - Q_1$*. Note that all points. including the outliers, are relatively close to the mean.*

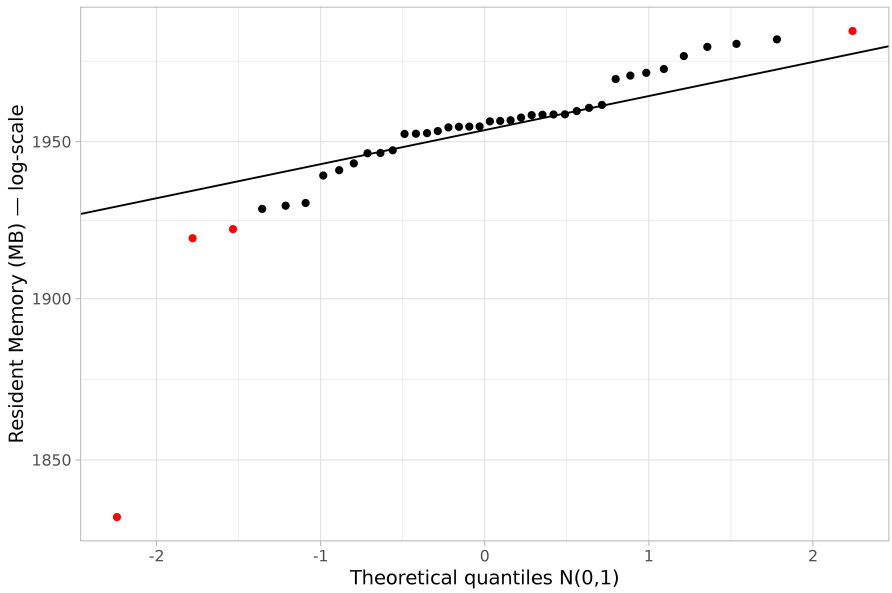
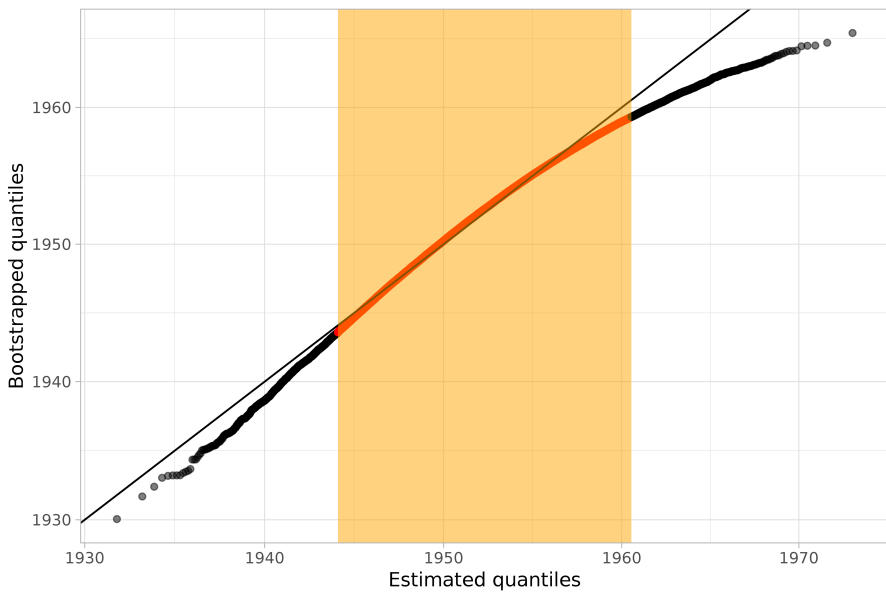


Figure A.4: Empirical distribution (10^6 bootstrap samples) of the geometric mean of the resident memory measurements in Figure A.3 compared with the estimated theoretical distribution. The line $y = x$ is drawn. The reported theoretical 95% CI (highlighted in orange) matches well with a corresponding range of points from the bootstrapped distribution (highlighted in red).



Appendix B

Supplemental evaluation of other proof assistants

In this appendix we include further examples of how our test cases type-check in other proof assistants.

B.1 Idris 2

These examples show the limitations of the blocked constant approach to the binder problem.

