Practical Heterogeneous Unification for Dependent Type Checking

Víctor López Juan
Practical Heterogeneous Unification for Dependent Type Checking
VÍCTOR LÓPEZ JUAN

© VÍCTOR LÓPEZ JUAN, 2021.

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
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 Myrén 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.


Y gracias a mis padres, por recibirme siempre en casa como si nunca me hubiera marchado.
2.19 Closing metasubstitution \((\text{CLOSE}(\Sigma))\) .................................. 51
2.20 Equality of metasubstitutions \((\Theta_1 \equiv \Theta_2)\) .......................... 53
2.21 Signature extensions \((\Sigma \subseteq \Sigma')\) ...................................... 54
2.22 Non-reducible terms ................................................................. 56
2.23 Rigidly occurring terms \((t[u])\) .................................................. 58
2.24 Out of scope features ............................................................... 61
2.25 Closing remarks ................................................................. 61

3 Unification for dependent type checking ........................................ 63
3.1 From type checking to unification ............................................. 66
3.2 Approaches to elaboration ..................................................... 67
3.3 Higher-order unification .......................................................... 68
3.4 \((\text{Un})\text{decidability of higher order unification} \) ...................... 69
3.5 Miller pattern unification ....................................................... 70
3.6 Dynamic pattern unification ................................................... 70
3.7 Extension to product types .................................................... 71
3.8 Interleaving type checking with unification .............................. 72
3.9 Strictly ordered, homogeneous constraints .................................. 73
3.10 Twin types ............................................................................ 74

4 Unifying without order ................................................................ 77
4.1 Two-sided internal constraints ................................................ 78
4.2 Heterogeneous equality ........................................................... 80
4.3 From type checking to internal constraints ............................... 82
4.4 Correctness of reduction rules ................................................. 84
4.5 A reduction rule toolkit ........................................................... 86
4.5.1 Syntactic equality check ...................................................... 86
4.5.2 Metavariable instantiation .................................................. 87
4.5.3 Type constructors .............................................................. 93
4.5.4 Constraint symmetry ........................................................ 94
4.5.5 Term conversion ............................................................... 94
4.5.6 Type conversion ............................................................... 95
4.5.7 Type-directed unification .................................................. 95
4.5.8 Strongly neutral terms ....................................................... 96
4.5.9 Metavariable argument killing ......................................... 98
4.5.10 Metavariable argument currying ..................................... 102
4.5.11 Metavariable \(\eta\)-expansion ............................................. 103
4.5.12 Context variable currying .............................................. 105
4.6 Example of constraint solving ............................................... 106
4.7 Limitations of the rule toolkit ................................................ 108
4.8 Beyond correctness ............................................................... 111
4.8.1 Open-world assumption .................................................. 111
4.8.2 Unsolvable problems ....................................................... 113
4.9 Extensibility and limitations .................................................. 114
4.9.1 Singleton types with \(\eta\)-equality ................................... 114
4.10 Concluding remarks ............................................................ 115
## 5 Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Twin types</td>
<td>118</td>
</tr>
<tr>
<td>5.2</td>
<td>Heterogeneous contexts</td>
<td>119</td>
</tr>
<tr>
<td>5.3</td>
<td>Context switching</td>
<td>120</td>
</tr>
<tr>
<td>5.4</td>
<td>Constraint representation</td>
<td>122</td>
</tr>
<tr>
<td>5.5</td>
<td>Constraint blocking</td>
<td>122</td>
</tr>
<tr>
<td>5.6</td>
<td>Constraint prioritization</td>
<td>124</td>
</tr>
<tr>
<td>5.7</td>
<td>Metavariable instantiation</td>
<td>126</td>
</tr>
<tr>
<td>5.8</td>
<td>Fast twin simplification</td>
<td>129</td>
</tr>
<tr>
<td>5.9</td>
<td>Unification of binders</td>
<td>131</td>
</tr>
<tr>
<td>5.10</td>
<td>Unification of spines</td>
<td>131</td>
</tr>
<tr>
<td>5.11</td>
<td>Singleton types with $\eta$-equality</td>
<td>134</td>
</tr>
<tr>
<td>5.12</td>
<td>Other changes</td>
<td>134</td>
</tr>
<tr>
<td>5.13</td>
<td>Beyond Agda</td>
<td>135</td>
</tr>
</tbody>
</table>

## 6 Evaluation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Implementation effort</td>
<td>137</td>
</tr>
<tr>
<td>6.2</td>
<td>Functional testing</td>
<td>138</td>
</tr>
<tr>
<td>6.3</td>
<td>Addressed bugs in Agda</td>
<td>140</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance evaluation</td>
<td>141</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Methodology</td>
<td>141</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Project benchmarks</td>
<td>141</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Context singleness check</td>
<td>142</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Other benchmarks</td>
<td>143</td>
</tr>
<tr>
<td>6.5</td>
<td>Limitations and future work</td>
<td>143</td>
</tr>
<tr>
<td>6.6</td>
<td>Comparison with other proof assistants</td>
<td>149</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Coq, Matita and Lean</td>
<td>149</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Idris 2</td>
<td>157</td>
</tr>
<tr>
<td>6.7</td>
<td>Concluding remarks</td>
<td>163</td>
</tr>
</tbody>
</table>

## 7 Conclusion

### A Statistical modelling of the benchmark data

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Statistical model</td>
<td>167</td>
</tr>
<tr>
<td>A.2</td>
<td>Adequacy of the model</td>
<td>169</td>
</tr>
</tbody>
</table>

### B Supplemental evaluation of other proof assistants

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>Idris 2</td>
<td>175</td>
</tr>
<tr>
<td>B.2</td>
<td>Matita</td>
<td>177</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Binder problem</td>
<td>177</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Spine problem</td>
<td>180</td>
</tr>
<tr>
<td>B.3</td>
<td>Lean</td>
<td>183</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Binder problem</td>
<td>183</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Spine problem</td>
<td>185</td>
</tr>
</tbody>
</table>

## Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>List of definitions, notations and problems</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>List of postulates</td>
<td>192</td>
</tr>
<tr>
<td></td>
<td>List of theorems, propositions, lemmas and remarks</td>
<td>192</td>
</tr>
<tr>
<td></td>
<td>List of examples</td>
<td>195</td>
</tr>
</tbody>
</table>
List of figures ................................. 196
List of code listings ............................. 196
List of tables ................................. 198

Bibliography .................................. 207
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 $\Gamma$) and $\Gamma \vdash u : B$ ($u$ has type $B$ in context $\Gamma$). 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 $\Gamma \vdash A : \text{Set} \approx B : \text{Set}$ 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. Metavars 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 uninstatiated 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.
1.1. PROBLEM STATEMENT

