University of Strathclyde

Department of Computer and Information Sciences

Type Inference, Haskell and Dependent Types

Adam Michael Gundry

Doctor of Philosophy

2013

Acknowledgements

I would like to thank my supervisor, Conor McBride, for everything he has taught me, for his support when I needed it, and for always being ready to talk. None of this would have been possible without him.

This work was supported by Microsoft Research through its PhD Scholarship Programme. I am particularly grateful to Simon Peyton Jones and Dimitrios Vytiniotis from Microsoft Research Cambridge for fascinating discussions about the ins and outs of Haskell and GHC.

My thanks for their feedback on this work, and for the many interesting discussions we shared, go to the rest of the F_C crew: Stephanie Weirich, Richard Eisenberg, José Pedro Magalhães and Iavor Diatchki. I thank Philippa Cowderoy, Ben Kavanagh and James McKinna for their help and insight. Nathan Collins kindly pointed out a bug in one of the appendices.

My colleagues in the Mathematically Structured Programming group have each contributed to this in their own way. My thanks go to all of them: Guillaume Allais, Stevan Andjelkovic, Bob Atkey, Pierre-Évariste Dagand, Clément Fumex, Neil Ghani, Peter Hancock, Patricia Johann, Clemens Kupke, Sam Lindley, Lorenzo Malatesta, Federico Orsanigo and Tim Revell. Stuart Hannah and Jason Smith, from the Strathclyde Combinatorics group, were always supportive.

My viva took place on Monday 25th November 2013, and the examiners were Andrew Kennedy and Anders Claesson. I greatly valued the opportunity to discuss the ideas with them, and thank them for their helpful comments. Of course, the remaining errors are my responsibility alone.

Finally, I thank my wife, Christine, for everything.

But peace to vain regrets! We see but darkly
Even when we look behind us, and best things
Are not so pure by nature that they needs
Must keep to all, as fondly all believe,
Their highest promise. If the mariner,
When at reluctant distance he hath passed
Some tempting island, could but know the ills
That must have fallen upon him had he brought
His bark to land upon the wished-for shore,
Good cause would oft be his to thank the surf
Whose white belt scared him thence, or wind that blew
Inexorably adverse: for myself
I grieve not; happy is the gownèd youth,
Who only misses what I missed, who falls
No lower than I fell.

The Prelude, or Growth of a Poet's Mind
William Wordsworth, 1850

Contents

List of Figures

Abstract

This thesis studies questions of type inference, unification and elaboration for languages that combine dependent type theory and functional programming. Languages such as modern Haskell have very expressive type systems, allowing the programmer a great deal of freedom. These require advanced type inference and unification algorithms to reconstruct details that were left implicit, and suitable representation of the evidence delivered by such algorithms.

The first part proposes an approach to unification and type inference, based on information increase in dependency-ordered contexts, and keeping careful track of variable scope. Two existing systems are reviewed: the Hindley-Milner type system, and units of measure in the style of Kennedy. Subtle issues relating to let-generalisation become clearer as a result. Using the same approach, an algorithm is described for Miller pattern unification in a full-spectrum dependent type theory, forming a foundation for the elaboration of dependently typed languages.

The second part introduces *inch*, a language that extends Haskell with type-level data and functions, and dependent product types. Type-level numbers and arithmetic operations are specifically considered, as a particularly useful source of applications, such as the perennial example of vectors (length-indexed lists). The increased expressivity in the source language is matched by a suitable core language of *evidence*, into which *inch* programs can be translated. This language is based on System $\mathrm{F_C}$, the existing core language used by GHC, adapted to clarify the relationships between the type and term levels. It gives a coherent operational semantics to both levels, allowing shared data and dependent functions, but retaining a clear phase distinction. The contextual approach of the first part of the thesis is used to specify the elaboration of *inch* into the *evidence* language, and applications of *inch* based on type-level arithmetic are demonstrated.

Part I

Foundations of type inference

Chapter 1

Introduction

This thesis explores the combination of the functional programming language Haskell with dependent type theory. It is addressed to the functional programmer who wants a language that provides stronger static guarantees and a more expressive type system than modern Haskell, while maintaining the phase distinction and useful, if not necessarily complete, type inference. I will assume the reader has some familiarity with Haskell or a similar functional language, but not necessarily a great deal of familiarity with type theory. Experience of advanced type system features such as generalised algebraic datatypes and higher-rank types would be beneficial.

Haskell is a functional language with Hindley-Milner type inference in the tradition of ML. Thanks to type inference, the burden of type annotations is minimised, if not necessarily eliminated.[1] Moreover, the typeclass system enables *term inference*: types function as a real aid to the programmer, not just a safety net that prevents bad programs, as the compiler can write runtime code for the user. For example, Haskell's Eq typeclass can be used to compute an equality test for complex structured data from the equality tests on the component types.

Dependent types allow term-level data into the static type system. This allows more precise invariants to be specified: for example, rather than the type of lists of arbitrary length, one can work with the type of vectors of a statically-known length. Term inference becomes easier, because the presence of terms in types leads to equational constraints on terms, and solving these constraints may allow the compiler to discover runtime-relevant values. While typeclasses allow terms to be discovered by evaluating logic programs, dependent types allow them to be discovered by solving equations in the underlying functional language.

[1] I include approaches requiring a little annotation, sometimes called 'type reconstruction', under the general term 'type inference'. Type inference is *pure* if no annotations are required.

Haskell is a good basis for extension with dependent types because it is already widely used as a testbed for type system extensions. Numerous advanced features, that push the boundaries of type inference, have been adopted in the Glasgow Haskell Compiler (GHC): notably higher-rank types, which allow universal quantification in the domain of a function, and generalised algebraic datatypes, which allow data constructors to introduce equational constraints on types. While such extensions make pure type inference infeasible, this can be a price worth paying, particularly given the huge increase in expressivity achieved, the potential for term inference and the value of annotations as machine-checked documentation.

1.0.1 Contexts, variable scope and let-generalisation

One of the main themes of this thesis is the proper management of variable scope, which is crucial for correctly implementing type inference. Type inference algorithms create existential variables to stand for unknown type expressions, then solve for these variables by unification. Once higher-rank types are available, it is necessary to carefully manage which universally quantified variables are in scope for each existential variable. Even in the Hindley-Milner system, however, variable dependencies are key to understanding the process of let-generalisation.

Let-generalisation is used to assign polymorphic types to definitions. In

$$\mathbf{let}\, f\ x = (x, x)\ \mathbf{in}\ (f\ \mathsf{True}, f\ 3) :: ((\mathsf{Bool}, \mathsf{Bool}), (\mathsf{Int}, \mathsf{Int}))$$

the term is well-typed because f is assigned the type $\forall\, a\, .\, a \rightarrow (a, a)$. This type is determined by inferring the type $\beta \rightarrow (\beta, \beta)$, where β is an existential variable, then quantifying over β. In more complex examples it is not always possible to quantify over all the existential variables, as they may have meaning outside the local scope of the let-binding. This will be examined in more detail in Chapter 2.

All this motivates taking more care over metavariables than is traditional for the Hindley-Milner system. I will introduce a notion of *context* that tracks metavariable declarations and imposes a dependency-respecting order upon them. Considering *contextualised* unification and type inference problems leads to a precise notion of the minimal commitment necessary to solve a problem, and reveals the underlying structure that makes sense of the let-generalisation step. This structure makes it easier to deal with systems where variable dependency is more subtle than in Hindley-Milner, such as units of measure in the style of Kennedy (2010), considered in Chapter 3. Contexts can be extended to contain universal as well as existential variables, a ‘mixed prefix’ in the language of Miller (1992), allowing the analysis to be extended to dependent types, as in Chapter 4.

1.0.2 Dependent types in GHC Haskell

Simulating dependent types in Haskell is a cottage industry (McBride, 2002), and recent extensions to GHC allow some dependent datatypes to be defined reasonably neatly. The standard example of vectors of a fixed length is given by:

```
data ℕ = Zero | Suc ℕ

data Vec :: * → ℕ → * where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Suc n)
```

Datatype promotion (Yorgey et al., 2012) allows the data*type* $\mathbb{N}$ to be used in the *kind* of Vec, and correspondingly the Zero and Suc *data* constructors appear in the *types* of Nil and Cons. Moreover, Vec a m is a generalised algebraic datatype or GADT (Peyton Jones et al., 2006), meaning that pattern matching on its constructors supplies information to the typechecker: a proof of the equation $m \sim$ Zero in the Nil branch, and a proof of $m \sim$ Suc n in the Cons branch.

This type-level knowledge of length is useful for expressing more precise invariants in types, leading to more reliable code. The tail function for vectors

```
tail :: Vec a (Suc n) → Vec a n
tail (Cons _ xs) = xs
```

statically enforces the invariant that its argument list must be non-empty, so this definition is total, and it is guaranteed to return a result of the right length.