Listing B.1: Rendering of Example 6.4 in Idris. We use an inductive family to model the atomic constant $\mathbb{F}$ as this is the most straightforward way we found to do so.

```
data F    : Bool -> Type where
  MkFT : F True

data G : Bool -> Bool -> Type where

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c : (alpha : F True -> Bool -> Bool) ->
           (beta  : F True -> Bool -> Bool) ->
           Pair
                ((x : Bool) -> (y : F (alpha MkFT True)) ->
                               (z : F True) -> G (beta z x) x)

                ((x : Bool) -> (y : F True)            ->
                               (z : F (beta  MkFT True)) -> G x (alpha y x))
     ) -> Type
c1 c = AreEqual _ (c _ _)
```

Output:

```
Idris 2, version 0.3.0
1/1: Building BlockedConstant (BlockedConstant.idr)
```

```
Error: While processing right hand side of c1. Can't solve constraint between: ?_
    MkFT True and True.

BlockedConstant.idr:17:20--17:25
   |
17 | c1 c = AreEqual _ (c _ _)
   |

[Exit code: 1]
```

Listing B.2: Example in Listing B.1 with the solution to the metavariable given explicitly.

```
data F   : Bool -> Type where
  MkFT : F True

data G : Bool -> Bool -> Type where

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c : (alpha : F True -> Bool -> Bool) ->
          (beta  : F True -> Bool -> Bool) ->
          Pair
                ((x : Bool) -> (y : F (alpha MkFT True)) ->
                               (z : F True) -> G (beta z x) x)

                ((x : Bool) -> (y : F True)            ->
                               (z : F (beta  MkFT True)) -> G x (alpha y x))
     ) -> Type
c1 c = AreEqual _ (c (\ a , b => b) (\ a , b => b))
```

Output:

```
Idris 2, version 0.3.0
1/1: Building BlockedConstant_solved (BlockedConstant_solved.idr)
[Exit code: 0]
```

Listing B.3: Example in Listing B.1 with both sides of the constraint swapped.

```
data F   : Bool -> Type where
  MkFT : F True

data G : Bool -> Bool -> Type where

data AreEqual : (A : Type) -> Pair A A -> Type

c1 : (c : (alpha : F True -> Bool -> Bool) ->
          (beta  : F True -> Bool -> Bool) ->
          Pair
                ((x : Bool) -> (y : F True)            ->
                               (z : F (beta  MkFT True)) -> G x (alpha y x))

                ((x : Bool) -> (y : F (alpha MkFT True)) ->
                               (z : F True) -> G (beta z x) x)
```

```
16      ) −> Type
17 c1 c = AreEqual _ (c _ _)
```

Output:

```
1 Idris 2, version 0.3.0
2 1/1: Building BlockedConstantS (BlockedConstantS.idr)
3 Error: While processing right hand side of c1. Can't solve constraint between: ?_
      MkFT True and True.
4
5 BlockedConstantS.idr:17:20--17:25
6      |
7  17 | c1 c = AreEqual _ (c _ _)
8      |
9
10 [Exit code: 1]
```

Listing B.4: Example in Listing B.3 with the solution to the metavariable given explicitly.

```
1 data F    : Bool −> Type where
2   MkFT : F True
3
4 data G : Bool −> Bool −> Type where
5
6 data AreEqual : (A : Type) −> Pair A A −> Type
7
8 c1 : (c : (alpha : F True −> Bool −> Bool) −>
9           (beta  : F True −> Bool −> Bool) −>
10          Pair
11                ((x : Bool) −> (y : F True)              −>
12                                (z : F (beta  MkFT True)) −> G x (alpha y x))
13
14                ((x : Bool) −> (y : F (alpha MkFT True)) −>
15                                (z : F True) −> G (beta z x) x)
16      ) −> Type
17 c1 c = AreEqual _ (c (\ a , b => b) (\ a , b => b))
```

Output:

```
1 Idris 2, version 0.3.0
2 1/1: Building BlockedConstantS_solved (BlockedConstantS_solved.idr)
3 [Exit code: 0]
```

B.2 Matita

We translate our test cases for Coq into Matita, showing that the behaviour of the two proof assistants on our test cases coincides.