Listing 1.1: Non-unique implicit argument

```plaintext
−− 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.
CHAPTER 1. INTRODUCTION

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 \( \Gamma \) \( (\Gamma \vdash t \approx u) \). 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 \( \lambda \)-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 \( \lambda \Pi \) calculus which only constructs approximately well-typed terms, although with the property that they do at least have \( \eta \)-long, \( \beta \)-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 (\( \Pi \)-types) and dependent pair types (\( \Sigma \)-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 \( \Pi \)-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 \( \lambda \)-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 \textit{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 \( \Sigma \)-types with \( \eta \)-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 \( \Gamma \vdash (x : A) \rightarrow B \approx (x : A') \rightarrow B' \). 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 (\( \Gamma, x : A \vdash B \text{ type} \) and \( \Gamma, x : A' \vdash B' \text{ type} \)). It is not self-evident what the type of \( x \) should be in \( \Gamma, x : ? \vdash B \approx B' \text{ type} \), 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 $A$ ($p \leadsto A$) 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. $\Gamma, x : A \vdash B \approx B'[p/x] \text{ type}$, where $p \leadsto x$ 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 $\eta$-equality for records (including the unit type), omitting the type of the bound variable in $\lambda$-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) $c : (x : A) \rightarrow (y : B) \rightarrow C$. 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 $p \leadsto t$ 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
CHAPTER 1. INTRODUCTION

$t'$ 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 $\beta$-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 $\beta$-reductions when a solution is substituted for a metavariable. In Agda [7] this is mitigated by removing the leading $\lambda$-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 $\beta$-normalized. This is not a concern for us, as we only consider terms in $\beta$-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
CHAPTER 1. INTRODUCTION

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 \( \eta \)-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.
\[ x, y, z ::= \text{variables} \]
\[ X, Y, Z | 0, 1, 2, \cdots \text{de Bruijn indices} \]
\[ \alpha, \beta, \gamma ::= \text{metavariables} \]
\[ a, b, c, A, B, C \]
\[ t, u, v, r ::= \text{terms and types} \]
\[ T, U, A, B | \text{Bool boolean type} \]
\[ | \Pi AB \text{ function type} \]
\[ | \Sigma AB \text{ record type} \]
\[ | \text{Set universe} \]
\[ | c \text{ data constructor} \]
\[ | \lambda t \lambda\text{-abstraction} \]
\[ | \langle t, u \rangle \text{ pair constructor} \]
\[ | f \text{ neutral terms} \]
\[ f, g ::= \text{neutral terms} \]
\[ | h \text{ elimination head} \]
\[ | f e \text{ eliminator} \]
\[ h ::= \text{elimination heads} \]
\[ | x, X, \cdots \text{variable head} \]
\[ | \alpha, \beta, \cdots \text{metavariable head} \]
\[ | a, b, \cdots \text{atom head} \]
\[ | \text{if boolean recursor head} \]
\[ e ::= \text{eliminators} \]
\[ | t \text{ term application} \]
\[ | .\pi_1 \ | .\pi_2 \text{ projections} \]
\[ c ::= \text{data constructors} \]
\[ | \text{true} \ | \text{false booleans} \]

*Figure 2.1: Syntax for terms. Metavariables (\(\alpha, \beta, \cdots\)) and atoms (\(a, b, \cdots\)) are drawn from disjoint and countably infinite sets of names.*
2.2. NOTATIONAL PRELIMINARIES

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

- \(\square\) 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.

- \(\square^n\) 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.

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

- \(\square \ldots \square\) denotes a sequence of \(n\) identical elements of the form \(\square\).

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

- \(\square \odot \ldots \square\) denotes a sequence of \(n\) identical elements of the form \(\square\) such that the operator \(\odot\) 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 \(\alpha \vec{x}^n = \alpha \vec{x}\) 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 \(i\)th element of a vector \(\vec{x}\).

- \(x_{i,j}\) denotes the \(j\)th element of a vector \(\vec{t}_{i,j}\).

**Notation (Vector slices: \(t_{i,j}\)).** 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 \(1 \leq i \leq j \leq n\), \(\vec{t}_{i,j}\) is a vector of length \(j - i + 1\) whose \(k\)th element is the \((i + k - 1)\)th element of \(\vec{x}\). Whenever \(i > j\), the expression \(\vec{t}_{i,j}\) 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 \(\Sigma = \alpha : \text{Bool}, \mathbb{A} : \text{Set}, a : \mathbb{A}\), we say that \(\alpha : \text{Bool} \in \Sigma\).
Chapter 2. A Dependently-typed Language

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

Notation (Partial functions: $F \downarrow y$, $F \lnot\downarrow F$). 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 $F \lnot\downarrow$ if and only if there exists a $y$ such that $F \downarrow y$. We denote such a $y$ by $F$ itself.

2.3 Signatures ($\Sigma$ sig)

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

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

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 $a$ is fresh for $\Sigma$ (or, that $\alpha$ is fresh for $\Sigma$) if there is no $B$ such that $a : B \in \Sigma$ (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 $\Sigma$ contains an element of the form $\alpha := t : A$, we say that the metavariable $\alpha$ is instantiated in the signature $\Sigma$. The term $t$ is the body of $\alpha$ in this signature.

**Definition 2.3** (Uninstantiated metavariable). Conversely, given $\alpha : A \in \Sigma$, we say that $\alpha$ is uninstantiated in $\Sigma$ 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 $\Lambda \alpha$ expands to the term $\Lambda (\lambda x.\lambda y.y)$. In §2.14 we give a full account of computation in well-typed terms, which includes metavariable expansion.

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

\[ \Sigma \text{ sig} \]
\hline
\text{∅} \text{ sig} \rightarrow \text{EMPTY} \\
\hline
\Sigma \text{ sig} \quad \alpha \text{ is fresh for } \Sigma \\
\Sigma ; \cdot \vdash A \text{ type} \quad \text{ATOM-DECL} \\
\hline
\Sigma \text{ sig} \\
\Sigma , \alpha : A \text{ sig} \\
\Sigma ; \cdot \vdash A \text{ type} \quad \text{META-DECL} \\
\hline
\Sigma \text{ sig} \\
\Sigma , \alpha : A \text{ sig} \\
\Sigma ; \cdot \vdash A \text{ type} \quad \Sigma ; \cdot \vdash t : A \quad \text{META-INST} \\
\hline

The typing relations \( \Sigma ; \cdot \vdash A \text{ type} \) and \( \Sigma ; \cdot \vdash t : A \) are defined in §2.5.

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

- If \( \Sigma_2 = a : A, \Sigma_2' \), then \( \Sigma_1 ; \cdot \vdash A \text{ type} \).
- If \( \Sigma_2 = \alpha : A, \Sigma_2' \), then \( \Sigma_1 ; \cdot \vdash A \text{ type} \).
- If \( \Sigma_2 = \alpha := t : A, \Sigma_2' \), then \( \Sigma_1 ; \cdot \vdash A \text{ type} \) and \( \Sigma_1 ; \cdot \vdash t : A \).

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

\[ \text{support}(\cdot) = \varepsilon \]
\[ \text{support}(\Sigma, a : A) = \text{support}(\Sigma) \cup \{a\} \]
\[ \text{support}(\Sigma, \alpha : A) = \text{support}(\Sigma) \cup \{\alpha\} \]
\[ \text{support}(\Sigma, \alpha := t : A) = \text{support}(\Sigma) \cup \{a\} \]

Notation (Signature concatenation: \( \Sigma_1, \Sigma_2 \)). Signatures can be syntactically viewed as lists. The concatenation of two signatures \( \Sigma_1 \) and \( \Sigma_2 \) is written \( \Sigma_1, \Sigma_2 \). Note that this is a purely syntactic operation. It is not implied that \( \Sigma_2 \) is well-formed on its own, even if \( \Sigma_1 \) and \( \Sigma_1, \Sigma_2 \) are.

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

\[ \text{AtomDecls}(\cdot) = \emptyset \]
\[ \text{AtomDecls}(\Sigma, a : A) = \{a\} \cup \text{AtomDecls}(\Sigma) \]
\[ \text{AtomDecls}(\Sigma, \alpha : A) = \text{AtomDecls}(\Sigma) \]
\[ \text{AtomDecls}(\Sigma, \alpha := t : A) = \text{AtomDecls}(\Sigma) \]

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

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

Definition 2.10 (Metavariables in a term: \text{metas}(t)) The set of metavariables occurring in a term \( t \) (written \text{metas}(t)) is the set of metavariables that occur syntactically in \( t \). The full definition is given in Figure 2.2.
METAS(\(\alpha\)) = \{\alpha\}
METAS(\(\alpha\)) = \emptyset
METAS(x) = \emptyset
METAS(if) = \emptyset
METAS(f u) = METAS(f) \cup METAS(u)
METAS(f \pi_1) = METAS(f)
METAS(f \pi_2) = METAS(f)
METAS(\(\lambda t\)) = METAS(t)
METAS(\(\Pi AB\)) = METAS(A) \cup METAS(B)
METAS(\(\Sigma AB\)) = METAS(A) \cup METAS(B)
METAS(Bool) = \emptyset
METAS(Set) = \emptyset
METAS(c) = \emptyset
METAS(\(\langle t_1, t_2 \rangle\)) = METAS(t_1) \cup METAS(t_2)

Figure 2.2: Metavariables occurring in a term.

ATOMS(\(\alpha\)) = \{\alpha\}
ATOMS(\(\alpha\)) = \emptyset
ATOMS(x) = \emptyset
ATOMS(if) = \emptyset
ATOMS(f u) = ATOMS(f) \cup ATOMS(u)
ATOMS(f \pi_1) = ATOMS(f)
ATOMS(f \pi_2) = ATOMS(f)
ATOMS(\(\lambda t\)) = ATOMS(t)
ATOMS(\(\Pi AB\)) = ATOMS(A) \cup ATOMS(B)
ATOMS(\(\Sigma AB\)) = ATOMS(A) \cup ATOMS(B)
ATOMS(Bool) = \emptyset
ATOMS(Set) = \emptyset
ATOMS(c) = \emptyset
ATOMS(\(\langle t_1, t_2 \rangle\)) = ATOMS(t_1) \cup ATOMS(t_2)

Figure 2.3: Atoms occurring in a term.

Definition 2.11 (Set of atoms in a term: ATOMS(t)). The set of atoms occurring in a term \(t\) (written 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: CONSTS(t)). The set of constants occurring in a term \(t\) (written CONSTS(t)) is the set of metavariables and atoms that occur syntactically in \(t\).

\[
\text{CONSTS}(t) = \text{METAS}(t) \cup \text{ATOMS}(t)
\]
2.4 Contexts (\(\Sigma \vdash \Gamma \text{ ctx}\))

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:

\[
\Gamma, \Delta, \Xi ::= \cdot \quad \text{empty context} \\
\mid \Gamma, A \quad \text{context variable}
\]

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.

\[
\frac{\Sigma \ | \ \cdot \ | \ \text{CTX-EMPTY}}{\Sigma \vdash \cdot \text{ ctx}} \\
\frac{\Sigma \vdash \Gamma \text{ ctx} \quad \Sigma;\Gamma \vdash A \text{ type}}{\Sigma \vdash \Gamma, A \text{ ctx} \quad \text{CTX-VAR}}
\]

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

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

**Definition 2.14 (Support of a context: \(|\Gamma|\)).** The support of a context \(\Gamma\) 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, “\(\cdot \vdash x : A, y : \mathbb{B} x, z : \mathbb{C} x y \text{ ctx}\)” denotes the judgment “\(\cdot \vdash A, \mathbb{B} 0, \mathbb{C} 1 0 \text{ ctx}\)”.

**Notation (Context concatenation: \(\Gamma_1, \Gamma_2\)).** Contexts are syntactically lists of types. The concatenation of two contexts \(\Gamma\) and \(\Delta\) is written \(\Gamma, \Delta\). This is a purely syntactic operation. It is not implied that \(\Delta\) is well-formed on its own, even if \(\Gamma\) and \(\Gamma, \Delta\) are.

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

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.

\[
\frac{\Sigma;\Gamma \vdash A : \text{Set}}{\Sigma;\Gamma \vdash A \text{ type} \quad \text{TYPE}}
\]
\[
\Sigma; \Gamma \vdash A \equiv B : \text{Set} \\
\Sigma; \Gamma \vdash A \equiv B \text{ type}
\]

What it means for a term \(A\) to be of type \(\text{Set}\) \((\Gamma \vdash A : \text{Set})\) is defined in §2.11. Correspondingly, what it means for two terms of type \(\text{Set}\) 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 \(\text{Set}\) are considered types, and the only way for a term to be a type is to be of type \(\text{Set}\). Therefore, the judgments \(\Sigma; \Gamma \vdash A : \text{Set}\) and \(\Sigma; \Gamma \vdash A\) \(\text{type}\) are equivalent for any \(\Sigma, \Gamma\) and \(A\); as are the judgments \(\Sigma; \Gamma \vdash A \equiv B : \text{Set}\) and \(\Sigma; \Gamma \vdash A \equiv B \text{ type}\).

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

## 2.6 Context equality \((\Sigma \vdash \Gamma \equiv \Gamma' \text{ ctx})\)

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 \(\Sigma \vdash \Gamma_1 \text{ ctx}\) and \(\Sigma \vdash \Gamma_2 \text{ ctx}\) are equal (written \(\Sigma \vdash \Gamma_1 \equiv \Gamma_2 \text{ ctx}\)) iff they have the same length, and the types of the variables are equal point-wise.

\[
\begin{align*}
\Sigma \vdash \cdot \equiv \cdot \quad &\text{CTX-EMPTY-EQ} \\
\Sigma \vdash \Gamma_1 \equiv \Gamma_2 \text{ ctx} \quad \Sigma; \Gamma_1 \vdash A_1 \equiv A_2 \text{ type} \\
\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2 \text{ ctx} \quad &\text{CTX-VAR-EQ}
\end{align*}
\]

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

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

## 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. \(x, y, z, \ldots\)) stand for natural numbers (e.g. 0, 1, 2, ...). This convention has the benefit of giving the same representation to all \(\alpha\)-equivalent terms. For instance, the informally written terms \([\lambda x. \lambda y. x]\) and \([\lambda z. \lambda x. z]\) 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 (\(\lambda, \Pi\) or \(\Sigma\)). For instance, the syntax \(\lambda x. \alpha. x (\lambda x. \lambda y. x)\) denotes the term \(\lambda. \alpha. 0 (\lambda. 1)\). The binders for \(\Pi\) and \(\Sigma\),
2.7. BINDERS AND VARIABLES

\[
\begin{align*}
\text{FV}(x) & = \{x\} \\
\text{FV}(\alpha) & = \emptyset \\
\text{FV}(\omega) & = \emptyset \\
\text{FV}(\text{if}) & = \emptyset \\
\text{FV}(f \ e) & = \text{FV}(f) \cup \text{FV}(e) \\
\text{FV}(\lambda t) & = \text{FV}(t)^{-1} \\
\text{FV}(\Pi AB) & = \text{FV}(A) \cup (\text{FV}(B) - 1) \\
\text{FV}(\Sigma AB) & = \text{FV}(A) \cup (\text{FV}(B) - 1) \\
\text{FV}(\text{Bool}) & = \emptyset \\
\text{FV}(\text{Set}) & = \emptyset \\
\text{FV}(c) & = \emptyset \\
\text{FV}((t_1, t_2)) & = \text{FV}(t_1) \cup \text{FV}(t_2) \\
\text{FV}(\pi_1) = \text{FV}(\pi_2) = \emptyset
\end{align*}
\]

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: \(\lambda \bar{x}^n.t, \Pi(x : A)^n B\)). We may use vector notation to bind several variables at the same time. For instance, \(\lambda \bar{x}^n.t\) denotes the term \(\lambda x_1 \lambda x_2 \ldots \lambda x_n.t\), and we use \(\Pi(x : A)^n B\) to denote the term \(\Pi(x_1 : A_1)\Pi(x_2 : A_2)\ldots\Pi(x_n : A_n)B\). When \(n = 0\), it is interpreted as an absence of binders: \(\lambda \bar{x}^0.t \equiv t\) and \(\Pi(x : A)^0 B \equiv B\).

Notation (Arrow notation for \(\Pi\)-types: \((x : A) \rightarrow B, A \rightarrow B\)). We may use \((x : A) \rightarrow B\) 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 \(A \rightarrow B\) instead.

Notation (Product notation for \(\Sigma\)-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 \(\{n - k | n \in X, n \geq k\}\).

Definition 2.18 (Free variables in a term: \(\text{FV}(t)\)). The free variables in a term \(t\) (written \(\text{FV}(t)\)) are the set of variables which are not bound by a binder (i.e. \(\lambda, \Pi\) or \(\Sigma\)). The full definition of \(\text{FV}(t)\) is given in Figure 2.4.

Definition 2.19 (Free variables of a context: \(\text{FV}(\Delta)\)). Given a (partial) context \(\Delta\), the set of free variables of \(\Delta\) (written \(\text{FV}(\Delta)\)) is defined as follows:

\[
\begin{align*}
\text{FV}(\cdot) & = \emptyset \\
\text{FV}(A, \Delta) & = \text{FV}(A) \cup (\text{FV}(\Delta) - 1)
\end{align*}
\]
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 \( \rho \) 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: \((+n)\)). For \( n \in \mathbb{N} \), the renaming \([0... \mapsto n...]\) maps variable \( x \) to variable \( x + n \):

\[
[0... \mapsto n...] : \mathbb{N} \to \mathbb{N} \quad x \mapsto x + n
\]

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

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

\[
[n... \mapsto 0...] : \mathbb{N}/\{0,...,n-1\} \to \mathbb{N} \quad x \mapsto x - n
\]

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:

\[
(\rho + n) : 0, 1, 2, ..., n - 1 \cup \{x + n \mid x \in A\} \to \mathbb{N} \\
x \mapsto x \quad \text{if} \quad x < n \\
x \mapsto \rho(x - n) + n \quad \text{if} \quad x \geq n
\]

**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.

\[
(((−1) + n) : \mathbb{N} - \{n\} \to \mathbb{N} \\
i \mapsto i \quad \text{if} \quad i < n \\
i \mapsto i - 1 \quad \text{if} \quad i > n
\]

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

\[
\begin{align*}
x \rho &= \rho(x) \\
\alpha \rho &= \alpha \\
\sigma \rho &= \sigma \\
\text{if } \rho &= \text{if} \\
(f e) \rho &= (f \rho) (e \rho) \\
(\lambda t) \rho &= \lambda(t(\rho + 1)) \\
(\Pi AB) \rho &= \Pi(A \rho)(B(\rho + 1)) \\
(\Sigma AB) \rho &= \Sigma(A \rho)(B(\rho + 1)) \\
\text{Bool } \rho &= \text{Bool} \\
\text{Set } \rho &= \text{Set} \\
c \rho &= c \\
\langle t_1, t_2 \rangle \rho &= \langle t_1 \rho, t_2 \rho \rangle \\
(.\pi_1) \rho &= .\pi_1 \\
(.\pi_2) \rho &= .\pi_2
\end{align*}
\]

Figure 2.5: Applying a renaming \( \rho \) to a term.

**Definition 2.26** (Application of a renaming to a term: \( t \rho, t^\rho \)). Let \( \rho : A \to \mathbb{N} \) be a renaming, and \( t \) a term such that \( \text{fv}(t) \subseteq A \). Then \( t \rho \) is a term where each free variable is renumbered according to \( \rho \). The full definition is stated in Figure 2.5.

For conciseness, we may sometimes write renaming applications as \( t^\rho \) instead of \( t \rho \). The two notations have identical meanings.

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

\[
\begin{align*}
(\cdot) \rho &= \cdot \\
(A, \Delta) \rho &= (A \rho), (\Delta (\rho + 1))
\end{align*}
\]

**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 \( \text{fv}(t) \subseteq A \), then \( \text{fv}(t \rho) = \rho(\text{fv}(t)) \).

**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 \( \rho_1 \) and \( \rho_2 \) (i.e. \( \rho_1 \rho_2 : \rho_1^{-1}(B) \to \mathbb{N}, \text{with } (\rho_1 \rho_2)(x) = \rho_2(\rho_1(x)) \)).

**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 \( \text{fv}(t) \subseteq \rho_1^{-1}(B) \), 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 \textbf{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 $\rho$ 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 $\lambda$-terms, namely those in $\beta$-normal form. Terms in $\beta$-normal form are those which do not contain $\beta$-redexes. Examples of $\beta$-redexes (which are not valid terms according to our syntax) are $[(\lambda.t) u], [\langle t, u \rangle . \pi_1]$ and $[(t, u) . \pi_2]$.

A definition of substitution which simply replaces each variable with its respective term would create $\beta$-redexes. Because our syntax only allows for $\beta$-normal terms, we need to define substitution in such a way that the result is also in $\beta$-normal form: i.e. a hereditary substitution [9, 10].

Note that terms such as “if $\mathbb{A}$ true $a$ $b$” and “if $\mathbb{A}$ false $a$ $b$” are not considered $\beta$-redexes, but are instead subject to $\delta$-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 ($\vec{e}^n[t/x] \Downarrow \vec{e}'^n$).** We write $\vec{e}^n[t] \Downarrow \vec{e}'^n$ 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 $\beta$-redexes. Because our syntax is restricted to $\beta$-normal terms, we need to ensure that the result of such an application is in $\beta$-normal form. That is, we need to perform a hereditary elimination.
2.9. HEREDITARY SUBSTITUTION AND ELIMINATION

\[ x[u/x] \downarrow u \]
\[ y[u/x] \downarrow y \]
\[ y[u/x] \downarrow (y-1) \quad \text{if} \quad x > y \]
\[ y[u/x] \downarrow (y-1) \quad \text{if} \quad x < y \]
\[ \alpha[u/x] \downarrow \alpha \]
\[ \circ[u/x] \downarrow \circ \]
\[ \text{if}[u/x] \downarrow \text{if} \]

\[ (f \cdot t)[u/x] \downarrow r \quad \text{if} \quad f[u/x] \downarrow r_1 \land \land t[u/x] \downarrow r_2 \land r_1 \triangle r_2 \downarrow r \]
\[ (f \cdot \pi_1)[u/x] \downarrow r \quad \text{if} \quad f[u/x] \downarrow r_1 \land \land \pi_1 \downarrow r \]
\[ (f \cdot \pi_2)[u/x] \downarrow r \quad \text{if} \quad f[u/x] \downarrow r_1 \land \land \pi_2 \downarrow r \]

\[ (\lambda t)[u/x] \downarrow (\lambda x) \quad \text{if} \quad t[u(+1)/x+1] \downarrow r \]
\[ (\Pi AB)[u/x] \downarrow (\Pi A'B') \quad \text{if} \quad A[u/x] \downarrow A' \land \land B[u(+1)/x+1] \downarrow B' \]
\[ (\Sigma AB)[u/x] \downarrow (\Sigma A'B') \quad \text{if} \quad A[u/x] \downarrow A' \land \land B[u(+1)/x+1] \downarrow B' \]
\[ \text{Bool}[u/x] \downarrow \text{Bool} \]
\[ \text{Set}[u/x] \downarrow \text{Set} \]
\[ c[u/x] \downarrow c \]
\[ \langle t_1, t_2 \rangle[u/x] \downarrow \langle t_1', t_2' \rangle \quad \text{if} \quad t_1[u/x] \downarrow t_1' \land \land t_2[u/x] \downarrow t_2' \]

(a) Hereditary substitution

\[ h \circ e \circ e' \downarrow (h \circ e') \]
\[ \langle t_1, t_2 \rangle \circ \pi_1 \downarrow t_1 \]
\[ \langle t_1, t_2 \rangle \circ \pi_1 \downarrow t_2 \]
\[ \lambda t @ u \downarrow r \quad \text{if} \quad t[u/0] \downarrow r \]

(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 \( t @ e \downarrow \) as a shorthand for \( \exists r. t @ e \downarrow r \). Furthermore, if in a given context, \( \exists r. t @ e \downarrow r \) holds, then we will denote such an \( r \) by \( t @ e \).

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

If \( t @ e_1 \ldots e_n \downarrow r \) holds for some \( r \), we denote such an \( r \) by \( t @ e_1 \ldots e_n \).

**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 \( \forall v \in \text{fv}(u_i). v \geq m \). We use \( t[\vec{u}/\vec{x}] \downarrow r \) as a shorthand for \( \exists t_1, \ldots, t_{n-1}. (t[u_1^{+(n-1)}/x_1] \downarrow t_1) \land (t_1[u_2^{+(n-2)}/x_2] \downarrow t_2) \land \ldots \land (t_{n-1}[u_n^{+(0)}/x_n] \downarrow r) \). For \( n = 0 \), \( t[\varepsilon/\varepsilon] \downarrow t \), and, for \( n = 1 \), \( t[\vec{u}/\vec{x}] \downarrow r \) 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 \( t[\vec{u}/\vec{x}] \).

**Remark 2.35** (Iterated application as substitution on body). We have \( (\lambda^n.t) @ \vec{u} \downarrow r \) 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: \( f @ \vec{e} \)). Given a neutral term \( f \) and an eliminator \( e \), by Definition 2.32 (hereditary elimination), \( f @ e \downarrow \). Given \( \vec{e} \) and applying the above remark iteratively, we have \( f @ \vec{e} \downarrow (f \vec{e}) \). Therefore, we can always write \( f @ \vec{e} \).

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

\[
\cdot[u/x] \downarrow \cdot \quad (A, \Delta)[u/x] \downarrow (A', \Delta) \quad \text{if} \quad A[u/x] \downarrow A' \quad \text{and} \quad \Delta'[u(\varepsilon/\varepsilon)+1/x+1] \downarrow \Delta'
\]

**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 \( \rho \) 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 \( t @ e \downarrow u \), then \( (t^\rho @ e^\rho) \downarrow u^\rho \).
**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, \( V[\vec{x}^n] = V[\ldots,n,(n-1),...,0 \mapsto \ldots,1,0,\vec{x}] \).

**Proof.** By induction on the structure of \( V \).

### 2.10 Head lookup (\( \Sigma;\Gamma \vdash \text{h} \Rightarrow A \))

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).

\[
\frac{\Sigma \vdash \Gamma \text{ctx}}{\Sigma;\Gamma \vdash n \Rightarrow A^{(+|\Gamma|)}} \quad \text{VAR}
\]

\[
\frac{\Sigma \vdash \Gamma \text{ctx} \quad \alpha : A \in \Sigma}{{\Sigma};\Gamma \vdash \alpha \Rightarrow A} \quad \text{META}_1
\]

\[
\frac{\Sigma \vdash \Gamma \text{ctx} \quad \alpha := t : A \in \Sigma}{{\Sigma};\Gamma \vdash \alpha \Rightarrow A} \quad \text{META}_2
\]

\[
\frac{\Sigma \vdash \Gamma \text{ctx} \quad \phi : A \in \Sigma}{{\Sigma};\Gamma \vdash \phi \Rightarrow A} \quad \text{ATOM}
\]

\[
\frac{\Sigma \vdash \Gamma \text{ctx} \quad \text{if} \Rightarrow \Pi(\Pi\text{BoolSet})(\Pi\text{Bool}(\Pi(1\text{ true})(\Pi(2\text{ false})(3\text{ 2}))))}{\Sigma;\Gamma \vdash \text{if} \Rightarrow (X : \text{Bool} \Rightarrow \text{Set}) \Rightarrow (y : \text{Bool}) \Rightarrow X \text{ true} \Rightarrow X \text{ false} \Rightarrow X \text{ y}} \quad \text{IF}
\]

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

**Remark.** In the conclusion of the rules ATOM, META$_1$, and 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 \( \Gamma \) under signature \( \Sigma \)” is written \( \Sigma;\Gamma \vdash t : A \).

**Notation** (Implicit signature). In those rules where the signature \( \Sigma \) is omitted, it is understood that all premises and the conclusion share the same signature \( \Sigma \). The rule then holds for any such \( \Sigma \). For instance, consider the first typing rule given below, the BOOL rule (left); and its implied full form (right).
Even though the premises and the conclusion of a rule often share the same context $\Gamma$, 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**

$$\frac{\Gamma \texttt{ctx}}{\Gamma \vdash \texttt{bool} : \texttt{Set}} \quad \text{BOOL} \quad \frac{\Sigma \vdash \Gamma \texttt{ctx}}{\Sigma;\Gamma \vdash \texttt{bool} : \texttt{Set}} \quad \text{BOOL}$$

**Term constructors**

$$\frac{\Gamma \texttt{ctx}}{\Gamma \vdash \texttt{true} : \texttt{bool}} \quad \text{TRUE}$$

$$\frac{\Gamma \texttt{ctx}}{\Gamma \vdash \texttt{false} : \texttt{bool}} \quad \text{FALSE}$$

$$\frac{\Gamma, \vdash \ t : \ A \quad \Gamma, A \vdash \ B : \ \texttt{Set}}{\Gamma \vdash \lambda \ t : \Pi A B \vdash \ B : \ \texttt{Set}} \quad \text{ABS}$$

$$\frac{\Gamma \vdash \ t : \ A \quad \Gamma, A \vdash \ B \quad \text{type} \quad \Gamma \vdash \ u : \ B[t] \downarrow}{\Gamma \vdash \langle \ t, \ u \rangle : \Sigma A B} \quad \text{PAIR}$$

**Neutral terms**

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

$$\frac{\Gamma \vdash \ f : \Sigma A B}{\Gamma \vdash \ f . \pi_1 : A} \quad \text{PROJ1}$$

$$\frac{\Gamma \vdash \ f : \Sigma A B}{\Gamma \vdash \ f . \pi_2 : B[f . \pi_1]} \quad \text{PROJ2}$$

$$\frac{\Gamma \vdash \ f : \Pi A B \quad \Gamma \vdash \ t : \ A \quad \Gamma \vdash \ B[t] \downarrow}{\Gamma \vdash \ f \ t : \ B[t]} \quad \text{APP}$$
2.12 The Set : Set judgment and normalization

Our theory includes $\Sigma;\Gamma \vdash \text{Set} : \text{Set}$ 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 $\Sigma;\Gamma \vdash \text{Set} : \text{Set}$ 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 $\Gamma \vdash A \equiv B : \text{Set}$.

\[
\begin{align*}
\Gamma \text{ ctx} & \quad \text{BOOL-EQ} \\
\Gamma \vdash \text{Bool} \equiv \text{Bool} : \text{Set} & \\
\Gamma \vdash A \equiv A' : \text{Set} & \quad \Gamma, A \vdash B \equiv B' : \text{Set} \quad \text{PI-EQ} \\
\Gamma \vdash \Pi A B \equiv \Pi A' B' : \text{Set} & \\
\Gamma \vdash A \equiv A' : \text{Set} & \quad \Gamma, A \vdash B \equiv B' : \text{Set} \quad \text{SIGMA-EQ} \\
\Gamma \vdash \Sigma A B \equiv \Sigma A' B' : \text{Set} & \\
\Gamma \text{ ctx} & \quad \text{SET-EQ} \\
\Gamma \vdash \text{Set} \equiv \text{Set} : \text{Set} & \\
\Gamma \text{ ctx} & \quad \text{TRUE-EQ} \\
\Gamma \vdash \text{true} \equiv \text{true} : \text{Bool} & \\
\Gamma \text{ ctx} & \quad \text{FALSE-EQ} \\
\Gamma \vdash \text{false} \equiv \text{false} : \text{Bool} & \\
\Gamma, A \vdash t \equiv u : B & \quad \text{ABS-EQ} \\
\Gamma \vdash \lambda t. t \equiv \lambda u. \Pi A B & \\
\Gamma \vdash t_1 \equiv u_1 : A & \\
\Gamma, A \vdash B \text{ type} & \quad \text{PAIR-EQ} \\
\Gamma \vdash B[t_1][\downarrow] & \quad \Gamma \vdash t_2 \equiv u_2 : B[t_1] \\
\Gamma \vdash \langle t_1, t_2 \rangle \equiv \langle u_1, u_2 \rangle : \Sigma A B
\end{align*}
\]
Elimination

\[
\frac{\Gamma \vdash h \Rightarrow A}{\Gamma \vdash h = h : A} \quad \text{HEAD-EQ}
\]

\[
\frac{\Gamma \vdash f \equiv g : \Sigma AB}{\Gamma \vdash f . \pi_1 
\equiv g . \pi_1 : A} \quad \text{PROJ1-EQ}
\]

\[
\frac{\Gamma \vdash f \equiv g : \Sigma AB}{\Gamma \vdash f . \pi_2 
\equiv g . \pi_2 : B[f . \pi_1]} \quad \text{PROJ2-EQ}
\]

\[
\frac{\Gamma \vdash f \equiv g : \Pi AB}{\Gamma \vdash t \equiv u : A \quad B[t] \D}{\Gamma \vdash ft \equiv gu : B[t]} \quad \text{APP-EQ}
\]

Remark. In the HEAD-EQ rule, \(h\) stands for either (i) a variable \(x\), (ii) a metavariable \(\alpha\) (iii) an atom \(a\), 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 \(\beta\)-redexes, which are disallowed by the syntax (§2.1).

\(\eta\)-conversion

\[
\frac{\Gamma \vdash f : \Pi AB}{\Gamma \vdash f = \lambda f^{(+1)} 0 : \Pi AB} \quad \text{ETA-ABS}
\]

\[
\frac{\Gamma \vdash f : \Sigma AB}{\Gamma \vdash f = \langle f . \pi_1, f . \pi_2 \rangle : \Sigma AB} \quad \text{ETA-PAIR}
\]

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

\(\delta\)-conversion

\[
\Sigma; \Gamma \vdash \alpha \bar{e} : T \quad \Sigma; \Gamma \vdash t : T \quad \alpha := t : A \in \Sigma \\
\Sigma; \Gamma \vdash \alpha \bar{e} = t' : T \quad t \downarrow \bar{e} \downarrow t' \quad \text{DELTA-META}
\]

\[
\Gamma \vdash \text{if } A \text{ true } u_t u_f \bar{e} : T \quad \Gamma \vdash u' : T \quad u_t \downarrow \bar{e} \downarrow u' \\
\Gamma \vdash \text{if } A \text{ true } u_t u_f \bar{e} = u' : T \quad \text{DELTA-IF-TRUE}
\]

\[
\Gamma \vdash \text{if } A \text{ false } u_t u_f \bar{e} : T \quad \Gamma \vdash u' : T \quad u_f \downarrow \bar{e} \downarrow u' \\
\Gamma \vdash \text{if } A \text{ false } u_t u_f \bar{e} = u' : T \quad \text{DELTA-IF-FALSE}
\]

Remark. Because the term syntax disallows the possibility of intermediate \(\beta\)-redexes (§2.1), the rules DELTA-META, DELTA-IF-TRUE and DELTA-IF-FALSE include an eliminator spine \(\bar{e}\) in the LHS of the equality. This way they can be applied to the entire neutral term that contains the \(\delta\)-redex.
### 2.14 Term reduction ($\rightarrow_{\delta\eta}, \rightarrow^{*}_{\delta\eta}$)

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: $\Sigma;\Gamma \vdash t \rightarrow_{\delta\eta} u : T$). Let $\Sigma$ be a signature, $\Gamma$ a context and $T$ a type. The relation $\rightarrow_{\delta\eta}$ is defined in Figure 2.7 on page 32, with the additional requirement that, whenever $\Sigma;\Gamma \vdash t \rightarrow_{\delta\eta} u : T$, 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 $\text{APP}_n$ 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 $\delta$-rules $\text{META}, \text{IP}_1$ and $\text{IP}_2$ only apply to the whole neutral term, thus precluding the creation of $\beta$-redexes, which are disallowed by the syntax.

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

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

### 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 ($\delta\eta$-normalization step). For each recursive occurrence of the form $\Sigma;\Gamma \vdash t \rightarrow_{\delta\eta} u : T$, there is an implicit condition that $\Sigma;\Gamma \vdash t : T$.

$$\begin{align*}
    (\Pi_1) & \quad \Sigma;\Gamma \vdash \Pi A B \rightarrow_{\delta\eta} \Pi A'B : T & \text{if } & \Sigma;\Gamma \vdash A \rightarrow_{\delta\eta} A' : \text{Set} \\
    (\Pi_2) & \quad \Sigma;\Gamma \vdash \Pi A B \rightarrow_{\delta\eta} \Pi A'B' : T & \text{if } & \Sigma;\Gamma, A \vdash B \rightarrow_{\delta\eta} B' : \text{Set} \\
    (\Sigma_1) & \quad \Sigma;\Gamma \vdash \Sigma A B \rightarrow_{\delta\eta} \Sigma A'B : T & \text{if } & \Sigma;\Gamma \vdash A \rightarrow_{\delta\eta} A' : \text{Set} \\
    (\Sigma_2) & \quad \Sigma;\Gamma \vdash \Sigma A B \rightarrow_{\delta\eta} \Sigma A'B' : T & \text{if } & \Sigma;\Gamma, A \vdash B \rightarrow_{\delta\eta} B' : \text{Set} \\
    (\lambda) & \quad \Sigma;\Gamma \vdash \lambda t . t \rightarrow_{\delta\eta} \lambda t' : T & \text{if } & \Sigma;\Gamma \vdash T \equiv \Pi A B \text{ type} \\
    & \quad \text{and } \Sigma;\Gamma, A \vdash t \rightarrow_{\delta\eta} t' : B \\
    (\langle, \rangle_1) & \quad \Sigma;\Gamma \vdash \langle t, u \rangle \rightarrow_{\delta\eta} \langle t', u \rangle : T & \text{if } & \Sigma;\Gamma \vdash T \equiv \Sigma A B \text{ type} \\
    & \quad \Sigma;\Gamma \vdash t \rightarrow_{\delta\eta} t' : A \\
    (\langle, \rangle_2) & \quad \Sigma;\Gamma \vdash \langle t, u \rangle \rightarrow_{\delta\eta} \langle t, u' \rangle : T & \text{if } & \Sigma;\Gamma \vdash T \equiv \Sigma A B \text{ type} \\
    & \quad \text{and } B[t] \Downarrow u' \\
    & \quad \text{and } \Sigma;\Gamma \vdash u \rightarrow_{\delta\eta} u' : B[t] \\
    (\eta-\Pi) & \quad \Sigma;\Gamma \vdash f \rightarrow_{\delta\eta} \lambda(f^{(+1)}) 0 : T & \text{if } & \Sigma;\Gamma \vdash T \equiv \Pi A B \text{ type} \\
    (\eta-\Sigma) & \quad \Sigma;\Gamma \vdash f \rightarrow_{\delta\eta} \langle f . \pi_1, f . \pi_2 \rangle : T & \text{if } & \Sigma;\Gamma \vdash T \equiv \Sigma A B \text{ type} \\
    (\text{APP}_n) & \quad \Sigma;\Gamma \vdash h \overline{e}^{n-1} t \overline{e}' \rightarrow_{\delta\eta} h \overline{e} u \overline{e}'' : T & \text{if } & \Sigma;\Gamma \vdash h \overline{e} : \Pi U V \\
    & \quad \text{and } \Sigma;\Gamma \vdash t \rightarrow_{\delta\eta} u : U \\
    (\text{META}) & \quad \Sigma;\Gamma \vdash \alpha \overline{e} \rightarrow_{\delta\eta} u : T & \text{if } & \alpha := t : A \in \Sigma \\
    & \quad \text{and } (t @ \overline{e}) \Downarrow u \\
    (\text{IF}_1) & \quad \Sigma;\Gamma \vdash \text{if } A \text{ true } t u \overline{e} \rightarrow_{\delta\eta} t' : T & \text{if } & (t @ \overline{e}) \Downarrow t' \\
    (\text{IF}_2) & \quad \Sigma;\Gamma \vdash \text{if } A \text{ false } t u \overline{e} \rightarrow_{\delta\eta} u' : T & \text{if } & (u @ \overline{e}) \Downarrow u' 
\end{align*}
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: \( \Delta \ \text{ctx} \), \( \Delta \vdash A \ \text{type} \), \( \Delta \vdash A \equiv B \ \text{type} \), \( \Delta_1 \equiv \Delta_2 \ \text{ctx} \), \( \Delta \vdash t : A \), \( \Delta \vdash t \equiv u : A \), and \( J_1 \wedge J_2 \).

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

- \( J = \Delta \ \text{ctx} \), and \( \Sigma \vdash \Gamma, \Delta \).
- \( J = \Delta_1 \equiv \Delta_2 \ \text{ctx} \) and \( \Sigma \vdash \Gamma, \Delta_1 \equiv \Gamma, \Delta_2 \ \text{ctx} \).
- \( J = \Delta \vdash A \ \text{type} \) and \( \Sigma; \Gamma, \Delta \vdash A \ \text{type} \).
- \( J = \Delta \vdash A \equiv B \ \text{type} \) and \( \Sigma; \Gamma, \Delta \vdash A \equiv B \ \text{type} \).
- \( J = \Delta \vdash t : A \) 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: \( \text{FV}(\Delta \vdash t : B) \), \( \text{FV}(J) \)). We can consider the variables free in an entire judgment:

\[
\begin{align*}
\text{FV}(\Delta \ \text{ctx}) &= \text{FV}(\Delta) \\
\text{FV}(\Delta_1 \equiv \Delta_2 \ \text{ctx}) &= \text{FV}(\Delta_1) \cup \text{FV}(\Delta_2) \\
\text{FV}(\cdot \vdash A \ \text{type}) &= \text{FV}(A) \\
\text{FV}(\cdot \vdash A \equiv B \ \text{type}) &= \text{FV}(A) \cup \text{FV}(B) \\
\text{FV}(\cdot \vdash t : B) &= \text{FV}(t) \cup \text{FV}(B) \\
\text{FV}(\cdot \vdash t \equiv u : B) &= \text{FV}(t) \cup \text{FV}(u) \cup \text{FV}(B) \\
\text{FV}(A, J) &= \text{FV}(A) \cup (\text{FV}(J) \setminus \{0\}) - 1 \\
\text{FV}(J_1 \wedge J_2) &= \text{FV}(J_1) \cup \text{FV}(J_2)
\end{align*}
\]

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

\[
\begin{align*}
\text{consts}(\Delta_1 \equiv \Delta_2 \ \text{ctx}) &= \text{consts}(\Delta_1) \cup \text{consts}(\Delta_2) \\
\text{consts}(\cdot \vdash t \equiv u : B) &= \text{consts}(t) \cup \text{consts}(u) \cup \text{consts}(B) \\
\text{consts}(A, J) &= \text{consts}(A) \cup \text{consts}(J) \\
\text{consts}(J_1 \wedge J_2) &= \text{consts}(J_1) \cup \text{consts}(J_2)
\end{align*}
\]

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:

\[
\begin{align*}
(\Delta \text{ ctx}) \rho &= (\Delta \rho) \text{ ctx} \\
(\Delta_1 \equiv \Delta_2 \text{ ctx}) \rho &= \Delta_1 \rho \equiv \Delta_2 \rho \text{ ctx} \\
(\vdash A \text{ type}) \rho &= \vdash A \rho \text{ type} \\
(\vdash A \equiv B \text{ type}) \rho &= \vdash A \rho \equiv B \rho \text{ type} \\
(\vdash t : B) \rho &= \vdash t \rho : B \rho \\
(A, J) \rho &= (A \rho, J(\rho + 1)) \\
(J_1 \land J_2) \rho &= (J_1 \rho) \land (J_2 \rho)
\end{align*}
\]

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.

\[
\begin{align*}
(\vdash t : B)[\!u/\!x] \Downarrow (\vdash t' : B') & \quad \text{if} \quad t[\!u/\!x] \Downarrow t' \text{ and } B[\!u/\!x] \Downarrow B' \\
(A, J)[\!u/\!x] \Downarrow (A', J') & \quad \text{if} \quad A[\!u/\!x] \Downarrow A' \text{ and } J[\!u(\rho + 1)/\!x + 1] \Downarrow J'
\end{align*}
\]

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 \(\Sigma; \Gamma, x : A, \Delta \vdash t : A\) and \(\Gamma \vdash u : B\), then \(\Delta[\!u/\!x] \Downarrow, t[\!u(\rho + \Delta)/\!x] \Downarrow, A[u(\rho + \Delta)/\!x] \Downarrow, \) and \(\Gamma, \Delta[\!u/\!x] \vdash t[\!u(\rho + \Delta)/\!x] : A[u(\rho + \Delta)/\!x]\).

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

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

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 \(\Delta[u_1/\!x] \Downarrow, \Delta[u_2/\!x] \Downarrow, t_1[u_1(\rho + \Delta)/\!x] \Downarrow, t_2[u_2(\rho + \Delta)/\!x] \Downarrow, B[u_1(\rho + \Delta)/\!x] \Downarrow, B[u_2(\rho + \Delta)/\!x] \Downarrow, \) \(\Sigma \vdash \Gamma, \Delta[u_1/\!x], B[u_1(\rho + \Delta)/\!x] \equiv \Gamma, \Delta[u_2/\!x], B[u_2(\rho + \Delta)/\!x] \text{ ctx}\) and \(\Sigma; \Gamma, \Delta[u_1/\!x] \vdash t_1[u_1(\rho + \Delta)/\!x] \equiv t_2[u_2(\rho + \Delta)/\!x] : B[u_1(\rho + \Delta)/\!x]\).

Remark 2.49 (Strengthening by substitution). If \(x \notin \text{fv}(J)\), then \(J[u/\!x] = J(\rho + 1)+x\).

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 \(|\Delta| \notin \text{fv}(V, \Xi \vdash t : A)\).

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

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

**Postulate 7** (Congruence of hereditary projection). If $\Sigma;\Gamma \vdash t \equiv u : \Sigma A B$, then $(t @ .\pi_1)\psi$, $(u @ .\pi_1)\psi$, $\Sigma;\Gamma \vdash t @ .\pi_1 \equiv u @ .\pi_1 : A$, and also $(t @ .\pi_2)\psi$, $(u @ .\pi_2)\psi$, $B[t @ .\pi_1]\psi$, and $\Sigma;\Gamma \vdash t @ .\pi_2 \equiv u @ .\pi_2 : B[t @ .\pi_1]$. 

**Postulate 8** (No infinite chains). If $\Sigma;\Gamma \vdash t : A$, then there is no infinite chain of reductions $\Sigma;\Gamma \vdash \ldots \rightarrow \delta \eta \ldots : A$ 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 $i \in \mathbb{N}$, $\Sigma;\Gamma \vdash u_i \rightarrow \delta \eta u_{i+1} : A$.

**Definition 2.50** (Set of free variables, strengthened: $fv_x(t)$). The set of free variables of $t$ strengthened by $x$ is denoted by $fv_x(t)$ and is defined as $fv_x(t) \doteq \{ y -1 \mid y \in fv(t), y > x \} \cup \{ y \mid y \in fv(t), y < x \}$.

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

- If $t[u/x] \Downarrow r$, then $fv(r) \subseteq fv_x(t) \cup fv(u)$.
- If $(t @ e) \Downarrow r$, then $fv(r) \subseteq fv(t) \cup fv(e)$.

**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 $\Sigma;\Gamma, V \vdash u : \Pi A B$, and $\bar{t}$ such that $\Sigma;\Gamma, V \vdash t_i : A[t_1,\ldots,t_{i-1}]$. Finally, let $\bar{v}$ be such that $\Sigma;\Gamma, V \vdash v : V$. Then $(u \circ \bar{t})[\bar{v}] = (u[v] \circ t_1[v] \ldots t_n[v])$. 

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

**Proof.** See the proof of Lemma 2.52 in the licentiate thesis [54].

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

**Lemma 2.53** (Σ inversion). If $\Sigma;\Gamma \vdash \Sigma A B : T$, then $\Sigma;\Gamma \vdash T \equiv \text{Set type}$ $\Sigma;\Gamma \vdash A : \text{Set}$ and $\Sigma;\Gamma, A \vdash B : \text{Set}$. Also, by Remark 2.15 (there is only set), if $\Sigma;\Gamma \vdash \Sigma A B \equiv \text{type}$, then $\Sigma;\Gamma \vdash A \equiv \text{type}$ and $\Sigma;\Gamma, A \vdash B \equiv \text{type}$.
Proof. Analogous to the proof for Lemma 2.52 (Π inversion). \hfill\square

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

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

- **Reflexivity**: If \( \Sigma;\Gamma \vdash t : A \), then \( \Sigma;\Gamma \vdash t \equiv t : A \).
- **Symmetry**: If \( \Sigma;\Gamma \vdash t \equiv u : A \), then \( \Sigma;\Gamma \vdash u \equiv t : A \).
- **Transitivity**: If \( \Sigma;\Gamma \vdash t \equiv u : A \) and \( \Sigma;\Gamma \vdash u \equiv v : A \), then 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 \([x]-\text{eq}\).
Symmetry and transitivity are rules themselves. \hfill\square

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

- **Reflexivity**: If \( \Sigma;\Gamma \vdash A \text{ type} \), then \( \Sigma;\Gamma \vdash A \equiv A \text{ type} \).
- **Symmetry**: If \( \Sigma;\Gamma \vdash A \equiv B \text{ type} \), then \( \Sigma;\Gamma \vdash B \equiv A \text{ type} \).
- **Transitivity**: If \( \Sigma;\Gamma \vdash A \equiv B : \text{type} \) and \( \Sigma;\Gamma \vdash B \equiv C \text{ type} \), then \( \Sigma;\Gamma \vdash A \equiv C \text{ type} \).

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

**Lemma 2.56** (Neutral inversion).

- If \( \Sigma;\Gamma \vdash f \ t \bar{e} : T \), then \( \Sigma;\Gamma \vdash f : \Pi AB \) and \( \Sigma;\Gamma \vdash t : A \text{ for some } A \text{ and } B, \text{ with } B[t]\downarrow \).
- If \( \Sigma;\Gamma \vdash f \ . \pi_1 \bar{e} : T \) then \( \Sigma;\Gamma \vdash f : \Sigma AB \).
- If \( \Sigma;\Gamma \vdash f \ . \pi_2 \bar{e} : T \), then \( \Sigma;\Gamma \vdash f : \Sigma AB \text{ for some } A, B, \text{ with } B[f]\downarrow \).

Proof. See the proof of Lemma 2.56 in the licentiate thesis [54]. \hfill\square

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

Proof. See the proof of Lemma 2.57 in the licentiate thesis [54]. \hfill\square

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

Proof. By induction on \( n \), using Lemma 2.57. \hfill\square
Lemma 2.59 (Abstraction equality inversion). We have \( \Sigma;\Gamma \vdash \lambda^n.t \equiv \lambda^n.u : \Pi \vec{A} \), if and only if \( \Sigma;\Gamma, \vec{A} \vdash t \equiv u : B \).

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 \( \Sigma;\Gamma \vdash \Sigma AB \equiv T \text{ type} \), \( \Sigma;\Gamma \vdash t_1 : A \). \( B[t_1] \downarrow \) 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 \( \Sigma \vdash \Gamma \text{ ctx} \), then \( \Sigma \vdash \Gamma \equiv \Gamma \text{ ctx} \)).

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

Lemma 2.62 (Context weakening). Let \( J \) be a judgment. If \( \Sigma;\Gamma_1 \vdash J \), \( \Sigma \vdash \Gamma_1, \Gamma_2 \text{ ctx} \), 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 \( \Sigma \vdash \Gamma \equiv \Gamma' \text{ ctx} \), then \( \Sigma;\Gamma' \vdash J \).

- Furthermore, if \( J = (t : A) \) (or \( J = (t \equiv u : A) \)), and \( \Sigma;\Gamma \vdash A \equiv A' \text{ type} \) (that is, \( \Sigma \vdash \Gamma, A \equiv \Gamma', A' \text{ ctx} \)), 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 \( \text{FV}(t) \subseteq \{0, \ldots, |\Gamma| - 1\} \).

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

Corollary 2.66 (The signature is closed). Let \( \Sigma \) be a well-formed signature (\( \Sigma \text{ sig} \)). If \( \alpha : A \in \Sigma \) or \( \alpha : A \in \Sigma \), then \( \text{FV}(A) = \emptyset \). Also, if \( \alpha := t : A \in \Sigma \), then \( \text{FV}(t) = \text{FV}(A) = \emptyset \).
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 $\Sigma^{\text{sig}}$, $\Sigma'^{\text{sig}}$, and all the declarations in $\Sigma$ are present in $\Sigma'$, then we say $\Sigma \subseteq \Sigma'$.

That is, we have $\Sigma \subseteq \Sigma'$ if, for every $\Sigma_1$, $\Sigma_2$:

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

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

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

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

Lemma 2.69 (Signature weakening). Let $\Sigma$, $\Sigma'$ 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 $\Sigma \vdash \Gamma \text{ctx}$, then $\Sigma^{\text{sig}}$.

(ii) If $\Sigma;\Gamma \vdash h \Rightarrow A$, then $\Sigma;\Gamma \vdash A^{\text{type}}$.

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

(iv) If $\Sigma;\Gamma \vdash A \equiv B^{\text{type}}$, then $\Sigma;\Gamma \vdash A^{\text{type}}$ and $\Sigma;\Gamma \vdash B^{\text{type}}$.

(v) If $\Sigma;\Gamma \vdash t : A$, then $\Sigma;\Gamma \vdash A^{\text{type}}$.

(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 $\text{consts}(J) \subseteq \text{decls}(\Sigma)$, then $\Sigma \vdash J$. 

2.15. PROPERTIES

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

- If $a : A \in \Sigma$, 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:

$$\frac{\Sigma \vdash \Gamma \text{ ctx} \quad \alpha := t : A \in \Sigma}{\Sigma; \Gamma \vdash \alpha \equiv t : A} \text{ Delta-meta}_0$$

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 $\beta$-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 \( \Sigma;\Gamma \vdash \vec{e}_1 \equiv \vec{e}_2 : B \), and either \( \Sigma;\Gamma \vdash \vec{e}_1 : B' \) or \( \Sigma;\Gamma \vdash \vec{e}_2 : B' \). Then \( \Sigma;\Gamma \vdash \vec{e}_1 \equiv \vec{e}_2 : B' \).

**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 \( \Sigma;\Gamma', y : A, \Gamma'' : A' \vdash x : ( \bowtie x, A, \Gamma'' \downharpoonright A' ) \). Then \( \Sigma;\Gamma', y : A, \Gamma'' : A' \vdash J[ x \mapsto y ] \). (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 \( t \bowtie \vec{e} \downharpoonright t', u \bowtie \vec{e} \downharpoonright u' \), and \( \Sigma;\Gamma \vdash t' \equiv u' : T \).

In particular, by reflexivity, for every \( t \) such that \( \Sigma;\Gamma \vdash t : A \), we have \( t \bowtie \vec{e} \downharpoonright t' \) 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: \( \text{APP}_0 \), \( \text{APP-EQ}_0 \)). The following rules are admissible:

\[
\frac{\Sigma;\Gamma \vdash f : \Pi A B \quad \Sigma;\Gamma \vdash t : A}{\Sigma;\Gamma \vdash ft : B[t]} \quad \text{APP}_0
\]

\[
\frac{\Sigma;\Gamma \vdash f \equiv g : \Pi A B \quad \Sigma;\Gamma \vdash t \equiv u : A}{\Sigma;\Gamma \vdash ft \equiv gu : B[t]} \quad \text{APP-EQ}_0
\]

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

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 \( t \). \( \square \)

*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]. \( \square \)

*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]. \( \square \)

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

*Proof*. Using Postulate 3. See the proof of Lemma 2.84 in the licentiate thesis [54]. \( \square \)

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

*Proof*. Using Definition 2.32 and Postulate 7. See the proof of Lemma 2.85 in the licentiate thesis [54]. \( \square \)

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 \( \delta\eta \)-reduct)*. If \( \Sigma; \Gamma \vdash t : A \) and \( \Sigma; \Gamma \vdash \delta\eta u : A \), then \( \Sigma; \Gamma \vdash t \equiv \delta\eta u : A \).

*Proof*. By structural induction on the derivation of \( \Sigma; \Gamma \vdash t \rightarrow^* \delta\eta u : A \). \( \square \)

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 \( \Sigma; \Gamma \vdash t \rightarrow^* \delta\eta v : A \) and \( \Sigma; \Gamma \vdash u \rightarrow^* \delta\eta v : A \).