Type families (Chakravarty et al., 2005), which approximate functions on the type level, allow the definition of operations on type-level data. Addition for type-level naturals can be defined, then used in the type of vector concatenation:

```
type family (m :: ℕ) + (n :: ℕ) :: ℕ
type instance Zero   + n = n
type instance Suc m + n = Suc (m + n)

append :: Vec a m → Vec a n → Vec a (m + n)
append Nil          ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

However, type families do not correspond exactly to term-level functions, because they are open, that is, defining equations can be added anywhere. They are not translated into case analysis, but are understood as rewrite rules on the syntax of type expressions. This gap between the term-level and type-level operational semantics is problematic for dependent types, where the same expression may be used both statically (in the typechecker) and dynamically (at runtime).

1.0.3 The value of Π: going beyond GHC Haskell

Vector concatenation relies only on (implicit) universal quantifiers and runtime functions. However, consider the vector version of the replicate function, which creates a vector of length n by repeating its second argument n times:

$$\begin{aligned}&\mathsf{replicate} :: \Pi\ (n :: \mathbb{N}) \to a \to \mathsf{Vec}\ a\ n\\ &\mathsf{replicate}\ \mathsf{Zero}\quad\ \ \_ = \mathsf{Nil}\\ &\mathsf{replicate}\ (\mathsf{Suc}\ n)\ x = \mathsf{Cons}\ x\ (\mathsf{replicate}\ n\ x)\end{aligned}$$

Here the result type $\mathsf{Vec}\ a\ n$ depends on n, but the operational behaviour of the function also makes uses of n, as it is defined by pattern matching. This shows the need for the dependent product Π: it is a function space where the value is available both statically and dynamically. GHC Haskell does not currently support Π, but it can be encoded in some cases. Adding Π to Haskell is the main contribution of part II of this thesis. Chapter 5 describes the resulting language.

1.0.4 Type inference and term inference

Dependent type theory offers a significant extension of the verification that can be performed by types: ultimately, the full power of constructive mathematics can be used to specify and prove properties of programs. However, this power comes at a cost. Inferring the most general type of the composition operator

$$(g \circ f)\ x = g\ (f\ x)$$

is straightforward in Haskell, where it has type

$$(b \to c) \to (a \to b) \to (a \to c)\,.$$

If the codomain of f may depend on the value of x, and g may depend on x and $f\ x$, then the type becomes more complicated. A possible type for composition is

$$\begin{aligned}&\{A : \mathsf{Set}\}\ \{B : A \to \mathsf{Set}\}\ \{C : (a : A) \to B\ a \to \mathsf{Set}\}\\ &\quad (g : \{a : A\}\ (b : B\ a) \to C\ a\ b)\ (f : (a : A) \to B\ a)\ (a : A) \to C\ a\ (f\ a)\end{aligned}$$

in Agda notation,[2] ignoring universe polymorphism. It is not reasonable to ask a machine to reconstruct this type from the definition.

As I have noted, types are not simply a form of statically-checked documentation or a policing system that prohibits bad programs, important as these roles

[2]A dependent function space (Π-type) is written $(x : S) \to T$ or $\{x : S\} \to T$, where x is bound in T. The type Set is a universe of small types, resembling the Haskell kind $*$.

are. In exchange for writing more expressive types, the programmer can be repaid by having to write less of their program: term inference becomes feasible. Typeclasses accomplish this to a certain extent, but the presence of computational data in types means that constraints on types can determine runtime information. The replicate function defined in Subsection 1.0.3 makes crucial runtime use of its natural number argument. If it is used in a context demanding a value of type Vec a 42, the programmer should not need to supply that argument explicitly!

Users of a dependently typed language, if they wish to prove properties of their programs, have much work to do in choosing appropriate representations of data structures and ways to enforce invariants. On the other hand, significant benefits can be gained with less work by selectively establishing invariants that use the type system to prevent certain errors, guaranteeing the absence of a class of bugs, if not the absence of bugs altogether. Perhaps the way forward lies in a mixed economy: a system that combines the flexibility of Haskell with the reliability of dependent type theory. This is the approach that I will pursue.

1.1 Outline

This thesis falls into two parts: the first develops foundations for describing and analysing type inference, and the second builds on this work to introduce the *inch* system, extending Haskell with dependent types. Reference implementations of the algorithms in part I and details of selected proofs are given in the appendices.

Part I: Foundations of type inference

In Chapter 2, I start at the very beginning with a rationalised reconstruction of type inference for the Hindley-Milner type system, and its constraint-solving algorithm, first-order unification. This introduces a method of contextualised problem-solving that sustains the later development. Paying careful attention to variable scope makes evident the underlying structure on first-order unification that explains let-generalisation. Furthermore, I describe how to elaborate Hindley-Milner terms into System F, representing term structure in the context.

Following on from this in Chapter 3, I extend the basic Hindley-Milner system with Kennedy-style units of measure. This requires unification in the equational theory of abelian groups. I show how the contextual structure introduced in Chapter 2 makes let-generalisation straightforward, even in this more complex setting where variable occurrence does not imply dependency on that variable.

Taking a different direction in Chapter 4, I apply the same techniques of contextualised problem-solving to higher-order unification, where the correct management of scope is crucial. I describe an algorithm for Miller pattern unification in a full-spectrum dependent type theory. Higher-order unification is needed for implementing type inference for dependently-typed programming languages, as constraint-solving must take place in the definitional equality of the type theory. Not all equations can be solved immediately, so the algorithm must represent constraints explicitly and make most general progress where possible.

Part II: Haskell with dependent types

Having constructed the foundations, I build on them in the second part to create *inch*, a language based on Haskell with Π-types and type-level data. In Chapter 5, I introduce the main features of the language by example and compare it to related work. This chapter contains a more thorough introduction to the encoding of dependently-typed programs in Haskell via GADTs and type families.

To explain *inch* formally, I build an *evidence* language in Chapter 6, based on GHC's intermediate language System F_C, but influenced by Martin-Löf Type Theory. The *evidence* language is a very explicit calculus for which typechecking is straightforward. I give a precise account of the phase distinction, as Π means that the categories of runtime and type-level data are no longer mutually exclusive. The operational semantics of the *evidence* language, with type safety proof, makes explicit the computational role of dependent Π-types. Also, I present a new approach to proving consistency of coercions (which witness type equalities).

In Chapter 7, I describe type inference for *inch* via elaboration into the *evidence* language, using the ideas of contextualised problem-solving from the first part of the thesis. In particular, the elaboration algorithm clarifies the management of implicit and explicit arguments. Elaboration relies on an underlying constraint solver, which I do not study in detail, though it would use similar techniques to the unification algorithms from Part I.

The payoff for all this work appears in Chapter 8, where I present applications of *inch*, using dependent types to provide stronger guarantees of correctness. I give examples of vector functions, merge sort and red-black tree insertion and deletion, and show how the time complexity of such programs can be statically checked. Additionally, I demonstrate an approach to units of measure as a library based on type-level integers, in contrast to the built-in treatment in Chapter 3.

Finally, some concluding remarks form Chapter 9.

Chapter 2

A rationalised reconstruction of Hindley-Milner type inference

In this chapter I rebuild first-order unification and Hindley-Milner type inference from the ground up. A key theme of this thesis is the proper understanding of scope, achieved by keeping variables (especially 'unification variables' or 'metavariables') in contexts. Applying the variables-in-contexts approach to a standard type inference problem allows me to emphasise this theme, before moving on to more advanced type systems. This chapter is based on the paper "Type inference in context" by Gundry, McBride, and McKinna (2010). Appendix A (page 197) contains a Haskell implementation of the algorithm described here.

The Hindley-Milner type system[1] (Milner, 1978) consists of the simply-typed λ-calculus plus 'let-expressions' for polymorphic definitions. For example,

$$\textbf{let}\, x = \lambda y.y \,\textbf{in}\, x\, x$$

is well-typed: x is given the polymorphic type $\forall\alpha.\, \alpha \to \alpha$, which is instantiated in two different ways, first at type $(\beta \to \beta) \to (\beta \to \beta)$ and second at type $\beta \to \beta$. In contrast, λ-bound variables are monomorphic, so $\lambda x.x\,x$ is ill-typed.

The syntax of terms and types is

$$\begin{array}{lcl} t,\ s & ::= & x \mid \lambda x.t \mid s\,t \mid \textbf{let}\, x = s \,\textbf{in}\, t \\ \tau,\ \upsilon & ::= & \alpha \mid \tau \to \upsilon \end{array}$$

where x and y range over term variables, and α and β range over type variables. For simplicity, the function arrow $\to$ is the only type constructor.

[1]The work of Hindley (1969) was in type inference for combinatory logic, unlike Milner's type system with let-polymorphism, but 'Hindley-Milner' is the name that has stuck.

To handle let-polymorphism, the context assigns each term variable a *type scheme* σ rather than a monomorphic type. A type scheme is a type wrapped in one or more $\forall$-quantified variables, with the syntax