B.2.1 Binder problem

Listing B.5: Rendering of the binder problem in Matita, together with the output from the type checker. We use the symbol _ to induce Matita to create a metavariable, which we name alpha.

```
include "basics/bool.ma".
include "basics/types.ma".

(* 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)
                                    ].

(* Atoms *)
axiom F : bool -> Type[0].
axiom G : U -> Type[0].

(* Metavariables and constraints *)
definition c1 :
      let alpha : bool → U ≝ λx.? in
      (Π (x : bool). F (El (alpha x)) → G (alpha x)) →
      (Π (x : bool). F true           → G mkSet) ≝
      λ y. y.
```

Output:

```
0.99.3
Info: New object: cic:/matita/minilang/Pi/U.con
Info: New object: cic:/matita/minilang/Pi/mkSet.con
Info: New object: cic:/matita/minilang/Pi/mkEl.con
Info: New object: cic:/matita/minilang/Pi/El.con
Info: New object: cic:/matita/minilang/Pi/F.con
Info: New object: cic:/matita/minilang/Pi/G.con
Error: ********** DISAMBIGUATION ERRORS: **********
***** Errors obtained during phases 4: *****
*Error at 643–644: The term
y
has type
(∀x:bool.F (El ?55[...]) →G ?55[...])
but is here used with type
(∀x:bool.F true→G mkSet)

***** Errors obtained during phases 3: *****
*Error at 643–644: The term
y
has type
(∀x:bool.F (El ?54[...]) →G ?54[...])
but is here used with type
(∀x:bool.F true→G mkSet)

***** Errors obtained during phases 2: *****
```

```
*Error at 643–644: The term
y
has type
(∀x:bool.F (El ?53[...]) →G ?53[...])
but is here used with type
(∀x:bool.F true→G mkSet)

***** Errors obtained during phases 1: *****
*Error at 643–644: The term
y
has type
(∀x:bool.F (El ?52[...]) →G ?52[...])
but is here used with type
(∀x:bool.F true→G mkSet)


[ ExitFailure : 1]
```

Listing B.6: Example Listing B.5 modified so that the codomains are equal.

```
include "basics/bool.ma".
include "basics/types.ma".

(* 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)
                              ].

(* Atoms *)
axiom F : bool -> Type[0].
axiom G : U -> Type[0].

(* Metavariables and constraints *)
definition c1 :
     let alpha : bool → U ≝ λx.? in
     (Π (x : bool). F (El mkSet) → G (alpha x)) →
     (Π (x : bool). F true       → G mkSet) ≝
     λ y. y.
```

Output:

```
0.99.3
Info: New object: cic:/matita/minilang/PiEasy/U.con
Info: New object: cic:/matita/minilang/PiEasy/mkSet.con
Info: New object: cic:/matita/minilang/PiEasy/mkEl.con
Info: New object: cic:/matita/minilang/PiEasy/El.con
Info: New object: cic:/matita/minilang/PiEasy/F.con
Info: New object: cic:/matita/minilang/PiEasy/G.con
Info: New object: cic:/matita/minilang/PiEasy/c1.con
Info: Compilation successful
```

Listing B.7: Example Listing B.5 modified so that the metavariable alpha *is solved from the start.*

```
include "basics/bool.ma".
include "basics/types.ma".

(* 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)
                              ].

(* Atoms *)
axiom F : bool -> Type[0].
axiom G : U -> Type[0].

(* Metavariables and constraints *)
definition c1 :
     let alpha : bool → U ≝ λx.mkSet in
     (Π (x : bool). F (El (alpha x)) → G (alpha x)) →
     (Π (x : bool). F true           → G mkSet) ≝
     λ y. y.
```

Output:

```
0.99.3
Info: New object: cic:/matita/minilang/PiSolved/U.con
Info: New object: cic:/matita/minilang/PiSolved/mkSet.con
Info: New object: cic:/matita/minilang/PiSolved/mkEl.con
Info: New object: cic:/matita/minilang/PiSolved/El.con
Info: New object: cic:/matita/minilang/PiSolved/F.con
Info: New object: cic:/matita/minilang/PiSolved/G.con
Info: New object: cic:/matita/minilang/PiSolved/c1.con
Info: Compilation successful
```