*Definition 2.87 (Full normal form: \( \Sigma; \Gamma \vdash \delta\eta t : A \)).* We say that a term \( t \) is in full normal form (written \( \Sigma; \Gamma \vdash \delta\eta t : A \)), if \( \Sigma; \Gamma \vdash t : A \) and there is no \( v \) such that \( \Sigma; \Gamma \vdash t \rightarrow^* \delta\eta v : A \).
Postulate 15 (Existence of a unique full normal form). If \( \Sigma; \Gamma \vdash t : A \), then there exists \( v \) such that \( \Sigma; \Gamma \vdash \triangledown t \xrightarrow{*}_{\delta\eta} : A \), and \( \Sigma; \Gamma \vdash v \xrightarrow{*}_{\delta\eta} : A \).

Remark 2.88 (Existence of a common normal form). Given \( \Sigma; \Gamma \vdash t \equiv u : A \), there exists \( v \) such that \( \Sigma; \Gamma \vdash t \xrightarrow{*}_{\delta\eta} : A \), \( \Sigma; \Gamma \vdash u \xrightarrow{*}_{\delta\eta} : A \), and \( \Sigma; \Gamma \vdash v \xrightarrow{*}_{\delta\eta} : A \).


Remark (Uniqueness of full normal form). Let \( v_1, v_2 \) be terms such that \( \Sigma; \Gamma \vdash v_1 \xrightarrow{*}_{\delta\eta} : A \) and \( \Sigma; \Gamma \vdash v_2 \xrightarrow{*}_{\delta\eta} : A \).

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

(ii) If there is \( t \) such that \( \Sigma; \Gamma \vdash t \xrightarrow{*}_{\delta\eta} v_1 : A \) and \( \Sigma; \Gamma \vdash t \xrightarrow{*}_{\delta\eta} v_2 : A \), 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 \( \Sigma; \Gamma \vdash T \equiv \text{Set} : \text{Set} \) and \( \Sigma; \Gamma \vdash T \equiv \Pi A_1 B_1 : \text{Set} \) and \( \Sigma; \Gamma \vdash T \equiv \Sigma A_2 B_2 : \text{Set} \) 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 \( \Sigma; \Gamma \vdash t \xrightarrow{n}_{\delta\eta} u : T \) and \( \Sigma \vdash \Gamma, T \equiv \Gamma', T' \text{ ctx} \), then \( \Sigma; \Gamma' \vdash t \xrightarrow{n}_{\delta\eta} u : T' \).

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 \( \lambda \)). If \( \Sigma; \Gamma \vdash \lambda.f \xrightarrow{n}_{\delta\eta} \lambda.g : T \), then there are \( A, B \) such that \( \Sigma; \Gamma, A \vdash f \xrightarrow{n}_{\delta\eta} g : B \) and \( \Sigma; \Gamma \vdash T \equiv \Pi A B \text{ type} \).

In fact, for any \( A', B' \) such that \( \Sigma; \Gamma \vdash T \equiv \Pi A'B' \text{ type} \), \( \Sigma; \Gamma, A' \vdash f \xrightarrow{n}_{\delta\eta} g : B' \).

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 \( \Sigma; \Gamma \vdash \langle f_1, f_2 \rangle \xrightarrow{n}_{\delta\eta} \langle g_1, g_2 \rangle : T \), then there are \( A, B \) such that \( \Sigma; \Gamma \vdash T \equiv \Pi \Sigma A B \text{ type} \), \( \Sigma; \Gamma \vdash f_1 \xrightarrow{m_1}_{\delta\eta} g_1 : A \) and \( \Sigma; \Gamma \vdash f_2 \xrightarrow{m_2}_{\delta\eta} g_2 : B[f_1] \), with \( m_1 + m_2 = n \).

In fact, for any \( A', B' \) such that \( \Sigma; \Gamma \vdash T \equiv \Pi A'B' \text{ type} \), there exist \( m'_1 \) and \( m'_2 \) such that \( \Sigma; \Gamma \vdash f_1 \xrightarrow{m'_1}_{\delta\eta} g_1 : A' \) and \( \Sigma; \Gamma \vdash f_2 \xrightarrow{m'_2}_{\delta\eta} g_2 : B'[f_2] \), with \( m'_1 + m'_2 = n \).

Proof. The proof is analogous to Remark 2.91 (inversion of reduction under \( \lambda \)).
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 \( t[y/x] \downarrow v \) for some \( v \), then \( t[(-1)+x][u[(-1)+x]/y] \downarrow_{u[(-1)+x]} \).

For all \( x, t \) and \( \vec{e} \), with \( x \notin \text{FV}(t) \) and \( x \notin \text{FV}(\vec{e}) \), if \( t[\vec{e}/x] \downarrow u \) for some \( u \), then \( t[(-1)+x][\vec{e}[(-1)+x]/x] \downarrow u[(-1)+x] \).

**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 \( \Sigma; \Gamma, A, \Delta \vdash t[\delta\eta] : B, |\Delta| \notin \text{FV}(t) \) and \( |\Delta| \notin \text{FV}(B) \), then \( \Sigma; \Gamma, \Delta[(-1)] \vdash t[(-1)+|\Delta|][\delta\eta] t'[(-1)+|\Delta|] : B[(-1)+|\Delta|] \).

**Proof.** By Remark 2.43 (free variables of \( \delta\eta \)-reduct), \( |\Delta| \notin \text{FV}(t[\delta\eta]) \subseteq \text{FV}(t) \).

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

### 2.16 Weak head normalization (\( \ \downarrow \ ))

In this section we define a special case of \( \delta\eta \)-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: \( \Sigma \vdash t[\downarrow \gamma] u \)). The relation \( \Sigma \vdash t[\downarrow \gamma] u \) (read as “\( u \) is the weak head normal form of \( t \) in signature \( \Sigma \)”)

**Remark 2.96** (WHNF reduction is deterministic). If \( \Sigma \vdash t[\downarrow \gamma] u_1 \), and \( \Sigma \vdash t[\downarrow \gamma] u_2 \), then \( u_1 = u_2 \).

**Proof.** By induction on the derivations, noting that given a signature \( \Sigma \) 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 \( \delta\eta \)-reduction). If \( \Sigma; \Gamma \vdash t : A \) and \( \Sigma \vdash t[\downarrow \gamma] u \), then \( \Sigma; \Gamma \vdash t[\delta\eta] u : A \).

**Proof.** By induction on the derivation for \( \Sigma \vdash t[\downarrow \gamma] u \).

**Lemma 2.98** (Equality of WHNF). If \( \Sigma; \Gamma \vdash t : A \) then there is \( u \) such that \( \Sigma \vdash t[\downarrow \gamma] u \), 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 \( \Sigma \vdash t[\downarrow \gamma] u \). Then, either:
Bool ↘ Bool
Σ𝐴𝐵 ↘ Σ𝐴𝐵
Π𝐴𝐵 ↘ Π𝐴𝐵
Set ↘ Set
c ↘ c
λ.𝑡 ↘ λ.𝑡
⟨𝑡, 𝑢⟩ ↘ ⟨𝑡, 𝑢⟩
𝑥  ⃗ 𝑒 ↘ 𝑥  ⃗ 𝑒
𝕒  ⃗ 𝑒 ↘ 𝕒  ⃗ 𝑒
𝛼  ⃗ 𝑒 ↘ 𝛼  ⃗ 𝑒
if α ≔ 𝑡 ∶ 𝐴 ∈ Σ and (𝑡 @ ⃗ 𝑒)⇓𝑢 and 𝑢↘𝑢'
if 𝛼 is uninstantiated in Σ
if 𝑛 ≤ 3
if 𝐴 𝑏 𝑡 𝑢  ⃗ 𝑒 ↘ 𝑡" if 𝑏 \true and (𝑡 @ ⃗ 𝑒)⇓𝑡′ and 𝑡′↘𝑡"
if 𝐴 𝑏 𝑡 𝑢  ⃗ 𝑒 ↘ 𝑢" if 𝑏 \false and (𝑢 @ ⃗ 𝑒)⇓𝑢′ and 𝑢′↘𝑢"
if 𝐴 𝑏 𝑡 𝑢  ⃗ 𝑒 ↘ if 𝐴’ 𝑡 𝑢  ⃗ 𝑒 if 𝑏 \𝑏’ and 𝑏’ \neq \true and 𝑏’ \neq \false

Figure 2.8: Inductive definition of the weak head normal form relation (Σ ⊢ _ ↘ _). The signature Σ is considered to be given implicitly.

- 𝑢 = Bool.
- 𝑢 = Σ𝐴𝐵 for some 𝐴, 𝐵.
- 𝑢 = Π𝐴𝐵 for some 𝐴, 𝐵.
- 𝑢 = Set.
- 𝑢 = c.
- 𝑢 = λ.𝑢’ for some term 𝑢’.
- 𝑢 = ⟨𝑢₁, 𝑢₂⟩ for some term 𝑢₁ and 𝑢₂.
- 𝑢 = 𝑥  ⃗ 𝑒 for some variable 𝑥, and vector of eliminators ⃗ 𝑒.
- 𝑢 = 𝛼  ⃗ 𝑒 for some atom 𝛼, and vector of eliminators ⃗ 𝑒.
- 𝑢 = 𝛼  ⃗ 𝑒 for some metavariable 𝛼 such that 𝛼 is not instantiated in Σ.
- 𝑢 = if 𝑒ⁿ, where 𝑛 ≤ 3.
- 𝑢 = if 𝐴 𝑏 𝑡 𝑢  ⃗ 𝑒, where neither Σ;Γ ⊢ 𝑏 \true : Bool nor Σ;Γ ⊢ 𝑏 \false : Bool.

Proof. By induction on the derivation for Σ ⊢ 𝑡 ↘ 𝑢, and the typing rules. □

Definition 2.100 (Head of a term: Set, Σ, Π, Bool, λ, 𝑡, 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, 𝑡, 𝑐, the recursor if or the pair constructor ⟨_, _, ⟩.
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 $\Sigma;\Gamma \vdash T : \text{Set}$.

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

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

**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 $\Sigma \vdash t \downarrow u$, then $\text{FV}(u) \subseteq \text{FV}(t)$.

**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: $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} \downarrow U)$). In a signature $\Sigma$ and context $\Gamma$, the elimination of the type of a head $h$ by a spine $\bar{e}$, resulting in a type $U$ (written $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} \downarrow U)$) is defined as follows:

- $\Sigma;\Gamma \vdash (h : \hat{\@} \varepsilon \downarrow T$ if $\Sigma;\Gamma \vdash h \Rightarrow T$
- $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} u \downarrow U$ if $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} \downarrow T$ and $\Sigma \vdash T \downarrow \Pi A B$ and $\Sigma;\Gamma \vdash u : A$ and $B[u] \downarrow U$
- $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} . \pi_1 \downarrow A$ if $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} \downarrow T$ and $\Sigma \vdash T \downarrow \Sigma A B$
- $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} . \pi_2 \downarrow B[h \bar{e}]$ if $\Sigma;\Gamma \vdash (h : \hat{\@} \bar{e} \downarrow T$ and $\Sigma \vdash T \downarrow \Sigma A B$

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

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

- $\Sigma;\Gamma \vdash T \hat{\@} \varepsilon \downarrow T$ if $\Sigma;\Gamma \vdash T \hat{\@} \bar{t} u \downarrow U$ if $\Sigma;\Gamma \vdash T \hat{\@} \bar{t} \downarrow T'$ and $\Sigma \vdash T' \downarrow \Pi A B$ and $\Sigma;\Gamma \vdash u : A$ and $B[u] \downarrow U$
Remark 2.105 (Type elimination without projections). If \( \Sigma; \Gamma \vdash h \Rightarrow T \), then \( \Sigma; \Gamma \vdash (h : : ) \hat{\circ} \vec{t} \downarrow T' \) if and only if \( \Sigma; \Gamma \vdash T \hat{\circ} \vec{t} \downarrow T' \).

The relation \( \hat{\circ} \) is consistent with the typing rules.

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

*Proof.* By induction on the derivation of \( \Sigma; \Gamma \vdash (h : ) \hat{\circ} \vec{e} \downarrow B \). Use Lemma 2.98 (equality of WHNF) and the \text{CONV} \rule.

**Lemma 2.107** (Type elimination inversion). Assume that \( \Sigma; \Gamma \vdash h \vec{e} : B \). Then \( \Sigma; \Gamma \vdash (h : ) \hat{\circ} \vec{e} \downarrow B' \) with \( \Sigma; \Gamma \vdash B' \equiv B \) \text{type}.

*Proof.* Using Definition 2.103, Lemma 2.101, Lemma 2.98, Postulate 10, 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 \( \Sigma; \Gamma \vdash t : A \), and \( \Sigma; \Gamma \vdash A \hat{\circ} \vec{u} \downarrow A' \). Then \( t \hat{\circ} \vec{u} \downarrow t' \), and \( \Sigma; \Gamma \vdash t' : A' \). Additionally, if \( \Sigma; \Gamma \vdash t_1 \equiv t_2 : A \), then \( t_1 \hat{\circ} \vec{u} \downarrow t'_1 \), \( t_2 \hat{\circ} \vec{u} \downarrow t'_2 \), 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 \( \Sigma; \Gamma \vdash A \hat{\circ} \vec{u} \downarrow A' \) 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 2.110** (Type of hereditary application). Assume \( \Sigma; \Gamma \vdash t : A \), and \( \Sigma; \Gamma \vdash A \hat{\circ} \vec{u} \downarrow A' \). Then there exist \( \vec{A} \) and \( \vec{B} \) such that \( \Sigma; \Gamma \vdash T \equiv \Pi \vec{A} \vec{B} \) \text{type} and \( \Sigma; \Gamma \vdash h \vec{u} : U \). Also, \( \Sigma; \Gamma \vdash A \equiv \Pi U V \) \text{type} and \( \Sigma; \Gamma \vdash h \vec{e} : \Pi U V \). Furthermore, if \( \Sigma; \Gamma \vdash h \vec{e} : \Pi U' V' \), then \( \Sigma; \Gamma \vdash U \equiv U' \) \text{type}, and \( \Sigma; \Gamma \vdash V \equiv V' \) \text{type}, and \( \Sigma; \Gamma \vdash u : U' \) \text{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 \( \Sigma; \Gamma \vdash T \equiv \Pi \vec{A} B \) \text{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 $\Sigma;\Gamma \vdash h \bar{e} . \pi_1 \bar{e}' : T$ or $\Sigma;\Gamma \vdash h \bar{e} . \pi_2 \bar{e}' : T$, then there are $A$, $U$ and $V$ such that $\Sigma;\Gamma \vdash (h :) @ \bar{e} \downarrow A$ and $\Sigma \vdash A \searrow \Sigma UV$. In particular, $\Sigma;\Gamma \vdash A \equiv \Sigma UV$ type, and $\Sigma;\Gamma \vdash h \bar{e} : \Sigma UV$.

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

Definition 2.114 (Type application, reversed: $\Sigma;\Gamma \vdash T \hat{\boxempty} \vec{u} \downarrow A$). Given a signature $\Sigma$ and context $\Gamma$, reverse elimination of a type $T$ by a spine $\vec{u}$, resulting in a type $A$ (written $T \hat{\boxempty} \vec{u} \downarrow A$) is defined as follows:

$$
\Sigma;\Gamma \vdash T \hat{\boxempty} R \varepsilon \downarrow T \\
\Sigma;\Gamma \vdash T \hat{\boxempty} R \vec{u} \downarrow T'
$$

if $\Sigma \vdash T \searrow \Pi AB$ and $\Sigma;\Gamma \vdash u : A$ and $B[u] \downarrow B'$ and $\Sigma;\Gamma \vdash B' \hat{\boxempty} R \vec{u} \downarrow T'$

Lemma 2.115 (Type application, reversed). We have $\Sigma;\Gamma \vdash T \hat{\boxempty} \vec{u} \downarrow A$ if and only if $\Sigma;\Gamma \vdash T \hat{\boxempty} R \vec{u} \downarrow A$.

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

Lemma 2.116 (Free variables in type application). If $\Sigma;\Gamma \vdash T \hat{\boxempty} \vec{u} \downarrow A$, or $\Sigma;\Gamma \vdash T \hat{\boxempty} R \vec{u} \downarrow A$ then $\text{fV}(T) \cup \text{fV}(u) \supseteq A$.

Proof. By Lemma 2.115 (type application, reversed), it suffices to show the property for the case $\Sigma;\Gamma \vdash T \hat{\boxempty} R \vec{u} \downarrow A$. We proceed by induction on the derivation of $\Sigma;\Gamma \vdash T \hat{\boxempty} R \vec{u} \downarrow A$, 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 $(t \hat{\boxempty} \bar{e}) \downarrow u$, then $(t^{\rho} \hat{\boxempty} \bar{e}^{\rho}) \downarrow u^{\rho}$.

Proof. By mutual induction on the derivations.

Lemma 2.118 (Commuting of renamings with WHNF). Let $\rho$ 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 renamings with reversed type application). The following hold:

(i) Assume $\Sigma;\Gamma' \vdash A$ type. If $\Sigma;\Gamma', \Gamma'' \vdash T \hat{\boxempty} R \vec{u} \downarrow U$, then $\Sigma;\Gamma', \Gamma''(\rho+1) \vdash T(\rho+1) + |\Gamma''| \hat{\boxempty} R \vec{u}(\rho+1) + |\Gamma''| \downarrow U(\rho+1) + |\Gamma''|$.

(ii) If $\Sigma;\Gamma', y : A, \Gamma''(\rho+1) \vdash T \hat{\boxempty} R \vec{u} \downarrow U$, with $\Sigma;\Gamma', y : A, \Gamma''(\rho+1) \vdash T(\rho+1) + |\Gamma''| \hat{\boxempty} R \vec{u}(\rho+1) + |\Gamma''| \downarrow U(\rho+1) + |\Gamma''|$, then $\Sigma;\Gamma', y : A, \Gamma''(\rho+1) \vdash T[x \mapsto y] \hat{\boxempty} R \vec{u}(x \mapsto y) \downarrow U[x \mapsto y]$.
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 \( \text{fv}(t) \subseteq \{ \vec{x} \} \). Then, \( \Sigma;\cdot \vdash \lambda \vec{y}^n.t[\vec{x} \mapsto \vec{y}] : A \) and \( (\lambda \vec{y}^n.t[\vec{x} \mapsto \vec{y}]) @ \vec{x}⇓t \).

**Proof.** We show the following (stronger) property:

For all \( \Delta' = T^y \) with \( y' = (|\Delta'| - 1), \ldots, 0 \), \( \Gamma = T^z \) with \( z = (|\Gamma| - 1), \ldots, 0 \), \( \{ \vec{x} \} \subseteq \{ \vec{z} \} \) with all variables in \( \vec{x} \) pairwise distinct, \( A', B' \) and \( t \), suppose:

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

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

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

(iv) \( \text{FV}(A') \subseteq \{ y' \} \) and

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

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

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 (\( \Theta \))

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.
2.18. METASUBSTITUTIONS

Definition 2.121. Metasubstitution: (Θ)

\[ Θ ::= \cdot \quad \text{empty metasubstitution} \]
\[ | Θ, 𝕒 : A \quad \text{atom} \]
\[ | Θ, α := t : A \quad \text{metavariable instantiation} \]

For a metasubstitution to be well-formed, we impose the additional condition that, even though the metasubstitution may contain atom declarations (e.g. 𝕒 : A) and also metavariable instantiations (e.g. α := 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 \text{wf}). A metasubstitution \( Θ \) is well-formed (written \( Θ \text{wf} \)) if it is well-formed as a signature, and none of the types and terms in it contain any metavariables:

\[ \frac{\text{EMPTY}}{Θ \text{wf}} \]

\[ \frac{\text{Θ \text{wf}}}{\text{Θ \text{wf}}} \]

\[ \frac{\text{Θ \text{wf}}}{\text{Θ \text{wf}}} \]

Remark 2.123 (Metasubstitutions are signatures). Given a metasubstitution \( Θ \), if \( Θ \text{wf} \) then \( Θ \text{sig} \).

Furthermore, if \( Θ \) is a metasubstitution such that, for all \( 𝕒 : A ∈ Θ \), \( \text{METAS}(A) = \emptyset \); and, for all \( α := t : A ∈ Θ \), \( \text{METAS}(t) \cup \text{METAS}(A) = \emptyset \); and \( Θ \text{sig} \), then \( Θ \text{wf} \).

(Note that, by Lemma 2.70 (piecewise well-formedness of typing judgments), if \( Θ; \cdot ⊢ t : A \) then \( Θ; \cdot ⊢ A \text{ type} \)).

Definition 2.124 (Metasubstitution subsumption: Θ \subseteq Θ'). We say \( Θ \subseteq Θ' \) if \( Θ \text{wf} \), \( Θ' \text{wf} \), and, when taking \( Θ \) and \( Θ' \) as signatures, \( Θ \subseteq Θ' \).

Definition 2.125 (Compatible metasubstitution: Θ \vdash Σ). We say that \( Θ \) is compatible with \( Σ \) (written \( Θ \vdash Σ \)) if \( Θ \text{wf} \), \( Σ \text{sig} \), \( \text{DECLS}(Θ) = \text{DECLS}(Σ) \), and, for every judgment \( J \), if \( Σ \vdash J \), then \( Θ \vdash J \).

Definition 2.126 (Declaration: (D)). Each of the elements of a signature is a declaration. If \( D \) is a declaration, then either \( D = 𝕒 : A \), \( D = α : A \) or \( D = α := t : A \).

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:

- \( D = 𝕒 : A \), and \( Θ; \cdot ⊢ 𝕒 : A \).
• \(D = \alpha : A\), 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 \(\Theta\), \(\Theta\) 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 \(\Theta_{\text{wf}}\) is compatible with a declaration \(D\) iff any of the following hold:

i) \(D = a : A\), and there is \(a : B \in \Theta\) 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 \(\Theta\) be a well-formed metasubstitution, and \(\Sigma\) be a well-formed signature. We have \(\Theta \vdash \Sigma\) if and only if \(\text{decls}(\Theta) = \text{decls}(\Sigma)\), and, for each declaration \(D \in \Sigma\), \(D\) is compatible with \(\Theta\).

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 \(\Sigma_{\text{sig}}\) be a well-formed extended signature, and \(\Theta_{\text{wf}}\) a well-formed metasubstitution such that \(\Theta \vdash \Sigma\). Let \(\Theta'\) be a metasubstitution such that \(\Theta'_{\text{wf}}\), and \(\Theta \subseteq \Theta'\). Then, for every \(D \in \Sigma\), \(\Theta'\) 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 \(\Theta\) to a set \(S\) (written \(\Theta_S\)) is a metasubstitution which assigns the same metavariable values as \(\Theta\), but only to those metavariables in \(S\).

\[
\begin{align*}
(\cdot)_S & = \cdot \\
(\Theta, a : A)_S & = \Theta_S, a : A \\
(\Theta, \alpha := t : A)_S & = \Theta_S, \alpha := t : A \quad \text{if} \ \alpha \in S \\
(\Theta, \alpha := t : A)_S & = \Theta_S \quad \text{otherwise}
\end{align*}
\]

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

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 \( \Theta \text{wf} \) and set of metavariables \( S \), \( \Theta_S \text{wf} \) and \( \Theta_S \subseteq \Theta \). In particular, for any signature \( \Sigma \), \( \Theta\Sigma \text{wf} \) 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 \( \Theta \) 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 \( \Sigma \), \( \text{AtomDecls}(\Theta) = \text{AtomDecls}(\Theta\Sigma) \) and \( \text{support}(\Theta\Sigma) = \text{support}(\Theta) \cap \text{support}(\Sigma) \).

Remark 2.136 (Nested metasubstitution restriction). Let \( \Theta \) 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 \( \Theta, \Theta' \) be metasubstitutions such that \( \Theta \subseteq \Theta' \), and \( J \) a judgment. (For instance, if \( \Theta = \Theta'_{S'} \).) If \( \Theta \vdash J \), then \( \Theta' \vdash J \).

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

Remark 2.138 (Metasubstitution strengthening). Assume \( \Theta \text{wf}, \Theta \subseteq \Theta' \).

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 (\( \text{CLOSE}(\Sigma) \))

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 \( \Sigma \) 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 \( \Theta \) from such a signature \( \Sigma \).

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

Definition 2.140 (Normalization to meta-free terms: \( \Sigma \vdash t \downarrow u \)). Given a closed signature \( \Sigma \) and a term \( \Sigma;\Gamma \vdash t : A \), we say that \( u \) is the metavariable-free normal form of term \( t \) (written \( \Sigma \vdash t \downarrow u \)) if \( u \) is the result of replacing all metavariables occurring in \( t \) by their bodies given in \( \Sigma \) (Figure 2.9).
\[ \Sigma \vdash \text{Bool} \quad \text{Bool} \]
\[ \Sigma \vdash \Pi A B \quad \Pi A' B' \quad \text{if} \quad \Sigma \vdash A \quad \text{and} \quad \Sigma \vdash B \]
\[ \Sigma \vdash \Sigma A B \quad \Sigma A' B' \quad \text{if} \quad \Sigma \vdash A \quad \text{and} \quad \Sigma \vdash B \]
\[ \Sigma \vdash \text{Set} \quad \text{Set} \]
\[ \Sigma \vdash \lambda.t \quad \lambda.t' \quad \text{if} \quad \Sigma \vdash t \quad \text{and} \quad \Sigma \vdash t' \]
\[ \Sigma \vdash \langle t, u \rangle \quad \langle t', u' \rangle \quad \text{if} \quad \Sigma \vdash t \quad \text{and} \quad \Sigma \vdash u \]
\[ \Sigma \vdash \alpha \vec{e}^n \quad v \quad \text{if} \quad \alpha := t : A \in \Sigma \quad \text{and} \quad t \downarrow \quad v' \quad \text{and} \quad \Sigma \vdash v' \]
\[ \Sigma \vdash h \vec{e}^n \quad h \vec{e}' \quad \text{if} \quad (h = x \quad \text{or} \quad h = \circ \quad \text{or} \quad h = \text{if}) \quad \text{and} \quad \forall i \in \{1, \ldots, n\}. \]
\[ e'_i := e_i = \pi_1 \]
\[ \text{or} \quad e'_i := e_i = \pi_2 \]
\[ \text{or} \quad (e_i = t_i \quad \text{and} \quad \Sigma \vdash t_i \quad \text{and} \quad e'_i := u_i) \]

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 \( \Sigma \), and a term \( \Sigma;\Gamma \vdash t : A \), there exists a unique term \( u \) such that \( \Sigma \vdash t \downarrow u \). For this \( u \), we have \( \text{metas}(u) = \emptyset \), \( \text{consts}(u) \subseteq \text{consts}(t) \), 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 \( \Sigma \text{ sig} \) 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 \( \Sigma \vdash t \downarrow A' \), \( \Sigma \vdash A \downarrow A' \), \( \text{metas}(A') = \emptyset \), \( \Sigma;\cdot \vdash A' \equiv A \text{ type} \), \( \Sigma;\cdot \vdash t' \equiv t : A' \). Also, by Remark 2.15 (there is only set), if \( \Sigma;\cdot \vdash A \text{ type} \), there is \( A' \) such that \( \Sigma \vdash A \downarrow A' \), \( \Sigma;\cdot \vdash A \text{ type} \), and \( \Sigma;\cdot \vdash A' \equiv A \text{ type} \).

Remark. If \( \text{metas}(t) = \emptyset \), then for any signature \( \Sigma \), \( \Sigma \vdash t \downarrow t \).

Definition 2.143 (Closing metasubstitution: \( \text{close}(\Sigma) \downarrow \Theta \)). Given a closed signature \( \Sigma \), whether a metasubstitution \( \Theta \) is a closing metasubstitution for \( \Sigma \) (written \( \text{close}(\Sigma) \downarrow \Theta \)) is inductively defined as follows:

\[
\begin{align*}
\text{close}(\cdot) & \downarrow \cdot, \\
\text{close}(\Sigma, a : A) & \downarrow (\Theta, a : A') \quad \text{if} \quad \text{close}(\Sigma) \downarrow \Theta \\
& \quad \text{and} \quad \Sigma \vdash A \downarrow A', \\
\text{close}(\Sigma, \alpha := t : A) & \downarrow (\Theta, \alpha := t' : A') \quad \text{if} \quad \text{close}(\Sigma) \downarrow \Theta \\
& \quad \text{and} \quad \Sigma \vdash t \downarrow A'.
\end{align*}
\]

Lemma 2.144 (Compatibility of closing metasubstitution). If \( \Sigma \text{ sig} \) is closed, then there is a syntactically unique metasubstitution \( \Theta \) such that \( \text{close}(\Sigma) \downarrow \Theta \), \( \Theta \text{ wf} \) and \( \Theta \vdash \Sigma \).

Proof. By induction on \( \Sigma \), 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 \( \Theta \) and \( \Theta' \) are equal (written \( \Theta \equiv \Theta' \)) if \( \Theta \text{ wf}, \Theta' \text{ wf}, \text{DECLS}(\Theta) = \text{DECLS}(\Theta') \), and each declaration in \( \Theta \) is judgmentally equal to a corresponding declaration in \( \Theta' \) (and vice versa).

More precisely, metasubstitution equality is the transitive closure of the following relation:

\[
\begin{align*}
\Theta_1, a : A, \Theta_2 & \equiv \Theta_1, a : A', \Theta_2 \quad \text{if} \quad \Theta_1;\cdot \vdash A \equiv A' \text{ type} \\
\Theta_1, \alpha := t : A, \Theta_2 & \equiv \Theta_1, \alpha := t : A', \Theta_2 \quad \text{if} \quad \Theta_1;\cdot \vdash A \equiv A' \text{ type} \\
\Theta_1, \alpha := t : A, \Theta_2 & \equiv \Theta_1, \alpha := t' : A, \Theta_2 \quad \text{if} \quad \Theta_1;\cdot \vdash t \equiv t' : A \\
\Theta & \equiv \Theta' \quad \text{if} \quad \Theta' \text{ is a well-formed reordering of } \Theta
\end{align*}
\]
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 \( \Sigma \) be a closed signature, and \( \Theta_1, \Theta_2 \) 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 \( \Theta^0, \Theta^1, \Theta \) such that \((\Theta^0, \Theta^1) \text{ wf}, (\Theta^0, \Theta) \text{ wf}, \Theta^0, \Theta^1 \vdash \Sigma \) and \( \text{close}(\Sigma) \downarrow (\Theta^0, \Theta) \), we have \( \Theta^0, \Theta^1 \equiv \Theta^0, \Theta \).

Note that, if \( \text{close}(\Sigma) \downarrow (\Theta^0, \Theta) \), then we have \( \Sigma = \Sigma^0, \Sigma' \) and \( \text{close}(\Sigma^0) \downarrow (\Theta^0) \).

We proceed by induction on the length of \( \Theta \), 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 \( \Sigma \) sig and \( \Sigma \) is closed, by Lemma 2.144 (compatibility of closing metasubstitution), there is \( \Theta \) such that \( \text{close}(\Sigma) \downarrow \Theta \). 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 \( \Sigma \) be a closed signature. Then \( \text{close}(\Sigma) \downarrow \Theta, \Theta \vdash \Sigma \), and, for any other \( \Theta' \) 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 \( \Theta^1 \Sigma \text{ wf}, (\Theta^2) \Sigma \text{ wf} \) and \( (\Theta^1) \Sigma \equiv (\Theta^2) \Sigma \).

Proof. By induction on the derivations of \( \Theta^1 \text{ wf}, \Theta^2 \text{ wf}, \Theta^1 \equiv \Theta^2 \), 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 \( \Sigma' \) is an extension of \( \Sigma \) (written \( \Sigma \sqsubseteq \Sigma' \)) if \( \Sigma' \) contains all the declarations in \( \Sigma \). The signature \( \Sigma' \) may instantiate some metavariables that are not already instantiated in \( \Sigma \), and/or replace some types and terms in \( \Sigma \).
by equal ones; but \( \Sigma' \) must not declare any new atoms (i.e. \( \text{ATOMDECLS}(\Sigma) = \text{ATOMDECLS}(\Sigma') \)). More formally:

**Definition 2.151** (Signature extension: \( \Sigma \sqsubseteq \Sigma' \)). Consider the signatures \( \Sigma \) and \( \Sigma' \). We say that \( \Sigma' \) extends \( \Sigma \) (written \( \Sigma' \sqsupseteq \Sigma \) or \( \Sigma \sqsubseteq \Sigma' \)), if \( \Sigma \text{ sig}, \Sigma' \text{ sig}, \) 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
\[
\Sigma_1 \sqsubseteq \Sigma_3 \\
\text{if } \Sigma_1 \sqsubseteq \Sigma_2 \text{ and } \Sigma_2 \sqsubseteq \Sigma_3
\]

### Normalization
\[
\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha : A', \Sigma_2 \\
\Sigma_1, \alpha := t : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := t : A', \Sigma_2 \\
\Sigma_1, \alpha := t : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := t' : A, \Sigma_2 \\
\text{if } \Sigma_1; \cdot \vdash A \equiv A' \text{ type} \\
\text{if } \Sigma_1; \cdot \vdash A \equiv A' \text{ type} \\
\text{if } \Sigma_1; \cdot \vdash t \equiv t' : A
\]

### Permutation
\[
\Sigma \sqsubseteq \Sigma' \\
\text{if } \Sigma' \text{ is a well-formed reordering of } \Sigma
\]

**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 \( \text{ATOMDECLS}(\Sigma) = \text{ATOMDECLS}(\Sigma') \) and \( \text{SUPPORT}(\Sigma) \subseteq \text{SUPPORT}(\Sigma') \).

**Remark 2.154** (Metasubstitution restriction to extension). If \( \Sigma \sqsubseteq \Sigma' \), then by Remark 2.153 (declarations in a signature extension), \( \text{SUPPORT}(\Sigma) \subseteq \text{SUPPORT}(\Sigma') \). 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 \( \Sigma = \Sigma_1, \Sigma_2, \Sigma \text{ sig}, \) and \( \Sigma_1' \) such that \( \Sigma_1 \sqsubseteq \Sigma_1' \) (in particular, \( \Sigma_1' \text{ sig} \)). Also, \( \text{DECLS}(\Sigma_2) \cap \text{DECLS}(\Sigma_2') = \emptyset \). (that is, all the new declarations in \( \Sigma_1' \) are fresh for \( \Sigma_2 \)). Then \( \Sigma_1, \Sigma_2 \sqsubseteq \Sigma_1', \Sigma_2 \) (in particular, \( \Sigma_1', \Sigma_2 \text{ sig} \)).

**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 \( \Theta \) be a metasubstitution, and \( \Sigma \) and \( \Sigma' \) signatures such that \( \Sigma \sqsubseteq \Sigma' \) and \( \Theta \vdash \Sigma' \). Then \( \Theta_{\Sigma} \text{ wf and } \Theta_{\Sigma} = \Sigma \).

**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 \( \bar{h} \bar{e} \) and \( \bar{h} \bar{e}' \) 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:

- \( x \bar{e} \).
- \( \alpha \bar{e} \).
- \( \bar{e}^n \), 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 \( \bar{h} \bar{e} \bar{e}' \) is strongly neutral, then \( \bar{h} \bar{e}_1 \) is also strongly neutral.

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

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

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

(iii) 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 \( \Sigma; \Gamma \vdash f_0 \rightarrow^{\delta_\eta} t : T \), and \( \Sigma; \Gamma \vdash t \rightarrow^{\delta_\eta} f_1 : T \), where \( f_0 \) is a strongly neutral term. Then \( t \) is a neutral term.

**Proof.** By Definition 2.41 (\( \delta_\eta \)-normalization step), there are only three possible cases for \( \Sigma; \Gamma \vdash f_0 \rightarrow^{\delta_\eta} t : T \):

\[
\begin{align*}
(\eta-\Pi) & \quad \Sigma; \Gamma \vdash f \rightarrow^{\delta_\eta} \lambda.f^{(+1)} 0 : T & \text{if } \Sigma; \Gamma \vdash T = \Pi A B \text{ type} \\
(\eta-\Sigma) & \quad \Sigma; \Gamma \vdash f \rightarrow^{\delta_\eta} \langle f .\pi_1, f .\pi_2 \rangle : T & \text{if } \Sigma; \Gamma \vdash T = \Sigma A B \text{ type} \\
(\text{APP}_n) & \quad \Sigma; \Gamma \vdash h \bar{e}^n -1 u \bar{e}' \rightarrow^{\delta_\eta} h \bar{e} v \bar{e}' : T & \text{if } \Sigma; \Gamma \vdash h \bar{e} : \Pi U V \\
& & \quad \text{and } \Sigma; \Gamma \vdash u \rightarrow^{\delta_\eta} v : U
\end{align*}
\]

However, if \( t \) is of the form \( \lambda .t_0 \) or \( \langle t_1, t_2 \rangle \), and \( \Sigma; \Gamma \vdash t \rightarrow^{\delta_\eta} u : T \), then necessarily \( u \) is of the form \( \lambda .u_0 \) or \( \langle u_1, u_2 \rangle \). But \( \Sigma; \Gamma \vdash t \rightarrow^{\delta_\eta} f_1 : T \), where \( f_1 \) is a neutral term. Therefore, we are necessarily in case \( \text{APP}_n \), where \( t \) is also a neutral term. \( \square \)
Remark 2.162 (Reduction preserves strongly neutral terms). If \( f \) is strongly neutral, and \( \Sigma;\Gamma \vdash f \rightarrow^{\delta_n} f' : T \) 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 \( f = h^1 \overrightarrow{e^n} \) and \( g = h^2 \overrightarrow{e^n} \), and \( \Sigma;\Gamma \vdash f \equiv g : T \). Then, \( h^1 = h^2 \), and for each \( i \in 1,\ldots,n \):

\[ \begin{align*}
&\text{If } e^1_i = t \text{ and } e^2_i = u, \text{ then there are } U \text{ and } V \text{ such that } \\
&\quad \Sigma;\Gamma \vdash h^1 \overrightarrow{e^1_{1\ldots,i-1}} \equiv h^2 \overrightarrow{e^2_{1\ldots,i-1}} : \Pi U V \text{ and } \Sigma;\Gamma \vdash t \equiv u : U.
\end{align*} \]

\[ \begin{align*}
&\text{If } e^1_i = e^2_i = \pi_1 \text{ or } e^1_i = e^2_i = \pi_2, \text{ then there are } U \text{ and } V \text{ such that } \\
&\quad \Sigma;\Gamma \vdash h^1 \overrightarrow{e^1_{1\ldots,i-1}} \equiv h^2 \overrightarrow{e^2_{1\ldots,i-1}} : \Sigma U V.
\end{align*} \]

Note that, by Lemma 2.75 (uniqueness of typing for neutrals) and Postulate 10 (injectivity of \( \Pi \)), the above hold for any \( U, V \) such that \( \Sigma;\Gamma \vdash h^1 \overrightarrow{e^1_{1\ldots,i-1}} \equiv h^2 \overrightarrow{e^2_{1\ldots,i-1}} : \Pi U V \) (first case) or \( \Sigma;\Gamma \vdash h^1 \overrightarrow{e^1_{1\ldots,i-1}} \equiv h^2 \overrightarrow{e^1_{1\ldots,i-1}} : \Sigma U V \) (second case).

Proof. By Postulate 14 (existence of a common reduct), there exists \( v \) such that \( \Sigma;\Gamma \vdash f \rightarrow^{M} v : T \) and \( \Sigma;\Gamma \vdash g \rightarrow^{N} v : T \). We show by induction on the sum of \( M \) and \( N \) that for all \( M, N \), and for all \( f, g \) such that \( \Sigma;\Gamma \vdash f \rightarrow^{M} v : T \) and \( \Sigma;\Gamma \vdash g \rightarrow^{N} v : T \), 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:

\[ \begin{align*}
&\text{• } x \overrightarrow{e}.
&\text{• } \overrightarrow{a} \overrightarrow{e}.
&\text{• if } e_1 e_2 \overrightarrow{e}, \text{ where } e_2 \text{ is a strongly neutral term.}
\end{align*} \]

Remark 2.165 (Extensions of irreducible terms). If \( f \) is an irreducible term, then so is \( f \overrightarrow{e} \) for any elimination spine \( \overrightarrow{e} \).

Lemma 2.166 (Reduction at \( \Pi \)-type). Assume \( \Sigma;\Gamma \vdash f : \Pi \overrightarrow{U^n} V \) for some \( \overrightarrow{U}, V \). Then there is \( v \) such that \( \Sigma;\Gamma \vdash \lambda \overrightarrow{z^n}.f \overrightarrow{z^n} \rightarrow^{\delta_n} \lambda \overrightarrow{z^n}.v : \Pi \overrightarrow{U^n} V \), and \( \Sigma;\Gamma \vdash \lambda \overrightarrow{z^n}.v \rightarrow^{\delta_n} \Pi \overrightarrow{U^n} V \).

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 $\Theta;\Gamma \vdash v \rightarrow_{\delta_\eta} : V$. Then $v$ is of the form $v_{nf}$ for some $v_{nf}$ generated by the following grammar:

$$t_{nf}, u_{nf}, v_{nf}, A_{nf}, B_{nf} ::= \text{Set}$$
$$\mid \text{Bool}$$
$$\mid \Pi A_{nf} B_{nf}$$
$$\mid \Sigma A_{nf} B_{nf}$$
$$\mid c$$
$$\mid \lambda t_{nf}$$
$$\mid \langle t_{nf}, u_{nf} \rangle$$
$$\mid h e_{nf} \quad \text{if } h e_{nf} \text{ is strongly neutral}$$

$$e_{nf} ::= t_{nf} | .\pi_1 | .\pi_2$$

Note that the converse does not hold; for instance:

$$\mathbb{A} : \text{Set}; x : \mathbb{A} \rightarrow \mathbb{A} \vdash x \rightarrow_{\delta_\eta} \lambda y. x \ y : \mathbb{A} \rightarrow \mathbb{A}$$

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. \qed

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, $\delta_\eta$-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:

(R-ID) $u[u]^0$
(R-\Pi_1) $(\Pi A B)[u]^n$ if $A[u]^n$
(R-\Pi_2) $(\Pi A B)[u]^{1+n}$ if $B[u]^n$
(R-\Sigma_1) $(\Sigma A B)[u]^n$ if $A[u]^n$
(R-\Sigma_2) $(\Sigma A B)[u]^{1+n}$ if $B[u]^n$
(R-\lambda) $(\lambda t)[u]^{1+n}$ if $t[u]^n$
(R-\langle,\rangle_1) $(t_1, t_2)[u]^n$ if $t_1[u]^n$
(R-\langle,\rangle_2) $(t_1, t_2)[u]^n$ if $t_2[u]^n$
(R-irred) $(f \ e)[f]^0$ if $f$ is an irreducible term
(R-strong) $(h \ e)[u]^n$ if $h \ e$ is strongly neutral

and there is $t \in \ e$ such that $t[u]^n$
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 \( \Delta \) (written \( \Sigma; \Gamma \vdash t[\Delta \vdash u : U] : T \)) if \( \Sigma; \Gamma \vdash t : T, \Sigma; \Gamma, \Delta \vdash u : U \), and \( \Sigma; \Gamma \vdash t[\Delta \vdash u : U] : T \), where the latter is defined as follows:

\[
\begin{align*}
\text{(TR-ID)} & \quad \Sigma; \Gamma \vdash u[\cdot \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash U \equiv T \type \\
\text{(TR-Π₁)} & \quad \Sigma; \Gamma \vdash (ΠA\!B)[\Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash A[\Delta \vdash u : U] : \Set \\
\text{(TR-Π₂)} & \quad \Sigma; \Gamma \vdash (ΠA\!B)[A', \Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash A' \equiv A \type \\
& \quad \text{and } \Sigma; \Gamma, A \vdash B[\Delta \vdash u : U] : \Set \\
\text{(TR-Σ₁)} & \quad \Sigma; \Gamma \vdash (ΣA\!B)[\Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash A[\Delta \vdash u : U] : \Set \\
\text{(TR-Σ₂)} & \quad \Sigma; \Gamma \vdash (ΣA\!B)[A', \Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash A' \equiv A \type \\
& \quad \text{and } \Sigma; \Gamma, A \vdash B[\Delta \vdash u : U] : \Set \\
\text{(TR-λ)} & \quad \Sigma; \Gamma \vdash (λ.t)[A, \Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash ΠA\!B \equiv T \type \\
& \quad \text{and } \Sigma; \Gamma, A \vdash t[\Delta \vdash u : U] : B \\
\text{(TR-⟨⟩₁)} & \quad \Sigma; \Gamma \vdash (⟨t₁, t₂⟩)[\Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash ΣA\!B \equiv T \type \\
& \quad \text{and } \Sigma; \Gamma, A \vdash t₁[\Delta \vdash u : U] : A \\
\text{(TR-⟨⟩₂)} & \quad \Sigma; \Gamma \vdash (⟨t₁, t₂⟩)[\Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash ΣA\!B \equiv T \type \\
& \quad \text{and } \Sigma; \Gamma \vdash t₂[\Delta \vdash u : U] : B[t₁] \\
\text{(TR-irred)} & \quad \Sigma; \Gamma \vdash (f \, \vec{e})[\cdot \vdash f : U]' : T & \text{if } f \text{ is an irreducible term} \\
\text{(TR-strong)} & \quad \Sigma; \Gamma \vdash (h \, \vec{e}_1 \, t \, \vec{e}_2)[\Delta \vdash u : U]' : T & \text{if } \Sigma; \Gamma \vdash h \, \vec{e}_1 : ΠA\!B \\
& \quad \text{and } \Sigma; \Gamma \vdash t[\Delta \vdash u : U] : A \\
& \quad \text{and } h \, \vec{e}_1 \, t \, \vec{e}_2 \text{ is strongly neutral}
\end{align*}
\]

**Lemma 2.170** (Typing of rigid occurrences). Suppose \( \Sigma; \Gamma \vdash t : T \) and \( t[\!u\!]^n \). Then there exist \( \Delta \) 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 (⟨⟩ inversion), or Lemma 2.111 (application inversion), respectively). \(\Box\)
Remark 2.171 (Free variables of rigid occurrence). If \( \Sigma; \Gamma \vdash t[\Delta \vdash u : U] : T \) then \( \text{FV}(u) - |\Delta| \subseteq \text{FV}(t) \).

Lemma 2.172 (Free variables in reduction of rigid occurrences). Let \( t \) and \( u \) be terms such that \( \Sigma; \Gamma \vdash t[\Delta \vdash u : U] : T \). If there is \( r \) such that \( \Sigma; \Gamma \vdash t \rightarrow r \) : \( T \), then there is \( v \) such that \( \Sigma; \Gamma, \Delta \vdash u \equiv v : U \), and \( \text{FV}(v) - |\Delta| \subseteq \text{FV}(r) \).

Proof. 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 \( \Sigma; \Gamma \vdash x \bar{e} \equiv t : U \), then \( x \in \text{FV}(t) \).

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 \( \bar{f} \) of irreducible neutral terms, a normal form term \( v_{nf} \), and a vector \( \bar{x} \) of variables fulfilling the hypothesis of Definition 2.34 (iterated hereditary substitution). Then we have \( v_{nf}[\bar{f}/\bar{x}] \downarrow u \) for some \( u \), and, for all \( i \in \{1, ..., n\} \), if \( x_i \in \text{FV}(v_{nf}) \), then there is \( m \) such that \( u[\bar{f}^i + m] \).

Proof. By Remark 2.36 (hereditary substitution by a neutral term), \( u \) exists. By induction on \( v_{nf} \) and Definition 2.31 (hereditary substitution), \( u[\bar{f}^i + m] \).

Lemma 2.175 (Preservation of irreducibles by normal forms). Suppose that \( \Theta; \Gamma, \bar{U} : V \), and \( \Theta; \Gamma, \bar{U} \vdash v \text{\( \rightarrow \delta_{\eta} \)} : V \). Let \( \bar{f}^n \) be a vector of irreducible terms, such that, for all \( i = 1, ..., n \), \( f_i^i = h_i^i e_i^i \), and, for some \( h, \Theta; \Gamma \vdash h : \Pi\bar{U}B \), \( \Theta; \Gamma \vdash h \bar{f} : B[\bar{f}] \). Take \( i \in \{1, ..., n\} \) such that \( x_i \in \text{FV}(v) \).

If \( \Theta; \Gamma \vdash v[\bar{f}] \equiv u : T \) for some \( u \) and type \( T \), and \( h_i = y \), then \( y \in \text{FV}(u) \).

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 \( \Theta; \Gamma, \bar{U} : V \), and \( \Theta; \Gamma, \bar{U} \vdash v \text{\( \rightarrow \delta_{\eta} \)} : V \).

Let \( \bar{f}^n \) and \( \bar{f}^n \) be two vectors of irreducible terms, such that, for all \( i = 1, ..., n \), \( f_i^i = h_i^i e_i^i \), and, for some \( h, \Theta; \Gamma \vdash h : \Pi\bar{U}B \), \( \Theta; \Gamma \vdash h \bar{f} : B[\bar{f}] \).

Let \( i \in \{1, ..., n\} \) such that \( x_i \in \text{FV}(v) \). If \( \Theta; \Gamma \vdash v[\bar{f}] \equiv v[\bar{f}'] : V[\bar{f}] \), then \( h_i = h_i' \).

Proof. Because \( \Theta; \Gamma, \bar{U} \vdash v \text{\( \rightarrow \delta_{\eta} \)} : V \), by Lemma 2.167 (characterization of normal forms), \( v \) is of the form \( v_{nf} \) for some \( v_{af} \).

Let \( \bar{x}^n = n - 1, ..., 0 \). It suffices to show the following (stronger) property, taking \( \Delta = \cdot \):

For all \( \Delta \) and \( u_{nf} \) if \( x_i^{\Delta} \in \text{FV}(u_{nf}) \) and for some \( T \), \( \Theta; \Gamma, \Delta \vdash u_{nf}[\bar{f}^{\Delta}/\bar{x}^{\Delta}] : T \), 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].
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 $\Sigma$ and context $\Gamma$, such that $\Sigma \vdash \Gamma$ ctx.

We say that $t$ is a term with holes in signature $\Sigma$ and context $\Gamma$ if, for each metavariable $\alpha$ occurring in $t$ ($\alpha \in \text{METAS}(t)$), either:

(a) $\alpha \in \text{DECLS}(\Sigma)$, or

(b) $\alpha$ 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 “$\text{HOLES}(_{-}, _{-}, _{-}) \Downarrow -$” as in Figure 3.1, we say that a term $t$ is a term with holes in signature $\Sigma$ and context $\Gamma$ if (HOLES($\Sigma$, $\Gamma$, $t$) $\Downarrow$ $H$) and METAS($t$) $\subseteq$ SUPPORT($\Sigma$) $\cup$ $H$. 

63
\[ \text{HOLES}(\Sigma, n, \text{Set}) \downarrow \emptyset \]
\[ \text{HOLES}(\Sigma, n, \text{Bool}) \downarrow \emptyset \]
\[ \text{HOLES}(\Sigma, n, \Pi A B) \downarrow (H_1 \cup H_2) \quad \text{if} \quad \text{HOLES}(\Sigma, n, A) \downarrow H_1 \]
\[ \text{and} \quad \text{HOLES}(\Sigma, n + 1, B) \downarrow H_2 \]
\[ \text{and} \quad H_1 \cap H_2 = \emptyset \]
\[ \text{HOLES}(\Sigma, n, \Sigma A B) \downarrow (H_1 \cup H_2) \quad \text{if} \quad \text{HOLES}(\Sigma, n, A) \downarrow H_1 \]
\[ \text{and} \quad \text{HOLES}(\Sigma, n + 1, B) \downarrow H_2 \]
\[ \text{and} \quad H_1 \cap H_2 = \emptyset \]
\[ \text{HOLES}(\Sigma, n, c) \downarrow \emptyset \]
\[ \text{HOLES}(\Sigma, n, \lambda.t) \downarrow H \quad \text{if} \quad \text{HOLES}(\Sigma, n + 1, t) \downarrow H \]
\[ \text{HOLES}(\Sigma, n, (t_1, t_2)) \downarrow (H_1 \cup H_2) \quad \text{if} \quad \text{HOLES}(\Sigma, n, t_1) \downarrow H_1 \]
\[ \text{and} \quad \text{HOLES}(\Sigma, n, t_2) \downarrow H_2 \]
\[ \text{and} \quad H_1 \cap H_2 = \emptyset \]
\[ \text{HOLES}(\Sigma, n, \alpha \,(n-1) \,(n-2) \,\ldots \,0) \downarrow \{\alpha\} \quad \text{if} \quad \alpha \notin \text{DECLS}(\Sigma) \]
\[ \text{HOLES}(\Sigma, n, h \,\overline{e}^m) \downarrow (\bigcup_{i=1}^{m} H_i) \quad \text{if} \quad \forall i \in \{1, \ldots, m\}. \text{HOLES}(\Sigma, n, e_i) \downarrow H_i \]
\[ \text{and} \quad \forall i, j. i \neq j. H_i \cap H_j = \emptyset \]
\[ \text{and} \quad (h = \alpha, \alpha \in \text{DECLS}(\Sigma)) \]
\[ \text{or} \quad h = x \,\text{or} \quad h = a \,\text{or} \quad h = \text{if} \]

\[ \text{HOLES}(\Sigma, n, .\pi_1) \downarrow \emptyset \]
\[ \text{HOLES}(\Sigma, n, .\pi_2) \downarrow \emptyset \]

\[ \text{Figure 3.1: Recursive definition of the set of holes in a term} \]
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 \( \Sigma \) and context \( \Gamma \), 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 \( \Sigma \) and context \( \Gamma \). Then \( \Sigma;\Gamma \vdash t : A \) is a well-formed type checking problem.

Definition 3.3 (Solution to a type checking problem: \( \Theta \models \Sigma;\Gamma \vdash t : A \)). We say that a metasubstitution \( \Theta \) is a solution to the type checking problem \( \Sigma;\Gamma \vdash t : A \) (written \( \Theta \models \Sigma;\Gamma \vdash t : A \)), if \( \Theta \) wf, \( \Theta \Sigma \models \Sigma \), \( \text{decls}(\Theta) = \text{decls}(\Sigma) \cup \text{holes}(\Sigma,|\Gamma|,t) \), 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 \( \Theta \) such that i) \( \Theta \models \Sigma;\Gamma \vdash t : A \), and ii) for any other metasubstitution \( \Theta' \) such that \( \Theta' \models \Sigma;\Gamma \vdash t : A \), 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 \( \Sigma = \Aleph : \text{Set}, c : (A : \text{Set}) \to A \to A \). We have \( \Sigma \) sig, \( \Sigma \vdash \cdot \) ctx, and \( \Sigma;\cdot \vdash A \) type.

Take \( t = \lambda x. c (\alpha A) x \). The metavariable \( \alpha \) does not occur in the signature \( \Sigma \), and is applied to all the variables in scope (here, only \( x \), because \( \Gamma \) is empty). By Definition 3.1 (term with holes), \( t \) is a term with holes in signature \( \Sigma \) and context \( \Gamma \), and the following is a well-formed type checking problem:

\[
\Sigma;\cdot \vdash^? \lambda x. c (\alpha x) x : A \to A
\]

If we take \( \Theta = \Aleph : \text{Set}, c : (A : \text{Set}) \to A \to A, \alpha := \lambda x. A : A \to \text{Set} \), then \( \Theta \Sigma \models \Sigma \), and \( \Theta;\cdot \vdash t : A \to A \). Therefore, \( \Theta \) is a solution to the type checking problem \( \Sigma;\cdot \vdash^? t : A \to A \).

Let \( \Theta' \) be another solution to the given problem. Then:

(i) \( \Theta' \models \Sigma, \) therefore \( \Theta';\cdot \vdash A : \text{Set} \) and \( c : (A : \text{Set}) \to A \to A \).

(ii) By Lemma 2.82 (\( \lambda \) inversion), Lemma 2.56 (neutral inversion), and Lemma 2.52 (\( \Pi \) inversion), \( \Theta' ; \cdot \vdash A \to A \) and \( \Theta' ; \cdot \vdash \alpha x \equiv \alpha A : A \to \text{Set} \).

By Lemma 4.35 (Miller’s pattern condition), \( \Theta' ; \cdot \vdash A \equiv \lambda x. A : A \to \text{Set} \).

By Lemma 2.130 (alternative characterization of a compatible metasubstitution), \( \Theta' \equiv \Theta \). By Lemma 2.148 (uniqueness of closing metasubstitution), \( \Theta' \equiv \Theta \). Therefore, \( \Theta \) 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:

\[
\Sigma;\cdot \vdash^? \lambda x. c (\alpha A) x : A \to A
\]

Both \( \Theta_1 \) and \( \Theta_2 \) (below) are solutions:
\[ \Theta_1 = \mathbb{A} : \text{Set}, c : (A : \text{Set}) \to A \to A, \alpha := \lambda x.\mathbb{A} : \mathbb{A} \to \text{Set} \]
\[ \Theta_2 = \mathbb{A} : \text{Set}, c : (A : \text{Set}) \to A \to A, \alpha := \lambda x.x : \mathbb{A} \to \text{Set} \]

However, by postulate Postulate 14 (existence of a common reduct), 
\[ [\Theta_1 ; \vdash \alpha \neq \lambda x.x : \text{Set}] \] thus \[ [\Theta_1 \neq \Theta_2] \]. By Lemma 2.147 (compatibility respects equality) this means \[ [\Theta_1 \neq \Theta_2] \]. Therefore neither \( \Theta_1 \) nor \( \Theta_2 \) 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 \( \Sigma \) (written \( \Sigma;\Gamma \vdash t : A \cong u : B \text{ wf} \)) when each of the two sides is well-typed.

\[
\frac{\Sigma;\Gamma \vdash t : A}{\Sigma;\Gamma \vdash u : B}
\]

Basic constraints thus defined are heterogeneous: each side may have a different type.

**Definition 3.8** (Solution to a basic constraint: \( \Theta \models \Sigma;\Gamma \vdash t : A \cong u : B \)). A metasubstitution \( \Theta \) is a solution to a well-formed basic constraint \( \Sigma;\Gamma \vdash t : A \cong u : B \text{ wf} \) (written \( \Theta \models \Sigma;\Gamma \vdash t : A \cong u : B \)) if \( \Theta \models \Sigma, \Theta;\Gamma \vdash A \equiv B : \text{Set} \) 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 \( \Sigma \) and a set of basic constraints of the form \( \Sigma;\Gamma_i \vdash t_i : A_i \cong u_i : B_i \) \((i \in \{1,\ldots,n\})\), is there a metasubstitution \( \Theta \) such that \( \Theta \models \Sigma \), and for each \( i \in \{1,\ldots,n\} \), \( \Theta \) 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 \( \Sigma' \), a term \( u \), and a set of basic constraints \( \vec{C} \).
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 $\Sigma'$, a term $u$ and a set of basic constraints $\vec{C}$, such that $\Sigma \subseteq \Sigma'$, $\text{ATOMDECLS}(\Sigma) = \text{ATOMDECLS}(\Sigma')$, $\text{SUPPORT}(\Sigma') \supseteq \text{METAS}(t)$, $\Sigma'; \Gamma \vdash u : A$, and each of the basic constraints $\vec{C} \in \vec{C}$ 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 $\Sigma'$ is the signature produced by the elaboration algorithm, $u$ the term, and $\vec{C}$ the basic constraints, then the following hold:

**Soundness** Let $\Theta$ be such that $\Theta \models \Sigma'$ and $\Theta \models \vec{C}$. Then $\Theta_{\Sigma', t} \vdash \Sigma; \Gamma \vdash t : A$, and $\Theta; \Gamma \vdash t \equiv u : A$.

**Completeness** Let $\Theta$ be a metasubstitution such that $\Theta \models \Sigma; \Gamma \vdash t : A$. Then there is a metasubstitution $\tilde{\Theta}$ such that $\tilde{\Theta}_{\Sigma', t} = \Theta$, $\tilde{\Theta} \models \Sigma'$, and $\tilde{\Theta} \models \vec{C}$.

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 $\Sigma'$ extending $\Sigma$ 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 $\Sigma \vdash p : A \rightarrow_{\delta} t$ when $\bar{S}$, where $\bar{S}$ 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 $\Sigma \vdash p : A \rightarrow_{\delta} t$ when $\bar{S}$, and a signature $\Sigma'$ such that $\Sigma'$ extends $\Sigma$ and $\Sigma'$ solves $\bar{S}$, we have that $\Sigma'; \vdash t : A$. 
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 $\Sigma'$ solving $\delta$ is equivalent with $\Sigma';\cdot \vdash A \equiv B$ type. Therefore, we can understand a blocked constant as the combination of a metavariable $\alpha$ 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 $\Sigma$ is thus extended into a signature $\Sigma'$, 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 $\lambda$-terms, where equality is given by the $\eta$ and $\beta$-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 $\Sigma$ when $\Sigma;\Gamma \vdash t \equiv u : T$.

Unification is first-order when the types of metavariables are all first-order (e.g. $\alpha : A_1 \to \ldots \to A_n$, with all $A_i$ 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:

$$\alpha : \text{Set}, \mathcal{F} : \text{Set} \to \text{Set}, \varnothing : \text{Set}; \cdot \vdash \mathcal{F} \alpha \approx \mathcal{F} \varnothing : \text{Set}$$

The problem has the solution $\Theta \equiv \mathcal{F} : \text{Set} \to \text{Set}, \varnothing : \text{Set}, \alpha := \varnothing : \text{Set}$. Indeed, replacing all occurrences of $\alpha$ by $\varnothing$ in $\mathcal{F} \alpha$ gives $\mathcal{F} \varnothing$.

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
3.4. (UN)DECIDABILITY OF HIGHER ORDER UNIFICATION

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 $\beta$-reduction.

Example 3.16 (Higher-order problem). Consider the following higher-order problem:

$$\alpha : \text{Set} \to \text{Set} \to \text{Set}, \ F : \text{Set} \to \text{Set}; \ x, y : \text{Set} \vdash \alpha \ x \ y \ \approx \ F \ x : \text{Set}$$

Note that $\alpha$ is of function type, and in fact occurs at the head of a term ($\alpha \ x \ y$). The problem has the solution $\Theta = F : \text{Set} \to \text{Set}, \alpha := \lambda x. \lambda y. F \ x : \text{Set}$. Finding this solution requires accounting for the fact that substituting $\lambda x. \lambda y. F \ x$ for $\alpha$ is not a purely syntactic substitution, which would give the (ungrammatical) term $[(\lambda x. \lambda y. \ F \ x) \ x \ y]$; but instead also involves a computation step (resulting in $F \ x$).

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 $\{x, y\}$, consider the corresponding higher-order unification problem:

$$\mathbb{A} : \text{Set},$$

$$\alpha : (\mathbb{A} \to \mathbb{A}) \to \ldots \to (\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A}),$$

$$\beta : (\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A}),$$

$$x, y : \mathbb{A} \to \mathbb{A} \vdash \alpha \ a_1 \ldots \ a_n \approx \alpha \ b_1 \ldots \ b_n : \mathbb{A}$$

$$x : \mathbb{A} \to \mathbb{A} \vdash \alpha \ x \ldots \ x \approx \lambda z. \ x (\beta \ x \ z) : \mathbb{A} \to \mathbb{A}$$

The tilde $\sim$ over a word denotes its encoding as a list of variables. For example, the encoding of the word $x y x$ is $x \ y \ x = \lambda z. (x \ (y \ (x \ z)))$.

By a combinatorial argument, because of the type of metavariable $\alpha$, in all well-typed solutions to the problem, the body of $\alpha$, when fully $\eta$-expanded, is of the form $\lambda w_1 \ldots \lambda w_n \ldots \lambda z. \ w_{i_1} \ldots (w_{i_p} \ z)).$ 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 cor-
responds 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 \( \beta \) and the second equation,
the problem always has at least one solution, namely, \( \Theta \equiv \text{Set, } \alpha := \lambda w_1 \ldots \lambda w_n \lambda z : (\text{A} \to \text{A}) \to \ldots \to (\text{A} \to \text{A}) \to \text{A} \to \text{A} \). 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 fol-
lowing higher-order unification problem:

\[
\text{A} : \text{Set}, \quad \alpha : (\text{A} \to \text{A}) \to (\text{A} \to \text{A}) \to \text{A} \to \text{A}; \\
u : \text{A} \to \text{A}, v : \text{A} \to \text{A} \vdash \alpha u v \approx \lambda z. v (v (u z))
\]

This problem has the following solution:

\[
\Theta = \text{Set, } \alpha := (\lambda u. \lambda v. \lambda s. v (v (u s))) : (\text{A} \to \text{A}) \to (\text{A} \to \text{A}) \to \text{A} \to \text{A}
\]

Note that all the metavariables in Example 3.17 are applied only to dis-
tinct 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 \( \alpha \), \((\beta x y)\), is not a variable.

\[
\text{A} : \text{Set, } \alpha : \text{A} \to \text{A} \to \text{A}, \beta : \text{A} \to \text{A} \to \text{A}; \\
x, y : \text{A} \vdash \alpha (\beta x y) y \approx y : \text{A} \\
x, y : \text{A} \vdash \beta x y \approx x : \text{A}
\]

However, the second constraint is in the pattern fragment. This means that,
in any solution to the problem, \( \beta \) is necessarily instantiated to the term \( \lambda x. \lambda y. x \)
(or a term judgmentally equal to said term). Following this assignment, the
3.7 Extension to product types

So far we have discussed unification for the simply typed $\lambda$-calculus. The pattern fragment can be extended to accommodate terms with product types ($\times$), including pairs $\langle t, u \rangle$, projections $\langle \pi_1, \pi_2 \rangle$ and the corresponding $\eta$-equality.

For example, the following problem is not in the pattern fragment, because the arguments are projected variables:

\[
\begin{align*}
\mathbb{A} &: \text{Set}, \mathbb{B} &: \text{Set}, \\
\alpha &: \mathbb{A} \to \mathbb{B} \to \mathbb{B}; \\
x &: \mathbb{A} \times \mathbb{B} \vdash \alpha (x \cdot \pi_1) (x \cdot \pi_2) \approx (x \cdot \pi_2) : \mathbb{B}
\end{align*}
\]

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 $x = \langle x_1, x_2 \rangle$.

\[
\begin{align*}
\mathbb{A} &: \text{Set}, \mathbb{B} &: \text{Set}, \\
\alpha &: \mathbb{A} \to \mathbb{B} \to \mathbb{B}; \\
x_1 &: \mathbb{A}, x_2 &: \mathbb{B} \vdash \alpha x_1 x_2 \approx x_2 : \mathbb{B}
\end{align*}
\]

Similarly, the following problem is also not in the pattern fragment, because the argument is not a single variable, but a pair:

\[
\begin{align*}
\mathbb{A} &: \text{Set}, \mathbb{B} &: \text{Set}, \\
\alpha &: \mathbb{A} \times \mathbb{B} \to \mathbb{B}; \\
x &: \mathbb{A}, y &: \mathbb{B} \vdash \alpha \langle x, y \rangle \approx y : \mathbb{B}
\end{align*}
\]

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 \( \alpha \). The problem then becomes the following, which is in the pattern fragment:

\[
\begin{align*}
&\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \\
&\alpha' : \mathbb{A} \to \mathbb{B} \to \mathbb{B}, \\
&\alpha := \lambda x.\alpha' (x .\pi_1) (x .\pi_2) : \mathbb{A} \times \mathbb{B} \to \mathbb{B}; \\
x : \mathbb{A}, y : \mathbb{B} \vdash \alpha' x y \approx y : \mathbb{B}
\end{align*}
\]

Because of \( \eta \)-equality, a term of record type is determined by its projections, therefore the set of possible solutions for \( \alpha \) 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 \( \Theta, \Theta ;\cdot \vdash \alpha' \equiv \lambda x.\lambda y.y : \mathbb{A} \to \mathbb{B} \to \mathbb{B} \) (see Lemma 4.35, Miller’s pattern condition). In fact, the new problem has the following unique solution:

\[
\Theta \equiv \mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha := \lambda x.(x .\pi_2) : \mathbb{A} \times \mathbb{B} \to \mathbb{B}, \alpha' := \lambda x.\lambda y.y : \mathbb{A} \to \mathbb{B} \to \mathbb{B}
\]

Therefore the original problem has the unique solution \( \Theta_{\{\alpha}\} = (\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha := \lambda x. x.\pi_2 : \mathbb{A} \times \mathbb{B} \to \mathbb{B}) \).

Abel and Pientka [2] extensively elaborate on this insight to extend dynamic pattern unification for a theory containing both dependent function (\( \Pi \)) and record (\( \Sigma \)) 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.
3.9 Strictly ordered, homogeneous constraints

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 \( \Pi \)-types: \( \Pi(x : A)B \) and \( \Pi(x : A')B' \). In order to obtain constraints that can be solved, we may want to simplify \( \leadsto \) the given constraint into two new constraints, one that unifies \( A \) and \( A' \), and another which unifies \( B \) and \( B' \):

\[
\Sigma; \Gamma \vdash \Pi(x : A)B : \text{Set} \leadsto \Pi(x : A')B' : \text{Set}
\]

\[
\Sigma; \Gamma \vdash A : \text{Set} \leadsto A' : \text{Set} \land \\
\Gamma, x : [?] \vdash B \equiv B' : \text{Set}
\]

The question is, in the second constraint, what the type \([?]\) 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 \( \Sigma; \Gamma \vdash t \equiv u : A \). 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 \( \Sigma; \Gamma \vdash t : A \equiv u : B \) is translated into two homogeneous internal constraints; namely \( \Sigma; \Gamma \vdash A \equiv B : \text{Set} \) and \( \Sigma; \Gamma \vdash t \equiv u : A \). 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 \( \Sigma' \supseteq \Sigma \) is found such that \( \Sigma' ; \Gamma \vdash A \equiv B : \text{Set} \), then the second constraint will be well-formed in this extended signature: \( \Sigma' ; \Gamma \vdash t \equiv u : A \), 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 \( \alpha : \text{Set} \rightarrow \text{Set} \) and \( \beta : \alpha \text{Bool} \), consider the constraint \( x : \text{Set} \vdash \langle \alpha x, \text{true} \rangle : \text{Set} \times \text{Bool} \equiv \langle \text{Bool}, \beta \rangle : \text{Set} \times \alpha \text{Bool} \). This constraint has the unique solution \( \alpha := \lambda x. \text{Bool} : \text{Set}, \beta := \text{true} : \text{Bool} \).
A strictly ordered approach based on homogeneous constraints would first solve $x : \text{Set} \vdash \text{Set} \times \text{Bool} \approx \text{Set} \times \alpha \text{Bool} : \text{Set}$, and once it is solved (and only then), attempt $x : \text{Set} \vdash (\alpha x, \text{true}) \approx (\text{Bool}, \beta) : \text{Set} \times \text{Bool}$. However, the first constraint has two possible, incompatible solutions: $\alpha := \lambda x.\text{Bool} : \text{Set}$ and $\alpha := \lambda x.x : \text{Set}$.

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.

$\Gamma ::= \cdot$ empty twin context

| $\Gamma, x : A$ variable of simple type
| $\Gamma, \hat{x} : A_1\ddagger A_2$ variable of twin type

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:

\[
\begin{align*}
h & ::= \ldots \text{neutral heads} \\
& \mid \hat{x} \quad \text{left twin variable} \\
& \mid \breve{x} \quad \text{right twin variable}
\end{align*}
\]

The rule \textsc{var} is replaced by the following two rules:

\[
\frac{\Gamma, x : A_1\ddagger A', \Delta \vdash \hat{x} \Rightarrow A}{\Gamma, \hat{x} : A_1\ddagger A', \Delta \vdash \hat{x} \Rightarrow A} \quad \text{VAR-LEFT}
\]

\[
\frac{\Gamma, x : A_1\ddagger A', \Delta \vdash \breve{x} \Rightarrow A'}{\Gamma, \breve{x} : A_1\ddagger A', \Delta \vdash \breve{x} \Rightarrow A'} \quad \text{VAR-RIGHT}
\]

In this new setting, the basic constraint from Problem 3.18 is simplified in this manner, where the type $B[x \mapsto \hat{x}]$ (respectively $B'[x \mapsto \breve{x}]$) is the result of syntactically replacing each occurrence of $x$ in $B$ by $\hat{x}$ (respectively, each occurrence of $x$ in $B'$ by $\breve{x}$):
Σ;Γ ⊢ Π(𝑥 ∶ 𝐴)𝐵 ∶ Set ≅ Π(𝑥 ∶ 𝐴′)𝐵′ ∶ Set
Σ;Γ ⊢ 𝐴 ∶ Set ≅ 𝑢 ∶ 𝐴′ ∶ Set ∧
Σ;Γ, 𝑥 ∶ 𝐴‡𝐴 ′ ⊢ 𝐵[𝑥 ↦ ́ 𝑥] ∶ Set ≅ 𝐵′[𝑥 ↦ ̀ 𝑥] ∶ Set

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. Excessi-
vie 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). How-
ever, 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 \( \Gamma_1 \ddagger \Gamma_2 \) is a pair of contexts \( \Gamma_1 \) and \( \Gamma_2 \), such that \( |\Gamma_1| = |\Gamma_2| \).

A twin context is well-formed if each of the sides \( \Gamma_1 \) and \( \Gamma_2 \) are well-formed. The precondition \( |\Gamma_1| = |\Gamma_2| \) follows from the use of the syntax \( \Gamma_1 \ddagger \Gamma_2 \). For
the sake of clarity, we reiterate it in the derivation rule.

\[
\begin{align*}
\Sigma \vdash \Gamma_1 \text{ ctx} & \quad \Sigma \vdash \Gamma_2 \text{ ctx} \quad (|\Gamma_1| = |\Gamma_2|) \\
\Sigma \vdash \Gamma_1 \ddagger \Gamma_2 \text{ wf}
\end{align*}
\]

Notation (Twin context). Given a well-formed twin context \( \Gamma_1 \ddagger \Gamma_2 \), it can also
be viewed as a context where each variable has two types:

\[
\begin{align*}
\Gamma_1 \ddagger \Gamma_2 & \quad ::= \quad \text{empty twin context} \\
& \quad | \quad \Gamma_1 \ddagger A_1, \Gamma_2 \ddagger A_2 \quad \text{variable of twin type}
\end{align*}
\]

Twin contexts can be concatenated by concatenating each of their sides.

Notation (Twin context concatenation). The concatenation of two twin-
contexts \( \Gamma_1 \ddagger \Gamma_2 \) and \( \Delta_1 \ddagger \Delta_2 \) is written \( (\Gamma_1 \ddagger \Gamma_2), (\Delta_1 \ddagger \Delta_2) \), and corresponds
to the twin context \( (\Gamma_1, \Delta_1) \ddagger (\Gamma_2, \Delta_2) \). Writing \( (\Gamma_1 \ddagger \Gamma_2), (\Delta_1 \ddagger \Delta_2) \) instead of
\( (\Gamma_1, \Delta_1) \ddagger (\Gamma_2, \Delta_2) \) 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
\( \Gamma_1 \ddagger \Gamma_2 \), two terms \( t \) and \( u \), and two types \( A \) and \( B \), an internal constraint is a
5-tuple of the form \( \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B \).

An internal constraint \( \mathcal{C} \) is well-formed in a signature \( \Sigma \) (written \( \Sigma; \mathcal{C} \text{ wf} \))
if and only if each side of the constraint is well-typed in the corresponding side
of the context.

\[
\begin{align*}
\Sigma \vdash \Gamma_1 \ddagger \Gamma_2 \text{ wf} & \quad \Sigma; \Gamma_1 \vdash t : A \quad \Sigma; \Gamma_2 \vdash u : B \\
\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \approx u : A \ddagger B \text{ wf}
\end{align*}
\]

A unification problem is a set of constraints sharing the same signature:
4.1. TWO-SIDED INTERNAL CONSTRAINTS

Definition 4.3 (Unification problem). Given a signature \( \Sigma \) 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.

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

Notation. The vector \( \vec{\mathcal{C}} \) may be understood as a conjunction of constraints; therefore we use \( \land \) as the element separator:

\[
\vec{\mathcal{C}}^n = \mathcal{C}_1 \land ... \land \mathcal{C}_n
\]

An empty vector of constraints is denoted by “□”:

\[
\vec{\mathcal{C}}^0 = □
\]

Definition 4.4 (Set of constants in a constraint or a vector of constraints: \( \text{consts}(\mathcal{C}), \text{consts}(\vec{\mathcal{C}}) \)).

\[
\text{consts}(\Gamma_1 \vdash \Gamma_2 \vdash t \approx u : A \vdash B) = \text{consts}(\Gamma_1) \cup \text{consts}(\Gamma_2)
\]
\[
\cup \text{consts}(t) \cup \text{consts}(u)
\]
\[
\cup \text{consts}(A) \cup \text{consts}(B)
\]

\[
\text{consts}(\vec{\mathcal{C}}^n) = \bigcup_{i=1}^{n} \text{consts}(\mathcal{C}_i)
\]

Definition 4.5 (Well-formed unification problem). A unification problem \( \Sigma; \vec{\mathcal{C}} \) is well-formed if \( \Sigma \) is well-formed, and each of the constraints in \( \vec{\mathcal{C}} \) is well-formed in \( \Sigma \).

\[
\Sigma \quad \text{sig} \quad \forall \mathcal{C} \in \vec{\mathcal{C}}, \Sigma; \mathcal{C} \quad \text{wf} \quad \Sigma; \vec{\mathcal{C}} \quad \text{wf}
\]

Remark 4.6 (No extraneous constants in constraint). If \( \Sigma; \mathcal{C} \quad \text{wf} \), then \( \text{consts}(\mathcal{C}) \subseteq \text{decls}(\Sigma) \). Also, if \( \Sigma; \vec{\mathcal{C}} \quad \text{wf} \), then \( \text{consts}(\vec{\mathcal{C}}) \subseteq \text{decls}(\Sigma) \).

Proof. By Definition 4.2 (well-formed internal constraint), Definition 4.5 (well-formed unification problem), and Lemma 2.72 (no extraneous constants). □

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 \( \Sigma \), \( \Sigma; \mathcal{C} \quad \text{wf} \) if and only if \( \Sigma \vdash J \); and \( \text{consts}(J) = \text{consts}(\mathcal{C}) \). Namely, \( J = (\Gamma_1 \vdash t : A) \land (\Gamma_2 \vdash u : B) \).

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 \( \Sigma \), we have \( \Sigma; \vec{\mathcal{C}} \quad \text{wf} \) if and only if \( \Sigma \vdash J \), and \( \text{consts}(J) \) 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 \( \vec{\mathcal{C}} = □ \), then let \( J \) be a judgment that holds in any signature, such as \( J \equiv \cdot \text{ctx} \).
Definition 4.9 (Solution to a constraint: \( \Theta \vdash \tilde{\mathcal{C}} \), \( \Theta \vdash \tilde{\mathcal{C}} \)). Let \( \Theta \) be a metasubstitution, and \( \mathcal{C} = \Gamma_1 \vdash t \equiv u : A \vdash B \) be an internal constraint.

We say that \( \Theta \) is a solution to \( \mathcal{C} \) (written \( \Theta \models \mathcal{C} \)) if and only if \( \Theta; \mathcal{C} \) \( \frac{\text{wf}}{\text{ctx}} \), \( \Theta; \Gamma_1 \vdash t \equiv u : A \) (or, equivalently, \( \Theta; \Gamma_1 \equiv \Gamma_2 \) \( \frac{\text{ctx}}{\text{type}} \) and \( \Theta; \Gamma_1 \vdash t \equiv u : A \)).

If \( \mathcal{C} \) is a vector of constraints, we say that \( \Theta \models \mathcal{C} \) if, for each \( \mathcal{C} \in \mathcal{C} \), \( \Theta \models \mathcal{C} \).

Remark 4.10 (Solution to a constraint as a judgment). Let \( \mathcal{C} \) be a constraint, \( \mathcal{C} = \Gamma_1 \vdash t \equiv u : A \vdash B \). Then there is a judgment \( J = (\Gamma_1, A \equiv \Gamma_2, B \frac{\text{ctx}}{\text{type}}) \frac{\text{ctx}}{\text{type}} (\Gamma_1 \vdash t \equiv u : A) \) such that \( \text{consts}(J) = \text{consts}(\mathcal{C}) \) and, for any well-formed metasubstitution \( \Theta \) \( \frac{\text{wf}}{\text{ctx}} \), \( \Theta \models \mathcal{C} \) if and only if \( \Theta \models J \).

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: \( \Theta \models \Sigma; \mathcal{C} \)). Let \( \Sigma; \mathcal{C} \) be a well-formed unification problem, and \( \Theta \) a well-formed metasubstitution. We say that \( \Theta \) is a solution to \( \Sigma; \mathcal{C} \) (written \( \Theta \models \Sigma; \mathcal{C} \)) if we have \( \Theta \models \Sigma \) and \( \Theta \models \mathcal{C} \).

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: \( \Sigma; \Gamma \vdash \mathcal{C} \)). Two terms \( \Sigma; \Gamma \vdash t : A \) and \( \Sigma; \Delta \vdash u : B \) are heterogeneously equal (written \( \Sigma; \Gamma \vdash \mathcal{C} \)), 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) \( \text{fv}(v) \subseteq \text{fv}(t) \cap \text{fv}(u) \).

Given such a witness \( v \), we can write \( \Sigma; \Gamma \vdash \mathcal{C} \). 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 $\Sigma;\Gamma_1 \vdash t \equiv u : A_1 \parallel A_2$ if and only if $\Sigma;\Gamma_1 \parallel t \equiv u : A_1$.

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).

\[
\begin{align*}
\mathbb{A} & : \text{Set, } a : \mathbb{A}, \alpha : \mathbb{A} \to \text{Set}; \\
x & : \mathbb{A} \to \mathbb{A} \parallel (\alpha \circ), z : (\alpha \circ) \parallel \mathbb{A} \to \mathbb{A} \\
\vdash \\
(x, \lambda y.z y) & \equiv (x, z) \equiv (\lambda y.x y, z) : \\
(\alpha \circ \times (\mathbb{A} \to \mathbb{A})) & \parallel ((\mathbb{A} \to \mathbb{A}) \times \alpha \circ)
\end{align*}
\]

Note that each side of the heterogeneous equality is equal to the witness $((x, z))$ 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 $\Sigma;\Gamma \parallel \Delta \vdash t \equiv t \equiv t : A \parallel B$ (even if $\Gamma \neq \Delta$).

Remark 4.16 (Symmetry of heterogeneous equality). Heterogeneous equality is symmetric. That is, given $\Sigma;\Gamma \parallel \Delta \vdash t \equiv t \equiv u : A \parallel B$, we also have $\Sigma;\Delta \parallel \Gamma \vdash u \equiv u \equiv t : B \parallel A$.

For our development, we are not concerned with whether the heterogeneous equality is transitive.

The heterogeneous equality is used to define when a constraint is satisfied in a given signature.

Definition 4.17 (Constraint satisfaction: $\Sigma \models \mathcal{C}$). Let $\Sigma$ be a signature, and $\mathcal{C}$ an internal constraint, $\mathcal{C} = \Gamma \parallel \Delta \vdash t \approx u : A \parallel B$. We say that $\mathcal{C}$ is satisfied in $\Sigma$ (written $\Sigma \models \mathcal{C}$), if $\Sigma;\mathcal{C} \text{wf}$ and the two sides of the constraint are heterogeneously equal. That is, $\Sigma;\Gamma \parallel \Delta \vdash t \equiv u : A \parallel B$.

We say that the constraints $\mathcal{C}$ are satisfied in signature $\Sigma$ (written $\Sigma \models \mathcal{C}$), if, $\Sigma$ is well-formed and for each $\mathcal{C} \in \mathcal{C}$, $\Sigma \models \mathcal{C}$.

Remark (Relationship between constraint solution and constraint satisfaction). Constraint satisfaction is a weaker notion than Definition 4.9 (solution to a constraint). If $\Theta \models \mathcal{C}$, then, in particular, $\Theta \models \mathcal{C}$.

When we can go in the other direction (that is, $\Theta \models \mathcal{C}$ implies $\Theta \models \mathcal{C}$), we say that $\mathcal{C}$ is an essentially homogeneous set of constraints. That is, even if each constraint is not necessarily homogeneous (e.g. $\Gamma_1 \parallel \Gamma_2 \vdash t \equiv u : A_1 \parallel A_2 \in \mathcal{C}$ 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 $\Theta$ 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 \( \vec{C} \) be a vector of constraints. We say that \( \vec{C} \) is an essentially homogeneous set of constraints iff for every metasubstitution \( \Theta \), such that \( \Theta \models \vec{C} \) we have \( \Theta \models \vec{C} \).

Definition 4.19 (Essentially homogeneous problem). A problem \( \Sigma; \vec{C} \) is essentially homogeneous iff \( \vec{C} \) 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 \( \Sigma \subseteq \Sigma' \), and \( \Sigma \models \vec{C} \). Then \( \Sigma' \models \vec{C} \).

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 \( \Theta \models \Sigma \) and \( \Sigma \models \vec{C} \). Then \( \Theta \models \vec{C} \).

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 \( \Sigma'; \vec{C} \) be a unification problem, where \( \Sigma' \) is a signature produced by the elaboration algorithm and \( \vec{C} \) contains, for each basic constraint of the form \( \Gamma \vdash u : A \cong v : B \) produced by the elaboration algorithm, the internal constraints \( \Gamma \vdash A \cong B : \text{Set}\\text{Set} \) and \( \Gamma \vdash u \approx v : A\approx B \). Then we say that \( \Sigma'; \vec{C} \) 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 \( \Sigma'; \vec{C} \) its elaboration into internal constraints by a well-formed elaboration algorithm.

Then, the following hold:
Well-formedness The problem $\Sigma';{\tilde{C}}$ is well-formed, $\Sigma \subseteq \Sigma'$ and $\text{DECLS}(\Sigma') \supseteq \text{DECLS}(\Sigma) \cup \text{METAS}(t)$.

Essential homogeneity The set of constraints $\tilde{C}$ is essentially homogeneous.

Proof.

Well-formedness By construction.

Essential homogeneity Assume $\Theta \models \tilde{C}$.

Let $C \in \tilde{C}$, Let $C = \Gamma_1 \vdash \Gamma_2 \vdash t \approx u : A\downarrow B$.

There are two possible cases:

i) $C = \Gamma\downarrow \Gamma \vdash A \approx B : \text{Set}\downarrow \text{Set}$: Assume $\Theta;\Gamma\vdash A \equiv \{V\} \equiv B : \text{Set}\downarrow \text{Set}$. This implies $\Theta;\Gamma \vdash A \equiv V : \text{Set}$ and $\Theta;\Gamma \vdash B \equiv V : \text{Set}$. By Lemma 2.70 (piecwise well-formedness of typing judgments) we have $\Theta \vdash \Gamma$ \text{ctx}. Also, by the set rule and Remark 2.15 (there is only set), $\Theta;\Gamma \vdash \text{Set}$, which gives $\Theta \vdash \Gamma, \text{Set}$ \text{ctx}$. By reflexivity of context equality $\Theta \vdash \Gamma, \text{Set}$ \text{ctx} and $\Theta;\Gamma\downarrow \Gamma \vdash A \equiv B : \text{Set}\downarrow \text{Set}$, by Lemma 4.13 (homogenization of heterogeneous equality), $\Theta \vdash C$.

ii) $C = \Gamma\downarrow \Gamma \vdash t \approx u : A\downarrow B$: By Definition 4.22 (elaboration into internal constraints), the elaboration algorithm produced a basic constraint $C' = \Gamma \vdash t : A \equiv u : B$, which means there is a constraint $C'' \in \tilde{C}$, $C'' = \Gamma\downarrow \Gamma \vdash A \approx B : \text{Set}\downarrow \text{Set}$. Because $\Theta \vdash C$, in particular, $\Theta \vdash C''$. This means $\Theta;\Gamma \vdash A \equiv A_0 : \text{Set}$ and $\Theta;\Gamma \vdash B \equiv A_0 : \text{Set}$ for some term $A_0$. By transitivity of equality, $\Theta;\Gamma \vdash B : \text{Set}$. By Remark 2.15, $\Theta;\Gamma \vdash A \equiv B$ \text{type}. By reflexivity of context equality, $\Theta \vdash \Gamma \equiv \Gamma$ \text{ctx}$. By Definition 2.16, $\Theta \vdash \Gamma, A \equiv \Gamma, B$ \text{ctx}$. By Lemma 4.13 (homogenization of heterogeneous equality), $\Theta \vdash C$.

Therefore, for all $C \in \tilde{C}$, $\Theta \vdash C$. Thus, $\tilde{C}$ 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 $\Sigma';\tilde{C}$ its elaboration into internal constraints by a well formed and correct elaboration algorithm. Then the following hold:

Soundness For each $\Theta$ such that $\Theta \models \Sigma';\tilde{C}$, we have $\Theta_{\Sigma;\Gamma} \text{wf}$ and $\Theta_{\Sigma;\Gamma} \models \Sigma;\Gamma \vdash^? t : A$.

Completeness If there is $\bar{\Theta}$ with $\bar{\Theta} \models \Sigma;\Gamma \vdash^? t : A$, then there is $\bar{\Theta}$ with $\bar{\Theta}_{\Sigma;\Gamma} = \Theta$ and $\bar{\Theta} \models \Sigma';\tilde{C}$.

Proof.
Soundness Assume $\Theta \vdash \Sigma'; \vec{c}$. 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 $\mathcal{C} = \Gamma \vdash t \equiv u : A \upharpoonright B \in \vec{c}$. Again by Definition 4.11 (solution to a unification problem), this means $\Theta \vdash \Sigma'; \Gamma, A \equiv \Gamma, B$ and $\Theta; \Gamma \vdash t \equiv u : A$.

By Definition 3.12 (correctness of an elaboration algorithm), $\Theta_{\Sigma'; t} \vdash \Gamma \vdash^? t : A$.

Completeness Assume there is a metasubstitution $\Theta$ such that $\Theta \vdash \Sigma; \Gamma \vdash^? t : A$. By Definition 3.12 (correctness of an elaboration algorithm), there is $\tilde{\Theta}$ such that $\tilde{\Theta} \Sigma \uplus t = \Theta$ and $\tilde{\Theta} \vdash \Sigma'$.

Let $\mathcal{C} \in \vec{C}$. We proceed by case analysis:

- $\mathcal{C} = \Gamma \vdash t \approx u : A \upharpoonright B : \text{Set} \upharpoonright \text{Set}$, and the elaboration algorithm produced a basic constraint $\tilde{\mathcal{C}} = \Gamma \vdash t : A \equiv u : B$.
  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), $\Sigma' \vdash \Gamma$ and $\Sigma' \vdash \Gamma, \text{Set} \equiv \Gamma, \text{Set} \equiv \Gamma, \text{Set}$. Because $\tilde{\Theta} \vdash \Sigma'$, we have $\tilde{\Theta} \vdash \Gamma, \text{Set} \equiv \Gamma, \text{Set}$. Also, by Definition 3.12 (correctness of an elaboration algorithm), $\tilde{\Theta} \vdash A \equiv B$ type. By Remark 2.15 (there is only set), $\tilde{\Theta} \vdash \Gamma \vdash A \equiv B : \text{Set}$; that is, $\tilde{\Theta} \vdash \mathcal{C}$.

- $\mathcal{C} = \Gamma \vdash t \approx u : A \upharpoonright B$, and the elaboration algorithm produced a basic constraint $\mathcal{C} = \Gamma \vdash t : A \equiv u : B$. By Definition 3.12 (correctness of an elaboration algorithm), $\tilde{\Theta} \vdash \Gamma \equiv B$ type and $\tilde{\Theta} \vdash \Gamma \vdash t \equiv u : A$.
  By Lemma 2.70 (piecewise well-formedness of typing judgments), we have $\tilde{\Theta} \vdash \Gamma$ and $\tilde{\Theta} \vdash \Gamma \equiv \Gamma$ and $\tilde{\Theta} \vdash \Gamma \vdash A \equiv B : \text{Set}$; that is, $\tilde{\Theta} \vdash \mathcal{C}$.

Therefore, $\tilde{\Theta} \vdash \Sigma'; \vec{c}$.

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 $\Sigma; \vec{c} \rightsquigarrow \Sigma'; \vec{D}$, where $\Sigma$ and $\Sigma'$ are signatures, and $\vec{c}$ and $\vec{D}$ are vectors of internal constraints.

A rule $\Sigma; \vec{c} \rightsquigarrow \Sigma'$ states that, under signature $\Sigma$, the constraints $\vec{c}$ reduce to a list of constraints $\vec{D}$, extending the signature to $\Sigma'$. 
4.4. CORRECTNESS OF REDUCTION RULES

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 \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\beta} \) is correct if, assuming that \( \Sigma; \vec{\alpha} \) is well-formed, we have:

**Well-formedness** The problem \( \Sigma'; \vec{\beta} \) is well-formed, and \( \Sigma \subseteq \Sigma' \) (in particular, \( \Sigma' \) sig).

**Soundness** For every \( \Sigma'' \) with \( \Sigma'' \sqsupseteq \Sigma' \), if \( \Sigma'' \models \vec{\beta} \) then \( \Sigma'' \models \vec{\alpha} \).

**Completeness** For each metasubstitution \( \Theta \) such that \( \Theta \models \Sigma; \vec{\alpha} \), there is a metasubstitution \( \Theta' \) such that \( \Theta = \Theta'_\Sigma \) and \( \Theta' \models \Sigma'; \vec{\beta} \).

**Remark.** The soundness property is stated in terms of constraint satisfaction (\( \models \)), and the completeness in terms of constraint solutions (\( \models \)).

We make this distinction to make the correctness proofs of the individual rules more succinct, as for soundness it suffices to show satisfaction (\( \Sigma'' \models \vec{\alpha} \)). However, for proving completeness, we use the stronger premise (\( \Theta \models \vec{\alpha} \)).

**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: \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \)). We say that the problem \( \Sigma; \vec{\alpha} \) reduces to \( \Sigma'; \vec{\alpha}' \) in one step (written \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \)), if, \( \vec{\alpha} = \vec{\alpha}_1 \land \vec{\alpha} \land \vec{\alpha}_2 \), \( \vec{\alpha}' = \vec{\alpha}_1 \land \vec{\beta} \land \vec{\alpha}_2 \), and \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\beta} \) is a correct rule.

**Definition 4.28** (Problem reduction: \( \Sigma'; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \)). We say that the problem \( \Sigma; \vec{\alpha} \) reduces to \( \Sigma'; \vec{\alpha}' \) if \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \), where \( \rightarrow \) is the reflexive, transitive closure of \( \rightarrow \).

**Lemma 4.29** (Correctness of problem reduction). Given a well-formed problem \( \Sigma; \vec{\alpha} \), such that \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \), then \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \) is a correct rule. That is, the following hold:

**Well-formedness** The problem \( \Sigma'; \vec{\alpha}' \) is well-formed, and \( \Sigma \subseteq \Sigma' \).

**Soundness** For every \( \Sigma'' \) with \( \Sigma'' \sqsupseteq \Sigma' \), if \( \Sigma'' \models \vec{\alpha}' \) then \( \Sigma'' \models \vec{\alpha} \).

**Completeness** For each metasubstitution \( \Theta \) such that \( \Theta \models \Sigma; \vec{\alpha} \), there is a metasubstitution \( \Theta' \) such that \( \Theta = \Theta'_\Sigma \) and \( \Theta' \models \Sigma'; \vec{\alpha}' \).

**Proof.** By induction on the length of the derivation of \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}' \), 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 \( \Sigma; \vec{\alpha} \), rules are applied iteratively, stopping if it produces a signature \( \Sigma' \) such that \( \Sigma; \vec{\alpha} \rightarrow \Sigma'; \vec{\alpha}'' \). If \( \Sigma' \) is closed (that is, instantiates all metavariables) we can obtain a solution to the original problem \( \Sigma; \vec{\alpha} \) by constructing the closing metasubstitution of \( \Sigma' \).
**Definition 4.30** (Solved problem). Let $\Sigma; \vec{D}$ be a problem. We say that $\Sigma; \vec{D}$ is a solved problem if $\Sigma; \vec{D}$ \textit{wf}, $\Sigma$ is a closed signature, and $\vec{D} = \square$.

**Theorem 4.31** (Correctness of unification). Let $\Sigma; \vec{C}$ be an essentially homogeneous, well-formed problem such that: $\Sigma; \vec{C} \rightsquigarrow \Sigma'; \square$, where $\Sigma'; \square$ is a solved problem (i.e. $\Sigma'$ is closed).

Then the following hold:

1. The signature $\Sigma'$ is well-formed.
2. There is $\Theta$ such that close($\Sigma'$) $\downarrow\Theta$ and $\Theta \Sigma \models \Sigma; \vec{C}$.
3. For every $\tilde{\Theta}$ such that $\tilde{\Theta} \models \Sigma; \vec{C}$, 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 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).

\[
\Sigma; \Gamma \vdash t \approx t : A \vdash A' \Rightarrow \Sigma; \square
\]

**Proof of correctness.** By Definition 4.26 (rule correctness), it suffices to show:

- **Well-formedness** Assume that $\Sigma; \Gamma \vdash t \approx t : A \vdash A'$ is well-formed. Then, $\Sigma \text{ sig}$. If $\Sigma$ is well-formed, then the problem $\Sigma; \square$ is also trivially well-formed. By Definition 2.151 (signature extension), $\Sigma \subset \Sigma$.

- **Soundness** Because $\Sigma; \Gamma \vdash t \approx t : A \vdash A'$ is well-formed, we have $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Gamma' \vdash t : A'$. Let $\Sigma'' \supseteq \Sigma$. 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), $\Sigma''; \Gamma \vdash t \equiv \{t\} \equiv t : A \vdash A'$. 

4.5. A REDUCTION RULE TOOLKIT

Completeness Assume $\Theta \vdash \Sigma; \Gamma' \vdash t : A \vdash A'$. In particular, $\Theta \vdash \Sigma$.

Let $\Theta' = \Theta$. We have $\Theta'_\Sigma = \Theta$. By assumption, $\Theta' \vdash \Sigma$. Because $\Theta' \vdash \Sigma$, then vacuously $\Theta' \vdash \Sigma; \Box$.

4.5.2 Metavariable instantiation

Metavariable instantiation is the bread-and-butter of higher-order unification.

Given a unification problem of the form $\Sigma'; \Box$ (with $\Sigma'$ 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:

$$[[\Sigma_1, \alpha : A, \Sigma_2; \Gamma_1 \vdash \alpha \bar{x}^n \approx t : B_1 \vdash B_2] \Rightarrow \Sigma_1, \alpha := u : A, \Sigma_2; \Box] \quad (\star)$$

This rule schema instantiates $\alpha$ to $u$ using the constraint $\Gamma_1 \vdash \alpha \bar{x}^n \approx t : B_1 \vdash B_2$.

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 Rule-Schema 2 (metavariable instantiation). In order to specify those preconditions and prove the correctness of the rule, we first need to introduce some new concepts.

Lemma 4.33 (General $\eta$-equality for $\Pi$-types). If $\Sigma; \Gamma \vdash u : \Pi \bar{A}^n B$, then $\Sigma; \Gamma \vdash u \equiv \lambda x^n \cdot (u^{+n}) \cdot \bar{x} : \Pi \bar{A}^n B$.

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 $\eta$-equality for pairs). Assume $\Sigma; \Gamma \vdash u : \Sigma AB$. Then $(u \cdot \bar{\pi}_1)\downarrow$, $(u \cdot \bar{\pi}_2)\downarrow$ and $\Sigma; \Gamma \vdash u \equiv \langle u \cdot \bar{\pi}_1, u \cdot \bar{\pi}_2 \rangle : \Sigma AB$.

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 \( \Sigma; \cdot \vdash u : \Pi \overrightarrow{A}^n.B, \Sigma; \cdot \vdash v : \Pi \overrightarrow{A}^n.B \) (in particular, \( u \) and \( v \) closed). Assume that, for all \( i \in \{1, \ldots, n\} \), \( \Sigma; \Gamma \vdash x_i : A_i[\overrightarrow{x}_{1,\ldots,i-1}] \), and \( \Sigma; \Gamma \vdash u @ \overrightarrow{x} \equiv v @ \overrightarrow{x} : B[\overrightarrow{x}] \), with all variables in \( \overrightarrow{x} \) pairwise distinct. Then \( \Sigma; \cdot \vdash u \equiv v : \Pi \overrightarrow{A}^n.B \).

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 \( \Sigma \vdash \Gamma_1, B_1 \equiv \Gamma_2, B_2 \text{ ctx} \). 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 \( \Gamma_1 \) and \( \Gamma_2 \) are heterogeneously equal in signature \( \Sigma \) modulo the sets of variables \( X_1 \) and \( X_2 \) (written \( \Sigma \vdash \Gamma_1 \equiv \Gamma_2 \equiv X_1, X_2 \)), if \( \Sigma \) is a well-formed signature, \( \Gamma_1 \) and \( \Gamma_2 \) 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 \( \Gamma_1 \) and \( \Gamma_2 \) are heterogeneously equal.

\[
\begin{align*}
\Sigma \vdash \cdot \equiv \{\cdot\} \equiv \emptyset, \emptyset \quad \text{EMPTY} \\
0 \notin X_1 \cup X_2 \quad \Sigma \vdash \Gamma_1 \equiv \Gamma \equiv X_1, X_2 \equiv \Gamma_2 \\
\quad \Sigma \vdash \Gamma_1, A_1 \equiv \{\Gamma, \text{Set}\} \equiv X_1, X_2 \equiv \Gamma_2, A_2 \quad \text{UNUSED} \\
0 \in X_2 - X_1 \quad \Sigma \vdash \Gamma_1 \equiv \{\Gamma\} \equiv X_1, X_2 - 1 \equiv \Gamma_2 \\
\quad \Sigma \vdash \Gamma_1, A_1 \equiv \{\Gamma, A_2\} \equiv X_1, X_2 \equiv \Gamma_2, A_2 \quad \text{USED-R} \\
0 \in X_1 - X_2 \quad \Sigma \vdash \Gamma_1 \equiv \{\Gamma\} \equiv (X_1 - 1) \cup \text{FV}(A_2), X_2 - 1 \equiv \Gamma_2 \\
\quad \Sigma \vdash \Gamma_1, A_1 \equiv \{\Gamma, A_1\} \equiv X_1, X_2 \equiv \Gamma_2, A_2 \quad \text{USED-L} \\
\end{align*}
\]

\[
\begin{align*}
a & \equiv \text{FV}(A_1) \cap \text{FV}(A_2) \\
X'_1 & \equiv (X_1 - 1) \cup a \\
X'_2 & \equiv (X_2 - 1) \cup a \\
\Sigma; \Gamma_1 \uplus \Gamma_2 \vdash A_1 \equiv \{A\} \equiv \text{Set} \uplus \text{Set} \quad \Sigma \vdash \Gamma_1 \equiv \{\Gamma\} \equiv X_1, X_2 \equiv \Gamma_2 \\
\quad \Sigma \vdash \Gamma_1, A_1 \equiv \{\Gamma, A\} \equiv X_1, X_2 \equiv \Gamma_2, A_2 \quad \text{USED}
\end{align*}
\]

Remark. In the rule UNUSED we use Set as the witness type because it is well-formed in any context \( \Gamma \). 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 $\Sigma; \Gamma_1 \equiv \Gamma_2 \equiv B \equiv B_2 : \text{Set} \uplus \text{Set}$ and $\Sigma \vdash \Gamma_1 \equiv \Gamma_2$. Then $\Sigma; \Gamma \vdash t : B$ and $\Sigma; \Gamma \vdash u : B$.

Proof. We will prove the following stronger property:

Suppose that $\Sigma \vdash \Gamma_1 \equiv \Gamma_2 \equiv X_1, X_2$. For every $\Delta$, if $\Sigma; \Delta \vdash t : B$ and $\text{fv}(\Delta \vdash t : B) \subseteq X_1$, then $\Sigma; \Gamma \vdash t : B$.

Also, if $\Sigma; \Delta \vdash u : B$ and $\text{fv}(\Delta \vdash u : B) \subseteq X_2$, then $\Sigma; \Gamma \vdash u : B$.

We proceed by induction on the length of $\Gamma$ (i.e., the structure of the derivation for $\Sigma \vdash \Gamma_1 \equiv \Gamma_2$).

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:

$$
\begin{align*}
\Gamma_1 &= \Gamma'_1, A_1 \\
\Gamma_2 &= \Gamma'_2, A_2 \\
\Gamma &= \Gamma', A
\end{align*}
$$

We consider four cases, one for each possible rule in the derivation of $\Sigma \vdash \Gamma_1 \equiv \Gamma_2$:

- Rule empty: Trivial.
- Rule unused:

$$
\begin{align*}
0 \notin X_1 \cup X_2 & \quad \Sigma \vdash \Gamma_1 \equiv (\Gamma'_1) \equiv X_1, X_2, \Gamma_2 \\
\Sigma \vdash \Gamma_1, A_1 \equiv (\Gamma'_1, \text{Set}) \equiv X_1, X_2, \Gamma_2, A_2 & \text{ Unused}
\end{align*}
$$

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

Also by assumption, $\Gamma'_1, A_1, \Delta \vdash t : B$, which, by Lemma 2.71 (variables of irrelevant type), implies $\Gamma'_1, \text{Set}, \Delta \vdash t : B$.

By Definition 2.45 (free variables of a scoped and typed term) and the assumption, $\text{fv}(\text{Set}, \Delta \vdash t : B) = \text{fv}(\Delta \vdash t : B) - 1 \subseteq X_1 - 1$.

From the premises, $\Sigma \vdash \Gamma'_1 \equiv (\Gamma'_1) \equiv X_1, X_2 - 1, \Gamma_2$. By the induction hypothesis, $\Gamma'_1, \text{Set}, \Delta \vdash t : B$; i.e., $\Gamma, \Delta \vdash t : B$.

By the same token, we show that, if $\Gamma_2, \Delta \vdash u : B$ with $\text{fv}(\Delta \vdash u : B) \subseteq X_2$, then $\Gamma, \Delta \vdash u : B$.

- Rule used-l:

$$
\begin{align*}
0 \in X_1 - X_2 & \quad \Sigma \vdash \Gamma_1 \equiv (\Gamma'_1) \equiv X_1, X_2 - 1, \Gamma_2 \\
\Sigma \vdash \Gamma_1, A_1 \equiv (\Gamma'_1, A_1) \equiv X_1, X_2, \Gamma_2, A_2 & \text{ Used-l}
\end{align*}
$$

From the premises, $\Sigma \vdash \Gamma_1 \equiv (\Gamma'_1) \equiv X_1, X_2 - 1, \Gamma_2$. By the induction hypothesis, $\Gamma'_1 \equiv (\Gamma'_1) \equiv X_1, X_2 - 1, \Gamma_2$. By the same token, we show that, if $\Gamma_2, \Delta \vdash u : B$ with $\text{fv}(\Delta \vdash u : B) \subseteq X_2$, then $\Gamma, \Delta \vdash u : B$. 

Assume $\Gamma, \Delta \vdash t : B$, i.e. $\Gamma', A, \Delta \vdash t : B$.

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

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. $\Gamma_2', A_1, \Delta \vdash u : B$.

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

Also from $\text{FV}(\Delta \vdash u : B) \subseteq X_2$ we deduce $\text{FV}(\text{Set}, \Delta \vdash u : B) = \text{FV}(\text{Set}) \cup (\text{FV}(\Delta \vdash u : B) - 1) = \text{FV}(\Delta \vdash u : B) - 1 \subseteq X_2 - 1$. By the induction hypothesis, $\Gamma', \text{Set}, \Delta \vdash u : B$.

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:

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

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

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

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

From the assumptions $\text{FV}(\Delta \vdash u : B) \subseteq X_2$ and $\Sigma; \Gamma_2', A_2, \Delta \vdash u : B$, by the same token, it follows that $\Sigma; \Gamma, \Delta \vdash u : B$.

Now that we have proven ($\star$), we can use it to prove the original lemma.

By hypothesis, $\Sigma; \Gamma_1'; \Gamma_2 \vdash B_1 \equiv B \equiv B_2 : \text{Set}$.

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

By Definition 2.45 (free variables of a scoped and typed term), $\text{FV}(\cdot \vdash t : B) = \text{FV}(t) \cup \text{FV}(B) \subseteq \text{FV}(t) \cup b$, where $\Sigma; \Gamma_1 \vdash B_1 \equiv B : \text{Set}$.

By Lemma 2.63 (preservation of judgments by type conversion), $\text{FV}(B_1) \subseteq \text{FV}(B) \subseteq \text{FV}(A_1) \cap \text{FV}(A_2) = a$.

By the induction hypothesis, $\Sigma; \Gamma_1', A_1, \Delta \vdash t : B$, i.e. $\Gamma_1', A_1, \Delta \vdash t : B$. Finally, by Lemma 2.71, we have $\Gamma_1', A_1, \Delta \vdash t : B$, i.e. $\Gamma, \Delta \vdash t : B$. 

- Rule used-r: Same as used-l, swapping “1” and “2”, and “t” and “u”.

- Rule used:
Also by hypothesis, \( \Sigma \vdash \Gamma_1 \equiv \{ \Gamma \} \equiv \Gamma_2 \). By applying (⋆) with \( \Delta = \cdot \), we have \( \Sigma; \Gamma \vdash t : B \).

By the same reasoning, from \( \Sigma; \Gamma_2 \vdash u : B_2 \) and using (⋆), 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).

\[
\Sigma; \Gamma_1 \vdash \alpha \vec{x}^n \approx t : B_1 \vdash B_2 \quad \leadsto \quad \Sigma_1, \alpha := \lambda \vec{y}.t' : A, \Sigma_2; \Box
\]

where

\[
\begin{align*}
\Sigma &= \Sigma_1, \alpha : A, \Sigma_2 \\
\vec{t}' &= t[\vec{x} \mapsto \vec{y}] \\
\end{align*}
\]

all variables in \( \vec{x} \) are pair-wise distinct \hspace{1cm} (1)

\[
\begin{align*}
\text{FV}(t) &\subseteq \vec{x} \hspace{1cm} (2a) \\
\text{CONSTS}(t) &\subseteq \text{DECLS}(\Sigma_1) \hspace{1cm} (2b) \\
\Sigma; \Gamma_1 \vdash B_1 \equiv B &\equiv B_2 : \text{Set} \vdash \text{Set} \hspace{1cm} (3a) \\
\Sigma \vdash \Gamma_1 \equiv \{ \Gamma \} \equiv \{ x_i | i = 1, \ldots, n \} \cup b, \text{FV}(t) \cup b &\equiv \Gamma_2 \hspace{1cm} (3b) \\
b &\equiv \text{FV}(B_1) \cap \text{FV}(B_2) \\
\end{align*}
\]

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 \( \alpha \).

**Proof of correctness.** Let \( \mathcal{C} = \Gamma_1 \vdash \alpha \vec{x}^n \approx t : B_1 \vdash B_2 \), and \( \Sigma' = \Sigma_1, \alpha := \lambda \vec{y}.t' : A, \Sigma_2 \).

By Definition 4.26 (rule correctness), assuming \( \Sigma; \mathcal{C} \text{ wf} \), it suffices to show:

**Well-formedness** Because no new constraints are added, the rule is well-formed if \( \Sigma' \text{ sig} \) and \( \Sigma' \supseteq \Sigma \).

(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 \( \Sigma; \Gamma \vdash A \therefore \vec{x} \downarrow B' \), 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), \( \Sigma' : \vdash \lambda \vec{y}.t' : A \).

(ii) By the assumption, \( \text{CONSTS}(t') \subseteq \text{DECLS}(\Sigma_1) \). Because \( \Sigma_1 ; \cdot \vdash A \text{ type} \), by Lemma 2.72 (no extraneous constants), \( \text{CONSTS}(A) \subseteq \text{DECLS}(\Sigma_1) \).

Because \( \Sigma_1 \subseteq \Sigma \), and \( \Sigma; \cdot \vdash \lambda \vec{y}.t' : A \), by Postulate 12 (signature strengthening), \( \Sigma_1 ; \cdot \vdash \lambda \vec{y}.t' : A \).

By Remark 2.5 (signature inversion), we have that \( \Sigma_1 \text{ sig} \). By Definition 2.4 (well-formed signature) and item (ii), \( \Sigma_1, \alpha := \lambda \vec{y}.t' : A \text{ sig} \).

By Definition 2.151 (signature extension), this gives \( \Sigma_1, \alpha : A \sqsubseteq \Sigma_1, \alpha := \lambda \vec{y}.t' : A \).

By Corollary 2.156 (horizontal composition of extensions), \( \Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq \Sigma_1, \alpha := \lambda \vec{y}.t' : A, \Sigma_2 \).
CHAPTER 4. UNIFYING WITHOUT ORDER

**Soundness** Take $\Sigma'' \trianglerighteq \Sigma'$. We need to show that $\Sigma'' \models \mathcal{C}$.

(i) By rule $\text{delta-meta}_0$, $\Sigma'' \trianglerighteq \Sigma' \trianglerighteq \Sigma \trianglerighteq \Gamma_1 \text{ ctx}$, by Lemma 2.155 (preservation of judgments under signature extensions), we have $\Sigma'' ; \Gamma_1 \vdash \alpha \equiv \lambda \overrightarrow{y}.t' : A$.

(ii) The term $\alpha$ is neutral. By Definition 2.32 (hereditary elimination), $\alpha @ \overrightarrow{x} \downarrow \alpha @ \overrightarrow{x}$.

(iii) By Remark 2.35 (iterated application as substitution on body) $(\lambda \overrightarrow{y}.t') @ \overrightarrow{x} \downarrow t'[\overrightarrow{y} \mapsto \overrightarrow{x}]$. By Lemma 2.40 (correspondence between renaming and substitution), we have $t'[\overrightarrow{y} \mapsto \overrightarrow{x}] = t[\overrightarrow{x} \mapsto \overrightarrow{y}] \implies t \downarrow t[\overrightarrow{x} \mapsto \overrightarrow{y}]$. Therefore, we have $(\lambda \overrightarrow{y}.t' @ \overrightarrow{x}) \downarrow t$.

(iv) By well-formedness of the original problem, $\Sigma ; \Gamma_1 \vdash \alpha \overrightarrow{x} : B_1$. Because $\Sigma'' \trianglerighteq \Sigma'$, by Lemma 2.155 (preservation of judgments under signature extensions), we also have $\Sigma'' ; \Gamma_1 \vdash \alpha \overrightarrow{x} : B_1$. By Lemma 2.109 (type application inversion), this means that there exists $B'_1$ such that $\Sigma'' ; \Gamma_1 \vdash A @ \overrightarrow{x} \downarrow B'_1$ and $\Sigma'' ; \Gamma_1 \vdash B'_1 \equiv B_1$ type.

By (i), (ii) (iii) and Lemma 2.110 (type of hereditary application), $\Sigma'' ; \Gamma_1 \vdash \alpha \overrightarrow{x} \equiv t : B'_1$.

(v) By (iv) and the $\text{conv-eq}$ rule, $\Sigma'' ; \Gamma_1 \vdash \alpha \overrightarrow{x} \equiv t : B_1$.