$$\sigma \quad ::= \quad \tau \mid \forall \alpha .\, \sigma$$

Morally, one should distinguish between the 'universally quantified' variables in type schemes, and 'existentially quantified' variables (known as 'metavariables', 'unification variables' or 'holes') for which solutions are found by unification during type inference. However, for this chapter I can conflate the two: variables are always bound in type schemes, while metavariables are always free in the context.

Milner's typing rules, as presented by Clément et al. (1986) adapted into algorithmic form, appear in Figure 2.1. The context A is an unordered set of type scheme bindings, with A_x denoting 'A minus any x binding': such contexts do not reflect lexical scope, so shadowing requires deletion and reinsertion.

Algorithm $\mathcal{W}$ is a well-known type inference algorithm for the Hindley-Milner system, due to Damas and Milner (1982), and based on the Unification Algorithm of Robinson (1965). Most presentations of Algorithm $\mathcal{W}$ have treated the underlying unification algorithm as a 'black box', but by considering both together I will show that the generalisation step (used when inferring the type of a let-expression) becomes straightforward (Section 2.3).

Why revisit Algorithm $\mathcal{W}$? As a first step towards a larger goal: explaining how to elaborate high-level *dependently typed* programs into fully explicit calculi, as in Chapter 7. Just as $\mathcal{W}$ specialises polymorphic type schemes, elaboration involves inferring *implicit arguments* by solving constraints, but with fewer algorithmic guarantees. Pragmatically, we need to account for stepwise progress in problem solving from states of partial knowledge. I seek local correctness criteria for type inference that guarantee global correctness.

2.0.1 The occurs check

Testing whether a variable occurs in a term is used by both Robinson unification and Algorithm $\mathcal{W}$. In unification, the check is usually necessary to ensure termination, let alone correctness: the equation $\alpha \equiv \alpha \to \beta$ has no finite solution because the right-hand side depends on the left, so it does not make a good definition for α.[2]

[2]Of course, this assumes types are inductively defined: coinductive systems, which allow infinitary types as the solutions of such equations, are outside the scope of this thesis.

$\boxed{A \vdash t : \sigma}$ *(term t has type scheme σ under assumptions A)*

$$\frac{x : \sigma \in A \qquad \sigma \succeq \tau}{A \vdash x : \tau} \qquad \frac{A \vdash t : \tau' \to \tau \qquad A \vdash t' : \tau'}{A \vdash t\, t' : \tau} \qquad \frac{A_x \cup \{x : \tau'\} \vdash t : \tau}{A \vdash \lambda x. t : \tau' \to \tau}$$

$$\frac{A \vdash t' : \tau' \qquad \sigma = \mathrm{gen}(A, \tau') \qquad A_x \cup \{x : \sigma\} \vdash t : \tau}{A \vdash \textbf{let}\, x = t' \,\textbf{in}\, t : \tau}$$

$\sigma \succeq \tau$ if τ is a generic instance of σ (specialising σ yields τ)

$$\mathrm{gen}(A, \tau) = \begin{cases} \forall \overline{\alpha_i}^{\,i \in 1..n}.\, \tau & (FV(\tau) \setminus FV(A) = \{\alpha_1, \ldots, \alpha_n\}) \\ \tau & (FV(\tau) \setminus FV(A) = \emptyset) \end{cases}$$

Figure 2.1: Milner's typing rules

In Algorithm $\mathcal{W}$, the occurs check is used to discover type dependencies just in time for generalisation. When inferring the type of $\textbf{let}\, x = t' \,\textbf{in}\, t$, the type of t' must first be inferred, then 'generic' type variables, those occurring in t' but not the enclosing bindings, must be quantified over. The idea is that type variables may be generalised over (and freely substituted) if they are not recording a necessary coincidence. For example, a typing derivation for $\lambda y. \textbf{let}\, x = y \,\textbf{in}\, x$ might have $\{y : \alpha\} \vdash y : \alpha$ for the definiens. One is certainly not free to generalise over α, as this would allow any type to be assigned to x! On the other hand, a derivation for $\textbf{let}\, x = \lambda y.y \,\textbf{in}\, x\, x$ could include $\emptyset \vdash \lambda y.y : \alpha \to \alpha$, and α must be generalised over for the whole expression to be well-typed.

In both unification and type inference, the occurs check is used to detect dependencies between variables. The traditional approach of leaving unification variables floating in space, without any structure, works for the Hindley-Milner system because there are no scoping conditions on candidate solutions for variables. This will not always be the case, so it is better to expose the structure and manage dependencies explicitly.

In further contrast to other presentations of unification and Hindley-Milner type inference, the algorithm I will describe is based on contexts carrying variable *definitions* as well as *declarations*. This allows the context to record the entire result of the algorithm.

2.1 A framework for contextual problem solving

Let me begin by revisiting unification for type expressions with free variables. In order to address the problem of solving equations, I must first explain which types are considered equal, raising the question of which things a given context admits as types, and which contexts make sense in the first place.

A context Θ is a dependency-ordered list of unknown type metavariables, definitions of metavariables and given term variables:

$$\Theta \quad ::= \quad \cdot \mid \Theta, \alpha : * \mid \Theta, \alpha := \tau : * \mid \Theta, x : \sigma \mid \Theta \fatsemi$$

It is divided into 'localities' by the $\fatsemi$ marker, the role of which will be explained in Subsection 2.1.2. I write Ξ for a context suffix containing only metavariables.

Contexts introduce named variables and ascribe properties to them, but the properties should first make sense. The rules in Figure 2.3 define the judgment $\Theta \vdash$ **ctx**, which checks that a context is *valid*, i.e. that every variable is distinct and each property is well-formed for the preceding context. Definitions $\alpha := \tau : *$ and term variable bindings $x : \sigma$ make sense only if the type τ or scheme σ is well-scoped, as verified by the judgment $\Theta \vdash \sigma : *$.

For example, the context $\alpha : *, \beta : *, x : \alpha \to \beta$ is valid, while $x : \alpha, \alpha : *$ is not, because α is not in scope for x. This dependency-ordering means that entries on the right are harder to depend on, and correspondingly easier to generalise.

Variables must not be duplicated in a context. In the rules, $\alpha \# \Theta$ means α is fresh for (does not occur in) Θ. I will usually ignore freshness issues: in practice, locally nameless representations (McBride and McKinna, 2004) are sufficient.

Metavariables definitions induce a nontrivial equational theory on types, as given in Figure 2.3. The definitions in a context represent a substitution in 'triangular form' (Baader and Snyder, 2001), that can be applied on demand to produce a type or type scheme that contains only unknown metavariables.

Unification is the problem of finding definitions for metavariables in order to make an equation hold. Type inference involves solving unification problems and finding a type that makes a typing judgment hold. Solutions to both problems should be 'most general' in that they should make the least commitment necessary to solve the equation or assign a type. In the following subsections, I will make this more precise by introducing a general notion of 'statements' that can be judged in contexts, and defining the permissible 'information increases' that move a context toward making a statement hold.

$$
\begin{array}{lrcl}
\text{Term variables} & x, y & & \\
\text{Type metavariables} & \alpha, \beta, \gamma & & \\
\text{Contexts} & \Theta & ::= & \cdot \mid \Theta, \alpha{:}{*} \mid \Theta, \alpha{:=}\tau : {*} \mid \Theta, x{:}\sigma \mid \Theta\fatsemi \\
\text{Suffixes} & \Xi & ::= & \cdot \mid \Xi, \alpha{:}{*} \mid \Xi, \alpha{:=}\tau : {*} \\
\text{Types} & \tau,\ \upsilon & ::= & \alpha \mid \tau \rightarrow \upsilon \\
\text{Type schemes} & \sigma & ::= & \tau \mid \forall\alpha.\,\sigma \\
\text{Terms} & t,\ s & ::= & x \mid \lambda x.t \mid s\,t \mid \mathbf{let}\,x{=}s\,\mathbf{in}\,t \\
\text{Statements} & J & ::= & \mathbf{ctx} \mid \sigma{:}{*} \mid \tau \equiv \upsilon{:}{*} \mid t{:}\sigma \mid \sigma \succ \sigma' \mid J \ \wedge\ J'
\end{array}
$$

Figure 2.2: Syntax

$\boxed{\Theta \vdash \mathbf{ctx}}$ *(Θ is a valid context)*

$$
\frac{}{\cdot \vdash \mathbf{ctx}}
\qquad
\frac{\begin{array}{c}\alpha\#\Theta \\ \Theta \vdash \mathbf{ctx}\end{array}}{\Theta, \alpha{:}{*} \vdash \mathbf{ctx}}
\qquad
\frac{\begin{array}{c}\alpha\#\Theta \\ \Theta \vdash \tau{:}{*}\end{array}}{\Theta, \alpha{:=}\tau : {*} \vdash \mathbf{ctx}}
\qquad
\frac{\begin{array}{c}x\#\Theta \\ \Theta \vdash \sigma{:}{*}\end{array}}{\Theta, x{:}\sigma \vdash \mathbf{ctx}}
\qquad
\frac{\Theta \vdash \mathbf{ctx}}{\Theta\fatsemi \vdash \mathbf{ctx}}
$$

$\boxed{\Theta \vdash \sigma{:}{*}}$ *(σ is a well-formed type scheme in Θ)*

$$
\frac{\Theta \ni \alpha{:}{*} \qquad \Theta \vdash \mathbf{ctx}}{\Theta \vdash \alpha{:}{*}}
\qquad
\frac{\Theta \vdash \tau{:}{*} \qquad \Theta \vdash \upsilon{:}{*}}{\Theta \vdash \tau \rightarrow \upsilon{:}{*}}
\qquad
\frac{\Theta, \alpha{:}{*} \vdash \sigma{:}{*}}{\Theta \vdash \forall\alpha.\,\sigma{:}{*}}
$$

$\boxed{\Theta \vdash \tau \equiv \upsilon : {*}}$ *(τ and υ are equal types in Θ)*

$$
\frac{\Theta \vdash \tau{:}{*}}{\Theta \vdash \tau \equiv \tau : {*}}
\qquad
\frac{\Theta \vdash \tau \equiv \upsilon : {*}}{\Theta \vdash \upsilon \equiv \tau : {*}}
\qquad
\frac{\Theta \vdash \tau_0 \equiv \tau_1 : {*} \qquad \Theta \vdash \tau_1 \equiv \tau_2 : {*}}{\Theta \vdash \tau_0 \equiv \tau_2 : {*}}
$$

$$
\frac{\Theta \vdash \mathbf{ctx} \qquad \Theta \ni \alpha{:=}\tau : {*}}{\Theta \vdash \alpha \equiv \tau : {*}}
\qquad
\frac{\Theta \vdash \tau \equiv \tau' : {*} \qquad \Theta \vdash \upsilon \equiv \upsilon' : {*}}{\Theta \vdash \tau \rightarrow \tau' \equiv \upsilon \rightarrow \upsilon' : {*}}
$$

Figure 2.3: Rules for context validity, well-formed schemes and type equality

2.1.1 Modelling statements-in-context

Having introduced contexts, now I will give a general picture of 'statements-in-context', allowing unification and type inference to be viewed in a uniform setting. A *statement* is an assertion that can be judged in a context, with grammar