B.2.2 Spine problem

Listing B.8: Rendering of the binder problem in Matita, together with the output from the type checker. We use the symbol _ *to induce Matita to create a metavariable, which we name* alpha*.*

```
include "basics/bool.ma".
include "basics/types.ma".

(* 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)
                                ].

(* Atoms *)
axiom Pred : bool -> U -> Type[0].

(* Metavariables and constraints *)
definition c1 :
     let alpha : bool → U ≝ λx.? in
     (Π (x : bool). Pred (El (alpha x)) (alpha x)) →
     (Π (x : bool). Pred  true          mkSet) ≝
     λ y. y.
```

Output:

```
0.99.3
Info:  New object: cic:/matita/minilang/SpineMini/U.con
Info:  New object: cic:/matita/minilang/SpineMini/mkSet.con
Info:  New object: cic:/matita/minilang/SpineMini/mkEl.con
Info:  New object: cic:/matita/minilang/SpineMini/El.con
Info:  New object: cic:/matita/minilang/SpineMini/Pred.con
Error: ********** DISAMBIGUATION ERRORS: **********
***** Errors obtained during phases 4: *****
*Error at 626—627: The term
y
has type
(∀x:bool.Pred (El ?55[...])  ?55[...])
but is here used with type
(∀x:bool.Pred true mkSet)

***** Errors obtained during phases 3: *****
*Error at 626—627: The term
y
has type
(∀x:bool.Pred (El ?54[...])  ?54[...])
but is here used with type
(∀x:bool.Pred true mkSet)

***** Errors obtained during phases 2: *****
*Error at 626—627: The term
y
has type
(∀x:bool.Pred (El ?53[...])  ?53[...])
but is here used with type
(∀x:bool.Pred true mkSet)

***** Errors obtained during phases 1: *****
*Error at 626—627: The term
y
has type
(∀x:bool.Pred (El ?52[...])  ?52[...])
but is here used with type
(∀x:bool.Pred true mkSet)
```

```
39
40
41 [ ExitFailure : 1]
```

Listing B.9: Example Listing B.8 modified so that the first arguments can be made equal.

```
1  include "basics/bool.ma".
2  include "basics/types.ma".
3
4  (* Shorthands *)
5  definition U                : Type[0] ≝ bool × bool.
6  definition mkSet            : U       ≝ ⟨true , true⟩.
7  definition mkEl (b : bool) : U       ≝ ⟨false , b⟩.
8
9  definition El (u : U) : bool ≝ match (\fst u) with
10          [ true  => true
11                | false => (\snd u)
12                                  ].
13
14 (* Atoms *)
15 axiom Pred : U -> bool -> Type[0].
16
17 (* Metavariables and constraints *)
18 definition c1 :
19       let alpha : bool → U ≝ λx.? in
20       (Π (x : bool). Pred (alpha x) (El (alpha x)) ) →
21       (Π (x : bool). Pred  mkSet     true) ≝
22       λ y. y.
```

Output:

```
1 0.99.3
2 Info:  New object: cic:/matita/minilang/SpineMiniEasy/U.con
3 Info:  New object: cic:/matita/minilang/SpineMiniEasy/mkSet.con
4 Info:  New object: cic:/matita/minilang/SpineMiniEasy/mkEl.con
5 Info:  New object: cic:/matita/minilang/SpineMiniEasy/El.con
6 Info:  New object: cic:/matita/minilang/SpineMiniEasy/Pred.con
7 Info:  New object: cic:/matita/minilang/SpineMiniEasy/c1.con
8 Info:  Compilation successful
```

Listing B.10: Example Listing B.8 modified so that the metavariable alpha is solved from the start.

```
1  include "basics/bool.ma".
2  include "basics/types.ma".
3
4  (* Shorthands *)
5  definition U                : Type[0] ≝ bool × bool.
6  definition mkSet            : U       ≝ ⟨true , true⟩.
7  definition mkEl (b : bool) : U       ≝ ⟨false , b⟩.
8
9  definition El (u : U) : bool ≝ match (\fst u) with
10          [ true  => true
```

```
                | false  => (\snd u)
                                ].

(* Atoms *)
axiom Pred : bool -> U -> Type[0].

(* Metavariables and constraints *)
definition  c1 :
      let alpha : bool → U ≝ λx.mkSet in
      (Π (x : bool). Pred (El (alpha x)) (alpha x)) →
      (Π (x : bool). Pred  true              mkSet) ≝
      λ y. y.
```