(vi) By well-formedness of the original problem, $\Sigma ; \Gamma_2 \vdash t : B_2$. Because $\Sigma'' \trianglerighteq \Sigma' \trianglerighteq \Sigma$, by Lemma 2.155 (preservation of judgments under signature extensions), we also have $\Sigma'' ; \Gamma_2 \vdash t : B_2$. By reflexivity, $\Sigma'' ; \Gamma_2 \vdash t \equiv t : B_2$.

(vii) By the premises of the rule, $\text{fv}(t) \subseteq \overrightarrow{x} = \text{fv}(\alpha \overrightarrow{x})$. Also, trivially, $\text{fv}(t) \subseteq \text{fv}(t)$.

By (v), (vi), and (vii), $\Sigma'' ; \Gamma_1 \vdash \alpha \overrightarrow{x} \equiv \{t\} \equiv t : B_1 \uplus B_2$.

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

**Completeness** Assume that $\Theta \vdash \Sigma ; \Gamma_1 \uplus \Gamma_2 \vdash \alpha \overrightarrow{x} \approx t : B_1 \uplus B_2$ holds; that is, $\Theta \vdash \Sigma, \Theta \vdash \Gamma_1, B_1 \equiv \Gamma_2, B_2 \text{ ctx}$ and $\Theta ; \Gamma_1 \vdash \alpha \overrightarrow{x} \equiv t : B_1$.

Take $\Theta' = \Theta$. Because $\Theta \vdash \Sigma, \Theta' = \Theta_2 = \Theta$. We need to show that $\Theta \vdash \Sigma' ; \sqcup$. Because there are no new constraints, it suffices to show $\Theta \vdash \Sigma'$. Because $\text{decls}(\Theta) = \text{decls}(\Sigma')$, by Lemma 2.130 (alternative characterization of a compatible metasubstitution), it suffices to show that, for each $D \in \Sigma'$, $\Theta$ 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, $\Theta$ is compatible with $D$.

If $D \notin \Sigma_1$ and $D \notin \Sigma_2$, then $D = (\alpha :: \lambda \overrightarrow{y}.t' : A)$. Let $u$ and $B$ be the term and type such that $\alpha :: u : B \in \Theta$. 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) $\Theta ; \cdot \vdash u \equiv \lambda \overrightarrow{y}.t' : B$.

(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 $\Sigma \vdash_1 \Gamma$ and $\Theta \vdash \Sigma$, we have $\Theta \vdash_1 \Gamma$ context. Because $\alpha := u : B \in \Theta$, by the rule Delta-meta, $\Theta; \Gamma \vdash_1 \alpha \equiv u : B$.

Because the original problem is well-formed, we have $\Sigma; \Gamma \vdash_1 \alpha \overrightarrow{x} : B_1$. Also, because $\alpha := u : B \in \Theta$, we have $\Theta; \Gamma \vdash \alpha \Rightarrow B$. By Lemma 2.109 (type application inversion), $\Sigma; \Gamma \vdash B \overrightarrow{x} \Downarrow B'_1$, and $\Sigma; \Gamma \vdash B'_1 \equiv B_1 : \text{Set}$.

By Lemma 2.110 (type of hereditary application), from $\Theta; \Gamma \vdash_1 \alpha \equiv u : B$ we have $\Theta; \Gamma \vdash_1 u \overrightarrow{x} \equiv u : B_1$. By assumption, $\Theta; \Gamma \vdash_1 \alpha \overrightarrow{x} \equiv t : B_1$. By symmetry and transitivity of equality, we have $\Theta; \Gamma \vdash_1 u \overrightarrow{x} \equiv t : B_1$.

By item (iii) of the soundness proof, $\lambda \overrightarrow{y}.t' \overrightarrow{x} \Downarrow t$. Therefore, by reflexivity, $\Theta; \Gamma \vdash_1 \lambda \overrightarrow{y}.t' \overrightarrow{x} \equiv t : B_1$. By symmetry and transitivity of equality, we have $\Theta; \Gamma \vdash_1 u \overrightarrow{x} \equiv \lambda \overrightarrow{y}.t' : \text{Set}$. By Lemma 4.35 (Miller’s pattern condition), this gives $\Theta; \Gamma \vdash_1 u \equiv (\lambda \overrightarrow{y}.t') : A$.

Because all of $u$, $(\lambda \overrightarrow{y}.t')$, and $A$ are closed terms, by Postulate 13 (context strengthening), $\Theta; \cdot \vdash_1 u \equiv (\lambda \overrightarrow{y}.t') : A$. Because $\Theta; \cdot \vdash_1 B \equiv A$ type, by the Conv-eq rule, $\Theta; \cdot \vdash_1 u \equiv (\lambda \overrightarrow{y}.t') : B$.

\[\square\]

4.5.3 Type constructors

When it comes to the judgmental equality, the type formers $\Pi$ and $\Sigma$ are injective. That is, two $\Pi$ types are equal as terms iff the domain and codomain are equal as terms (Postulate 10). Correspondingly, two $\Sigma$-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 $\Pi$-type or a $\Sigma$-type.

Rule-Schema 3 (Injectivity of $\Pi$).

\[
\begin{align*}
\Sigma; \Gamma \vdash \Pi AB & \equiv \Pi A'B' : \text{Set} \xrightarrow{\Rightarrow} \\
\Sigma; \Gamma \vdash A & \equiv A' : \text{Set} \wedge \\
\Gamma \vdash A', A \equiv B' & : \text{Set} \\
\end{align*}
\]

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

Rule-Schema 4 (Injectivity of $\Sigma$).

\[
\begin{align*}
\Sigma; \Gamma \vdash AB & \equiv \Sigma A'B' : \text{Set} \xrightarrow{\Rightarrow} \\
\Sigma; \Gamma \vdash A & \equiv A' : \text{Set} \wedge \\
\Gamma \vdash A', A \equiv B' & : \text{Set} \\
\end{align*}
\]

Proof. We follow the same reasoning as in the proof for Rule-Schema 3, using Lemma 2.53 ($\Sigma$ inversion) and Postulate 11 (injectivity of $\Sigma$). \[\square\]

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).

\[ \Sigma; \Gamma \vdash \text{Bool} \approx \text{Bool} : \text{Set} \not\vdash \Sigma; \Box \]

Proof of correctness. This rule schema is a special case of Rule-Schema 1 (syntactic equality).

Rule-Schema 6 (Set).

\[ \Sigma; \Gamma \vdash \text{Set} \approx \text{Set} : \text{Set} \not\vdash \Sigma; \Box \]

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).

\[ \Sigma; \Gamma \vdash t \approx u : A \not\vdash \Sigma ; \Gamma' \vdash u \approx t : A' \]

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 \( \Sigma; \vec{\Gamma} \vdash \vec{\Delta} \), we get a rule \( \Sigma; \vec{\Gamma}' \vdash \vec{\Delta}' \); where for each \( \vec{\Gamma}'_i = \Gamma_1 \vdash t \approx u : A_1 \not\vdash \Gamma_2 \), we have \( \vec{\Gamma}'_i \equiv \Gamma_2 \vdash t \approx u : A_1 \not\vdash \Gamma_1 \), and for each \( \vec{\Delta}'_i = \Gamma_1 \vdash t \approx u : A_1 \not\vdash \Gamma_2 \), we have \( \vec{\Delta}'_i \equiv \Gamma_2 \vdash t \approx u : A_1 \not\vdash \Gamma_1 \).

4.5.5 Term conversion

Consider the following problem:

\[ \Sigma; \cdot \vdash a \approx \lambda x.a x : (A \rightarrow A) \not\vdash (A \rightarrow A) \]

\[ \Sigma = A : \text{Set}, a : A \rightarrow A \]

Observe that, by the \texttt{eta-abs} rule, \( \Sigma; \cdot \vdash a \equiv \lambda x.a x : A \rightarrow A \) (***). If we replace the LHS of the constraint in (*)& with the RHS of (**), we can use Rule-Schema 1 (syntactic equality) to solve problem (*).

In general, given a constraint \( \Sigma; \Gamma \vdash t \approx u : A \not\vdash \Gamma' \), 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 \( \text{FV}(t') \subseteq \text{FV}(t) \). This condition is in particular fulfilled when \( \Sigma; \Gamma \vdash t \rightarrow_{\delta \eta} t' : A \), which by Remark 2.43 (free variables of \( \delta \eta \)-reduct) implies \( \text{FV}(t') \subseteq \text{FV}(t) \).
4.5. A REDUCTION RULE TOOLKIT

Rule-Schema 8 (Term conversion).

\[ \Sigma; \Gamma \vdash t \approx u : A \Rightarrow \Sigma; \Gamma' \vdash t' \approx u : A' \]

where \( \Sigma; \Gamma \vdash t \equiv t' : A \)

\[ \text{fv}(t) \supseteq \text{fv}(t') \]

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).

\[ \Sigma; \Gamma \vdash t \approx u : A \Rightarrow \Sigma; \Gamma_0 \vdash t' \approx u : A_0 \]

where \( \Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0 \text{ ctx} \)

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).

\[ \Sigma; \Box \Rightarrow \Sigma' ; \Box \]

where \( \Sigma \sqsubseteq \Sigma' \text{ and } \Sigma' \sqsubseteq \Sigma \text{ ctx} \)


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 \( \lambda \)-abstraction, or by a pair constructor.

Rule-Schema 11 (\( \lambda \)-abstraction).

\[ \Sigma; \Gamma \vdash \lambda. t \approx \lambda. u : \Pi A B \Rightarrow \Sigma; \Gamma' \vdash t' \approx u : B' \]

Proof of correctness. Using Lemma 2.82 (\(\lambda\) inversion), Lemma 2.83 (injectivity of \(\lambda\)) and Postulate 10 (injectivity of \(\Pi\)). 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).
\[
\Sigma; \Gamma \vdash \langle t_1, t_2 \rangle \approx \langle u_1, u_2 \rangle : \Sigma AB \vdash A'B' \leadsto \\
\Sigma; \Gamma \vdash t_1 \approx u_1 : A \dvdash A' \land \Gamma \vdash t_2 \approx u_2 : B[t_1]\dvdash B'[u_1] \\
\text{where } B[t_1] \downarrow \text{ and } B'[u_1] \downarrow
\]

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).
\[
\Sigma; \Gamma \vdash c \approx c : \text{Bool} \dvdash \text{Bool} \leadsto \Sigma; \Box
\]

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 \(\Sigma \equiv \mathbb{A} : \text{Set}, a : \mathbb{A}, b : \mathbb{A}\), and consider the following three examples:

**Example 4.39** (Strong neutral unification).
\[
\Sigma, \alpha : \mathbb{A}; x : \text{Bool} \vdash \lambda.\mathbb{A} x \alpha \equiv \lambda.\mathbb{A} x a \equiv b : \mathbb{A} \dvdash \mathbb{A}
\]

By Definition 4.11 (solution to a unification problem), this problem has a solution \(\Theta \equiv \Sigma, \alpha := b : \mathbb{A}\). In effect, we can obtain such a solution by requiring each of the arguments to \(\lambda\) on the LHS to be equal to the corresponding argument on the RHS.

\[
\Sigma, \alpha : \mathbb{A}; x : \text{Bool} \vdash \lambda.\mathbb{A} \equiv \lambda.\mathbb{A} : (\text{Bool} \to \mathbb{A}) \dvdash (\text{Bool} \to \mathbb{A}) \\
x : \text{Bool} \vdash x \equiv x : \text{Bool} \\
x : \text{Bool} \vdash a \equiv a : \mathbb{A} \dvdash \mathbb{A} \\
x : \text{Bool} \vdash \alpha \equiv b : \mathbb{A} \dvdash \mathbb{A}
\]

By applying Rule-Schema 1 (syntactic equality) and Rule-Schema 2 (meta-variable instantiation), we have \(\Sigma, \alpha := b : \mathbb{A}; \Box\). By definition, close(\(\Sigma, \alpha := b : \mathbb{A}\) \(\downarrow \Theta\).

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:

\[ \Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \vdash (\lambda \cdot \mathcal{A}) \beta \mathcal{a} \equiv (\lambda \cdot \mathcal{A}) \beta \mathcal{a} : \mathcal{A} \]

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

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

\[ \Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \vdash \lambda \cdot \mathcal{A} \equiv \lambda \cdot \mathcal{A} : \text{Bool} \rightarrow \mathcal{A} \]
\[ \vdash \alpha \equiv \beta : \text{Bool} \]
\[ \vdash \mathcal{a} \equiv \mathcal{b} : \mathcal{A} \]
\[ \vdash \mathcal{b} \equiv \mathcal{a} : \mathcal{A} \]

However, \( \Theta \) is no longer a solution of the resulting problem, as this would imply \( \Theta; \vdash \mathcal{a} \equiv \mathcal{b} : \mathcal{A} \).

\[ \blacksquare \]

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

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

\[ \Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \vdash \beta \equiv \text{true} : \text{Bool} \]
\[ \vdash (\lambda \cdot \mathcal{A}) \alpha \mathcal{a} \equiv (\lambda \cdot \mathcal{A}) \beta \mathcal{a} : \mathcal{A} \]

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

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

\[ \Sigma, \alpha : \text{Bool}, \beta : \text{Bool}; \vdash \beta \equiv \text{true} : \text{Bool} \]
\[ \vdash \lambda \cdot \mathcal{A} \equiv \lambda \cdot \mathcal{A} : \text{Bool} \rightarrow \mathcal{A} \]
\[ \vdash \alpha \equiv \beta : \text{Bool} \]
\[ \vdash \mathcal{a} \equiv \mathcal{a} : \mathcal{A} \]
\[ \vdash \mathcal{a} \equiv \mathcal{a} : \mathcal{A} \]

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

\[ \Sigma, \beta := \text{true} : \text{Bool}, \alpha := \beta : \text{Bool}; \square \]

And \( \text{close}(\Sigma, \beta := \text{true} : \text{Bool}, \alpha := \beta : \text{Bool}) \downarrow \Theta_1 \). However, the resulting solution \( (\Theta_1) \) is not a unique solution to the original problem.

\[ \blacksquare \]

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).

\[
\begin{align*}
\Sigma; \Gamma \vdash h \overline{e} \approx h \overline{e'} : T_1 \vdash T' \\
\Sigma; \bigwedge_{i \in J} \Gamma \vdash t_i \approx u_i : B_i \vdash B'_i
\end{align*}
\]

where

\[J \subseteq \{1, \ldots, n\}\]

h \overline{e} and h \overline{e'} are strongly neutral for each i \in \{1, \ldots, 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 \(e_i = t_i, e'_i = u_i\),

\[
\begin{align*}
\Sigma; \Gamma \vdash h \@ \overline{e}_{1,...,i-1} \downarrow V_i \text{ and } \Sigma \vdash V_i \downarrow \Pi B_i C_i, \text{ and } \\
\Sigma; \Gamma' \vdash h \@ \overline{e'}_{1,...,i-1} \downarrow V'_i \text{ and } \Sigma \vdash V_i \downarrow \Pi B'_i C'_i
\end{align*}
\]

\[\square\]


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 \(\Sigma \triangleq \Delta : \text{Set}, a : \Delta, f : \Delta \to \Delta\):

\[
\begin{align*}
\Sigma, \alpha : \text{Bool} \to \Delta, \beta : \Delta, x : \text{Bool} \vdash \beta \approx f(\alpha x) : \Delta \downarrow \Delta \\
\downarrow \vdash \alpha \text{ true} \approx a : \Delta \downarrow \Delta
\end{align*}
\]

This problem has solution \(\Theta = \Sigma, \alpha := \lambda x. a : \text{Bool} \to \Delta, \beta := f a : \Delta\). However, there is no clear way of finding this solution with the rules described so far:

(i) In the first constraint, \(x \in \text{FV}(f(\alpha x))\), but \(x\) is not in the arguments of \(\beta\); and in the second constraint, \(\alpha\) 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 \( \beta \). Thus, we may (correctly) assume that \( x \) is not actually used by \( \alpha \). Under this assumption we may “kill” the argument of \( \alpha \) as follows:

\[
\begin{align*}
\Sigma, \gamma : \mathbb{A}, \alpha := \lambda x. \gamma : \text{Bool} & \to \mathbb{A}, \beta : \mathbb{A}; x : \text{Bool} \vdash \beta \approx \Gamma (\alpha x) : A \ddagger \mathbb{A} \\
\cdot \vdash \alpha \text{ true} & \approx a : A \ddagger \mathbb{A}
\end{align*}
\]

By applying Rule-Schema 8 (term conversion) twice, we obtain:

\[
\begin{align*}
\Sigma, \gamma : \mathbb{A}, \alpha := \lambda x. \gamma : \text{Bool} & \to \mathbb{A}, \beta : \mathbb{A}; x : \text{Bool} \vdash \beta \approx \Gamma \gamma : A \ddagger \mathbb{A} \\
\cdot \vdash \gamma \approx a & : A \ddagger \mathbb{A}
\end{align*}
\]

And now, by applying Rule-Schema 2 (metavariable instantiation) twice, we obtain:

\[
\begin{align*}
\Sigma, \gamma := a : \mathbb{A}, \alpha := \lambda x. \gamma : \text{Bool} & \to \mathbb{A}, \beta := \Gamma \gamma : \mathbb{A} \ddagger \mathbb{A}
\end{align*}
\]

Finally, by Definition 2.143 (closing metasubstitution), \( \text{CLOSE}(\Sigma, \gamma := a : A, \alpha := \lambda x. \gamma : \text{Bool} \to A, \beta := \Gamma a : A) \downarrow \Theta' \), and \( \Theta'_{\Sigma, \alpha : \text{Bool} \to A, \beta : A} = \Theta \).

**Example 4.43** (Bad pruning). The approach in Example 4.42 is not always correct.

Consider the following problem, where \( \Sigma \cong \Delta : \text{Set}, a : \mathbb{A}, \Gamma : \mathbb{A} \to \mathbb{A} :\)

\[
\begin{align*}
\Sigma, \alpha : \mathbb{A} & \to \mathbb{A}, \beta : \mathbb{A}, \gamma : \mathbb{A} \to \mathbb{A}; x : \text{bool} \vdash \beta \approx \gamma (\alpha x) : A \ddagger \mathbb{A} \\
\cdot \vdash \alpha x & \approx x : A \ddagger \mathbb{A} \\
\cdot \vdash \gamma x & \approx a : A \ddagger \mathbb{A}
\end{align*}
\]

The problem has the solution \( \Theta \cong \Sigma, \alpha := \lambda x. x : \mathbb{A} \to \mathbb{A}, \beta := a : \mathbb{A}, \gamma := \lambda x. a : \mathbb{A} \to \mathbb{A} \).

If we kill the first argument of \( \alpha \), we obtain the following problem:

\[
\begin{align*}
\Sigma, \delta : \mathbb{A}, \alpha := \lambda x. \delta : \mathbb{A} & \to \mathbb{A}, \beta : \mathbb{A}, \gamma : \mathbb{A} \to \mathbb{A}; x : \text{bool} \vdash \beta \approx \gamma \delta : A \ddagger \mathbb{A} \\
\cdot \vdash \delta & \approx x : A \ddagger \mathbb{A} \\
\cdot \vdash \gamma x & \approx a : A \ddagger \mathbb{A}
\end{align*}
\]
Because metavariables can only be instantiated to closed terms, the constraint \( \vdash \delta \approx x : A \downarrow A \) is unsolvable. The resulting problem does not have the solution \( \Theta \). 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: \( \Sigma \vdash \text{kill}(\alpha, n) \mapsto \Sigma' \)). We say that killing the \( n \)-th argument of metavariable \( \alpha \) in signature \( \Sigma \) yields signature \( \Sigma' \) (written \( \Sigma \vdash \text{kill}(\alpha, n) \mapsto \Sigma' \)), if all the following hold:

(i) \( n \in \mathbb{N}, n \geq 1 \),

(ii) \( \Sigma \text{ sig}, \Sigma = \Sigma_1, \alpha : T, \Sigma_2 \) for some \( \Sigma_1, \Sigma_2 \) and \( T \).

(iii) \( \Sigma' = \Sigma_1, \beta : \Pi \bar{A}^{n-1} U, \alpha := \lambda \bar{x}^{n-1} . \lambda y. \beta \bar{x} : T, \Sigma_2 \) for some \( \beta, \bar{A} \) and \( U \) with \( \beta \notin \text{decls}(\Sigma) \); and

(iv) \( \Sigma_1; \vdash T \equiv \Pi \bar{A}^{n-1} \Pi BU^{(+1)} \) type for some \( B \).

**Lemma 4.45** (Well-formedness of killing). Assume \( \Sigma \vdash \text{kill}(\alpha, n) \mapsto \Sigma' \), where \( \Sigma = \Sigma_1, \alpha : T, \Sigma_2, \) and \( \Sigma' = \Sigma_1, \beta : T', \alpha := t : T, \Sigma_2 \) for some \( \Sigma_1, \Sigma_2, T, T' \) and \( t \). Then \( \Sigma' \) 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 \( \alpha \) 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 \( \Sigma \vdash \text{kill}(\alpha, n) \mapsto \Sigma' \) where for some \( T_\alpha, T_\beta \) and \( t_\alpha \), 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 \( \Theta \) be a metasubstitution such that \( \Theta \vdash \Sigma \).

If there is \( v \) and \( T \) such that \( \Theta ; \vdash \alpha \equiv \lambda^n v : T, \) with \( 0 \notin \text{FV}(v) \), then there are \( u_\beta \) and \( U_\beta \) such that, for \( \Theta' = \Theta, \beta := u_\beta : U_\beta \), we have:

(i) \( \Theta' \text{ wf} \),

(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]. □
4.5. A REDUCTION RULE TOOLKIT

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 \( \Theta; \Gamma \vdash \alpha \bar{f} \equiv \alpha \bar{f}' : A \), where \( \bar{f} = f_1 f_2 \ldots f_{1+m} \) and \( \bar{f}' = f'_1 f'_2 \ldots f'_{1+m} \). Also, assume that \( f_{n+1} = x \bar{e} \) and \( f'_{n+1} = y \bar{e}' \), and \( x \neq y \).

Then there are \( v \) and \( T \) such that \( \Theta; \cdot \vdash \alpha \equiv \lambda v : T \), where \( 0 \notin \text{fv}(v) \).

**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]. \( \square \)

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

\[
\Sigma; \Gamma' \vdash \alpha \bar{f} x = \alpha \bar{f}' y : A \rightarrow A' \\
\rightarrow \Sigma'; \Gamma' \vdash \beta \bar{g} \Rightarrow \beta \bar{g}' : A \rightarrow A'
\]

where

\( x \neq y \) \hspace{1cm} (1)

all terms in \( \bar{f} \), \( \bar{f}' \), \( \bar{g} \), \( \bar{g}' \) are irreducible \hspace{1cm} (2)

\( \Sigma \vdash \text{KILL}(\alpha, n+1) \mapsto \Sigma' \), where \( \Sigma = \Sigma_1, \alpha : U, \Sigma_2 \)

and \( \Sigma' = \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2 \) \hspace{1cm} (3)

Condition 3 ensures that the resulting problem is well formed, while conditions 1 and 2 ensure the resulting problem has the same solutions as the original problem.

**Proof of correctness.** 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]. \( \square \)

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).**

\[
\Sigma_1, \alpha : U, \Sigma_2; \Gamma' \vdash \alpha \bar{f} = \alpha \bar{g} : A \rightarrow A' \\
\rightarrow \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2; \Gamma' \vdash \beta \bar{g} \Rightarrow \beta \bar{g}' : A \rightarrow A'
\]

where

\( \bar{f} = f_1 \ldots f_{n} \) and \( \bar{g} = g_1 \ldots g_{n} \)

\( f_i = h \bar{e}, g_i = h' \bar{e}' \) \hspace{1cm} (4)

all terms in \( \bar{f} \) and \( \bar{g} \) are irreducible

\( \Sigma_1, \alpha : U, \Sigma_2 \vdash \text{KILL}(\alpha, i) \mapsto \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2 \)

\( i \in \{1, \ldots, n\} \)
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 \( \vec{f} = \vec{f}_1^m (y(+) \vec{e}) \vec{f}_2^m \) where, for all \( f \in \vec{f}_1 \) or \( f \in \vec{f}_2 \), \( f \) is irreducible, and let \( \alpha \) be a metavariable.

Assume that \( t_2[\alpha \vec{f}]^k, \Theta; \Gamma \vdash t_1 \equiv t_2 : A \), and \( y \notin \text{fv}(t_1) \).
Then there exist \( \Delta \) and \( B \) such that \( |\Delta| = k \Theta; \Gamma, \Delta \vdash \alpha \vec{f} : B \), and there exist \( v_0 \) and \( T \) such that \( \Theta; \cdot \vdash \alpha \equiv \lambda^{n+1}.v_0 : T \) and \( 0 \notin \text{fv}(v_0) \).

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).

\[
\Sigma; \Gamma \vdash v \approx t : A \leftrightarrow A' \Rightarrow \Sigma'; \Gamma \vdash v \approx t : A \leftrightarrow A'
\]

where
\[
t[\alpha \vec{f}_1^m (y \vec{e}) \vec{f}_2^m]_k \text{ for } k \in \mathbb{N} \tag{1}
\]
\[
y \notin \text{fv}(v) \tag{2}
\]
\[
every \ f \in \vec{f}_1 \text{ or } f \in \vec{f}_2 \text{ is irreducible} \tag{3}
\]
\[
\Sigma \vdash \text{kil}(\alpha, n + 1) \mapsto \Sigma', \text{ where } \Sigma = \Sigma_1, \alpha : U, \Sigma_2 \tag{4}
\]

and \( \Sigma' = \Sigma_1, \beta : T, \alpha := u : U, \Sigma_2 \)


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 \( y_2 : V[y_1] \).
4.5. A REDUCTION RULE TOOLKIT

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

$$\beta$$ is fresh for $$\Sigma$$

$$\Sigma_1 ; \cdot \vdash T \equiv \Pi \tilde{A}^n \Pi (\Sigma U V) B \text{ type}$$

$$B ((+2) + 1)(1, 0)/0 \Downarrow B'$$

$$T_\beta = \Pi \tilde{A}^n \Pi U IV B'$$

$$t_\alpha = \lambda x^n y. \beta \tilde{x}^n (y . \pi_1) (y . \pi_2)$$

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 $$\Sigma ; \Gamma_1 \vdash \alpha (x, y) \approx t : A_1 \uparrow A_2$$. Cases where metavariables are applied to $$\lambda$$-abstractions, such as constraints of the form $$\Sigma ; \Gamma_1 \vdash \alpha (\lambda x . y . x) \approx t : A_1 \uparrow A_2$$, may sometimes be tackled by using Rule-Schema 8 (term conversion) to $$\eta$$-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, $$\eta$$-contraction is type-preserving, and all well-typed terms are judgmentally equal to their $$\eta$$-contracted form, when it exists.

4.5.11 Metavariavle $$\eta$$-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:

$$A : \text{Set}, a : A, b : A, \alpha : A \rightarrow A \rightarrow A \times A;$$

$$x : A \uparrow A, y : A \uparrow A \vdash \alpha x y . \pi_1 \approx x : A \uparrow A,$$

$$x : A \uparrow A, y : A \uparrow A \vdash \alpha x y . \pi_2 \approx y : A \uparrow A$$

Rule-Schema 2 (metavariable instantiation) cannot be applied to any of the constraints, because the metavariable $$\alpha$$ is applied to eliminators which are not variables ($$\pi_1$$ and $$\pi_2$$, respectively). However, if we instantiate $$\alpha$$ with $$\alpha := \lambda x . \lambda y . (\alpha_1 x y, \alpha_2 x y) : A \rightarrow A \rightarrow A \times A$$ (where $$\alpha_1 : A \rightarrow A \rightarrow A$$ and $$\alpha_2 : A \rightarrow A \rightarrow A$$ are fresh metavariables) and apply Rule-Schema 8 (term
conversion) to normalize the constraints, then the problem becomes:

\[
\begin{align*}
\mathbb{A} : \text{Set}, a : \mathbb{A}, b : \mathbb{A}, \\
\alpha_1 : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \\
\alpha_2 : \mathbb{A} \to \mathbb{A} \to \mathbb{A}, \\
\alpha := \lambda x. \lambda y. (\alpha_1 x \, y, \alpha_2 x \, y) : \mathbb{A} \to \mathbb{A} \to \mathbb{A} \times \mathbb{A}; \\
x : \mathbb{A} \alpha \mathbb{A}, y : \mathbb{A} \alpha \mathbb{A} \vdash \alpha_1 x \, y \approx x : \mathbb{A} \alpha \mathbb{A}, \\
x : \mathbb{A} \alpha \mathbb{A}, y : \mathbb{A} \alpha \mathbb{A} \vdash \alpha_2 x \, y \approx y : \mathbb{A} \alpha \mathbb{A}
\end{align*}
\]

Then Rule-Schema 2 can be applied both constraints.

\begin{itemize}
\item Example 4.50. Consider the following unification problem:

\[
\begin{align*}
\mathbb{A} : \text{Set}, a : \mathbb{A}, \\
\mathbb{B} : \mathbb{A} \to \text{Set}, b : \mathbb{B} \, a, \\
\alpha : \mathbb{A} \to \mathbb{B} \, a \to \Sigma \mathbb{A}(\mathbb{B} \, a); \\
\cdot \vdash \alpha \, b \, \pi_1 \approx \alpha : \mathbb{A} \times \mathbb{A} \\
x : \mathbb{A} \alpha \mathbb{A}, y : \mathbb{B} \alpha \mathbb{A} \vdash \alpha \, x \, y \approx \langle x, y \rangle : \Sigma \mathbb{A}(\mathbb{B} \, a) \mathbb{A}(\alpha \, b \, \pi_1))
\end{align*}
\]

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

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

\[
\begin{align*}
\mathbb{A} : \text{Set}, a : \mathbb{A}, \\
\mathbb{B} : \mathbb{A} \to \text{Set}, b : \mathbb{B} \, a, \\
\alpha_1 : \mathbb{A} \to \mathbb{B} \, a \to \mathbb{A}, \\
\alpha_2 : \mathbb{A} \to \mathbb{B} \, a \to \mathbb{B} \, a, \\
\alpha := \lambda x. \lambda y. (\alpha_1 x \, y, \alpha_2 x \, y) : \mathbb{A} \to \mathbb{B} \, a \to \Sigma \mathbb{A}(\mathbb{B} \, a); \\
\cdot \vdash \alpha_1 \, b \approx \alpha : \mathbb{A} \times \mathbb{A} \\
x : \mathbb{A} \alpha \mathbb{A}, y : \mathbb{B} \alpha \mathbb{A} \vdash \alpha_1 \, x \, y \approx x : \mathbb{A} \alpha \mathbb{A} \times \mathbb{A} \\
x : \mathbb{A} \alpha \mathbb{A}, y : \mathbb{B} \alpha \mathbb{A} \vdash \alpha_2 \, x \, y \approx y : \mathbb{B} \alpha \mathbb{A} \times \mathbb{A}(\alpha_1 \, b)
\end{align*}
\]

Then, by applying Rule-Schema 2 (metavariable instantiation), we can instantiate \( \alpha_1 \) 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:
4.5. A REDUCTION RULE TOOLKIT

Rule-Schema 19 (Metavariable $\eta$-expansion).

$\Sigma_1, \alpha : U, \Sigma_2 ; \Box \Rightarrow$

$\Sigma_1, \alpha_1 : \Pi T. A, \alpha_2 : \Pi T. B[\alpha_1 \vec{x}], \alpha := \lambda \vec{x}^n. \langle \alpha_1 \vec{x}, \alpha_2 \vec{x} \rangle : U, \Sigma_2 ; \Box$

where

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

$\Sigma_1 ; \vdash U \equiv \Pi T^n. \Sigma AB$ type

$\alpha_1$ and $\alpha_2$ 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 ($\Pi$ inversion), Lemma 2.53 ($\Sigma$ 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 meta-substitution), 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 $\eta$-equality for pairs) and Lemma 4.33 (general $\eta$-equality for $\Pi$-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 $\Sigma$-types from metavariable arguments, Abel and Pientka [2] show how one can remove $\Sigma$-types from constraint contexts. This will help us instantiate metavariables whose arguments contain projections.

**Example 4.51.** Consider the following problem:

$\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B};$

$x : (\mathbb{A} \times (\mathbb{A} \to \mathbb{B})) \vdash (\mathbb{A} \times (\mathbb{A} \to \mathbb{B})) \vdash \alpha (x.\pi_1) (x.\pi_2) \approx (x.\pi_2) (x.\pi_1) : \mathbb{B} \uparrow \mathbb{B}$

Rule-Schema 2 (metavariable instantiation) may not be applied, as the arguments of $\alpha$ 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:

$\mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}, \alpha : \mathbb{A} \to (\mathbb{A} \to \mathbb{B}) \to \mathbb{B};$

$x_1 : \mathbb{A} \uparrow \mathbb{A}, x_2 : (\mathbb{A} \to \mathbb{B}) \vdash (\mathbb{A} \to \mathbb{B}) \vdash \alpha x_1 x_2 \approx x_2 x_1 : \mathbb{B} \uparrow \mathbb{B}$

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).

\[
\Sigma;\Gamma_1 \vdash \Gamma_2, x : \Sigma U_1 V_1 \vdash \Sigma U_2 V_2, \Delta_1 \vdash \Delta_2 \vdash t_1 \approx t_2 : A_1 \vdash A_2 \Rightarrow
\Sigma;\Gamma_1 \vdash \Gamma_2, x_1 : U_1 \vdash U_2, x_2 : V_1 \vdash V_2, \Delta_1' \vdash \Delta_2' \vdash t_1' \approx t_2' : A_1' \vdash A_2'
\]

where

\[
(\Delta_1 \vdash t_1 : A_1) \langle (1,0) \rangle [\sigma_{\gamma_1 \gamma_2}] \Downarrow (\Delta_1' \vdash t_1' : A_1')
\]

\[
(\Delta_2 \vdash t_2 : A_2) \langle (1,0) \rangle [\sigma_{\gamma_1 \gamma_2}] \Downarrow (\Delta_2' \vdash t_2' : A_2')
\]

Informally, we have \([\Delta_1' = \Delta_1[\sigma_{\gamma_1 \gamma_2}], t_1' = t_1[\sigma_{\gamma_1 \gamma_2}], t_2' = t_2[\sigma_{\gamma_1 \gamma_2}]]\), 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 \( r \), \( t[\sigma_{xy}] \Downarrow r \). Then:

- If \( z \notin \text{FV}(t) \), then \( \text{FV}(r) = \text{FV}_z(t) \).
- If \( z \in \text{FV}(t) \), then \( \text{FV}_z(t) \subseteq \text{FV}(r) \subseteq \text{FV}_z(t) \cup \{x,y\} \).

*Proof.* By induction on the derivation for \( t[\sigma_{xy}] \Downarrow r \), and Remark 2.28 (renaming and free variables). \(\Box\)

**Lemma 4.53** (Free variables in substitution by irreducible). Let \( t \) be a term such that, for some \( r \), \( t[\sigma_{xe}] \Downarrow r \), where \( e = \pi_1 \) or \( e = \pi_2 \). Then:

- If \( z \notin \text{FV}(t) \), then \( \text{FV}(r) = \text{FV}_z(t) \).
- If \( z \in \text{FV}(t) \), then \( \text{FV}_z(t) \subseteq \text{FV}(r) \subseteq \text{FV}_z(t) \cup \{x\} \).

*Proof.* By induction on the derivation for \( t[\sigma_{xe}] \Downarrow r \), and Remark 2.28 (renaming and free variables). \(\Box\)


**Remark 4.54.** Note that, if \( \Sigma;\mathcal{C} \text{ wf} \), then by Postulate 1 (typing of hereditary substitution), \( \Delta_1', \Delta_2', t_1', t_2', A_1' \) 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.

\[
\begin{align*}
\text{U} & \triangleq \text{Bool} \times \text{Bool} : \text{Set} \\
\text{set} & \triangleq \langle \text{true}, \text{false} \rangle : \text{U} \\
\text{el} (b : \text{Bool}) & \triangleq \langle \text{false}, b \rangle : \text{U} \\
\text{El} (u : \text{U}) & \triangleq \text{if} (\lambda. \text{Bool}) (u . \pi_1) \text{ true} (u . \pi_2) : \text{Bool}
\end{align*}
\]

These shorthands mimic some sort of inductive data type (U) with two constructors (set, el) and an inductively-defined function of type U → 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 $\alpha$ occurs in a term $(\lambda y.(\alpha x))$, which has to be unified with another term $(\lambda y.\text{set})$, whose type $(\mathbb{F} (\text{El} (\alpha x)) \rightarrow \text{U})$ contains the same metavariable $\alpha$.

\[
\begin{align*}
\mathbb{F} : \text{Bool} \rightarrow \text{Set}, \mathbb{P} : (X : \text{Set}) \rightarrow (x : X) \rightarrow \text{Set}, \alpha : \text{Bool} \rightarrow \text{U} \\
x : \text{Bool} \vdash \mathbb{P} (\mathbb{F} (\text{El} (\alpha x)) \rightarrow \text{U}) (\lambda y.\text{set}) : \text{Set} \cong \mathbb{P} (\mathbb{F} \text{ true} \rightarrow \text{U}) (\lambda y.(\alpha x)) : \text{Set}
\end{align*}
\]

By Definition 4.22, the constraint in Example 4.55 decomposes into two internal constraints, yielding the problem $\Sigma(0) ; C_1(0), C_2(0)$, where:

\[
\begin{align*}
\Sigma(0) & \triangleq \mathbb{F} : \text{Bool} \rightarrow \text{Set}, \mathbb{P} : (X : \text{Set}) \rightarrow (x : X) \rightarrow \text{Set}, \alpha : \text{Bool} \rightarrow \text{U} \\
C_1(0) & \triangleq x : \text{Bool} \vdash \text{set} \cong \text{Set} : \text{Set} \\
C_2(0) & \triangleq x : \text{Bool} \vdash (\lambda y.\text{set}) \cong (\lambda y.(\alpha x)) : \text{Set} \\
\end{align*}
\]

The constraints can be refined following the steps below:

1. By Rule-Schema 1 (syntactic equality), $\Sigma(0) ; C_1(0), C_2(0) \rightarrow \Sigma(0) ; \Box$. By Rule-Schema 14 (strongly neutral terms), $\Sigma(0) ; C_1(0), C_2(0) \rightarrow \Sigma(1), \Sigma(1), \Sigma(1), \Sigma(1)$, where:

\[
\begin{align*}
C_1(1) & \triangleq x : \text{Bool} \vdash \mathbb{F} (\text{El} (\alpha x)) \rightarrow \text{U} \cong \mathbb{F} \text{ true} \rightarrow \text{U} : \text{Set} \\
C_2(1) & \triangleq x : \text{Bool} \vdash (\lambda y.\text{set}) \cong (\lambda y.(\alpha x)) : \text{Set} \\
\end{align*}
\]

Therefore, $\Sigma(0) ; C_1(0), C_2(0) \rightarrow \Sigma(0) ; C_1(1), C_2(1)$.
2. By Rule-Schema 3 (injectivity of $\Pi$), $\Sigma^{(0)}; \mathcal{C}_1^{(1)} \leadsto \Sigma^{(0)}; \mathcal{C}_1^{(2)}, \mathcal{C}_2^{(2)}$, where:

\[
\mathcal{C}_1^{(2)} \equiv x : \text{Bool} \vdash \mathcal{F}(\text{El}(\alpha x)) \equiv \mathcal{F}\text{true : Set}
\]

\[
\mathcal{C}_2^{(2)} \equiv x : \text{Bool}, y : \mathcal{F}(\text{El}(\alpha x)) \vdash \text{true : Set}
\]

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

\[
\mathcal{C}_3^{(2)} \equiv x : \text{Bool}, y : (\mathcal{F}(\text{El}(\alpha x))) \vdash \text{set} \approx \alpha x : \text{U}\text{U}
\]

Therefore, $\Sigma^{(0)}; \mathcal{C}_1^{(1)}, \mathcal{C}_2^{(1)} \leadsto \Sigma^{(0)}; \mathcal{C}_1^{(3)}, \mathcal{C}_2^{(3)}$.

3. By Rule-Schema 14 (strongly neutral terms), $\Sigma^{(0)}; \mathcal{C}_1^{(2)} \leadsto \Sigma^{(0)}; \mathcal{C}_1^{(3)}$, where:

\[
\mathcal{C}_1^{(3)} \equiv x : \text{Bool} \vdash \text{El}(\alpha x) \approx \text{true : Bool}
\]

By Rule-Schema 1 (syntactic equality), $\Sigma^{(0)}; \mathcal{C}_2^{(2)} \leadsto \Sigma^{(1)}; \square$.

By Rule-Schema 2 (metavariable instantiation): $\Sigma^{(1)}; \mathcal{C}_3^{(2)} \leadsto \Sigma^{(1)}; \square$, where:

\[
\Sigma^{(1)} \equiv \mathcal{F} : \text{Bool} \rightarrow \text{Set}, \mathcal{P} : (X : \text{Set}) \rightarrow (x : X) \rightarrow \text{Set}, \alpha := \lambda x.\text{set} : \text{Bool} \rightarrow \text{U}
\]

Therefore, $\Sigma^{(0)}; \mathcal{C}_1^{(2)}, \mathcal{C}_2^{(2)} \leadsto \Sigma^{(1)}; \mathcal{C}_1^{(3)}$.

4. By Rule-Schema 8 (term conversion), $\Sigma^{(1)}; \mathcal{C}_1^{(3)} \leadsto \Sigma^{(1)}; \mathcal{C}_1^{(4)}$, where:

\[
\mathcal{C}_1^{(4)} \equiv x : \text{Bool} \vdash \text{true} \approx \text{true : Bool}
\]

5. Finally, by Rule-Schema 1 (syntactic equality), $\Sigma^{(1)}; \mathcal{C}_1^{(4)} \leadsto \Sigma^{(1)}; \square$.

Because $\Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)} \leadsto \Sigma^{(1)}; \square$ and $\Sigma^{(1)}$ is closed, by Theorem 4.31, there exists a unique solution $\Theta$ such that $\Theta \models \Sigma^{(0)}; \mathcal{C}_1^{(0)}, \mathcal{C}_2^{(0)}$, where:

\[
\Theta = \mathcal{F} : \text{Bool} \rightarrow \text{Set}, \mathcal{P} : (X : \text{Set}) \rightarrow (x : X) \rightarrow \text{Set}, \alpha := \lambda x.\text{set} : \text{Bool} \rightarrow \text{U}
\]

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 ways 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 $\text{Set} \upharpoonright \text{Set}$.

Consider the following signature:

$$
\Sigma_0 \triangleq \alpha_p : (A : \text{Set}) \rightarrow (P : \text{Set}) \rightarrow ((x : A) \rightarrow P x) \rightarrow ((x : A) \rightarrow P x)
$$

$$
\varepsilon : (A : \text{Set}) \rightarrow A \rightarrow A \rightarrow \text{Set}
$$

$$
\text{refl} : (A : \text{Set}) \rightarrow (x : A) \rightarrow \varepsilon A x x
$$

$$
\text{elim} : (A : \text{Set}) \rightarrow (x : A) \rightarrow (y : A) \rightarrow (P : (x' : A) \rightarrow (y' : A) \rightarrow \varepsilon A x'y' \rightarrow \text{Set}) \rightarrow ((x'' : A) \rightarrow P x'' x'' (\text{refl} A x'')) \rightarrow (z : \varepsilon A x y) \rightarrow P x y z
$$

$$
\Sigma \triangleq \Sigma_0, \alpha : (A : \text{Set}) \rightarrow (x : A) \rightarrow (P : A \rightarrow \text{Set}) \rightarrow P x \rightarrow P x \rightarrow P x
$$

$$
\beta : (A : \text{Set}) \rightarrow (x : A) \rightarrow (P : A \rightarrow \text{Set}) \rightarrow P x \rightarrow P x \rightarrow P x
$$

$$
\gamma : (A : \text{Set}) \rightarrow (x : A) \rightarrow (P : A \rightarrow \text{Set}) \rightarrow P x \rightarrow \gamma A x P p
$$

$$
\varepsilon : (A : \text{Set}) \rightarrow (x : A) \rightarrow (P : A \rightarrow \text{Set}) \rightarrow P x \rightarrow (x' : \gamma A x P p) \rightarrow (y' : \gamma A x P p) \rightarrow \varepsilon (\gamma A x P p) x'y' \rightarrow \text{Set}
$$

$$
\varphi : (A : \text{Set}) \rightarrow (x : A) \rightarrow (P : A \rightarrow \text{Set}) \rightarrow P x \rightarrow ((x'' : (\gamma A x P p)) \rightarrow P x'' x'' (\text{refl} (\gamma A x P p) x''))
$$

Let $\Gamma \triangleq A : \text{Set}, x : A, P : A \rightarrow \text{Set}, p : P x$, and $\vec{y} \triangleq A x P p$ in $\Gamma$ (that is, $\vec{y} \triangleq 3210$).

Consider the following constraints:

$$
\mathcal{C}_1 \triangleq \Gamma \vdash \varepsilon \vec{y} (\delta \vec{y}) (\delta \vec{y}) (\text{refl} (\gamma \vec{y}) (\delta \vec{y})) \approx P x \rightarrow P x : \text{Set}
$$

$$
\mathcal{C}_2 \triangleq \Gamma \vdash \alpha \vec{y} p \approx \text{elim} A x x (\lambda y. \lambda z. (P y \rightarrow P z)) (\lambda. \lambda z. z) (\text{refl} A x) p : P x
$$

$$
\mathcal{C}_3 \triangleq \Gamma \vdash \beta \vec{y} p \approx p : P x
$$

$$
\mathcal{C}_4 \triangleq \Gamma \vdash \text{elim} (\gamma \vec{y}) (\delta \vec{y}) (\delta \vec{y}) (\varepsilon \vec{y}) (\varphi \vec{y}) (\text{refl} (\gamma \vec{y}) (\delta \vec{y})) \approx \alpha \vec{y}:
$$

$$
(\varepsilon \vec{y} (\gamma \vec{y}) (\delta \vec{y}) (\delta \vec{y}) (\text{refl} (\gamma \vec{y}) (\delta \vec{y}))) \vdash (P x \rightarrow P x)
$$

$$
\mathcal{C}_5 \triangleq \Gamma \vdash \varphi \vec{y} (\delta \vec{y}) \approx \beta \vec{y}:
$$

$$
(\varepsilon \vec{y} (\gamma \vec{y}) (\delta \vec{y}) (\delta \vec{y}) (\text{refl} (\gamma \vec{y}) (\delta \vec{y}))) \vdash (P x \rightarrow P x)
$$

As they stand, there is no clear way to make progress using only the rules in §4.5. The arguments to the metavariables $\varepsilon$, $\alpha$ and $\beta$ 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):

\[
\Sigma' \equiv \Sigma_0, \beta : ", \gamma : ", \delta : ", \epsilon : ", \varphi : ", \\
\alpha := \lambda \bar{y}. \text{elim} (\gamma \bar{y}) (\delta \bar{y}) (\delta \bar{y}) (\epsilon \bar{y}) (\varphi \bar{x}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})) : ">

We do not have \( \Sigma' \text{ sig} \). If this were the case, from the definition of well-formed signature, we would need to have, in particular:

\[
\beta : ", \gamma : ", \delta : ", \epsilon : ", \varphi : "; \vdash \\
\lambda \bar{y}. \text{elim} (\gamma \bar{y}) (\delta \bar{y}) (\delta \bar{y}) (\epsilon \bar{y}) (\varphi \bar{x}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})) : \\
(A : \text{Set}) \rightarrow (x : A) \rightarrow (P : A \rightarrow \text{Set}) \rightarrow P x \rightarrow P x \rightarrow P x
\]

By Corollary 2.58 and Postulate 10, this would entail:

\[
\beta : ", \gamma : ", \delta : ", \epsilon : ", \varphi : "; A : \text{Set}, x : A, P : A \rightarrow \text{Set}, z : P x \vdash \\
\text{elim} (\gamma \bar{y}) (\delta \bar{y}) (\delta \bar{y}) (\epsilon \bar{y}) (\varphi \bar{x}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})) : P x \rightarrow P x
\]

By the typing rules, we have:

\[
\beta : ", \gamma : ", \delta : ", \epsilon : ", \varphi : "; A : \text{Set}, x : A, P : A \rightarrow \text{Set}, z : P x \vdash \\
\text{elim} (\gamma \bar{y}) (\delta \bar{y}) (\delta \bar{y}) (\epsilon \bar{y}) (\varphi \bar{x}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})) : \\
(\epsilon \bar{y} (\gamma \bar{y}) (\delta \bar{y}) (\delta \bar{y}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})))
\]

By Lemma 2.75, we have that \( P x \rightarrow P x \) and \( \epsilon \bar{y} (\gamma \bar{y}) (\delta \bar{y}) (\delta \bar{y}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})) \) would need to be equal as types. However, these types cannot become equal until the metavariable \( \epsilon \) 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:

\[
\mathcal{C}_1 \equiv " \\
\mathcal{C}_2 \equiv \Gamma \vdash \text{elim} (\gamma \bar{y}) (\delta \bar{y}) (\epsilon \bar{y}) (\varphi \bar{x}) (\text{refl} (\gamma \bar{y}) (\delta \bar{y})) \bar{y} p \approx \\
\text{elim} A x x (\lambda y. \lambda z. \lambda (P y \rightarrow P z)) (\lambda \lambda z. z) (\text{refl} A x) p : P x
\]

\[
\mathcal{C}_3 \equiv " \\
\mathcal{C}_5 \equiv "
\]

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:
\[ \Sigma'' \equiv \Sigma_0, \]
\[ \beta : "', \gamma : \lambda \vec{y}. A : ", \delta : \lambda \vec{y}. x, \]
\[ \varepsilon := (\lambda \vec{y}. \lambda z_1. \lambda z_2. \lambda.(P \, z_1 \rightarrow P \, z_2)) : "', \]
\[ \varphi := \lambda \vec{y}. \lambda. \lambda z. z : "', \]
\[ \alpha := \lambda \vec{y}. \text{elim}(\gamma \, \vec{y}) (\delta \, \vec{y}) (\delta \, \vec{y}) (\varepsilon \, \vec{y}) (\varphi \, \vec{x}) \]
\[ \mathcal{C}'_{1} \equiv \Gamma \vdash P \, x \rightarrow P \, x \approx P \, x \rightarrow P \, x : " \]
\[ \mathcal{C}'_{3} \equiv " \]
\[ \mathcal{C}'_{5} \equiv \Gamma \vdash \lambda x.x \approx \beta \, \vec{y} : (P \, x \rightarrow P \, x) \downarrow (P \, x \rightarrow P \, x) \]

By applying Rule-Schema 2 (metavariable instantiation) we obtain the body of \( \beta \). 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:

\[ \Sigma''' \equiv \Sigma_0, \]
\[ \alpha := \lambda \vec{y}. \text{elim} A \, x \, x \,(\lambda z_1. \lambda z_2. \lambda.(P \, z_1 \rightarrow P \, z_2)) \,(\lambda. \lambda z. z) : " \]
\[ \beta := \lambda \vec{y}. \lambda x. x : "', \]
\[ \gamma := \lambda \vec{y}. A : "', \]
\[ \delta := \lambda \vec{y}. x : "', \]
\[ \varepsilon := \lambda \vec{y}. \lambda z_1. \lambda z_2. \lambda.(P \, z_1 \rightarrow P \, z_2) : "', \]
\[ \varphi := \lambda \vec{y}. \lambda. \lambda z. z : "'. \]

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 \( \Sigma_1 \) is defined as follows:

\[
\Sigma_1 \overset{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \alpha : \mathbb{A}
\]

And let \( \Sigma'_1 \) be defined as follows:

\[
\Sigma'_1 \overset{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \alpha := \mathbb{a} : \mathbb{A}
\]

Because, in the empty context, \( \mathbb{a} \) is the only value of type \( \mathbb{A} \), the rule \( \Sigma_1; \square \Rightarrow \Sigma'_1; \square \) is in fact a correct rule (we will not prove this).

Now, let \( \Theta_1 \overset{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a} : \mathbb{A}, \alpha := \mathbb{a} : \mathbb{A} \). Then, \( \Theta_1 \) 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 \( \Sigma_2; \vec{C} \):

\[
\Sigma_2; \vec{C} \overset{\text{def}}{=} \Sigma_1, \mathbb{b} : \mathbb{A}; \vec{C} \quad \vdash \alpha \equiv \mathbb{b} : \mathbb{A}^{\mathbb{A}}
\]

If we generalized the inference \( \Sigma_1; \square \Rightarrow \Sigma'_1; \square \) to the extended problem \( \Sigma_2; \vec{C} \), we would obtain the unification rule \( (\Sigma_1, \mathbb{b} : \mathbb{A}; \vec{C} \Rightarrow \Sigma'_1, \mathbb{b} : \mathbb{A}; \vec{C}) \). Applying this rule results in the problem \( (\Sigma_1, \mathbb{b} : \mathbb{A}; \vec{C}) \), which has one unique solution \( \Theta_2 \overset{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{a}, \mathbb{b} : \mathbb{A}, \alpha := \mathbb{b} : \mathbb{A} \).

By definition, \( \Theta_2; \vdash \Theta_2 \neq \mathbb{b} : \mathbb{A} \), which means \( \Theta_2 \) is not a solution for \( \Sigma'_1, \mathbb{b} : \mathbb{A}; \vec{C} \). The rule \( (\Sigma_1, \mathbb{b} : \mathbb{A}; \vec{C} \Rightarrow \Sigma'_1, \mathbb{b} : \mathbb{A}; \vec{C}) \) is thus not complete, and thus not correct.

Because it does not remain correct under extensions, we say that the rule \( \Sigma_1; \square \Rightarrow \Sigma'_1; \square \) 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 \( \Sigma; \vec{C} \Rightarrow \Sigma'; \vec{D} \). Let \( \Sigma_1 \) be such that \( \Sigma, \Sigma_1 \text{ sig} \) and \( \text{decls}(\Sigma_1) \cap \text{decls}(\Sigma') = \emptyset \). Then \( \Sigma, \Sigma_1; \vec{C} \Rightarrow \Sigma', \Sigma_1; \vec{D} \).

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. \( \square \)

This open-world assumption can be generalized to sequences of rule applications:

Remark 4.57 (Open-world assumption for problem reduction). Suppose \( \Sigma; \vec{C} \Rightarrow \Sigma'; \vec{D} \). Let \( \Sigma_1 \) be such that \( \Sigma, \Sigma_1 \text{ sig} \) and \( \text{decls}(\Sigma_1) \cap \text{decls}(\Sigma') = \emptyset \). and let \( \vec{E} \) be such that \( \Sigma, \Sigma_1; \vec{E} \text{ wf} \). Then, \( \Sigma, \Sigma_1; \vec{C}, \vec{E} \Rightarrow \Sigma', \Sigma_1; \vec{D}, \vec{E} \).
4.8. BEYOND CORRECTNESS

Proof. By induction on the derivation for $\Sigma; \vec{c} \rightsquigarrow^* \Sigma'; \vec{D}$, 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 $\Sigma; \vec{c}$ is unsolvable if there does not exist $\Theta$ such that $\Theta \models \Sigma; \vec{c}$.

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 $\Sigma; \vec{c} \rightsquigarrow^n \Sigma'; \vec{D}$, and there is no solution $\Theta'$ such that $\Theta' \models \Sigma'; \vec{D}$, then there is no solution $\Theta$ such that $\Theta \models \Sigma; \vec{c}$.

Proof. Proceed by induction on $n$.

- Case 0: Proceed by contradiction; assume there exists $\Theta$ such that $\Theta \models \Sigma; \vec{c}$. Then $\Theta' = \Theta$ fulfills $\Theta' \models \Sigma'; \vec{D}$, which is a contradiction. Therefore, there is no $\Theta$ such that $\Theta \models \Sigma; \vec{c}$.

- Case $1 + n$: Then we have $\Sigma; \vec{c} \rightsquigarrow^n \Sigma''; \vec{c} \rightsquigarrow \Sigma'; \vec{D}$. By Lemma 4.29 (correctness of problem reduction), $\Sigma''; \vec{c} \rightsquigarrow \Sigma'; \vec{D}$ is a correct rule; in particular, it is complete. Proceed by contradiction; assume there exists $\Theta''$ such that $\Theta'' \models \Sigma''; \vec{c}$. By completeness, there exists $\Theta'$ such that $\Theta'' = \Theta'_\Sigma$, and $\Theta' \models \Sigma'; \vec{D}$. This is a contradiction; therefore, there is no $\Theta''$ such that $\Theta'' \models \Sigma''; \vec{c}$. By the induction hypothesis, there is no $\Theta$ such that $\Theta \models \Sigma''; \vec{c}$.

In general, deciding whether a problem $\Sigma; \vec{c}$ 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:

$T_1 \overset{\text{def}}{=} \{c\}$

$T_2 \overset{\text{def}}{=} \{f \mid f \text{ is strongly neutral}\}$

$T_3 \overset{\text{def}}{=} \{\Pi AB\}$

$T_4 \overset{\text{def}}{=} \{\Sigma AB\}$

$T_5 \overset{\text{def}}{=} \{\text{Bool}\}$

$T_6 \overset{\text{def}}{=} \{\text{Set}\}$
Let $\Sigma; \vec{C}$ be a problem, and $C \in \vec{C}$ a constraint, $C = \Gamma' \vdash t \approx u : A'$. 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 $\Theta$ such that $\Theta \models \Sigma; \vec{C}$.

Proof. Assume there is $\Theta$ such that $\Theta \models \Sigma; \vec{C}$. Then, in particular, $\Theta; \Gamma \vdash t \equiv u : A$. By Postulate 14 (existence of a common reduct), there exists $r$ such that $\Theta; \Gamma \vdash t \rightarrow^*_{\delta\eta} r : A$ and $\Theta; \Gamma \vdash u \rightarrow^*_{\delta\eta} r : A$. 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. $\square$

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.

\[
\alpha : \text{Bool} \rightarrow \text{Bool} ; \cdot \vdash \alpha \text{true} \approx \text{true} : \text{Bool} \land \\
\cdot \vdash \alpha \text{true} \approx \text{false} : \text{Bool}
\]

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 $\eta$-equality is described below.

4.9.1 Singleton types with $\eta$-equality

If a type Unit has a single element $\langle \rangle$, then an $\eta$-rule for that type would have the form:

\[
\frac{\Sigma; \Gamma \vdash f : \text{Unit}}{\Sigma; \Gamma \vdash f \rightarrow^\eta \langle \rangle : \text{Unit}}
\]
Adding such a rule to the system is convenient, among other things, because, if one has a metavariable $\alpha : \Pi \vec{A}.\text{Unit}$, one can instantiate it directly as $\alpha := \lambda \vec{x}.\langle \rangle : \Pi \vec{A}.\text{Unit}$. If we consider the unit type above as a record type with no fields, this is a generalization of Rule-Schema 19 (metavariable $\eta$-expansion).

However, this rule has ramifications on the theory. In particular, one would have $\cdot ; x : \text{Bool} \rightarrow \text{Unit} \vdash x \text{true} \equiv x \text{false} : \text{Unit}$ without $\text{true} = \text{false}$, 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 $\eta$-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 $\eta$-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 $\Gamma_1 \vdash \Gamma_2 t \approx u : A_1 \equiv A_2$, 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 term 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`:

```agda
data ContextSide = LHS, RHS, Single, Both

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].

```agda
newtype ISet a = ISet { runISet :: IntSet }

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:

```agda
data Type = El Sort Term
```

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

```agda
data TwinT' a =
  Single { unSingleT :: OnSide 'Both a } |
  TwinT { twinPid :: ISet ProblemId }
```

5.2. HETEROGENEOUS CONTEXTS

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 \( \text{Set}\|\text{Set} \) 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:

```haskell
data Context_ = Empty
            | 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 \( \Gamma \equiv A_1 \|\!, A_1 \|\!, A_2 \|\!, A_2 \|\!, ..., A_n \|\!, A_n \|\!, \Gamma_1 \|\!, \Gamma_2 \). Let \( p_i \) be is the problem set of entry \( A_i \|\!, A_i \|\!, \Gamma_i \|\!, \Gamma_i \), and \( P_i = p_i \cup \cup p_i \) the problem set of the context \( \Gamma^{(i)} \equiv A_1 \|\!, A_1 \|\!, A_2 \|\!, A_2 \|\!, ..., A_i \|\!, A_i \|\!, \Gamma_1 \|\!, \Gamma_2 \). 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 \( P_i \), 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`:

```haskell
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 \( \Gamma \) of the current constraint or term.

In our development, terms also live in a single-sided context \( \Gamma \), but constraints may live in a twin context \( \Gamma_1 \uplus \Gamma_2 \). Enforcing this distinction in Agda 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 \( \Gamma \) is represented equivalently to the heterogeneous context \( \Gamma \uplus \emptyset \Gamma \).

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 happened 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, 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 term 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 appro-
priate 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 mecha-
nism 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 null noPids then
                SingleT . OnSide @'Both <$> (switchSide @'Single (reduce' a))
            else
                SingleT . OnSide @'Both <$> (switchSide @'Single (reduce' a))

                reduce' TwinT { twinPid = ∅
                                , twinLHS = OnSide @'LHS a}
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:

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

A constraint of the form \( \Gamma_1 \vdash \Gamma_2 \vdash t \approx u : A_1 \vdash A_2 \) can be represented as \( \text{ValueCmp} \_ \ (A_1 \vdash A_2) \ (\text{OnSide} \ @\text{LHS} t) \ (\text{OnSide} \ @\text{RHS} u) \). The context \( \Gamma_1 \vdash \Gamma_2 \) 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 \( \text{ElimCmp} \_ \) 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 \text{MonadConstraint} \( m \). Such a monad \( m \) gives access to the context of the current constraint being solved (that is, \text{MonadConstraint} \( m \) entails \text{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 \((\alpha_1 \lor \alpha_2 \lor \ldots \lor \alpha_n)\). 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 \(\{u_1, \ldots, u_n\}\)  
\hspace{1cm} -- notation: \(u_1 \land \ldots \land u_n\)

2 \hspace{1cm} | UnblockOnAny \(\{u_1, \ldots, u_n\}\)  
\hspace{1cm} -- notation: \(u_1 \lor \ldots \lor u_n\)

3 \hspace{1cm} | UnblockOnMeta \(\alpha\)  
\hspace{1cm} -- notation: \(\alpha\), where \(\alpha\) is a metavariable

4 \hspace{1cm} | UnblockOnProblem \(p\)  
\hspace{1cm} -- 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 \(\alpha\) has already been solved, and automatically turn UnblockOnMeta \(\alpha\) 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 “\(\alpha \land p\)”. The unblocker may be updated to \(\alpha\) when all the constraints in problem \(p\) are solved, and then to \(\top\) once \(\alpha\) is instantiated. The associated constraint will then be eligible to be retried.

We have preserved the workings of unblockers in Agda when implementing Agda\(_\varepsilon\), 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. 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 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, 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 ($\mathbb{N}^+ := \{1, \ldots, n\}$).