J	$::=$		
	$\mid$	$\mathbf{ctx}$	context validity
	$\mid$	$\sigma : *$	well-formed type scheme
	$\mid$	$\tau \equiv \upsilon : *$	equivalent types
	$\mid$	$t : \sigma$	well-typed term
	$\mid$	$\sigma \succ \sigma'$	generic instantiation of type schemes
	$\mid$	$J \wedge J'$	conjunction of statements

The rules for valid contexts, well-formed type schemes and type equality are given in Figure 2.3. The rules for well-typed terms and generic instantiation of type schemes will be given in Section 2.3 (Figures 2.6 and 2.7). The conjunction statement has a single introduction rule and admissible elimination rules:

$$\frac{\Theta \vdash J \qquad \Theta \vdash J'}{\Theta \vdash J \wedge J'} \qquad\qquad \frac{\Theta \vdash J \wedge J'}{\Theta \vdash J} \qquad \frac{\Theta \vdash J \wedge J'}{\Theta \vdash J'}$$

Each statement J has a corresponding *sanity condition*, $\mathbf{San}\, J$, whose truth is necessary for J to make sense. For example, the sanity condition for a typing statement is that the type is well-formed. Sanity conditions cannot be presupposed when writing the rules; rather, care must be taken to ensure them. The sanity conditions are given by the following lemma.

Lemma 2.1 (Sanity conditions). *If* $\Theta \vdash J$ *then* $\Theta \vdash \mathbf{San}\, J$*, where*

$$\begin{array}{rcl}
\mathbf{San}\,\mathbf{ctx} & \mapsto & \mathbf{ctx} \\
\mathbf{San}\,(\sigma : *) & \mapsto & \mathbf{ctx} \\
\mathbf{San}\,(\tau \equiv \upsilon) & \mapsto & \tau : * \ \wedge \ \upsilon : * \\
\mathbf{San}\,(t : \sigma) & \mapsto & \sigma : * \\
\mathbf{San}\,(\sigma \succ \sigma') & \mapsto & \sigma : * \ \wedge \ \sigma' : * \\
\mathbf{San}\,(J \wedge J') & \mapsto & \mathbf{San}\, J \ \wedge \ \mathbf{San}\, J'
\end{array}$$

Proof. By structural induction on derivations. The sanity condition for the **ctx** statement is uninformative, as it merely says that $\Theta \vdash$ **ctx** implies itself. □

Sanity conditions capture the requirements for a statement to be 'meaningful', before one can ask whether it is 'true' (Martin-Löf, 1996).

$\boxed{\theta : \Theta_0 \sqsubseteq \Theta_1}$ *(θ is a metasubstitution from Θ_0 to Θ_1)*

$$\frac{}{[\,] : \cdot \sqsubseteq \Xi} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \tau : *}{(\theta, \tau/\alpha) : \Theta_0, \alpha{:}* \sqsubseteq \Theta_1} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \tau \equiv \theta\,\upsilon : *}{(\theta, \tau/\alpha) : \Theta_0, \alpha{:=}\upsilon : * \sqsubseteq \Theta_1}$$

$$\frac{\theta : \Theta_0 \sqsubseteq \Theta_1}{\theta : \Theta_0, x{:}\sigma \sqsubseteq \Theta_1, x{:}\theta\,\sigma, \Xi} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1}{\theta : \Theta_0 \fatsemi \sqsubseteq \Theta_1 \fatsemi \Xi}$$

$\boxed{\theta \equiv \theta' {:} \Theta_0 \sqsubseteq \Theta_1}$ *(θ and θ' are equivalent metasubstitutions from Θ_0 to Θ_1)*

$$\frac{}{\cdot \equiv \cdot {:} \cdot \sqsubseteq \Theta_1} \qquad \frac{\theta \equiv \theta' {:} \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \tau \equiv \tau' {:} *}{(\theta, \tau/\alpha) \equiv (\theta', \tau'/\alpha) {:} \Theta_0, \alpha{:}* \sqsubseteq \Theta_1}$$

$$\frac{\theta \equiv \theta' {:} \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \tau \equiv \theta\,\upsilon {:} * \qquad \Theta_1 \vdash \tau \equiv \tau' {:} *}{(\theta, \tau/\alpha) \equiv (\theta', \tau'/\alpha) {:} \Theta_0, \alpha{:=}\upsilon : * \sqsubseteq \Theta_1}$$

$$\frac{\theta \equiv \theta' {:} \Theta_0 \sqsubseteq \Theta_1}{\theta \equiv \theta' {:} \Theta_0, x{:}\sigma \sqsubseteq \Theta_1, x{:}\theta\,\sigma, \Xi} \qquad \frac{\theta \equiv \theta' {:} \Theta_0 \sqsubseteq \Theta_1}{\theta \equiv \theta' {:} \Theta_0 \fatsemi \sqsubseteq \Theta_1 \fatsemi \Xi}$$

Figure 2.4: Metasubstitutions

2.1.2 An information order for contexts

In order to describe algorithms that make incremental progress by modifying the context (substituting for variables or turning unknowns into definitions), I must specify what constitutes progress. This amounts to giving an 'information order' on contexts, so that increasing in the order makes a context 'more informative', i.e. more statements hold.

Let Θ_0 and Θ_1 be valid contexts. An *information increase* or *metasubstitution from* Θ_0 *to* Θ_1 is a finite map θ from metavariables in Θ_0 to well-formed types in Θ_1, that respects the structure and dependency order of Θ_0. Figure 2.4 gives rules for the judgment $\theta : \Theta_0 \sqsubseteq \Theta_1$ that explains when θ is a metasubstitution. This can be understood by looking at the form of Θ_0 in each rule. If it is empty, then Θ_1 may contain metavariable declarations Ξ but no fixed structure. If the last entry in Θ_0 is a metavariable, then θ must give a well-formed type in Θ_1 to substitute for the metavariable, which should agree with the existing definition (if any). If the last entry is a term variable or $\fatsemi$ marker, then Θ_1 must have the same structure. Recall that a context suffix Ξ contains only metavariable declarations, not term variables or $\fatsemi$ markers, so it may always be added without changing the underlying structure.

Metasubstitutions act on types and statements in the obvious way, extending the action on variables

$$\theta\,\alpha \;\mapsto\; \tau \qquad \text{if} \qquad \tau/\alpha \in \theta$$

homomorphically on syntax. The identity metasubstitution $\iota : \Theta \sqsubseteq \Theta'$ where Θ' includes all the variables of Θ, usually just written $\Theta \sqsubseteq \Theta'$, replaces each variable with itself. A finite list of type-metavariable pairs, such as $[\tau/\alpha]$, represents a metasubstitution that is the identity except where specified.

Equivalence of metasubstitutions, written $\theta \equiv \theta' : \Theta_0 \sqsubseteq \Theta_1$ or simply $\theta \equiv \theta'$ when the contexts are obvious, means that the corresponding types are equal, as shown in Figure 2.4.

Stable statements

Intuitively, substituting a type τ for a metavariable α should not be able to falsify any existing equations. More generally, making contexts more informative should preserve derivability of judgments. What is it about the design of the deduction system that ensures this?

A statement J is *stable* if it is preserved by metasubstitution, i.e., if

$$\Theta_0 \vdash J \quad \text{and} \quad \theta : \Theta_0 \sqsubseteq \Theta_1 \quad \Rightarrow \quad \Theta_1 \vdash \theta\, J.$$

That is, a simultaneous substitution on syntax extends to apply to derivations of stable statements: information increase is really the extension of simultaneous substitution from variables-and-terms to declarations-and-derivations.

As context entries ascribe properties to variables, so statements ascribe properties to expressions. Each entry corresponds directly to a statement: $\alpha : *$ and $x : \sigma$ are both entries and statements, while $\alpha := \tau : *$ corresponds to $\alpha \equiv \tau : *$. A context entry causes the corresponding judgment to hold, that is, the rule

$$\frac{\Theta \ni J}{\Theta \vdash J}\ \textsc{lookup}$$

is admissible. Compare this to the variable rule of a type theory: as variables embed in terms, so contextual properties of variables embed in judgments.

There is a systematic technique to ensure the stability of statements by construction of the deduction system: the only rules using information from the context should correspond to LOOKUP, asserting that an entry in the context holds as a statement. It is then enough to check that recursive hypotheses occur in strictly positive positions, so they are stable by induction.

Lemma 2.2 (Stability)**.** *If $\Theta_0 \vdash J$ then J is stable.*

Proof. By structural induction on derivations. □

Stability means that information increases are closed under composition, where $\theta_2 \cdot \theta_1$ is defined by applying θ_2 to every type in θ_1.

Lemma 2.3 (Category of contexts)**.** *Contexts form a category with information increases as morphisms. In particular,*

$$\theta_1 : \Theta_0 \sqsubseteq \Theta_1 \quad \textit{and} \quad \theta_2 : \Theta_1 \sqsubseteq \Theta_2 \quad \Rightarrow \quad \theta_2 \cdot \theta_1 : \Theta_0 \sqsubseteq \Theta_2.$$

Proof. It is straightforward to verify that composition is associative and has identity ι. To show closure under composition, proceed by induction on Θ_0.

If Θ_0 is empty, then θ_1 is trivial, so $\theta_2 \cdot \theta_1$ is trivial. Moreover Θ_1 consists only of metavariable declarations, so the same applies to Θ_2.

If $\Theta_0 = \Theta_0', \alpha :*$ then $\theta_1 = \theta_1', \tau/\alpha$ where $\theta_1' : \Theta_0' \sqsubseteq \Theta_1$ and $\Theta_1 \vdash \tau : *$. Now induction gives $\theta_2 \cdot \theta_1' : \Theta_0' \sqsubseteq \Theta_2$ and $\Theta_2 \vdash \theta_2\, \tau : *$ by stability, so $\theta_2 \cdot \theta_1 : \Theta_0 \sqsubseteq \Theta_2$ since $\theta_2 \cdot (\theta_1, \tau/\alpha) = (\theta_2 \cdot \theta_1), (\theta_2\, \tau)/\alpha$. The case where Θ_0 ends with a defined metavariable is similar, using stability of the equality statement.

If $\Theta_0 = \Theta_0', x : \sigma$ then $\Theta_1 = \Theta_1', x : \theta_1\, \sigma, \Xi_1$ and $\theta_1 : \Theta_0' \sqsubseteq \Theta_1'$. Similarly $\Theta_2 = \Theta_2', x : (\theta_2 \cdot \theta_1)\, \sigma, \Xi_2$ and $\theta_2 : \Theta_1' \sqsubseteq \Theta_2'$. Now induction gives $\theta_2 \cdot \theta_1 : \Theta_0' \sqsubseteq \Theta_2'$. □

Preserving structure in the context: the $⨾$ separator

The unification and type inference algorithms given later will exploit the declaration order in the context, moving declarations left as little as possible. Thus the rightmost entries will be the ‘most local’. Moving a declaration left (making it ‘more global’) reduces the choice of solutions, but increases the visibility of the variable, widening its scope. The ordering constraints will be particularly useful for implementing type inference for the let-expressions, in order to generalise over ‘local’ type variables but not ‘global’ variables.

A *locality* is a section of a context Θ that contains only metavariables, so term variables and the marker $⨾$ separate localities. The definition of metasubstitution $\theta : \Theta_0 \sqsubseteq \Theta_1$ makes the localities of Θ_0 and Θ_1 correspond, so that declarations in any prefix of Θ_0 can be interpreted over the corresponding prefix of Θ_1. Thus if $\theta : \Theta_0 \fatsemi \Theta'_0 \sqsubseteq \Theta$ then $\Theta = \Theta_1 \fatsemi \Theta'_1$ where $\theta|_{\Theta_0} : \Theta_0 \sqsubseteq \Theta_1$. (Here $\theta|_{\Theta_0}$ is the metasubstitution θ restricted to the metavariables in Θ_0.)