Output:

```
0.99.3
Info:  New object: cic:/matita/minilang/SpineMiniSolved/U.con
Info:  New object: cic:/matita/minilang/SpineMiniSolved/mkSet.con
Info:  New object: cic:/matita/minilang/SpineMiniSolved/mkEl.con
Info:  New object: cic:/matita/minilang/SpineMiniSolved/El.con
Info:  New object: cic:/matita/minilang/SpineMiniSolved/Pred.con
Info:  New object: cic:/matita/minilang/SpineMiniSolved/c1.con
Info:  Compilation successful
```

B.3 Lean

We translate our test cases for Coq into Lean, showing that the behaviour of the two proof assistants on our test cases coincides.

B.3.1 Binder problem

Listing B.11: Rendering of the binder problem in Lean, together with the output from the type checker. We use the symbol `_` *to induce Lean to create a metavariable, which we name* `alpha`*.*

```
-- Shorthands
def U    : Type := prod 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

-- Atoms
constant F : bool -> Type
constant G : U -> Type

-- Metavariables and constraints
def c1
    (c :  forall  (alpha : bool -> U),
         ( forall  (x : bool), F (El (alpha x))  -> G (alpha x))) :
```

```
19          ( forall  (x : bool), F  tt              -> G set' )
20   :=  c _
```

Output:

```
1  Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
2  ./lean/Pi.lean:20:6: error: type mismatch, term
3    c ?m_1
4  has type
5    Π (x : bool), F (El (?m_1 x)) → G (?m_1 x)
6  but is expected to have type
7    bool → F tt → G set'
8  [Exit code: 1]
```

Listing B.12: Example Listing B.11 modified so that the codomains are equal.

```
1  -- Shorthands
2  def U    : Type := prod bool bool
3  def set' : U  := (tt, tt)
4  def el'  (b : bool) : U := (ff, b)
5  def El (u : U) : bool :=
6    match u.1 with
7      | tt  := tt
8      | ff  := u.2
9    end
10
11 -- Atoms
12 constant F : bool -> Type
13 constant G : U -> Type
14
15 -- Metavariables and constraints
16 def c1
17     (c :  forall  (alpha : bool -> U),
18          ( forall  (x : bool), F (El set')  -> G (alpha x))) :
19          ( forall  (x : bool), F  tt        -> G set' )
20   :=  c _
```

Output:

```
1  Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
2  [Exit code: 0]
```

Listing B.13: Example Listing B.11 modified so that the metavariable alpha *is solved from the start.*

```
1  -- Shorthands
2  def U    : Type := prod bool bool
3  def set' : U  := (tt, tt)
4  def el'  (b : bool) : U := (ff, b)
5  def El (u : U) : bool :=
6    match u.1 with
7      | tt  := tt
8      | ff  := u.2
9    end
10
```

```
-- Atoms
constant F : bool -> Type
constant G : U -> Type

-- Metavariables and constraints
def c1
    (c : forall (alpha : bool -> U),
        ( forall (x : bool), F (El (alpha x))  -> G (alpha x))) :
        ( forall (x : bool), F  tt             -> G set' )
  := c (fun z, set')
```

Output:

```
Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
[Exit code: 0]
```

B.3.2 Spine problem

Listing B.14: Rendering of the binder problem in Lean, together with the output from the type checker. We use the symbol _ to induce Lean to create a metavariable, which we name alpha.

```
-- Atoms
constant F : bool -> Type

-- Shorthands
def U    : Type := prod 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

constant Pred : bool -> U -> Type

-- Metavariables and constraints
def c1
    (c : forall (alpha : bool -> U),
      ( forall (x : bool), Pred (El (alpha x))  (alpha x) )) :
      ( forall (x : bool), Pred  tt              set'     ) := c _
```

Output:

```
Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
./lean/SpineMini.lean:21:63: error: type mismatch, term
  c ?m_1
has type
  Π (x : bool), Pred (El (?m_1 x)) (?m_1 x)
but is expected to have type
  bool → Pred tt set'
[Exit code: 1]
```