```haskell
1 data Unblocker = ...
2     | UnblockOnEffort \mathbb{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:

```haskell
1 unblocksOnEffort :: Unblocker \rightarrow \mathbb{N} \cup \{0, \infty\}
2 unblocksOnEffort (UnblockOnEffort e) = e
3 unblocksOnEffort UnblockOnMeta{} = \infty
4 unblocksOnEffort UnblockOnProblem{} = \infty
5 unblocksOnEffort (UnblockOnAll \{u_1, \ldots, u_n\}) =
6     \max \{\{0\} \cup \{unblocksOnEffort u_1, \ldots, unblocksOnEffort u_n\}\}
7 unblocksOnEffort (UnblockOnAny \{u_1, \ldots, u_n\}) =
8     \min \{\{\infty\} \cup \{unblocksOnEffort u_1, \ldots, unblocksOnEffort u_n\}\}
```

For instance, the unblocker $10$ requires effort $10$, as does $10 \lor \alpha$. The unblocker $\top$ requires effort $0$; the unblocker $1 \land \alpha$ requires effort $\infty$ (as the metavariable $\alpha$ 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.

```haskell
1 getEffortLevel :: MonadConstraint m \Rightarrow m \mathbb{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 whether the currently-allowed effort level is high enough:

```haskell
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 :: ℕ`: The effort level in the constraint environment when the constraint was postponed. If the unblocker `u` is `⊤`, 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 `⊤`), 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. `⊤`), and the effort level in its environment is increased by the required amount (δ):

```haskell
−− 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)]
```
--- · Ignore constraints that require infinity effort (∞)
--- (this means they cannot be unblocked by only increasing effort),
--- · Ignore constraints that require no extra effort (0)
--- (otherwise we could produce an infinite loop in Agda)

```
let (pcs₂, pcs₃) = partition (λ (e,_) → (e < ∞) && (e > 0)) pcs

case sortBy (compare `on` fst) (pcs₂) of
  [] → return False
  (δ, pc):pcs₄ → do
    --- Increase the effort by the required δ, and unblock the constraint
    let pc₁ = pc{e = e pc + δ, u = ⊤}
    Set the list of unsolved constraints to (pc₁:map snd pcs₄)
    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; thus the full ordered set of effort levels could be summarized as $0 < 10 < ∞$. 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. $\Gamma_1 ⊨ \alpha \vec{x} \equiv v : B_1 \dashv B_2$) requires the conditions $Σ; Γ_1 \vdash Γ_2 \vdash B_1 \equiv B \equiv B_2 : \text{Set}; \text{Set}$ (3a) and $Σ \vdash Γ_1 \equiv Γ \equiv \{x_i | i = 1, \ldots, n\} \cup b. \text{FV}(t) \cup b. \Gamma_2$, where $b = \text{FV}(B_1) \cap \text{FV}(B_2)$ (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.
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 $B_1\imp B_2$ returns $\top$, it means that $\Sigma;\Gamma_1\vdash B_1 \equiv \{B\} \equiv B_2 : \text{Set}\imp\text{Set}$ for some $B$. As per Definition 4.12, we call $B$ the interpolant. In Agda, 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, we need to know the intersection of the free variables of $\text{fv}(B_1)$ and $\text{fv}(B_2)$ (see Rule-Schema 2). Thus, we need to compute these two sets in order to check (3b). The more variables in $\text{fv}(B_1) \cap \text{fv}(B_2)$, 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 $\text{fv}(B_1) \cap \text{fv}(B_2)$, 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 $\alpha$, as instantiating the metavariable $\alpha$ to a suitable function (e.g. $\alpha := \lambda z. \text{true}$) 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 $\text{VarSetBlocked}$. In the actual implementation, $\text{VarSetBlocked}$ 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 $\text{VarSetBlocked}$ 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. $\text{fv}(t)$) or a type $A$ (i.e. $\text{fv}(A)$) 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.

```
checkTwinEqual :: (MonadConstraint m) ⇒ TwinT → m Unblocker
checkTwinEqual (SingleT t) = return AlwaysUnblock
checkTwinEqual TwinT{twinPid,twinLHS,twinRHS} = do
  pids ← {𝑝 ∈ twinPid | there are unsolved constraints associated with 𝑝}
  if null pids then
    return ⊤
  else
    bs ← ( strive 10 >>= \case
      ExtraEffort δ → return (UnblockOnEffort δ)
      Doable → Check whether twinLHS and twinRHS are convertible
    )
    return (((\𝑝∈pids UnblockOnProblem 𝑝) ∨ bs) ∨ bs)
```
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

```haskell
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, 𝐴_1‡ 𝐴_2 → In context Γ $ do
          −− If the variable 0 is present in both fvL and fvR ...
          if u_1 ≝ fvL 0 ≠ ⊤ and u_2 ≝ fvR 0 ≠ ⊤ then
            checkTwinEqual a >>= \case
            ⊤ → do
              −− Expand all metavariables in the type
              tA ← instantiateFull (𝐴_1‡ 𝐴_2)
              let fvA = freeVarsInterpolant tA
              −− Continue checking the other variable,
              −− plus those which must occur in any interpolant of 𝐴_1 and 𝐴_2.
              go (x ↦ (u_1 ∨ fvA(x)) ∧ fvL(x + 1)) (x ↦ (u_2 ∨ fvA(x)) ∧ fvR(x + 1))
              u_0 → return (u_0 ∨ u_1 ∨ u_2)
          −− If the variable 0 is present only in fvL ...
          else if u_1 ≝ fvL 0 ≠ ⊤ then
            fvA ← freeVarsBlocked <$> instantiateFull 𝐴_1
            go (x ↦ (u_1 ∨ 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 u_2 ≝ 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))
```

---

CHAPTER 5. IMPLEMENTATION
5.9. Unification of binders

When applying Rule-Schema 3 (injectivity of Π) for solving a constraint \( \Gamma_1 \vdash \Pi A_1 B_1 \approx \Pi A_2 B_2 : \mathsf{Set} \vdash \mathsf{Set} \), we compare the domains and the codomains. Listing 5.5 shows how this rule is applied in Agda.\(\varepsilon\). 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.\(\varepsilon\), 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} → do
  pids ← Get unsolved problems in twinPid
  if null pids then
    getConstraint_ >>= \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
      pids_2 ← Get unsolved problems in ctxPids
      −− We update the context entry so that it contains only
      −− those problems which are actually unsolved.
      In context (Entry pids_2 ty ctx) $ do
        −− If all problems in the twin and in the context are solved,
        −− to simplify the twin.
        𝜅 (if ISet.null pids_2 then (SingleT (OnSide @'Both twinLHS))
          −− Otherwise, we return the twin as—is. We update the twinPid
          else b{twinPid=pids_1})
      else
        𝜅 b{twinPid=pids_1}
```

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

```
... −− The domain of the \( \Pi \)-type contains additional information besides
  −− the type, such as modalities.
(Pi A_1 B_1, Pi A_2 B_2) → do
  −− Let ctx be the context of the current constraint
  ctx ← getContext_
  pid ← Attempt to solve the constraint ctx ⊢ A_1 ≈ A_2 : \text{Set} \dagger \text{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 A_1
                          , twinRHS = OnSide @'RHS A_2
                        })
  Solve the constraint ctx' ⊢ B_1 ≈ B_2 : \text{Set} \dagger \text{Set}
  ...
```
5.10. UNIFICATION OF SPINES

Constraints unifying strongly neutral terms (Definition 2.158) may be represented using the constructor \texttt{ElimCmp}. In this text, we use the notation \texttt{ElimCmp} \((A_1 t_{p_1}^1 A_2) (f_1 t_{p_2}^2 f_2) (t_1 e_1^1) (t_2 e_2^2)\) to represent the following constraint:

\begin{align}
\text{ElimCmp}_- & \ (A_1 t_{p_1}^1 A_2) (f_1 t_{p_2}^2 f_2) (t_1 e_1^1) (t_2 e_2^2) \\
\text{Done} & \\
\text{ElimCmp}_- & \ (A_1 t_{p_1}^1 A_2) (f_1 t_{p_2}^2 f_2) (t_1 e_1^1) (t_2 e_2^2) \\
& \ i \leftarrow \text{Attempt constraint } \Gamma \vdash t_1 \approx t_2 : A_1 t_{p_1}^1 A_2 \text{ and get the problem id} \\
& \text{Solve the constraint } \texttt{ElimCmp}_- (B_1[t_1] t_{p_2}^{1 \cup (i)} B_2[t_2]) ((f_1 t_1) t_{p_2}^{2 \cup (i)} (f_2 t_2)) (e_1^1) (e_2^2) \\
\text{ElimCmp}_- & \ (A_1 t_{p_1}^1 A_2) ((f_1 . \pi_1) t_{p_2} (f_2 . \pi)) (e_1) (e_2) \\
\text{ElimCmp}_- & \ (A_1 t_{p_1}^1 A_2) (f_1 t_{p_2}^2 f_2) (\pi_2 e_1^1) (\pi_2 e_2^2) \\
& \text{Solve } \texttt{ElimCmp}_- (B_1[f_1 . \pi_1] t_{p_2}^{1 \cup (i)} B_2[f_2 . \pi]) ((f_1 . \pi_2) t_{p_2} (f_2 . \pi_2)) (e_1^2) (e_2^2) \\
\text{ElimCmp}_- & \ (A_1 t_{p_1}^1 A_2) (f_1 t_{p_2}^2 f_2) (\pi_2 e_1^1) (\pi_2 e_2^2) \\
& \text{Solve } \texttt{ElimCmp}_- (B_1[f_1 . \pi_1] t_{p_2}^{1 \cup (i)} B_2[f_2 . \pi]) ((f_1 . \pi_2) t_{p_2} (f_2 . \pi_2)) (e_1^2) (e_2^2) \\
\text{ElimCmp}_- & \ (A_1 t_{p_1}^1 A_2) (f_1 t_{p_2}^2 f_2) (\pi_2 e_1^1) (\pi_2 e_2^2) \\
\text{Done} & \\
\end{align}

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.

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 \(\Gamma_1 \vdash \overline{h e_1^1} t \overline{e_2^2} \approx \overline{h e_1^2} . \pi_1 e_2^2 : T_1\), with \(\overline{e_1^1}\) and \(\overline{e_1^2}\) having the same length, and ditto for \(\overline{e_2^2}\) and \(\overline{e_2^1}\). In Agda-\(\varepsilon\), 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-\(\varepsilon\), 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 $\eta$-equality

Agda implements an $\eta$-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

```plaintext
1 isSingletonRecord :: (MonadConstraint m) ⇒ Type → m Bool
2 isSingletonRecord_ :: (MonadConstraint m) ⇒ TwinT → m Bool
3 isSingletonRecord_ 𝐴₁‡𝐴₂ = do
4   x ← switchSide @'LHS (isSingletonRecord 𝐴₁)
5   if x then
6     return True
7   else
8     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
<table>
<thead>
<tr>
<th>Task</th>
<th>Time (weeks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infrastructure (§5.1)</td>
<td>3</td>
</tr>
<tr>
<td>Unification of binders (§5.9)</td>
<td>2</td>
</tr>
<tr>
<td>Metavariable instantiation (§5.7)</td>
<td>1</td>
</tr>
<tr>
<td>Type-directed equality on terms (§5.12)</td>
<td>2</td>
</tr>
<tr>
<td>Context currying (Rule-Schema 20)</td>
<td>1</td>
</tr>
<tr>
<td>Unification of elimination spines (§5.10)</td>
<td>4</td>
</tr>
<tr>
<td>Head-injectivity analysis (§5.12)</td>
<td>1½</td>
</tr>
<tr>
<td>Projection-like functions (§5.12)</td>
<td>1½</td>
</tr>
<tr>
<td>Fixing of major bugs, refactoring and testing</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>18</strong></td>
</tr>
</tbody>
</table>

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.$\varepsilon$ 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
6.2. FUNCTIONAL TESTING

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

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 intermittent 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Agda.ε</th>
<th>Δ%</th>
<th>Agda (baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>agda-prelude</td>
<td>110 ± 1.2</td>
<td>+5.1% ... +1.8%</td>
<td>106 ± 1.2</td>
</tr>
<tr>
<td>agda-stdlib</td>
<td>531 ± 7.8</td>
<td>+0.64% ... -2.8%</td>
<td>537 ± 5.3</td>
</tr>
<tr>
<td>HoTT-Intro</td>
<td>223 ± 2.2</td>
<td>+5.9% ... +3.1%</td>
<td>214 ± 2.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Project</th>
<th>Agda.ε</th>
<th>Δ%</th>
<th>Agda (baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>agda-prelude</td>
<td>683 ± 15</td>
<td>+6% ... -0.92%</td>
<td>666 ± 17</td>
</tr>
<tr>
<td>agda-stdlib</td>
<td>1870 ± 13</td>
<td>-3.3% ... -4.8%</td>
<td>1950 ± 8.2</td>
</tr>
<tr>
<td>HoTT-Intro</td>
<td>2100 ± 37</td>
<td>-2.4% ... -6.9%</td>
<td>2210 ± 36</td>
</tr>
</tbody>
</table>

**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
6.5. LIMITATIONS AND FUTURE WORK

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Agda.( \varepsilon )</th>
<th>( \Delta % )</th>
<th>Agda (baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>categories/Categories</td>
<td>0.441 ± 0.012</td>
<td>+4.5% ... -2.4%</td>
<td>0.436 ± 0.01</td>
</tr>
<tr>
<td>categories/Language</td>
<td>0.553 ± 0.012</td>
<td>+6.9% ... +0.22%</td>
<td>0.534 ± 0.013</td>
</tr>
<tr>
<td>categories/Language3</td>
<td>0.373 ± 0.0089</td>
<td>+4.4% ... -2.3%</td>
<td>0.37 ± 0.0092</td>
</tr>
<tr>
<td>categories/Primitive</td>
<td>0.271 ± 0.0057</td>
<td>+7% ... +0.55%</td>
<td>0.261 ± 0.0062</td>
</tr>
<tr>
<td>examples/Cwf</td>
<td>2.65 ± 0.065</td>
<td>+5% ... -1.9%</td>
<td>2.61 ± 0.064</td>
</tr>
<tr>
<td>examples/FunctorComposition</td>
<td>0.396 ± 0.0074</td>
<td>+5.2% ... -1.1%</td>
<td>0.388 ± 0.0099</td>
</tr>
<tr>
<td>examples/Issue4044</td>
<td>0.587 ± 0.016</td>
<td>+9.7% ... +2%</td>
<td>0.555 ± 0.014</td>
</tr>
<tr>
<td>misc/Coverage</td>
<td>0.407 ± 0.0082</td>
<td>+4.5% ... -2%</td>
<td>0.403 ± 0.011</td>
</tr>
<tr>
<td>misc/Ids</td>
<td>5.92 ± 0.15</td>
<td>+0.55% ... -5.5%</td>
<td>6.07 ± 0.12</td>
</tr>
<tr>
<td>misc/LateMetaVariableInstantiation</td>
<td>0.126 ± 0.0026</td>
<td>+2.7% ... -3.1%</td>
<td>0.127 ± 0.0028</td>
</tr>
<tr>
<td>misc/SlowOccurrences</td>
<td>1.14 ± 0.027</td>
<td>+4.7% ... -1.7%</td>
<td>1.12 ± 0.025</td>
</tr>
<tr>
<td>monad/Monad</td>
<td>37.9 ± 0.85</td>
<td>+2.5% ... -3.4%</td>
<td>38.1 ± 0.81</td>
</tr>
<tr>
<td>proj/Data</td>
<td>0.265 ± 0.0061</td>
<td>+9% ... +2.6%</td>
<td>0.251 ± 0.0052</td>
</tr>
<tr>
<td>proj/Nested</td>
<td>0.26 ± 0.0052</td>
<td>+6.4% ... -0.044%</td>
<td>0.252 ± 0.0063</td>
</tr>
<tr>
<td>proj/Record</td>
<td>0.273 ± 0.0054</td>
<td>+6.8% ... +0.61%</td>
<td>0.263 ± 0.0062</td>
</tr>
<tr>
<td>std-lib/Any</td>
<td>119 ± 2.9</td>
<td>+4.6% ... -2.2%</td>
<td>117 ± 2.8</td>
</tr>
<tr>
<td>Syntacticosmos/Syntacticosmos</td>
<td>3.88 ± 0.098</td>
<td>+5.8% ... -0.8%</td>
<td>3.79 ± 0.082</td>
</tr>
</tbody>
</table>
Table 6.8: Resident memory when type-checking examples from the benchmark suite included with the Agda implementation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Agda,ε</th>
<th>Δ%</th>
<th>Agda (baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>categories/Categories</td>
<td>105 ± 0.018</td>
<td>+1.7% ... +1.6%</td>
<td>103 ± 0.074</td>
</tr>
<tr>
<td>categories/Language</td>
<td>108 ± 0.018</td>
<td>+2.5% ... +2.5%</td>
<td>106 ± 0.018</td>
</tr>
<tr>
<td>categories/Language3</td>
<td>96.4 ± 0.22</td>
<td>+2.9% ... +2.1%</td>
<td>94.1 ± 0.29</td>
</tr>
<tr>
<td>categories/Primitive</td>
<td>90.7 ± 0.1</td>
<td>+3.6% ... +3.4%</td>
<td>87.7 ± 0.029</td>
</tr>
<tr>
<td>examples/CwF</td>
<td>234 ± 0.86</td>
<td>+3.7% ... +2.6%</td>
<td>227 ± 0.85</td>
</tr>
<tr>
<td>examples/FunctorComposition</td>
<td>103 ± 0.09</td>
<td>+3.5% ... +2.4%</td>
<td>99.7 ± 0.49</td>
</tr>
<tr>
<td>examples/Issue4044</td>
<td>119 ± 0.19</td>
<td>+2.1% ... +1.7%</td>
<td>117 ± 0.11</td>
</tr>
<tr>
<td>misc/Coverage</td>
<td>99.2 ± 0.039</td>
<td>+3.6% ... +3.5%</td>
<td>95.8 ± 0.04</td>
</tr>
<tr>
<td>misc/Ids</td>
<td>1220 ± 8</td>
<td>+0.87% ... -0.74%</td>
<td>1220 ± 5.9</td>
</tr>
<tr>
<td>misc/LateMetaVariableInstantation</td>
<td>83.1 ± 0.056</td>
<td>+3.3% ... +3.2%</td>
<td>80.5 ± 0.024</td>
</tr>
<tr>
<td>misc/SlowOccurrences</td>
<td>188 ± 0.068</td>
<td>+2.1% ... +1.9%</td>
<td>184 ± 0.11</td>
</tr>
<tr>
<td>monad/Monad</td>
<td>335 ± 3.3</td>
<td>+1.9% ... -0.73%</td>
<td>333 ± 3.1</td>
</tr>
<tr>
<td>proj/Data</td>
<td>90 ± 0.071</td>
<td>+3.3% ... +3.1%</td>
<td>87.2 ± 0.058</td>
</tr>
<tr>
<td>proj/Nested</td>
<td>89.5 ± 0.044</td>
<td>+3.5% ... +3.3%</td>
<td>86.6 ± 0.038</td>
</tr>
<tr>
<td>proj/Record</td>
<td>90 ± 0.058</td>
<td>+3.4% ... +3.2%</td>
<td>87.1 ± 0.088</td>
</tr>
<tr>
<td>std-lib/Any</td>
<td>813 ± 2.8</td>
<td>+0.17% ... -0.9%</td>
<td>816 ± 3.5</td>
</tr>
<tr>
<td>Syntacticocosmos/Syntacticocosmos</td>
<td>120 ± 0.33</td>
<td>+3.6% ... +2.7%</td>
<td>116 ± 0.35</td>
</tr>
</tbody>
</table>
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$\varepsilon$ 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$\varepsilon$ are adopted by the main Agda implementation, the users will need to modify their existing code to give the arguments that Agda$\varepsilon$ 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-\(\varepsilon\) 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-\(\varepsilon\) than the adapted benchmark does in Agda, and than the original categories/Language benchmark does in either Agda-\(\varepsilon\) or Agda (which performs adequately in both). We presume that Agda-\(\varepsilon\) might be solving certain
6.6. COMPARISON WITH OTHER PROOF ASSISTANTS

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. 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, 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:
\[ U \equiv \text{Bool} \times \text{Bool} : \text{Set} \]
\[ \text{set} \equiv (\text{true}, \text{false}) : U \]
\[ \text{El} (u : U) \equiv \text{if} (\lambda. \text{Bool}) (u.\pi_1) \text{true} (u.\pi_2) : \text{Bool} \]

The problem is then as follows:
\[
\mathcal{F} : \text{Bool} \rightarrow \text{Set}, \mathcal{G} : U \rightarrow \text{Set}, \alpha : \text{Bool} \rightarrow U;
\]
\[
x : \text{Bool} \vdash \mathcal{F} (\text{El} (\alpha x)) \rightarrow \mathcal{G} (\alpha x) : \text{Set} \equiv \mathcal{F} \text{true} \rightarrow \mathcal{G} \text{set} : \text{Set}
\]

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 \( \alpha \).

1 (\text{* Shorthands *})
2 Definition \( U \) : \text{Set} := \text{bool} * \text{bool}.
3 Definition \( \text{set}' \) : \( U \) := (\text{true}, \text{true}).
4 Definition \( \text{el}' \) (\( b \) : \text{bool}) : \( U \) := (\text{false}, \( b \)).
5 Definition \( \text{El} \) (\( u \) : \( U \)) : \text{bool} :=
6 match (\text{fst} \( u \)) with
7 | \text{true} => \text{true}
8 | \text{false} => (\text{snd} \( u \))
9 end.

10 (\text{* Atoms *})
11 Axiom \( F \) : \text{bool} \rightarrow \text{Set}.
12 Axiom \( G \) : \( U \rightarrow \text{Set} \).

13 (\text{* Metavariables and constraints *})
14 Definition \( c1 \) :
15 \hspace{1em} let \( \alpha \) : \text{bool} \rightarrow \( U \) := \_ in
16 \hspace{1em} \forall (x : \text{bool}),
17 \hspace{1em} (F (\text{El} (\alpha \ x)) \rightarrow G (\alpha \ x)) \rightarrow
18 \hspace{1em} (F \text{true} \rightarrow G \text{set'}) :=
19 \hspace{1em} \text{fun} \ x \ y \Rightarrow y.

Output:

1 The Coq Proof Assistant, version 8.13.2 (May 2021)
2 compiled on May 28 2021 15:28:51 with OCaml 4.11.1
3 File "Pi.v", line 21, characters 16–17:
4 Error:
5 In environment
6 \( x \) : \text{bool}
7 \( y \) : \( F \) (\text{El} (?u \ x)) \rightarrow G (?u \ x)
8 The term "\( y "\) has type "\( F \) (\text{El} (?u \ x)) \rightarrow G (?u \ x)"
9 while it is expected to have type "\( F \) \text{true} \rightarrow G \text{set}’".

[Exit code: 0]
In this example, the definition of the function \( c_1 \) forces the type of the second argument \( (F (\text{El} (\alpha x)) \rightarrow G \text{set'}) \) and the type of the result of the function \( (F \text{true} \rightarrow G \text{set'}) \) 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 \( \Pi \)-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.

```coq
(* Shorthands *)
Definition U : Set := bool * bool.
Definition set' : U := (true, true).
Definition el' (b : bool) : U := (false, b).

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

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

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 \( \text{let alpha : bool -> U := fun x ==> set'} \), 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.

```coq
(* Shorthands *)
Definition U : Set := bool * bool.
Definition set' : U := (true, true).
Definition el' (b : bool) : U := (false, b).

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

(* Metavariables and constraints *)
Definition c1 :
  let alpha : bool -> U := _ in
  fun x y =>
  Output:
```
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:

\[
U \triangleq \text{Bool} \times \text{Bool} : \text{Set} \\
\text{set} \triangleq \langle \text{true}, \text{false} \rangle : U \\
\text{E1} (u : U) \triangleq \text{if} (\lambda \text{Bool}) (u . \pi_1) \text{true} (u . \pi_2) : \text{Bool}
\]

The problem is then as follows:

\[
\mathbb{P} : \text{Bool} \rightarrow U \rightarrow \text{Set}, \alpha : \text{Bool} \rightarrow U ; \\
x : \text{Bool} \vdash \mathbb{P} (\text{E1} \, (\alpha \, x)) \, (\alpha \, x) : \text{Set} \cong \mathbb{P} \text{true set} : \text{Set}
\]
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 $U : \text{Set} := \text{bool} \times \text{bool}$.
Definition $\text{set}' : U := (\text{true}, \text{true})$.
Definition $\text{el}' (b: \text{bool}) : U := (\text{false}, b)$.

Definition $\text{El} (u : U) : \text{bool} :=$
\begin{verbatim}
  match (fst u) with
  | true => true
  | false => (snd u)
end.
\end{verbatim}

Axiom $\text{Pred} : U \rightarrow \text{bool} \rightarrow \text{Set}$.

(* Metavariables and constraints *)
Definition $c1 :$
\begin{verbatim}
  let alpha : bool \rightarrow U := _ in
  forall (x: bool),
  (Pred (alpha x) (El (alpha x))) \rightarrow
  (Pred set' true) :=
  fun x y => y.
\end{verbatim}

Output:
Listing 6.15: Rendering of Example 6.2 in Coq, with the solution to the metavariable given, together with the output from the type checker.

(* Atoms *)
Axiom $F : \text{bool} \rightarrow \text{Set}$.

(* Shorthands *)
Definition $U : \text{Set} := \text{bool} \times \text{bool}$.
Definition $\text{set}' : U := (\text{true}, \text{true})$.
Definition $\text{el}' (b : \text{bool}) : U := (\text{false}, b)$.

Definition $\text{El} (u : U) : \text{bool} :=$
\begin{verbatim}
  match (fst u) with
  | true => true
  | false => (snd u)
end.
\end{verbatim}

Axiom $\text{Pred} : \text{bool} \rightarrow U \rightarrow \text{Set}$.

(* Metavariables and constraints *)
Definition $c1 :$
\begin{verbatim}
  let alpha : bool \rightarrow U := \_ in
  forall (x: bool),
  (Pred (El (alpha x)) (alpha x)) \rightarrow
  (Pred set' true) :=
  fun x y => y.
\end{verbatim}

Output:
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 corresponding type checker output are detailed in Appendix B.3.
Listing 6.16: Example of first-order unification in Coq

1 Inductive t : nat -> Type :=
2 | nil : t 0
3 | cons : forall (n:nat), bool -> t n -> t (S n).
4
5 Notation "[ x ; y ; .. ; z ]" := (cons _ x (cons _ y .. (cons _ z nil) ..)).
6
7 Fixpoint append (m n : nat) (x : t m) (y : t n) : t (m + n) :=
8 match x
9 | nil => y
10 | cons _ b x' => cons _ b (append _ _ x' y)
11 end.
12
13 Fixpoint repeat (n : nat) (b : bool) : t n :=
14 match n with
15 | O => nil
16 | S m => cons m b (repeat m b)
17 end.
18
19 Definition mkVec (m n : nat) : t (m + n) :=
20 append _ _ (repeat _ true) (repeat _ false).
21
22 Definition list1 : t (0 + 2) := mkVec _ _.
23 Eval cbv in list1.
24
25 Definition list2 : t (2 + 0) := mkVec _ _.
26 Eval cbv in list2.
27
28 Definition list3 : _ :=
29 let a := mkVec _ _ in
30 append (0+2) (2+0) a a.
31 Eval cbv in list3.
32

Output:

1 The Coq Proof Assistant, version 8.13.2 (May 2021)
2 compiled on May 28 2021 15:28:51 with OCaml 4.11.1
3 = [false; false ]
4 : t (0 + 2)
5 = [true; true]
6 : t (2 + 0)
7 = [false; false; false; false]
8 : t (0 + 2 + (2 + 0))
9 [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.

```idris
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).

\[
\begin{align*}
\mathcal{F} &: \text{Bool} \to \text{Set}, \\
\mathcal{f} &: \mathcal{F} \text{ true}, \\
\alpha &: \mathcal{F} \text{ true} \to \text{Bool} \to \text{Bool} \\
\gamma &: \mathcal{F} \text{ true} \to \text{Bool} \to \text{Bool} \\
\gamma &: \mathcal{F} \text{ true} \to \text{Bool} \to \text{Bool} \\
x &: \text{Bool} \vdash (z : \mathcal{F} (\alpha \mathcal{f} \text{ true})) \to \mathcal{F} x \\
&\approx (z : \mathcal{F} \text{ true}) \to \mathcal{F} (\alpha z x) : \text{Set} \uplus \text{Set}
\end{align*}
\]

Applying the rules Rule-Schema 3 (injectivity of \(\Pi\)), 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. \texttt{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 \(\mathcal{f}\) is rendered as a data constructor \texttt{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).

\[
\begin{align*}
\mathcal{F} &: \text{Bool} \to \text{Set}, \\
\mathcal{f} &: \mathcal{F} \text{ true}, \\
\mathcal{G} &: \text{Bool} \to \text{Bool} \to \text{Set}, \\
\alpha &: \mathcal{F} \text{ true} \to \text{Bool} \to \text{Bool} \\
\beta &: \mathcal{F} \text{ true} \to \text{Bool} \to \text{Bool} \\
\gamma &: \mathcal{F} \text{ true} \to \text{Bool} \to \text{Bool} \\
x &: \text{Bool} \vdash (y : \mathcal{F} (\alpha \mathcal{f} \text{ true})) \to (z : \mathcal{F} \text{ true}) \to \mathcal{G} (\beta z x) x \\
&\approx (y : \mathcal{F} \text{ true}) \to (z : \mathcal{F} (\beta \mathcal{f} \text{ true})) \to \mathcal{G} x (\alpha y x) : \text{Set} \uplus \text{Set}
\end{align*}
\]

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 \( \mathcal{f} \) due to the lack of a more straightforward way to define constants of arbitrary type in Idris 2.

```idris
data F : Bool \rightarrow Type where
  MkFT : F True

data AreEqual : (A : Type) \rightarrow Pair A A \rightarrow Type

c1 : (c : (alpha : F True \rightarrow Bool \rightarrow Bool) \rightarrow
  Pair ((x : Bool) \rightarrow (z : F True) \rightarrow F (alpha z x))
  ((x : Bool) \rightarrow (z : F (alpha MkFT True)) \rightarrow F x)) \rightarrow Type

c1 c = AreEqual _ (c _)
```

Output:

Idris 2, version 0.3.0
1/1: Building BlockedConstantLs (BlockedConstantLs.idr)
[Exit code: 0]


```idris
data F : Bool \rightarrow Type where
  MkFT : F True

data AreEqual : (A : Type) \rightarrow Pair A A \rightarrow Type

c1 : (c : (alpha : F True \rightarrow Bool \rightarrow Bool) \rightarrow
  Pair ((x : Bool) \rightarrow (z : F True) \rightarrow F (alpha z x))
  ((x : Bool) \rightarrow (z : F (alpha MkFT True)) \rightarrow F x)) \rightarrow 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.

```idris
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

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.

```idris
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')

Output:

Idris 2, version 0.3.0
1/1: Building SpineMini (SpineMini.idr)
```
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 \(\beta\)-normal form, which results in an infinite loop (Listing 6.23).

**Example 6.5** (Spine problem causing ill-typed instantiation). Consider the following shorthands:

\[
F z \triangleq \text{if} (\lambda.\text{Set}) z \text{ Bool} (\text{Bool} \to \text{Bool})
\]

\[
f z x y \triangleq \text{if} (\lambda z. F z) z x y
\]

Using them, we define the following constraints:

\[
\mathbb{P} : (y : \text{Bool}) \to F y \to \text{Bool} \to ((\text{Bool} \to \text{Bool}) \to \text{Bool}) \to \text{Set},
\]

\[
\alpha : \text{Bool} \to \text{Bool},
\]

\[
\beta : (\text{Bool} \to \text{Bool}) \to \text{Bool},
\]

\[
\gamma : (\text{Bool} \to \text{Bool}) \to \text{Bool},
\]

\[
; x : \text{Bool}, z : \text{Bool} \to \text{Bool} \vdash
\]

\[
\approx \mathbb{P} (\alpha \text{ true}) (f (\alpha \text{ true}) \text{ true}) \text{ false} (\lambda x. (\beta x)) : \text{Set}
\]

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 \((\lambda x. x x)\) in the terms leading to the type error.