As a consequence, moving a metavariable 'left of a $\fatsemi$ separator', into a new locality, is an irrevocable commitment. For example, $\Theta \fatsemi \alpha :*, \Theta' \sqsubseteq \Theta, \alpha :* \fatsemi \Theta'$ holds but the converse direction does not.

The $\fatsemi$ separators do not affect the statements that are provable in a context, however: $\Theta \fatsemi \Theta' \vdash J$ if and only if $\Theta, \Theta' \vdash J$.

Just as with $\fatsemi$ separators, given variables in the context are preserved by metasubstitution, and their type schemes must be updated appropriately. It would be possible for the definition of $\theta : \Theta_0 \sqsubseteq \Theta_1$ to require Θ_1 to assign a term variable x all the types that Θ_0 assigns it, but allow x to become more polymorphic and acquire new types. For example, the identity 'information increase'

$$\Theta, x : \tau \to \tau \sqsubseteq \Theta, x : \forall\alpha.\, \alpha \to \alpha$$

could be permitted. This notion certainly retains stability: every variable lookup can be simulated in the more general context. However, it allows term variables to be assigned arbitrarily generalised type schemes, which are incompatible with the known and intended value of those variables. As Wells (2002) points out, Hindley-Milner type inference is not in this respect compositional. He carefully distinguishes principal *typings*, given the right to demand more polymorphism, from Milner's principal *type schemes* and analyses how the language of types must be extended to express principal typings.

2.1.3 Constraints: problems at ground mode

I have described the information-increasing steps that a problem-solving algorithm can take, but how are problems themselves represented? Given any statement J for which the corresponding sanity conditions of Lemma 2.1 hold, it is reasonable to ask for the least information increase needed to make J hold.

Formally, a *constraint problem* is a pair of a context Θ_0 and a statement J, where $\Theta_0 \vdash \mathbf{San}\, J$. A *solution* to such a problem is then a context Θ_1 and an information increase $\theta : \Theta_0 \sqsubseteq \Theta_1$ such that $\Theta_1 \vdash \theta\, J$. Such a solution is *minimal* if, for any other solution $\theta' : \Theta_0 \sqsubseteq \Theta'$, there exists a metasubstitution $\zeta : \Theta' \sqsubseteq \Theta_1$ such that $\theta' \equiv \zeta \cdot \theta$ (say θ' *factors through* θ with *cofactor* ζ).

In this setting, a *unification problem* is a constraint problem where J is an equation, that is, a pair of a context Θ_0 and an equation $\tau \equiv \upsilon$, where $\Theta_0 \vdash \tau :*$ and $\Theta_0 \vdash \upsilon :*$. A solution to the problem (a *unifier*) is given by a context Θ_1 and a metasubstitution $\theta : \Theta_0 \sqsubseteq \Theta_1$ such that $\Theta_1 \vdash \theta\, \tau \equiv \theta\, \upsilon : *$. A minimal solution is a most general unifier.

Information increase allows variables to become more informative either by definition or by substitution. The algorithms presented here exploit only the former, always choosing solutions of the form $\Theta_0 \sqsubseteq \Theta_1$. However, I will show the solutions are minimal with respect to arbitrary information increases: making progress by definition alone is enough to capture all possible solutions.

Stability permits *sound* sequential problem solving: if $\theta_0 : \Theta_0 \sqsubseteq \Theta_1$ solves J and $\theta_1 : \Theta_1 \sqsubseteq \Theta_2$ solves $\theta_0\, J'$ then $\theta_1 \cdot \theta_0 : \Theta_0 \sqsubseteq \Theta_2$ solves $J \ \wedge \ J'$. Perhaps more surprisingly, composite problems acquire *minimal* solutions similarly. This allows a 'greedy' minimal commitment strategy for problem solving.[3]

Lemma 2.4 (The Optimist's lemma)**.** *If $\theta_0 : \Theta_0 \sqsubseteq \Theta_1$ is a minimal solution of J and $\theta_1 : \Theta_1 \sqsubseteq \Theta_2$ is a minimal solution of $\theta_0\, J'$ then $\theta_1 \cdot \theta_0 : \Theta_0 \sqsubseteq \Theta_2$ is a minimal solution of $J \ \wedge \ J'$.*

Proof. Any solution $\zeta : \Theta_0 \sqsubseteq \Theta$ to $(\Theta_0, J \ \wedge \ J')$ must solve (Θ_0, J), and hence factor through $\theta_0 : \Theta_0 \sqsubseteq \Theta_1$. But its cofactor solves $(\Theta_1, \theta_0\, J')$, and hence factors through $\theta_1 : \Theta_1 \sqsubseteq \Theta_2$. $\square$

I will use this lemma to prove that the unification algorithm delivers most general unifiers. It also expresses the underlying reason why type inference gives principal solutions, although a more general result is needed there, because statements have outputs and the second statement may depend on the first.

This sequential approach to problem solving is not the only decomposition justified by stability. The account of unification by McAdam (1998) amounts to a concurrent, transactional decomposition of problems. One context is extended by multiple substitutions, which are then unified to produce a single substitution.

Another reassuring property of problem solving is that minimal solutions are well-defined up to isomorphism. A metasubstitution $\theta : \Theta \sqsubseteq \Theta'$ is an *isomorphism* if there exists $\theta^{-1} : \Theta' \sqsubseteq \Theta$ such that $\theta^{-1} \cdot \theta \equiv \iota$ and $\theta \cdot \theta^{-1} \equiv \iota$. The following lemma allows the contexts Θ_0 and Θ_1 to be replaced with the isomorphic Θ and Θ', while retaining minimality.

Lemma 2.5 (Isomorphism lemma)**.** *Suppose Θ, Θ', Θ_0 and Θ_1 are contexts, J is a well-formed statement in Θ_0 and $\zeta : \Theta \sqsubseteq \Theta_0$ and $\zeta' : \Theta_1 \sqsubseteq \Theta'$ are isomorphisms. If $\theta : \Theta_0 \sqsubseteq \Theta_1$ is a minimal solution of J then $\zeta' \cdot \theta \cdot \zeta : \Theta \sqsubseteq \Theta'$ is a minimal solution of $\zeta^{-1}\, J$.*

[3]The 'optimistic optimisation' of McBride (1999).

Proof. Composition gives that $\zeta' \cdot \theta \cdot \zeta : \Theta \sqsubseteq \Theta'$ is a metasubstitution, and since $\Theta_1 \vdash \theta\, J$ we have $\Theta' \vdash \zeta'\,(\theta\, J)$ by stability (Lemma 2.2), so $\Theta' \vdash (\zeta' \cdot \theta \cdot \zeta)\,(\zeta^{-1}\, J)$. Hence $\zeta' \cdot \theta \cdot \zeta$ is a solution of $\zeta^{-1}\, J$.

To see that it is minimal, suppose $\theta'' : \Theta \sqsubseteq \Theta''$ is such that $\Theta'' \vdash \theta''\,(\zeta^{-1}\, J)$. Now $\theta'' \cdot \zeta^{-1}$ is a solution of J, so by minimality of θ there must be some ζ'' such that $\zeta'' : \Theta_1 \sqsubseteq \Theta''$ and $\zeta'' \cdot \theta \equiv \theta'' \cdot \zeta^{-1}$. Hence $(\zeta'' \cdot \zeta'^{-1}) \cdot (\zeta' \cdot \theta \cdot \zeta) \equiv \theta''$ so the required cofactor is $\zeta'' \cdot \zeta'^{-1} : \Theta' \sqsubseteq \Theta''$. □

2.2 Unification for the syntactic equational theory

Having set the scene, I will now present the unification algorithm itself. The algorithm starts by structurally decomposing a constraint into multiple constraints on variables, which can be solved sequentially (by the Optimist's lemma). Each remaining constraint is either an equation between two variables (a flex-flex constraint) or between a metavariable and another type (a flex-rigid constraint). Either way, it is solved by moving through the context from right to left (most local to most global), updating the constraint or context appropriately.

For example, consider the context $\alpha : *, \beta : *, \alpha' := \beta : *, \gamma : *$ and problem $\alpha \to \beta \equiv \alpha' \to (\gamma \to \gamma)$. This equation decomposes into two constraints on variables, $\alpha \equiv \alpha'$ and $\beta \equiv \gamma \to \gamma$. The first is solved thus:

$$\begin{array}{lllll} & \alpha : *, & \beta : *, & \alpha' {:=} \beta, & \gamma : *, & [\alpha \equiv \alpha'] \\ & \alpha : *, & \beta : *, & \alpha' {:=} \beta, & [\alpha \equiv \alpha'], & \gamma : * \\ & \alpha : *, & \beta : *, & [\alpha \equiv \beta], & \alpha' {:=} \beta, & \gamma : * \\ \twoheadrightarrow & \alpha : *, & \beta {:=} \alpha, & \alpha' {:=} \beta, & \gamma : * & \end{array}$$

To solve $\alpha \equiv \alpha'$, the algorithm ignores γ since it does not occur in the constraint, moves past α' by updating the constraint to $\alpha \equiv \beta$, then defines β.

Solving the flex-rigid constraint $\beta \equiv \gamma \to \gamma$ requires γ to be moved back through the context, since it occurs in the constraint but cannot be instantiated:

$$\begin{array}{llllll} & \alpha : *, & \beta {:=} \alpha, & \alpha' {:=} \beta, & \gamma : *, & [\beta \equiv \gamma \rightarrowtriangle \gamma] \\ & \alpha : *, & \beta {:=} \alpha, & \alpha' {:=} \beta, & [\gamma : * \mid \beta \equiv \gamma \rightarrowtriangle \gamma] & \\ & \alpha : *, & \beta {:=} \alpha, & [\gamma : * \mid \beta \equiv \gamma \rightarrowtriangle \gamma], & \alpha' {:=} \beta & \\ & \alpha : *, & [\gamma : * \mid \alpha \equiv \gamma \rightarrowtriangle \gamma], & \beta {:=} \alpha, & \alpha' {:=} \beta & \\ \twoheadrightarrow & \gamma : *, & \alpha {:=} \gamma \rightarrowtriangle \gamma, & \beta {:=} \alpha, & \alpha' {:=} \beta & \end{array}$$

Here the algorithm ignores α', moves past the definition of β by updating the constraint to $\alpha \equiv \gamma \to \gamma$, then defines α after pasting in γ. In general, when solving an 'flex-rigid' equation between a metavariable and a type, the algorithm must accumulate the type's dependencies as it finds them, performing the occurs check to ensure a solution exists. This is how variables move outward through localities, acquiring a more global relevance.

The unification algorithm is formally defined by the rules in Figure 2.5. Each inference rule can be read clockwise from the bottom-left: the inputs to the rule determine the inputs to the first premise, then the outputs from the first premise determine the inputs to the second premise, and so on, until the outputs from all the premises determine the outputs of the conclusion.