Listing B.15: Example Listing B.14 modified so that the first arguments can be made equal.

```
-- Atoms
constant F : bool -> Type

-- Shorthands
def U    : Type := prod 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

constant Pred : U -> bool -> Type

-- Metavariables and constraints
def c1
    (c : forall (alpha : bool -> U),
      ( forall (x : bool), Pred (alpha x) (El (alpha true)))) :
      ( forall (x : bool), Pred  set'       tt                ) := c _
```

Output:

```
Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
[Exit code: 0]
```

Listing B.16: Example Listing B.14 modified so that the metavariable alpha is solved from the start.

```
-- Atoms
constant F : bool -> Type

-- Shorthands
def U    : Type := prod 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

constant Pred : bool -> U -> Type

-- Metavariables and constraints
def c1
    (c : forall (alpha : bool -> U),
      ( forall (x : bool), Pred (El (alpha x))   (alpha x) )) :
      ( forall (x : bool), Pred  tt                 set'      ) := c (fun z, set')
```

Output:

```
Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
[Exit code: 0]
```

Index

List of Definitions, Notations and Problems

List of Postulates

List of Theorems, Propositions, Lemmas and Remarks

List of Examples

List of Figures

List of Listings

List of Tables

Bibliography

[1] Andreas Abel. Type checker normalizes too much. Agda Issue #4125, October 2019. URL https://github.com/agda/agda/issues/4125 .

[2] 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 .

[3] 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 .

[4] 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 .

[5] 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 .

[6] Andreas Abel, Guillaume Allais, Jesper Cockx, Nils Anders Danielsson, Philipp Hausmann, Fredrik Nordvall Forsberg, Ulf Norell, Víctor López Juan, Andrés Sicard-Ramírez, , Andrea Vezzosi, et al. Agda 2, 2021. URL https://github.com/agda/agda .

[7] Andreas Abel, Nils Anders Danielsson, Jesper Cockx, Ulf Norell, et al. Agda, 2021. URL https://github.com/agda/agda . Commit 47179128.

[8] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In *International Workshop on Types for Proofs and Programs*, pages 1–16. Springer, 2004. doi:10.1007/11617990_1 .

[9] 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/ .

[10] Robin Adams. Lambda-free logical frameworks. *CoRR*, abs/0804.1879, 2008. URL http://arxiv.org/abs/0804.1879 .

[11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *International Conference on Automated Deduction (CADE 2011)*, 2011. doi:10.1007/978-3-642-22438-6_7 .

[12] Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0 .

[13] Guillaume Brunerie, Nils Anders Danielsson, Ulf Norell, and Jesper Cockx. Equality checking uses too much memory in 2.6.0 (compared to 2.5.4). Agda Issue #4044, September 2019. URL `https://github.com/agda/agda/issues/4044` .

[14] Jacques Carette, Jason Hu, Reed Mullanix, Sandro Stucki, Guillaume Allais, et al. The agda-categories library, 2021. URL `https://github.com/martinescardo/TypeTopology` .

[15] Jesper Cockx and "palkarz". Type checker explosion. Agda Issue #3554, February 2019. URL `https://github.com/agda/agda/issues/3554` .

[16] The Idris Community. Idris 2 documentation – Implementation overview, 2020. URL `https://idris2.readthedocs.io/en/latest/implementation/overview.html` .

[17] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3): 285–326, 1984. doi:10.1007/BF00244273 .

[18] Catarina Coquand. The homepage of the Agda type checker, 1998. URL `https://web.archive.org/web/20070909205649/http://www.cs.chalmers.se/~catarina/agda/` .

[19] Catarina Coquand and Thierry Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages*, 1999. URL `https://www.eecs.uottawa.ca/~afelty/LFM99/CoquandCoquand.pdf` .

[20] 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` .

[21] William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957. doi:10.2307/2963593 .

[22] Luis Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1984. URL `http://hdl.handle.net/1842/13555` .

[23] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery. ISBN 0897910656. doi:10.1145/582153.582176 .

[24] Nils Anders Danielsson. The following code is accepted by Agda 2.6.2-f9a1816, but rejected by 57e31fe […]. Agda PR #5234, March 2021. URL `https://github.com/agda/agda/pull/5234#issuecomment-797034181` .