```idris
F : \text{Bool} \to \text{Type}
F \text{ True} = \text{ Bool}
F \text{ False} = \text{ Bool} \to \text{Bool}
mkF : (b : \text{Bool}) \to \text{Bool} \to (\text{Bool} \to \text{Bool}) \to \text{ F b}
mkF \text{ True} x y = x
mkF \text{ False} x y = y
data P : (b : \text{Bool}) \to F b \to \text{Bool} \to ((\text{Bool} \to \text{Bool}) \to \text{ Bool}) \to \text{ Type where}
data AreEqual : (A : \text{Type}) \to \text{Pair} A A \to \text{ Type}
c1 : (c :
    (alpha : \text{Bool} \to \text{Bool}) \to
    (beta : (\text{Bool} \to \text{Bool}) \to \text{Bool}) \to
    (gamma : (\text{Bool} \to \text{Bool}) \to \text{Bool}) \to
```

Pair ((x : Bool) ->
  (z : Bool -> Bool) ->
  P True (beta z) (alpha x) gamma)

((x : Bool) ->
 (z : Bool -> Bool) ->
P (alpha True) (mkF (alpha True) True z) False (\x => x (beta x)))

-> Type

c1 c = AreEqual _ (c _ _ _)

Output:

Idris 2, version 0.3.0
1/1: Building SpineProblemLittleOmega (SpineProblemLittleOmega.idr)
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)).

Mismatch between: True and False.

SpineProblemLittleOmega.idr:28:20—28:27

| 28 | c1 c = AreEqual _ (c _ _ _)
|_________________________

[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.

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

G : Bool -> Type
G True = Bool -> Bool
G False = (Bool -> Bool) -> Bool

mkG : (b : Bool) -> (Bool -> Bool) -> ((Bool -> Bool) -> Bool) -> G b
mkG True x y = x
mkG False x y = y

data P : (b : Bool) -> F b -> G b -> Bool ->
  ((Bool -> Bool) -> Bool) -> Bool -> Type where
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 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 $a \in \{Agda.\varepsilon, Agda\}$, 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 $(\prod_{i=1}^{n} y_{i}^{a,b})^{\frac{1}{n}}$. This can be rewritten as $\exp\left(\frac{1}{n} \sum_{i=1}^{n} \log(y_{i}^{a,b})\right)$, where $\frac{1}{n} \sum_{i=1}^{n} \log(y_{i}^{a,b})$ 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 $\log(y_{i}^{a,b})$. 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 ($n \rightarrow \infty$), the mean $\frac{1}{n} \sum_{i=1}^{n} \log(y_{i}^{a,b})$ tends to the expected value $\mathbb{E}\left[\log(y_{i}^{a,b})\right] = \mu_{a,b}$.

Confidence intervals for average resource usage

In the absence of additional information about the distribution of $\log(y_{i}^{a,b})$ ($i = 1, ..., n$), we assume that they are independent, normally distributed random variables with mean $\mu_{a,b}$ and variance $\sigma_{a,b}^{2}$ (i.e.
log\( (y_{a,b}^i) \sim \text{Normal}(\mu_{a,b}, \sigma_{a,b}^2) \). 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 \( \exp(\mu_{a,b}) \) can thus be understood as the result of taking the geometric mean of an infinite number of measurements. The distribution of \( \log(y_{a,b}^i) \) can be understood as arising from each measurement being subject to an independent, normally-distributed measurement error \( \log(\varepsilon_{a,b}^i) \sim \text{Normal}(0, \sigma_{a,b}^2) \). This measurement error increases (or decreases) the measured value \( y_{a,b}^i \) relative to the “true” value \( \exp(\mu_{a,b}) \) by a factor of \( \varepsilon_{a,b}^i \).

Under the normality assumption, we estimate \( \mu_{a,b} \) by \( \hat{\mu}_{a,b} \equiv \frac{1}{n} \sum_{i=1}^{n} \log(y_{a,b}^i) \). If we define \( \hat{\sigma}_{a,b}^2 \equiv \frac{1}{n-1} \sum_{i=1}^{n} (\log(y_{a,b}^i) - \hat{\mu}_{a,b})^2 \), then the 95% confidence interval (\( \alpha \equiv 0.05 \)) for \( \mu_{a,b} \) is \( \hat{\mu}_{a,b} \pm \frac{\hat{\sigma}_{a,b}}{\sqrt{n}} \cdot t_{n-1,1-\alpha/2} \), where \( t_{n-1,1-\alpha/2} \) 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 \( \exp\left(\hat{\mu}_{a,b} \pm \frac{\hat{\sigma}_{a,b}}{\sqrt{n}} \cdot t_{n-1,1-\alpha/2} \right) \). For readability, we use the larger, less precise interval which is centered at \( \exp(\hat{\mu}_{a,b}) \), namely \( \exp(\hat{\mu}_{a,b}) \pm \exp(\hat{\mu}_{a,b}) \left( \exp\left(\frac{\hat{\sigma}_{a,b}}{\sqrt{n}} \cdot t_{n-1,1-\alpha/2} \right) - 1 \right) \).

Confidence intervals for average ratios of resource usage
In order to compare the resource usage of Agda and Agda for a given benchmark \( b \), we observe the ratio \( \frac{y_{\text{Agda},\varepsilon}^{Agda,b}}{y_{\text{Agda},\varepsilon}^{Agda,b}} \). By the same token as when modelling the average resource usage, we will study the logarithm of this ratio, i.e. \( \log\left(\frac{y_{\text{Agda},\varepsilon}^{Agda,b}}{y_{\text{Agda},\varepsilon}^{Agda,b}}\right) \). The expected value of this ratio is \( r_b \equiv E \left[ \log\left(\frac{y_{\text{Agda},\varepsilon}^{Agda,b}}{y_{\text{Agda},\varepsilon}^{Agda,b}}\right) \right] = \mu_{\text{Agda},\varepsilon}^{Agda,b} - \mu_{\text{Agda},\varepsilon}^{Agda,b} \), which we can estimate by \( \hat{r}_b = \hat{\mu}_{\text{Agda},\varepsilon}^{Agda,b} - \hat{\mu}_{\text{Agda},\varepsilon}^{Agda,b} \). The quantity \( \exp(\hat{r}_b) \) can be interpreted as the geometric mean of the ratios \( \frac{y_{\text{Agda},\varepsilon}^{Agda,b}}{y_{\text{Agda},\varepsilon}^{Agda,b}} \) \( (i = 1, \ldots, n) \) as \( n \to \infty \), and it can be estimated by \( \exp(\check{r}_b) \).

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:

\[
\hat{\sigma}_b \equiv \sqrt{\frac{\hat{\sigma}_{\text{Agda},\varepsilon}^{Agda,b}^2}{n} + \frac{\hat{\sigma}_{\text{Agda},b}^2}{n}}
\]

then \( \frac{\check{r}_b - \hat{r}_b}{\hat{\sigma}_b} \) has a t-distribution with \( \nu \) degrees of freedom, where \( \nu \) is approximated by:

\[
\nu \equiv (n - 1) \left( \frac{\hat{\sigma}_{\text{Agda},\varepsilon}^{Agda,b}^2 + \hat{\sigma}_{\text{Agda},b}^2}{\hat{\sigma}_{\text{Agda},\varepsilon}^{Agda,b}^4 + \hat{\sigma}_{\text{Agda},b}^4} \right)^2
\]
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 $f$ as this is the most straightforward way we found to do so.

```idris
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 ((x : Bool) -> (y : F (alpha MkFT True)) ->
11            (z : F True) -> G (beta z x) x)
12
13    ((x : Bool) -> (y : F True) ->
14     (z : F (beta MkFT True)) -> G x (alpha y x))
15 ) -> Type
16 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)
   ) -> 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))
   ) -> Type

c1 c = AreEqual _ (c \ a , b => b) (\ a , b => b))

(3x : F True) -> G (beta z x) x)
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.

```plaintext
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 (∧st u) with
    [ true => true
      | false => (∧nd u)
    ].

(* Atoms *)
axiom F : bool −> Type[0].
aviom G : U −> Type[0].

(* Metavariables and constraints *)
definition c1 :
    let alpha : bool → U ≝ λx.? in
    (II (x : bool). F (El (alpha x)) → G (alpha x)) →
    (II (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/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: *****
```
Listing B.6: Example Listing B.5 modified so that the codomains are equal.

```matita
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
  (\Pi (x : bool). F (El mkSet) → G (alpha x)) →
  (\Pi (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.

```plaintext
1 include "basics/bool.ma".
2 include "basics/types.ma".

(* Shorthands *)
3 definition U : Type[0] ≝ bool × bool.
4 definition mkSet : U ≝ (true , true).
5 definition mkEl (b : bool) : U ≝ (false , b).
6
7 definition El (u : U) : bool ≝ match (\fst u) with
8     [ true => true
9     | false => (\snd u)
10     ].

(* Atoms *)
11 axiom F : bool → Type[0].
12 axiom G : U → Type[0].

(* Metavariables and constraints *)
13 definition c1 :
14     let alpha : bool → U ≝ \x.mkSet in
15     (\x : bool). F (El (alpha x)) → G (alpha x)) →
16     (\x : bool). F true → G mkSet) ≝
17     \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.

```plaintext
1 include "basics/bool.ma".
2 include "basics/types.ma".

(* Shorthands *)
3 definition U : Type[0] ≝ bool × bool.
4 definition mkSet : U ≝ (true , true).
5 definition mkEl (b : bool) : U ≝ (false , b).
```
B.2. MATITA

definition El (u : U) : bool ≝ match (∀ f 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:

0.993
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
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)
Listing B.9: Example Listing B.8 modified so that the first arguments can be made equal.

```plaintext
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 (∧st u) with
  | true => true
  | false => (∧nd u).

(* Atoms *)
axiom Pred : U −> bool −> Type[0].

(* Metavariabes and constraints *)
definition c1 :
  let alpha : bool → U ≝ λx.? in
  (II (x : bool). Pred (alpha x) (El (alpha x))) →
  (II (x : bool). Pred mkSet true) ≝
λ y. y.
```

Output:

0.99.3
Info: New object: cic:/matita/minilang/SpineMiniEasy/U.con
Info: New object: cic:/matita/minilang/SpineMiniEasy/mkSet.con
Info: New object: cic:/matita/minilang/SpineMiniEasy/mkEl.con
Info: New object: cic:/matita/minilang/SpineMiniEasy/El.con
Info: New object: cic:/matita/minilang/SpineMiniEasy/Pred.con
Info: New object: cic:/matita/minilang/SpineMiniEasy/c1.con
Info: Compilation successful

Listing B.10: Example Listing B.8 modified so that the metavariable alpha is solved from the start.

```plaintext
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 (∧st u) with
  | true => true
```


B.3. LEAN

Let

\[ \text{false} \Rightarrow (\\text{snd} \ u) \]

:\text{false} = \Rightarrow (\\text{snd} \ u)

(* Atoms *)

axiom Pred : bool \rightarrow U \rightarrow Type[0].

(* Metavariabes and constraints *)

definition c1 :

let alpha : bool \rightarrow U \equiv \lambda x.mkSet in

(\forall (x : bool). Pred (El (alpha x)) (alpha x)) \rightarrow

(\forall (x : bool). Pred true \mkSet) \equiv

\lambda 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/Pred.con
Info : New object: cic:/matita/minilang/SpineMiniSolved/El.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 \rightarrow Type
constant G : U \rightarrow Type

--- Metavariabes and constraints

def c1

(c : \forall (alpha : bool \rightarrow U),

(\forall (x : bool), F (El (alpha x)) \rightarrow G (alpha x))):

| false => (\snd u)
| false => (\snd u)
APPENDIX B. SUPPLEMENTAL EVALUATION

\[
\begin{align*}
( \forall (x : \text{bool}), \text{F} \ x & \rightarrow \text{G} \text{set'} ) \\
:= c \_ \\
\end{align*}
\]

Output:

Lean (version 3.28.0, commit 5a3bb32c05bc, Release)

/.lean/Pi.lean:20:6: error : type mismatch, term

\[ c \ ?m_1 \]

has type

\[ \Pi (x : \text{bool}), \text{F} (\text{El} \ (?m_1 x)) \rightarrow \text{G} (?m_1 x) \]

but is expected to have type

\[ \text{bool} \rightarrow \text{F} \ x \rightarrow \text{G} \text{set'} \]

[Exit code: 1]

Listing B.12: Example Listing B.11 modified so that the codomains are equal.

```lean
-- 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 : \text{bool} \rightarrow U),
   (\forall (x : \text{bool}), \text{F} \ (\text{El} \ \text{set'} )) \rightarrow \text{G} \ (\text{alpha} \ x))) :
  (\forall (x : \text{bool}), \text{F} \ tt \rightarrow \text{G} \text{set'})
:= c \_
```

Output:

Lean (version 3.28.0, commit 5a3bb32c05bc, Release)

[Exit code: 0]

Listing B.13: Example Listing B.11 modified so that the metavariable \text{alpha} is solved from the start.

```lean
-- 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
```
B.3. LEAN

11  —— Atoms
12  constant F : bool → Type
13  constant G : U → Type
14
15  —— Metavariables and constraints
16  def c1
17    (c : ∀ (alpha : bool → U),
18     (forall (x : bool), F (El (alpha x)) → G (alpha x))) :
19    (forall (x : bool), F tt → G set') := c (fun z, set')

Output:

1 Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
2 [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.

1  —— Atoms
2  constant F : bool → Type
3
4  —— Shorthands
5  def U : Type := prod bool bool
6  def set' : U := (tt, tt)
7  def el' (b : bool) : U := (ff, b)
8
9  def El (u : U) : bool :=
10    match u.1 with
11    | tt := tt
12    | ff := u.2
13  end
14
15  constant Pred : bool → U → Type
16
17  —— Metavariables and constraints
18  def c1
19    (c : ∀ (alpha : bool → U),
20     (forall (x : bool), Pred (El (alpha x)) (alpha x))) :
21    (forall (x : bool), Pred tt set') := c _

Output:

1 Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
2 ./lean/SpineMini.lean:21:63: error: type mismatch, term
3 c ?m_1
4 has type
5 U (x : bool), Pred (El (?m_1 x)) (?m_1 x)
6 but is expected to have type
7 bool → Pred tt set'
8 [Exit code: 1]
Listing B.15: Example Listing B.14 modified so that the first arguments can be made equal.

```lean
-- 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

-- Metavariabes 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.

```lean
-- 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

-- Metavariabes and constraints
def c1 (c : forall (alpha : bool → U),
  (forall (x : bool), Pred (El (alpha x)) (alpha x))) :
  (forall (x : bool), Pred set' tt) := c (fun z, set')
```

Output:
1 Lean (version 3.28.0, commit 5a3bb32c05bc, Release)
2 [Exit code: 0]
Index

List of Definitions, Notations and Problems

Notation (Vector notation: $\vec{t}$) .............................................. 15
Notation (Neutral terms in vector form: $h \vec{e}$) ...................... 15
Notation (Vector elements: $t_i$) .............................................. 15
Notation (Vector slices: $\vec{t}_{i_1, \ldots, j_1}$) .......................... 15
Notation (Vector membership: $\_ \in \_$) .................................. 15
Notation (Ungrammatical terms: $\lceil t \rceil$) ......................... 16
Notation (Partial functions: $F \downarrow y, F \downarrow y, F$) ............ 16
2.1 Definition (Fresh declaration) ............................................. 16
2.2 Definition (Instantiated metavariable, body of a metavariable) 16
2.3 Definition (Uninstantiated metavariable) ............................ 16
2.4 Definition (Well-formed signature: $\Sigma \mathsf{sig}$) ............... 16
2.6 Definition (Support of a signature: $\mathsf{Support}(\Sigma)$) ...... 17
Notation (Signature concatenation: $\Sigma_1, \Sigma_2$) ................. 17
2.7 Definition (Atom declarations of a signature: $\mathsf{AtomDecls}(\Sigma)$) 17
2.8 Definition (Constants declared by a signature: $\mathsf{Decl}(\Sigma)$) 17
2.10 Definition (Metavariabes in a term: $\mathsf{Metas}(t)$) ........... 17
2.11 Definition (Set of atoms in a term: $\mathsf{Atoms}(t)$) .......... 18
2.12 Definition (Set of constants of a term: $\mathsf{Consts}(t)$) ...... 18
2.14 Definition (Support of a context: $|\Gamma|)$ ......................... 19
Notation (Variable names in contexts: $\Gamma, x : A$) .......... 19
Notation (Context concatenation: $\Gamma_1, \Gamma_2$) .................. 19
2.16 Definition (Equality of contexts) ...................................... 20
Notation (Names for de Bruijn indices: $\lambda x.t, \Pi(x : A)B, \ldots$) 20
Notation (N-ary binders: $\lambda \vec{x}^n.t, \Pi(\overline{x}^n : A)B$) .......... 21
Notation (Arrow notation for $\Pi$-types: $(x : A) \rightarrow B, A \rightarrow B$) 21
Notation (Product notation for $\Sigma$-types: $(x : A) \times B, A \times B$) 21
Notation (Strengthening of a set of variables: $X - 1, X - k$) .... 21
2.18 Definition (Free variables in a term: $\mathsf{FV}(t)$) ............. 21
2.19 Definition (Free variables of a context: $\mathsf{FV}(\Delta)$) ......... 21
Notation (Membership of names in set of free variables: $x \in \mathsf{FV}(t), x \notin \mathsf{FV}(t), \mathsf{FV}(t) \subseteq \{ \vec{x} \}$) ............. 22
2.20 Definition (Renaming) ................................................... 22
2.21 Definition (Inline renamings: $[\ldots \mapsto \ldots]$) ............. 22
2.22 Definition (Weakening: $\langle +n \rangle)$ ............................. 22

189
2.23 Definition (Strengthening: \((-n)\)) ........................................... 22
2.24 Definition (Weakening of renamings: \((\rho + n)\)) ......................... 22
2.26 Definition (Application of a renaming to a term: \(t \rho, t^\rho\)) .......... 23
2.27 Definition (Renaming of a context: \(\Gamma \rho\)) ............................ 23
2.28 Notation (Composition of renamings: \(\rho_1 \rho_2\)) ................. 23
2.29 Definition (Application of a renaming to a term: \(t \rho, t^\rho\)) .......... 23
2.30 Notation (Composition of renamings: \(\rho_1 \rho_2\)) ................. 23
2.31 Definition (Hereditary substitution: \(t [u/x] \downarrow r\)) ............ 24
2.32 Notation (Names for de Bruijn indices in hereditary substitution) .......... 24
2.33 Definition (Hereditary substitution as a partial function: \(t [u/x]\Lsh, t [u/x]\Lr) ...... 25
2.34 Notation (Hereditary elimination as a partial function: \((t @ e) \Lsh, t @ e) \Lr) ...... 25
2.35 Definition (Iterated hereditary substitution: \(t [\vec{u}/\vec{x}] \Lsh \vec{r} \Lr) ...... 26
2.36 Notation (Names for de Bruijn indices in hereditary substitution) .......... 26
2.37 Definition (Iterated hereditary elimination: \(t @ \vec{e} \Lsh \vec{r} \Lr) ...... 26
2.38 Notation (Signature judgment: \(\Sigma \vdash J\)) ...... 33
2.39 Definition (Free variables of a scoped and typed term: \(\text{fv}(\Delta \vdash t : B)\), \(\text{fv}(J)\)) ............................................. 33
2.40 Notation (Signature subsumption: \(\Sigma \subseteq \Sigma'\)) .................. 38
2.41 Definition (\(\delta\eta\)-normalization step: \(\Sigma; \Gamma \vdash t \rightarrow \delta\eta u : T\)) .... 31
2.42 Definition (Iterated \(\delta\eta\)-reduction: \(\Sigma; \Gamma \vdash t \rightarrow \delta\eta^* u : A\)) ... 31
2.43 Definition (Judgment: \(\Sigma; \Gamma \vdash J\)) .................................. 33
2.44 Notation (Signature judgment: \(\Sigma \vdash J\)) ...... 33
2.45 Definition (Type elimination: \(\Sigma; \Gamma \vdash (\ell \vdash) \hat{\otimes} @ \vec{e} \Lsh U\)) ...... 45
2.46 Definition (Type application: \(\Sigma; \Gamma \vdash T \hat{\otimes} @ \vec{t} \Lsh U\)) ...... 45
2.47 Definition (Type application, reversed: \(\Sigma; \Gamma \vdash T \hat{\otimes} R @ \vec{t} \Lsh U\)) ...... 47
2.48 Definition (Well-formed reordering) ..................................... 38
2.49 Definition (Well-formed reorderings) ................................... 38
2.50 Definition (Signature elimination: \(\Sigma \subseteq \Sigma'\)) ................. 38
2.51 Definition (Well-formed reorderings) ................................... 38
2.52 Definition (Well-formed reorderings) ................................... 38
2.53 Definition (Restriction of a metasubstitution to a set of metavariables: \(\Theta \subseteq \Theta'\)) .................. 49
2.54 Definition (Compatible metasubstitution: \(\Theta \vdash \Sigma\)) ............. 49
2.55 Definition (Declaration: \((D)\)) ........................................ 49
2.56 Definition (Compatibility of a metasubstitution with a declaration: \(\Theta \text{ compatible with } D\)) ................. 49
2.57 Definition (Restriction of a metasubstitution to a set of metavariables: \(\Theta_{\Sigma'}, \Theta_{\Sigma U}\)) .......................... 50
2.58 Definition (Closed signature) ........................................... 51
2.59 Definition (Normalization to meta-free terms: \(\Sigma \vdash t \Ll u\)) ........ 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.143</td>
<td>Definition (Closing metasubstitution: (\text{CLOSE}(\Sigma) \downarrow \Theta))</td>
<td>53</td>
</tr>
<tr>
<td>2.145</td>
<td>Definition (Equality of metasubstitutions: (\Theta \equiv \Theta'))</td>
<td>53</td>
</tr>
<tr>
<td>2.151</td>
<td>Definition (Signature extension: (\Sigma \subseteq \Sigma'))</td>
<td>55</td>
</tr>
<tr>
<td>2.158</td>
<td>Definition (Strongly neutral term)</td>
<td>56</td>
</tr>
<tr>
<td>2.164</td>
<td>Definition (Irreducible terms)</td>
<td>57</td>
</tr>
<tr>
<td>2.168</td>
<td>Definition (Rigid occurrence)</td>
<td>58</td>
</tr>
<tr>
<td>2.169</td>
<td>Definition (Typed rigid occurrence)</td>
<td>59</td>
</tr>
<tr>
<td>3.1</td>
<td>Definition (Term with holes)</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>Definition (Well-formed type checking problem: (\Sigma; \Gamma \vdash \sharp t : A))</td>
<td>65</td>
</tr>
<tr>
<td>3.3</td>
<td>Definition (Solution to a type checking problem: (\Theta \vdash \Sigma; \Gamma \vdash \sharp t : A))</td>
<td>65</td>
</tr>
<tr>
<td>3.4</td>
<td>Definition (Unique solution to a type checking problem)</td>
<td>65</td>
</tr>
<tr>
<td>3.7</td>
<td>Definition (Basic constraint)</td>
<td>66</td>
</tr>
<tr>
<td>3.8</td>
<td>Definition (Solution to a basic constraint: (\Theta \vdash \Sigma; \Gamma \vdash \sharp u : B))</td>
<td>66</td>
</tr>
<tr>
<td>3.9</td>
<td>Problem (Unification of dependently-typed terms)</td>
<td>66</td>
</tr>
<tr>
<td>3.10</td>
<td>Definition (Elaboration algorithm)</td>
<td>66</td>
</tr>
<tr>
<td>3.11</td>
<td>Definition (Well-formedness of an elaboration algorithm)</td>
<td>67</td>
</tr>
<tr>
<td>3.12</td>
<td>Definition (Correctness of an elaboration algorithm)</td>
<td>67</td>
</tr>
<tr>
<td>3.13</td>
<td>Definition (Homogeneous constraint)</td>
<td>67</td>
</tr>
<tr>
<td>3.14</td>
<td>Definition (Blocked constant)</td>
<td>67</td>
</tr>
<tr>
<td>3.18</td>
<td>Problem (Binder problem)</td>
<td>73</td>
</tr>
<tr>
<td>3.19</td>
<td>Definition (Homogeneous constraint)</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Definition (Twin contexts)</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Definition (Well-formed internal constraint: (\Sigma; \Gamma_1 \vdash \sharp u : A))</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>Definition (Unification problem)</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>Definition (Set of constants in a constraint or a vector of constraints: (\text{consts}(\mathcal{C}), \text{consts}(\mathcal{C})))</td>
<td>79</td>
</tr>
<tr>
<td>4.5</td>
<td>Definition (Well-formed unification problem)</td>
<td>79</td>
</tr>
<tr>
<td>4.9</td>
<td>Definition (Solution to a constraint: (\Theta \vdash \mathcal{C}, \Theta \vdash \mathcal{C}))</td>
<td>80</td>
</tr>
<tr>
<td>4.11</td>
<td>Definition (Solution to a unification problem: (\Theta \vdash \Sigma; \mathcal{C}))</td>
<td>80</td>
</tr>
<tr>
<td>4.12</td>
<td>Definition (Heterogeneous equality: (\Sigma; \Gamma_1 \vdash \sharp u : A))</td>
<td>80</td>
</tr>
<tr>
<td>4.17</td>
<td>Definition (Constraint satisfaction: (\Sigma \models \mathcal{C}, \Sigma \models \mathcal{C}))</td>
<td>81</td>
</tr>
<tr>
<td>4.18</td>
<td>Definition (Essentially homogeneous set of constraints)</td>
<td>82</td>
</tr>
<tr>
<td>4.19</td>
<td>Definition (Essentially homogeneous problem)</td>
<td>82</td>
</tr>
<tr>
<td>4.22</td>
<td>Definition (Elaboration into internal constraints)</td>
<td>82</td>
</tr>
<tr>
<td>4.25</td>
<td>Definition (Reduction rule)</td>
<td>84</td>
</tr>
<tr>
<td>4.26</td>
<td>Definition (Rule correctness)</td>
<td>85</td>
</tr>
<tr>
<td>4.27</td>
<td>Definition (One-step problem reduction: (\Sigma; \mathcal{E} \Rightarrow \Sigma'; \mathcal{E}'))</td>
<td>85</td>
</tr>
<tr>
<td>4.28</td>
<td>Definition (Problem reduction: (\Sigma'; \mathcal{E} \Rightarrow \Sigma'; \mathcal{E}'))</td>
<td>85</td>
</tr>
<tr>
<td>4.30</td>
<td>Definition (Solved problem)</td>
<td>86</td>
</tr>
<tr>
<td>4.32</td>
<td>Problem (Metavariable instantiation)</td>
<td>87</td>
</tr>
<tr>
<td>4.36</td>
<td>Definition (Heterogeneously equal contexts modulo variables)</td>
<td>88</td>
</tr>
</tbody>
</table>
### List of Postulates

<table>
<thead>
<tr>
<th>Postulate</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postulate (Typing of hereditary substitution)</td>
<td>34</td>
</tr>
<tr>
<td>Postulate (Typing of hereditary application)</td>
<td>34</td>
</tr>
<tr>
<td>Postulate (Typing of hereditary projection)</td>
<td>34</td>
</tr>
<tr>
<td>Postulate (Congruence of hereditary substitution)</td>
<td>34</td>
</tr>
<tr>
<td>Postulate (Hereditary substitution commutes)</td>
<td>34</td>
</tr>
<tr>
<td>Postulate (Congruence of hereditary application)</td>
<td>35</td>
</tr>
<tr>
<td>Postulate (Congruence of hereditary projection)</td>
<td>35</td>
</tr>
<tr>
<td>Postulate (No infinite chains)</td>
<td>35</td>
</tr>
<tr>
<td>Postulate (Commuting of hereditary substitution and application)</td>
<td>35</td>
</tr>
<tr>
<td>Postulate (Injectivity of $\Pi$)</td>
<td>35</td>
</tr>
<tr>
<td>Postulate (Injectivity of $\Sigma$)</td>
<td>36</td>
</tr>
<tr>
<td>Postulate (Signature strengthening)</td>
<td>38</td>
</tr>
<tr>
<td>Postulate (Context strengthening)</td>
<td>39</td>
</tr>
<tr>
<td>Postulate (Existence of a common reduct)</td>
<td>41</td>
</tr>
<tr>
<td>Postulate (Existence of a unique full normal form)</td>
<td>42</td>
</tr>
</tbody>
</table>
2.54 Lemma (Term equality is an equivalence relation) ..... 36
2.55 Remark (Type equality is an equivalence relation) ..... 36
2.56 Lemma (Neutral inversion) ..... 36
2.57 Lemma (Type of $\lambda$-abstraction) ..... 36
2.59 Lemma (Abstraction equality inversion) ..... 37
2.60 Lemma (Type of a pair) ..... 37
2.61 Remark (Reflexivity of context equality) ..... 37
2.62 Lemma (Context weakening) ..... 37
2.63 Lemma (Preservation of judgments by type conversion) ..... 37
2.64 Lemma (Equality of contexts is an equivalence relation) ..... 37
2.65 Lemma (No extraneous variables in term) ..... 37
2.66 Lemma (Signature weakening) ..... 38
2.69 Lemma (Piecewise well-formedness of typing judgments) ..... 38
2.70 Lemma (Variables of irrelevant type) ..... 39
2.72 Lemma (No extraneous constants) ..... 39
2.73 Remark (Signature piecewise well-formed) ..... 39
2.74 Remark (Simplified \textsc{delta-meta} rule: \textsc{delta-meta}$_0$) ..... 39
2.75 Lemma (Uniqueness of typing for neutrals) ..... 40
2.76 Lemma (Variable types say everything) ..... 40
2.79 Lemma (Typing and congruence of elimination) ..... 40
2.80 Lemma (Simplified \textsc{app}, \textsc{app-eq}: \textsc{app}$_0$, \textsc{app-eq}$_0$) ..... 40
2.81 Remark (Cancellation of weakening with substitution) ..... 41
2.82 Lemma ($\lambda$ inversion) ..... 41
2.83 Lemma (Injectivity of $\lambda$) ..... 41
2.84 Lemma (\langle , \rangle-inversion) ..... 41
2.85 Lemma (Injectivity of \langle , \rangle) ..... 41
2.86 Lemma (Equality of $\delta\eta$-reduct) ..... 41
2.87 Remark (Existence of a common normal form) ..... 42
2.89 Remark (Disjointness of primitive types) ..... 42
2.90 Lemma (Reduction under equal context) ..... 42
2.91 Remark (Inversion of reduction under $\lambda$) ..... 42
2.92 Remark (Inversion of reduction under $\langle , \rangle$) ..... 42
2.93 Remark (Strengthening of hereditary substitution and elimination) ..... 43
2.94 Remark (Strengthening of reduction) ..... 43
2.96 Remark (WHNF reduction is deterministic) ..... 43
2.97 Remark (WHNF reduction is $\delta\eta$-reduction) ..... 43
2.98 Lemma (Equality of WHNF) ..... 43
2.99 Lemma (Term in WHNF) ..... 43
2.101 Lemma (Nose of weak-head normal form) ..... 45
2.102 Remark (Preservation of free variables by WHNF) ..... 45
2.105 Remark (Type elimination without projections) ..... 46
2.106 Lemma (Type elimination) ..... 46
2.107 Lemma (Type elimination inversion) ..... 46
2.108 Remark (Uniqueness of head type lookup) ..... 46
2.109 Lemma (Type application inversion) ..... 46
2.110 Lemma (Type of hereditary application) ..... 46
2.111 Lemma (Application inversion) ..... 46
2.112 Lemma (Iterated application inversion) ..... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Theorem/Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.113</td>
<td>Lemma (Projection inversion)</td>
</tr>
<tr>
<td>2.115</td>
<td>Lemma (Type application, reversed)</td>
</tr>
<tr>
<td>2.116</td>
<td>Lemma (Free variables in type application)</td>
</tr>
<tr>
<td>2.117</td>
<td>Lemma (Commuting of renamings with hereditary substitution and elimination)</td>
</tr>
<tr>
<td>2.118</td>
<td>Lemma (Commuting of renamings with WHNF)</td>
</tr>
<tr>
<td>2.119</td>
<td>Lemma (Commuting of renaming with reversed type application)</td>
</tr>
<tr>
<td>2.120</td>
<td>Lemma (Typing of metavariable bodies)</td>
</tr>
<tr>
<td>2.123</td>
<td>Remark (Metasubstitutions are signatures)</td>
</tr>
<tr>
<td>2.128</td>
<td>Remark (Compatibility with a declaration as a judgment: ( J = D ))</td>
</tr>
<tr>
<td>2.129</td>
<td>Remark (Alternative characterization of compatibility of a metasubstitution with a declaration)</td>
</tr>
<tr>
<td>2.130</td>
<td>Lemma (Alternative characterization of a compatible metasubstitution)</td>
</tr>
<tr>
<td>2.131</td>
<td>Remark (Compatibility of extended metasubstitutions with declarations)</td>
</tr>
<tr>
<td>2.133</td>
<td>Remark (Restriction to a compatible signature)</td>
</tr>
<tr>
<td>2.134</td>
<td>Remark (Subsumption of restriction)</td>
</tr>
<tr>
<td>2.135</td>
<td>Remark (Declarations in a metasubstitution restriction)</td>
</tr>
<tr>
<td>2.136</td>
<td>Remark (Nested metasubstitution restriction)</td>
</tr>
<tr>
<td>2.137</td>
<td>Remark (Metasubstitution weakening)</td>
</tr>
<tr>
<td>2.138</td>
<td>Remark (Metasubstitution strengthening)</td>
</tr>
<tr>
<td>2.141</td>
<td>Lemma (Existence of meta-free normal form)</td>
</tr>
<tr>
<td>2.142</td>
<td>Lemma (Compatibility of closing metasubstitution)</td>
</tr>
<tr>
<td>2.143</td>
<td>Lemma (Metasubstitution equality is an equivalence relation)</td>
</tr>
<tr>
<td>2.147</td>
<td>Lemma (Compatibility respects equality)</td>
</tr>
<tr>
<td>2.148</td>
<td>Lemma (Uniqueness of closing metasubstitution)</td>
</tr>
<tr>
<td>2.150</td>
<td>Lemma (Equality of restricted metasubstitutions)</td>
</tr>
<tr>
<td>2.152</td>
<td>Remark (Signature extension is reflexive and transitive)</td>
</tr>
<tr>
<td>2.153</td>
<td>Remark (Declarations in a signature extension)</td>
</tr>
<tr>
<td>2.154</td>
<td>Remark (Metasubstitution restriction to extension)</td>
</tr>
<tr>
<td>2.155</td>
<td>Lemma (Preservation of judgments under signature extensions)</td>
</tr>
<tr>
<td>2.157</td>
<td>Lemma (Restriction of a metasubstitution to an extended signature)</td>
</tr>
<tr>
<td>2.159</td>
<td>Remark (Prefixes of strongly neutral terms)</td>
</tr>
<tr>
<td>2.160</td>
<td>Remark (Closure of strongly neutral terms)</td>
</tr>
<tr>
<td>2.161</td>
<td>Remark (Intermediate steps of reduction of strong neutrals)</td>
</tr>
<tr>
<td>2.162</td>
<td>Remark (Reduction preserves strongly neutral terms)</td>
</tr>
<tr>
<td>2.163</td>
<td>Lemma (Injectivity of elimination for strongly neutral terms)</td>
</tr>
<tr>
<td>2.165</td>
<td>Remark (Extensions of irreducible terms)</td>
</tr>
<tr>
<td>2.166</td>
<td>Lemma (Reduction at II-type)</td>
</tr>
<tr>
<td>2.167</td>
<td>Lemma (Characterization of normal forms)</td>
</tr>
<tr>
<td>2.170</td>
<td>Lemma (Typing of rigid occurrences)</td>
</tr>
<tr>
<td>2.171</td>
<td>Remark (Free variables of rigid occurrence)</td>
</tr>
<tr>
<td>2.172</td>
<td>Lemma (Free variables in reduction of rigid occurrences)</td>
</tr>
</tbody>
</table>
List of Examples

2.25 Example (Strengthening by a variable: \((-1) + n\)) \hspace{1cm} 22

3.5 Example (Dependent type checking with metavariables, unique solution) \hspace{1cm} 65

3.6 Example (Dependent type checking with metavariables, no unique solution) \hspace{1cm} 65

3.15 Example (First-order problem) \hspace{1cm} 68

3.16 Example (Higher-order problem) \hspace{1cm} 69
List of Figures

2.1 Syntax for terms ............................................. 14
2.2 Metavariables occurring in a term. ........................... 18
2.3 Atoms occurring in a term. .................................. 18
2.4 Free variables in a term. .................................... 21
2.5 Applying a renaming \( \rho \) to a term. ......................... 23
2.6 Hereditary substitution and elimination ....................... 25
2.7 Cases for \( \delta \eta \)-reduction ................................. 32
2.8 Inductive definition of the weak head normal form relation. 44
2.9 Inductive definition of the meta-free normal form of a term. 52

3.1 Recursive definition of the set of holes in a term ............. 64

6.5 Increase in resource usage when type-checking selected projects 143
6.6 Impact of context-singleness optimization on resource usage 146
6.9 Increase in resource usage when type-checking selected benchmarks ................................................. 147

A.1 Quantile-quantile plot of CPU time measurements ............. 170
A.2 Empirical distribution of the geometric mean of the CPU time measurements ........................................ 171
A.3 Quantile-quantile plot of resident memory measurements .... 172
A.4 Empirical distribution of the geometric mean of the resident memory measurements ................................ 173
List of Listings

1.1 Non-unique implicit argument ........................................ 3
5.1 Check whether two sides of a twin type are equal ............... 128
5.2 Checking of heterogeneous equality of the local context ......... 130
5.3 Implementation in Agda of the check for the preconditions of Rule-Schema 2 (metavariable instantiation) .................... 131
5.4 Simplifying a twin type or term based on whether the associated constraints have been solved ............................ 132
5.5 Unifying two II-types .................................................. 132
5.6 Solving elimination constraints ...................................... 133
5.7 Checking for singleton types ........................................... 134

6.10 Rendering of Example 6.1 in Coq, together with the output from the type checker ..................................................... 150
6.11 Example in Listing 6.10 modified so that the codomains are equal 151
6.12 Example in Listing 6.10 modified so that the solution to the metavariable is given ...................................................... 151
6.13 Rendering of Example 6.2 in Coq, together with the output from the type checker ..................................................... 153
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 153
6.15 Rendering of Example 6.2 in Coq, with the solution to the metavariable given, together with the output from the type checker. 154
6.16 Example of first-order unification in Coq ......................... 156
6.17 Rendering of Example 6.1 in Idris, together with the output from the type checker ..................................................... 157
6.18 Idris 2 rendition of Example 6.3 (Left version) .................. 159
6.19 Idris 2 rendition of Example 6.3 (Right version). Compare with Listing 6.18 .............................................................. 159
6.20 Example in Listing 6.19 with the solution given explicitly .... 160
6.21 Rendering of Example 6.2 in Idris, together with the output from the type checker ..................................................... 160
6.22 Example 6.5 translated into Idris .................................... 161
6.23 Expanded version of Example 6.5 translated into Idris ........ 162

B.1 Rendering of Example 6.4 in Idris .................................. 175
B.2 Example in Listing B.1 with the solution to the metavariable given explicitly .............................................................. 176
B.3 Example in Listing B.1 with both sides of the constraint swapped 176
B.4 Example in Listing B.3 with the solution to the metavariable given explicitly .............................................................. 177
B.5 Rendering of the binder problem in Matita, together with the output from the type checker ........................................... 177
B.6 Example Listing B.5 modified so that the codomains are equal 179
B.7 Example Listing B.5 modified so that the metavariable alpha is solved from the start .................................................... 180
B.8 Rendering of the binder problem in Matita, together with the
output from the type checker. ........................................ 180
B.9 Example Listing B.8 modified so that the first arguments can
be made equal. ............................................................ 182
B.10 Example Listing B.8 modified so that the metavariable alpha is
solved from the start. .................................................... 182
B.11 Rendering of the binder problem in Lean, together with the
output from the type checker. .......................................... 183
B.12 Example Listing B.11 modified so that the codomains are equal. 184
B.13 Example Listing B.11 modified so that the metavariable alpha
is solved from the start. ................................................ 184
B.14 Rendering of the binder problem in Lean, together with the
output from the type checker. ........................................... 185
B.15 Example Listing B.14 modified so that the first arguments can
be made equal. .......................................................... 185
B.16 Example Listing B.14 modified so that the metavariable alpha
is solved from the start. ................................................ 186

List of Tables

6.1 Effort required to implement the functionality required in Chap-
ter 5 ................................................................. 138
6.2 List of modules in Agda, in descending total number of lines
added or changed. ...................................................... 139
6.3 CPU time for type-checking existing projects developed with
Agda. .................................................................. 142
6.4 Resident memory when type-checking existing projects devel-
oped with Agda. ....................................................... 142
6.7 CPU usage when type-checking examples from the benchmark
suite included with the Agda implementation. ................. 144
6.8 Resident memory when type-checking examples from the bench-
mark suite included with the Agda implementation. .......... 145
Bibliography