The *unify* judgment $\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1$ means that given inputs Θ_0, τ and υ, unification succeeds with solution $\Theta_0 \sqsubseteq \Theta_1$. The inputs must satisfy the sanity conditions $\Theta_0 \vdash \tau : *$ and $\Theta_0 \vdash \upsilon : *$. Symmetric variants of the INST and DEFINE rules have been omitted.

The *instantiate* judgment $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$ means that given inputs Θ_0, Ξ, α and τ, instantiating α with τ succeeds, yielding solution $\Theta_0 \sqsubseteq \Theta_1$. The idea is that the bar ($|$) represents progress in examining context elements in order, and Ξ contains exactly those declarations on which τ depends. Formally, the inputs must satisfy the following conditions, where the set $\mathsf{fmv}(\tau)$ records those metavariables occurring free in type τ.

Definition 2.1. The quadruple $(\Theta_0, \Xi, \alpha, \tau)$ *satisfies the input conditions* if

- $\Theta_0 \vdash \alpha : *$ where α is a metavariable,
- $\Theta_0, \Xi \vdash \tau : *$ where τ is not a metavariable, and
- Ξ contains only metavariable declarations $\beta : *$ with $\beta \in \mathsf{fmv}(\tau)$.

The main point of these conditions is to ensure that Ξ contains only genuine dependencies of τ, so moving Ξ back in the context will not sacrifice generality.

Observe that no rule applies to deduce

$$\Theta_0, \alpha : * \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \text{ with } \alpha \in \mathsf{fmv}(\tau),$$

where the algorithm fails. This is an occurs check failure: α and τ cannot unify if α occurs in τ, and τ is not a variable. Given the single type constructor symbol (the function arrow $\to$), there are no failures due to rigid-rigid mismatch, but adding these will not significantly complicate matters.

The unification algorithm is implemented in Appendix A.2 (page 200).

$\boxed{\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1}$ *(unifying τ with υ in Θ_0 results in Θ_1)*

$$\frac{\Theta_0 \vdash \tau_0 \equiv \upsilon_0 : * \dashv \Theta_1 \qquad \Theta_1 \vdash \tau_1 \equiv \upsilon_1 : * \dashv \Theta_2}{\Theta_0 \vdash (\tau_0 \to \tau_1) \equiv (\upsilon_0 \to \upsilon_1) : * \dashv \Theta_2}\ \textsc{decompose}$$

$$\frac{\tau \text{ non-variable} \qquad \Theta_0 \,|\, \cdot \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0 \vdash \alpha \equiv \tau : * \dashv \Theta_1}\ \textsc{inst}$$

$$\frac{}{\Theta, \alpha{:}* \vdash \alpha \equiv \alpha : * \dashv \Theta, \alpha{:}*}\ \textsc{idle} \qquad \frac{\alpha \neq \beta}{\Theta, \alpha{:}* \vdash \alpha \equiv \beta : * \dashv \Theta, \alpha{:=}\beta : *}\ \textsc{define}$$

$$\frac{\Theta_0 \vdash [\tau/\gamma]\,\alpha \equiv [\tau/\gamma]\,\beta : * \dashv \Theta_1}{\Theta_0, \gamma{:=}\tau : * \vdash \alpha \equiv \beta : * \dashv \Theta_1, \gamma{:=}\tau : *}\ \textsc{subs}$$

$$\frac{\Theta_0 \vdash \alpha \equiv \beta : * \dashv \Theta_1 \qquad \alpha \neq \gamma \qquad \beta \neq \gamma}{\Theta_0, \gamma{:}* \vdash \alpha \equiv \beta : * \dashv \Theta_1, \gamma{:}*}\ \textsc{skip-ty}$$

$$\frac{\Theta_0 \vdash \alpha \equiv \beta : * \dashv \Theta_1}{\Theta_0, x{:}\sigma \vdash \alpha \equiv \beta : * \dashv \Theta_1, x{:}\sigma}\ \textsc{skip-tm} \qquad \frac{\Theta_0 \vdash \alpha \equiv \beta : * \dashv \Theta_1}{\Theta_0 ⨟ \vdash \alpha \equiv \beta : * \dashv \Theta_1 ⨟}\ \textsc{skip-semi}$$

$\boxed{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}$ *(instantiating α with τ in Θ_0, Ξ results in Θ_1)*

$$\frac{\alpha \notin \mathsf{fmv}(\tau)}{\Theta_0, \alpha{:}* \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_0, \Xi, \alpha{:=}\tau : *}\ \textsc{inst-define}$$

$$\frac{\Theta_0, \Xi \vdash [\upsilon/\beta]\,\alpha \equiv [\upsilon/\beta]\,\tau : * \dashv \Theta_1}{\Theta_0, \beta{:=}\upsilon : * \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1, \beta{:=}\upsilon : *}\ \textsc{inst-subs}$$

$$\frac{\Theta_0 \,|\, \beta{:}*, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \qquad \alpha \neq \beta \qquad \beta \in \mathsf{fmv}(\tau)}{\Theta_0, \beta{:}* \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}\ \textsc{inst-depend}$$

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \qquad \alpha \neq \beta \qquad \beta \notin \mathsf{fmv}(\tau)}{\Theta_0, \beta{:}* \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1, \beta{:}*}\ \textsc{inst-skip-ty}$$

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0, x{:}\sigma \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1, x{:}\sigma}\ \textsc{inst-skip-tm}$$

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0 ⨟ \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 ⨟}\ \textsc{inst-skip-semi}$$

Figure 2.5: Algorithmic rules for unification

2.2.1 Correctness of syntactic unification

The contextual problem-solving discipline I have introduced allows soundness to be linked with generality, showing that unification produces minimal solutions.

Lemma 2.6 (Soundness and generality of unification).
(a) If $\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *is a minimal solution of* $\tau \equiv \upsilon$.

(b) If $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$ *then* $\Theta_0, \Xi \sqsubseteq \Theta_1$ *is a minimal solution of* $\alpha \equiv \tau$.

Proof. By induction on the structure of derivations. The key idea is that the type variables of Θ_0 and Θ_1 are the same, and whenever $\theta : \Theta_0 \sqsubseteq \Theta'$ is a solution, the definitions made in Θ_1 must hold as equations in Θ' for the problem to be solved, so θ can be rearranged to produce the necessary cofactor $\zeta : \Theta_1 \sqsubseteq \Theta'$. For details, see Appendix D.1 (page 236). □

A lemma about the occurs check is needed for completeness of unification.

Lemma 2.7 (Occurs check). *Let* α *be a metavariable and* τ *a non-metavariable type in* Θ *such that* $\alpha \in \mathsf{fmv}(\tau)$. *There is no context* Θ' *and metasubstitution* $\theta : \Theta \sqsubseteq \Theta'$ *such that* $\Theta' \vdash \theta\,\alpha \equiv \theta\,\tau : *$.

Proof. Suppose otherwise. By expanding definitions in Θ' we have a type containing no defined metavariables that is equal to a proper subterm of itself, but induction on the definition of equality shows that this is impossible. □

Exposing the structure underlying unification makes termination of the algorithm evident (McBride, 2003). Each unification or instantiation step either shortens the overall context, shortens the uninspected context left of the bar (for instantiation) or preserves the context and decomposes types.

Lemma 2.8 (Completeness of unification).
(a) If $\theta : \Theta_0 \sqsubseteq \Theta'$, $\Theta_0 \vdash \upsilon : * \;\wedge\; \tau : *$ *and* $\Theta' \vdash \theta\,\upsilon \equiv \theta\,\tau : *$, *then there is some context* Θ_1 *such that* $\Theta_0 \vdash \upsilon \equiv \tau : * \dashv \Theta_1$.

(b) Moreover, if $\theta : \Theta_0, \Xi \sqsubseteq \Theta'$ *is such that* $\Theta' \vdash \theta\,\alpha \equiv \theta\,\tau : *$ *and the input conditions (Definition 2.1) are satisfied, then* $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$.

Proof. Since the algorithm terminates, it suffices to show that it covers every case such that a solution can exist. Each step preserves solutions: if the equation in a conclusion can be solved, so can those in its premises. The only omitted case is

$$\Theta_0, \alpha{:}* \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \quad \text{with } \alpha \in \mathsf{fmv}(\tau),$$

but Lemma 2.7 implies that this has no solutions. □

$\boxed{\Theta \vdash t : \sigma}$ *(term t has type scheme σ in Θ)*

$$\frac{\Theta \ni x : \sigma \qquad \Theta \vdash \mathbf{ctx}}{\Theta \vdash x : \sigma} \qquad \frac{\Theta, x : \tau \vdash t : \upsilon}{\Theta \vdash \lambda x.t : \tau \to \upsilon} \qquad \frac{\Theta \vdash t : \tau \to \upsilon \quad \Theta \vdash s : \tau}{\Theta \vdash t\, s : \upsilon}$$

$$\frac{\Theta \vdash s : \sigma \quad \Theta, x : \sigma \vdash t : \sigma'}{\Theta \vdash \mathbf{let}\, x = s\, \mathbf{in}\, t : \sigma'} \qquad \frac{\Theta, \alpha : * \vdash t : \sigma}{\Theta \vdash t : \forall \alpha.\, \sigma} \qquad \frac{\Theta \vdash t : \forall \alpha.\, \sigma \quad \Theta \vdash \tau : *}{\Theta \vdash t : [\tau/\alpha]\, \sigma} \qquad \frac{\Theta \vdash t : \tau \quad \Theta \vdash \tau \equiv \upsilon : *}{\Theta \vdash t : \upsilon}$$

Figure 2.6: Declarative rules for type assignment

$\boxed{\Theta \vdash \sigma \succ \sigma'}$ *(σ is more general than σ' in Θ)*

$$\frac{\Theta \vdash \tau \equiv \upsilon : *}{\Theta \vdash \tau \succ \upsilon} \qquad \frac{\alpha \notin \mathsf{fmv}(\sigma) \quad \Theta, \alpha : * \vdash \sigma \succ \sigma'}{\Theta \vdash \sigma \succ \forall \alpha.\, \sigma'} \qquad \frac{\Theta \vdash \tau : * \quad \Theta \vdash [\tau/\alpha]\, \sigma \succ \upsilon}{\Theta \vdash \forall \alpha.\, \sigma \succ \upsilon}$$

Figure 2.7: Generic instantiation for type schemes

2.3 Type inference with generalisation made easy

The deduction rules for the typing statement $t : \sigma$ are given in Figure 2.6. Type inference involves making this statement hold, but unlike unification, the type should be an *output* of problem-solving along with the solution context. The definition of constraint problems in Subsection 2.1.3 is insufficiently general. Instead, each parameter in a statement has a *mode*, either ‘input’ or ‘output’.

A *type inference problem* consists of a context Θ_0 and a term t; a solution is a metasubstitution $\theta : \Theta_0 \sqsubseteq \Theta_1$ and a type τ such that $\Theta_1 \vdash t : \tau$. Such a solution is *most general* or *minimal* if any other solution $(\theta' : \Theta_0 \sqsubseteq \Theta', \upsilon)$ factors through it with cofactor ζ, such that $\Theta' \vdash \upsilon \equiv \zeta\, \tau : *$.

Similarly, a *type scheme inference problem* consists of a context Θ_0 and a term t; a solution is a metasubstitution $\theta : \Theta_0 \sqsubseteq \Theta_1$ and a scheme σ such that $\Theta_1 \vdash t : \sigma$. Such a solution is *most general* or *minimal* if any other solution $(\theta' : \Theta_0 \sqsubseteq \Theta', \sigma')$ factors through it with cofactor ζ such that $\Theta' \vdash \zeta\, \sigma \succ \sigma'$.

Here $\sigma \succ \sigma'$ is the generic instantiation relation, defined in Figure 2.7, meaning that any type which is an instance of σ' is also an instance of σ.

Type schemes arise by quantifying a context suffix (a list of type metavariables) Ξ over a type τ, written $\forall\Xi.\tau$ and defined by