[25] 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` .

[26] Nils Anders Danielsson, Matthew Daggitt, Guillaume Allais, et al. The Agda standard library, 2021. URL `https://github.com/agda/agda-stdlib` . Commit `d00ba755c`, file "EverythingSafe.agda".

[27] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972. ISSN 1385-7258. doi:10.1016/1385-7258(72)90034-0 .

[28] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[29] 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` .

[30] 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` .

[31] Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206(1-2):1–50, 1998. doi:10.1016/S0304-3975(97)00141-2 .

[32] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5), May 2021. ISSN 0360-0300. doi:10.1145/3450952 .

[33] Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2012. doi:10.1145/2430532.2364522 .

[34] 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 .

[35] 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` .

[36] Andrey Petrovych Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, August 1958. ISSN 0001-0782. doi:10.1145/368892.368907 .

[37] Martin Escardo et al. TypeTopology, 2021. URL https://github.com/martinescardo/TypeTopology .

[38] Free Software Foundation. Bison 3.7.6 user manual, 2021. URL https://www.gnu.org/software/bison/manual/html_node/index.html .

[39] 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 .

[40] Georges Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[41] 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 .

[42] Eichii Goto. Monocopy and associative algorithms in an extended LISP. Technical report, Information Science Laboratory, Faculty of Science, University of Tokyo, 1974. URL https://www.cs.utexas.edu/users/hunt/research/hash-cons/hash-cons-papers/monocopy-goto.pdf .

[43] 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 .

[44] 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/ .

[45] Adam Gundry and Conor McBride. A tutorial implementation of dynamic pattern unification. Unpublished, 2012. URL http://adam.gundry.co.uk/pub/pattern-unify/ .

[46] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. ISSN 00029947. doi:10.2307/1995158 .

[47] 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 .

[48] Gérard 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 .

[49] 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 .

[50] 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 .

[51] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE. URL `https://hal.inria.fr/hal-01238879` .

[52] Haskell Libraries. Data.Intset. containers: Assorted concrete container types, 2020. URL `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-IntSet.html` .

[53] Víctor López Juan and Nils Anders Danielsson. Practical dependent type checking using twin types. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2020, page 11–23, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380515. doi:10.1145/3406089.3409030 .

[54] Víctor López Juan. *Practical Unification for Dependent Type Checking*. Licentiate Thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2020. URL `https://research.chalmers.se/publication/519011` .

[55] Víctor López Juan. Add arguments that cannot be solved with heterogeneous unification, 2021. URL `https://github.com/agda/agda-stdlib/commit/d00ba755c319d2237815a16c23e69eac44d70d77` .

[56] Víctor López Juan, Francesco Mazzoli, Nils Anders Danielsson, Ulf Norell, Andrea Vezzosi, Andreas Abel, et al. Tog$^+$, 2020. URL `https://lopezjuan.com/project/togt/` .

[57] Kaivalya M. Dixit. *Overview of the SPEC Benchmarks — Chapter 9: The Benchmark Handbook for Database and Transaction Processing Systems*. 2nd edition, 1993. ISBN 1-55860-292-5. URL `https://jimgray.azurewebsites.net/BenchmarkHandbook/chapter9.pdf` .

[58] 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. URL `https://search.ebscohost.com/login.aspx?direct=true&db=cat07470a&AN=clc.fefa91c4.66a5.4b2c.9f34.115f4716efed&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds` .

[59] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975. doi:10.1016/S0049-237X(08)71945-1 .

[60] Francesco Mazzoli and Andreas Abel. Type checking through unification, 2016. arXiv:1609.09709v1 .

[61] 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` .

[62] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003. doi:10.1017/S0956796803004957 .

[63] Conor McBride. Epigram, 2004. URL `https://web.archive.org/web/20120717070845/http://www.e-pig.org/darcs/Pig09/web/` .

[64] 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 .

[65] Conor McBride. Basics of bidirectionalism. Blog post, 2018. URL `https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/` .

[66] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi:10.1017/S0956796803004829 .

[67] 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` .

[68] 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 .

[69] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. doi:10.1016/0022-0000(78)90014-4 .

[70] 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` .

[71] 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 .

[72] nLab authors. FinSet, 2021. URL `https://ncatlab.org/nlab/show/FinSet` .