$$\begin{array}{rcl} \forall\cdot.\tau & \mapsto & \tau \\ \forall(\alpha{:}*,\Xi).\tau & \mapsto & \forall\alpha.\,(\forall\Xi.\tau) \\ \forall(\alpha{:=}\upsilon:*,\Xi).\tau & \mapsto & [\upsilon/\alpha]\,(\forall\Xi.\tau) \end{array}$$

Any scheme $\sigma = \forall\overline{\alpha_i}^{\,i}.\,\tau$ can be viewed in this way, using the suffix $\overline{\alpha_i{:}*}^{\,i}$.

Lemma 2.9. $\Theta \vdash t{:}(\forall\Xi.\tau)$ *if and only if* $\Theta,\Xi \vdash t{:}\tau$.

Proof. Straightforward induction on Ξ. □

2.3.1 The Generalist's lemma

Recall that $⨾$ markers divide the context into localities. In the type inference algorithm, the metavariables that can be generalised are exactly those in the current locality. This relies on the following lemma, which states that a minimal solution to a type scheme inference problem can be found from a minimal solution to a type inference problem.

Crucially, a substitution for variables in a locality cannot depend on variables in a 'more local' one: for example, $[\beta/\alpha,\beta/\beta] : \alpha{:}*\,⨾\,\beta{:}* \sqsubseteq \cdot\,⨾\,\beta{:}*$ is forbidden. This allows any $\theta : \Theta\,⨾\,\Xi \sqsubseteq \Theta'\,⨾\,\Xi'$ to be restricted to variables in Θ, so that $\theta|_{\Theta} : \Theta \sqsubseteq \Theta'$.

Lemma 2.10 (The Generalist's lemma)**.** *If* $\theta : \Theta_0⨾ \sqsubseteq \Theta_1⨾\Xi$ *is a minimal solution of the type inference problem for* t *with output* τ*, then* $\theta : \Theta_0 \sqsubseteq \Theta_1$ *is a minimal solution of the type scheme inference problem for* t *with output* $\forall\Xi.\tau$*.*

Proof. If $\theta : \Theta_0⨾ \sqsubseteq \Theta_1\,⨾\,\Xi$ then $\theta : \Theta_0 \sqsubseteq \Theta_1$ by definition of $\sqsubseteq$. Furthermore, $\Theta_1 \vdash t{:}(\forall\Xi.\tau)$ holds iff $\Theta_1\,⨾\,\Xi \vdash t{:}\tau$ by Lemma 2.9.

For minimality, suppose $\theta' : \Theta_0 \sqsubseteq \Theta'$ is an information increase and $\forall\overline{\alpha_i}^{\,i}.\,\upsilon$ is a scheme such that $\Theta' \vdash t{:}\forall\overline{\alpha_i}^{\,i}.\,\upsilon$. Then $\Theta',\overline{\alpha_i{:}*}^{\,i} \vdash t{:}\upsilon$. Now $\theta' : \Theta_0⨾ \sqsubseteq \Theta'⨾\overline{\alpha_i{:}*}^{\,i}$ and $\Theta'\,⨾\,\overline{\alpha_i{:}*}^{\,i} \vdash t : \upsilon$, so by minimality of the hypothesis there is a cofactor $\zeta : \Theta_1\,⨾\,\Xi \sqsubseteq \Theta'\,⨾\,\overline{\alpha_i{:}*}^{\,i}$ such that $\theta' = \zeta\cdot\theta$ and $\Theta'\,⨾\,\overline{\alpha_i{:}*}^{\,i} \vdash \zeta\,\tau \equiv \upsilon : *$. Then $\zeta|_{\Theta_1} : \Theta_1 \sqsubseteq \Theta'$, $\theta' \equiv \zeta|_{\Theta_1}\cdot\theta$ and $\Theta' \vdash \zeta|_{\Theta_1}\,(\forall\Xi.\tau) \succ \forall\overline{\alpha_i}^{\,i}.\,\upsilon$ as required. □

2.3.2 Transforming type assignment into type inference

The typing rules in Figure 2.6 do not directly lead to a type inference algorithm, as they permit unrestricted generalisation and instantiation of type schemes. To

$\boxed{\Theta \vdash t : \tau}$

$$\frac{\Theta \ni x : \sigma \qquad \Theta \vdash \sigma \succ \tau}{\Theta \vdash x : \tau}\ \textsc{var} \qquad\qquad \frac{\Theta, x : \tau \vdash t : \upsilon}{\Theta \vdash \lambda x.t : \tau \to \upsilon}\ \textsc{lam}$$

$$\frac{\Theta \vdash s : \tau' \to \tau \qquad \Theta \vdash t : \tau'}{\Theta \vdash s\, t : \tau}\ \textsc{app} \qquad\qquad \frac{\Theta \fatsemi \Xi \vdash s : \upsilon \qquad \Theta, x : (\forall \Xi . \upsilon) \vdash t : \tau}{\Theta \vdash \mathbf{let}\, x = s\, \mathbf{in}\, t : \tau}\ \textsc{let}$$

Figure 2.8: Transformed rules for type assignment

resolve this, an equivalent system (assigning types rather than type schemes) is given in Figure 2.8, where instantiation occurs only at variables, and generalisation at let-bindings. This transformation is well known: a clear presentation is given by Clément et al. (1986) resulting in the rules of Figure 2.1.

From the transformed rules, an algorithm can be constructed to match. To convert a rule into algorithmic form, proceed clockwise starting from the inputs to the conclusion. For each premise, ensure that the problem inputs are fully specified (by the inputs to the conclusion and the outputs of previous premises), inserting metavariables to stand for unknown inputs. Instead of pattern matching on problem outputs, ensure there are schematic variables in output positions, and reintroduce unification constraints as necessary.

The type inference judgment $\Theta_0 \vdash t : \tau \dashv \Theta_1$ and the scheme inference judgment $\Theta_0 \vdash t : \sigma \dashv \Theta_1$ are defined by the rules in Figure 2.9. As they are structural on terms, they yield a terminating algorithm. The Optimist's lemma means that sequential solution of problems delivers a minimal solution, and the Generalist's lemma makes it easy to reduce type scheme inference problems to type inference problems.

The λ-rule now generates a metavariable for the argument type. The rule for application assigns types to the function and argument separately, then inserts an equation with a fresh name for the codomain type.

The type inference algorithm is implemented in Appendix A.3 (page 202).

2.3.3 Correctness of type inference

Since the algorithmic rules correspond directly to the transformed declarative system in Figure 2.8, it is easy to prove soundness, completeness and generality of type inference with respect to this system.

$\boxed{\Theta_0 \vdash t : \sigma \dashv \Theta_1}$ *(term t in context Θ_0 has inferred scheme σ in context Θ_1)*

$$\frac{\Theta_0 \fatsemi \vdash t : \tau \dashv \Theta_1 \fatsemi \Xi}{\Theta_0 \vdash t : (\forall \Xi . \tau) \dashv \Theta_1} \text{ INFER-GEN}$$

$\boxed{\Theta_0 \vdash t : \tau \dashv \Theta_1}$ *(term t in context Θ_0 has inferred type τ in context Θ_1)*

$$\frac{x : (\forall \overline{\alpha_i}^{\,i} . \upsilon) \in \Theta_0}{\Theta_0 \vdash x : \upsilon \dashv \Theta_0, \overline{\alpha_i : *}^{\,i}} \text{ INFER-VAR}$$

$$\frac{\Theta_0, \alpha : *, x : \alpha \vdash t : \tau \dashv \Theta_1, x : \alpha, \Xi}{\Theta_0 \vdash \lambda x . t : \alpha \rightarrow \tau \dashv \Theta_1, \Xi} \text{ INFER-LAM}$$

$$\frac{\Theta_0 \vdash s : \upsilon \dashv \Theta_1 \qquad \Theta_1 \vdash t : \upsilon' \dashv \Theta_2 \qquad \Theta_2, \alpha : * \vdash \upsilon \equiv \upsilon' \rightarrow \alpha : * \dashv \Theta_3}{\Theta_0 \vdash s\, t : \alpha \dashv \Theta_3} \text{ INFER-APP}$$

$$\frac{\Theta_0 \vdash s : \sigma \dashv \Theta_1 \qquad \Theta_1, x : \sigma \vdash t : \tau \dashv \Theta_2, x : \sigma, \Xi}{\Theta_0 \vdash \mathbf{let}\, x = s \,\mathbf{in}\, t : \tau \dashv \Theta_2, \Xi} \text{ INFER-LET}$$

Figure 2.9: Algorithmic rules for type inference

Lemma 2.11 (Soundness and generality of type inference)**.** *If $\Theta_0 \vdash t : \tau \dashv \Theta_1$, then $\Theta_0 \sqsubseteq \Theta_1$ is a minimal solution to the type inference problem for t with output τ. Similarly, if $\Theta_0 \vdash t : \sigma \dashv \Theta_1$ then $\Theta_0 \sqsubseteq \Theta_1$ is a minimal solution to the type scheme inference problem for t with output σ.*

Proof. By induction on derivations, using the Optimist's lemma (2.4) and Generalist's lemma (2.10). For details, see Appendix D.1 (page 237). □

Lemma 2.12 (Completeness of type inference)**.**

(a) If (Θ_0, t) is a type inference problem with solution $(\theta : \Theta_0 \sqsubseteq \Theta', \upsilon)$, then $\Theta_0 \vdash t : \tau \dashv \Theta_1$ for some Θ_1 and τ.

(b) If (Θ_0, t) is a scheme inference problem with solution $(\theta : \Theta_0 \sqsubseteq \Theta', \sigma')$, then $\Theta_0 \vdash t : \sigma \dashv \Theta_1$ for some Θ_1 and σ.

Proof. By induction on the derivation of $\Theta' \vdash t : \upsilon$ or $\Theta' \vdash t : \sigma'$ in the transformed declarative system of Figure 2.8. Each case corresponds directly to an algorithmic rule. For details, see Appendix D.1 (page 238). □

2.4 Elaboration, zipper style

Elaboration is a step beyond type inference, where instead of merely generating a type corresponding to the source term, a representation of the term in a more explicit calculus is generated. This might seem excessive for the simple Hindley-Milner system, but for more complex type systems (particularly those involving dependent types) the distinction is helpful. In Chapter 7, I will discuss elaboration of a Haskell-like language. Here, to introduce the idea of elaboration, I show how to elaborate Hindley-Milner terms into explicitly-typed predicative System F. This algorithm is implemented in Appendix A.4 (page 204).

The grammar of System F terms is

$$e \quad ::= \quad x \mid \lambda x\!:\!\sigma.e \mid \Lambda\alpha\!:\!*.e \mid e\,e' \mid e\,\tau$$

where λ-bound variables have type annotations, and type abstraction and application are explicit. The type system is standard, and hence omitted; it is essentially a syntax-directed version of the declarative system in Figure 2.6.

So far in this chapter, the context structure has carried the 'linguistic' context of term variables and type metavariables, but the type inference algorithm has separately managed the 'syntactic' context (the structure of the term). Variable bindings and the $\fatsemi$ marker are vestiges of the syntactic context: a variable represents the fact that type inference is taking place under a λ- or let-binding, and a $\fatsemi$ marker represents 'being under a let-definiens'. Let me take this idea to its natural conclusion, identifying the syntactic and linguistic contexts into a single data structure that represents progress through a type inference problem.

Huet (1997) taught us how to use a 'zipper' data structure to represent a position in a tree, such as a term. The path to the current location is represented as a list of layers, where each layer corresponds to choosing a single branch at a node, and stores the subtrees rooted at the other branches. McBride (2001) observed that the type of the zipper can be *computed* by differentiation, and further refined the structure to represent left-to-right progress through a term (McBride, 2008). Terms 'to the left' of the current location have been elaborated to a typed System F term, while those 'to the right' have not yet been visited. Thus the syntax of layers is given by