[73] 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` .

[74] Ulf Norell. More simplification, splitting class into two separate types.[...]. Agda Issue #2709, August 2017. URL `https://github.com/agda/agda/issues/2709#issuecomment-324371998` .

[75] Ulf Norell. Internal error during instance search. Agda Issue #3870, June 2019. URL `https://github.com/agda/agda/issues/3870` .

[76] Ulf Norell. Workaround for too optimistic conversion checking, 2019. URL `https://github.com/agda/agda/commit/237aa1f9ab2b5bead729d7d6c3695569186b4d8b` .

[77] Ulf Norell. More fine-grained blocking. Pull Request #4782, June 2020. URL `https://github.com/agda/agda/pull/4782` .

[78] Ulf Norell. More constraint overhaul. Pull Request #4840, August 2020. URL `https://github.com/agda/agda/pull/4840` .

[79] 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` .

[80] Ulf Norell and Víctor López Juan. Internal error in the presence of unsatisfiable constraints. Agda Issue #3027, April 2018. URL `https://github.com/agda/agda/issues/3027` .

[81] Ulf Norell et al. Programming library for Agda, 2020. URL `https://github.com/UlfNorell/agda-prelude` . Commit `fcca646d6`, file "Everything.agda".

[82] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.5452` .

[83] Kent Petersson. A programming system for type theory. Technical Report 9, Programming Methodology Group, University of Gothenburg and Chalmers University of Technology, 1984. URL `https://search.ebscohost.com/login.aspx?direct=true&db=cat07470a&AN=clc.79a415d4.e762.4772.aee2.af6bd2e2057f&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds` . ISSN 0282-2083.

[84] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 252–265, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919793. doi:10.1145/268946.268967 .

[85] 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 .

[86] Git Software Project. Documentation of git-diff – Version 2.3.1, 2021. URL `https://git-scm.com/docs/git-diff/2.31.1` .

[87] 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/` .

[88] 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 .

[89] Egbert Rijke and Ali Caglayan. Introduction to homotopy type theory, 2021. URL `https://github.com/HoTT-Intro/Agda` . Commit `75d334968`, file "book.agda".

[90] 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 .

[91] F. E. Satterthwaite. An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2(6):110–114, 1946. ISSN 00994987. URL `http://www.jstor.org/stable/3002019` .

[92] Christian Sattler, Jesper Cockx, and Ulf Norell. Universe level constraints not solved. Agda Issue #4345, 2020. URL `https://github.com/agda/agda/issues/4345` .

[93] GHC Team. Datatype promotion. Glasgow Haskell Compiler, 2020. URL `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/data_kinds.html` .

[94] GHC Team. Visible type application. Glasgow Haskell Compiler, 2020. URL `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_applications.html` .

[95] Idris Development Team. Idris: A language for type-driven development, 2021. URL `https://www.idris-lang.org/` .

[96] The Agda Development Team. Generalization over unsolved metavariables, 2019. URL `https://agda.readthedocs.io/en/v2.6.0.1/language/generalization-of-declared-variables.html#generalization-over-unsolved-metavariables` .

[97] The Agda Development Team. Agda user manual – Cubical, 2021. URL `https://agda.readthedocs.io/en/v2.6.1.3/language/cubical.html` .

[98] The Agda Development Team. Agda user manual – Sized types, 2021. URL `https://agda.readthedocs.io/en/v2.6.1.3/language/sized-types.html` .

[99] The Agda Development Team. Agda user manual – Universe levels, 2021. URL `https://agda.readthedocs.io/en/v2.6.1.3/language/universe-levels.html` .

[100] The Coq Development Team. Coq source code – kernel/constr.ml, 2020. URL `https://github.com/coq/coq/blob/V8.13.2/kernel/constr.ml` .

[101] The Coq Development Team. Coq 8.13.1, March 2021. URL `https://coq.inria.fr/news/coq-8-13-1-is-out.html` .

[102] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3 (ICFP), July 2019. doi:10.1145/3341691 .

[103] B. L. Welch. The generalization of Student's problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 01 1947. ISSN 0006-3444. doi:10.1093/biomet/34.1-2.28 .

[104] 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 .

[105] 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 .