$$\ell \quad ::= \quad []\,t \mid (e\!:\!\tau)[] \mid \lambda x\!:\!\tau.[] \mid \mathbf{let}\,x\!=\![]\,\mathbf{in}\,t \mid \mathbf{let}\,x\!:\!\sigma\!=\!e\,\mathbf{in}\,[]$$

where a hole $[]$ represents the current position. Contexts are adapted to include layers rather than variables or $\fatsemi$ markers:

$$\Theta \quad ::= \quad \cdot \mid \Theta, \alpha\!:\!* \mid \Theta, \alpha\!:\!* := \tau \mid \Theta, \ell$$

$$\begin{array}{llllll}
\Theta & \downarrow x & \mapsto & \Theta, \overline{\alpha_j{:}*}^j & \uparrow x\,\overline{\alpha_j}^j : \tau & \text{if } \Theta \ni x{:}\forall\overline{\alpha_j}^j.\tau \\
\Theta & \downarrow s\,t & \mapsto & \Theta, []t & \downarrow s & \\
\Theta & \downarrow \lambda x.t & \mapsto & \Theta, \alpha{:}*, \lambda x{:}\alpha.[] & \downarrow t & \\
\Theta & \downarrow \mathbf{let}\, x{=}s\,\mathbf{in}\, t & \mapsto & \Theta, \mathbf{let}\, x{=}[]\,\mathbf{in}\, t & \downarrow s &
\end{array}$$

$$\begin{array}{lllll}
\Theta, \lambda x{:}\upsilon.[], \Xi & \uparrow e : \tau & \mapsto & \Theta, \Xi & \uparrow \lambda x{:}\upsilon.e : \upsilon \to \tau \\
\Theta, \mathbf{let}\, x{=}[]\,\mathbf{in}\, t, \Xi & \uparrow e : \tau & \mapsto & \Theta, \mathbf{let}\, x{:}\forall\Xi.\tau{=}\Lambda\Xi.e\,\mathbf{in}\,[] & \downarrow t \\
\Theta, \mathbf{let}\, x{:}\sigma{=}e'\,\mathbf{in}\,[], \Xi & \uparrow e : \tau & \mapsto & \Theta, \Xi & \uparrow (\lambda x{:}\sigma.e)\; e' : \tau \\
\Theta, []t, \Xi & \uparrow e : \tau & \mapsto & \Theta, \Xi, (e{:}\tau)[] & \downarrow t \\
\Theta, (e'{:}\upsilon)[], \Xi & \uparrow e : \tau & \mapsto & \Theta' & \uparrow e'\, e : \beta \\
 & & & \multicolumn{2}{l}{\text{where } \Theta, \Xi, \beta{:}* \vdash \upsilon \equiv \tau \to \beta : * \dashv \Theta'}
\end{array}$$

Figure 2.10: Elaboration as state-transformation

Now that Θ represents the entire context of an elaboration problem, elaboration can be implemented tail-recursively as an state-transforming automaton. Figure 2.10 shows the elaboration algorithm. It is divided into two modes:

- The 'downwards' mode $\Theta \downarrow t$ takes a context and a source term which is being elaborated. If it is a variable, control switches to the 'upwards' mode, otherwise it moves into an appropriate subterm by extending the context.
- The 'upwards' mode $\Theta \uparrow e : \tau$ takes a context and an elaborated System F term with its type. It examines the context to move outwards, refocus on the next subterm to elaborate, then switch back to downwards mode.

The algorithm should be invoked in downwards mode with the empty context and the original term to be elaborated. Eventually, if the term is well-typed, the upwards mode will run out of layers and terminate with the elaborated version of the term and its type.

This explicit representation of partial progress through an elaboration problem is very useful when constraints cannot always be solved immediately, as in a dependently typed setting. Elaboration is no longer a left-to-right march through the term structure, but may involve back-and-forth refocusing as the elaborator finds places where progress can be made. This is the basis of the implementation of elaboration in Epigram.

2.5 Discussion

In this chapter, I have given an implementation of Hindley-Milner type inference involving all the same steps as Algorithm $\mathcal{W}$, but not necessarily in the same order. In particular, the dependency panic that seizes $\mathcal{W}$ in the let-rule becomes an invariant that the unification algorithm maintain a well-founded context.

The algorithm is presented as a problem transformation system locally preserving solutions, hence finding a most general global solution if any solutions exist at all. Accumulating solutions to decomposed problems is justified simply by stability of solutions on information increase. The discipline of problem solving established here is happily complete for Hindley-Milner type inference, but in any case couples soundness with generality.

Maintain context validity, make definitions anywhere and only where there is no choice, so the solutions you find will be general and generalisable locally: this is a key design principle for elaboration of high-level code in systems like Epigram and Agda, and bugs arise from its transgression. The account given here of 'current information' in terms of contexts and their information ordering provides a principled means to investigate and repair these troubles.

There is, however, some way to go. Algorithm $\mathcal{W}$ is a conveniently structural type inference process for 'finished' expressions in a setting where unification is complete. Each subproblem is either solved or rejected on first inspection—there is never a need for a 'later, perhaps' outcome. As a result, 'direct style' recursive programming is adequate to the task. If problems could get stuck, how might an algorithm abandon them and return to them later? By storing their *context*, of course! In Chapter 4, I will take exactly this approach to deal with higher-order unification problems.

First, though, I will extend the framework in another direction: handling units of measure with the equational theory of abelian groups. Variable dependency becomes more subtle in the presence of a nontrivial equational theory, and so maintaining a well-founded context (in order to make generalisation straightforward) is even more crucial.

2.5.1 Related work

The idea of assertions consuming an input context and producing an output context goes back at least to Pollack (1990). Nipkow and Prehofer (1995) use unordered input and output contexts to pass information about Haskell typeclass inference, with a conventional substitution-based presentation of unification.

The work of Dunfield and Krishnaswami (2013) on higher-rank polymorphism in a bidirectional type system, based on earlier work by Dunfield (2009), uses well-founded contexts that contain existential type variables (amongst other things). They rely on a notion of context extension in a similar way to my definition of information increase between input and output contexts, and while their treatment of unification is different (since they are dealing with subtyping for higher-rank polymorphism, rather than let-generalisation) there are some similarities with the approach I have described.

An alternative approach to generalisation, used in some ML implementations for the sake of efficiency, involves assigning numeric 'ranks' to type variables based on the number of bindings they are introduced under, then generalising over variables whose rank is sufficiently large. Rémy (1992) implemented an algorithm based on counting let-bindings as part of the OCaml typechecker, and Kiselyov (2013) gives a clear explanation of Rémy's algorithm which relates it to region-based memory management. Kuan and MacQueen (2007) formalised and compared approaches that count let- and λ-bindings; they attribute the idea for counting λ-bindings to Damas (1984). The algorithm I described manages ranks implicitly, by representing type variables in an ordered context, in which the $\fatsemi$ marker corresponds to increasing the rank.

Chapter 3

Unification and type inference for units of measure

In the previous chapter, I described a ‘problem solving’ rationalisation of syntactic unification and Hindley-Milner type inference that provides a more refined account of dependency analysis. Term and type variables live in a dependency-ordered context. Problems are solved in small steps, each of which is most general and involves minimal extra dependency. This makes let-generalisation particularly easy: simply ‘skim off’ generalisable type variables from the end of the context, as nothing can depend on them.

I now move on to consider one of the many extensions of the Hindley-Milner system, namely units of measure in the style of Kennedy (1996a,b, 2010). My approach to type inference gives a clearer account of the subtle issues surrounding generalisation in the presence of a nontrivial equational theory on types. This chapter is based on work presented at TFP 2011 (Gundry, 2011). A Haskell implementation of the unification algorithm described here is given in Appendix B.

Consider this Haskell function, traditionally of type $\mathsf{Float} \to \mathsf{Float}$:

$$\begin{aligned}&\mathsf{distanceTravelled}\ t = \mathsf{velocity} * t + (\mathsf{acceleration} * t * t)\ /\ 2\\ &\quad \textbf{where}\ \{\mathsf{velocity} = 2.0; \mathsf{acceleration} = 3.6\}\end{aligned}$$

Kennedy (1996b) shows how to check units of measure for such terms: with velocity and acceleration annotated with their units ($\mathbf{m} * \mathbf{s}^{-1}$ and $\mathbf{m} * \mathbf{s}^{-2}$), the system could infer the type $\mathsf{Float}\langle\mathbf{s}\rangle \to \mathsf{Float}\langle\mathbf{m}\rangle$ for the whole function. Type inference relies on unification, but units need a more liberal equational theory than syntactic equality, as $\mathbf{m} * \mathbf{s}^{-1} * \mathbf{s}$ should mean the same thing as $\mathbf{m}$. Kennedy uses the theory of abelian groups. He has introduced units of measure with polymorphism into the functional programming language F# (Syme, 2010).

3.0.1 A troublesome example

Algorithm $\mathcal{W}$ relies on dependency analysis for let-generalisation. Using the occurs check to identify generalisable variables (those that are free in the type but not the typing environment) is problematic for the equational theory of abelian groups, as *variable occurrence does not imply variable dependency.* Later I will show another way of looking at this: given the equation $\alpha \equiv \tau$, where α is a metavariable and τ is a type, the solution $[\tau/\alpha]$ is not necessarily most general! In this chapter, I will give an analysis of dependency that exposes and resolves the difficulties with generalisation.

Kennedy (2010, p. 292) gives the example (notation adapted):

$$\lambda x.\mathbf{let}\, y = \mathrm{div}\, x \,\mathbf{in}\, (y\, \mathrm{mass}, y\, \mathrm{time}), \qquad \text{where}$$
$$\mathrm{div} : \forall \alpha : \mathcal{U}. \forall \beta : \mathcal{U}. \mathbb{F}\langle \alpha * \beta \rangle \to \mathbb{F}\langle \alpha \rangle \to \mathbb{F}\langle \beta \rangle, \qquad \mathrm{mass} : \mathbb{F}\langle \mathbf{kg} \rangle, \qquad \mathrm{time} : \mathbb{F}\langle \mathbf{s} \rangle.$$

Here $\mathbb{F}\langle \nu \rangle$ is a type of numbers with units ν, defined in Subsection 3.0.2. If one adds constraint solving for units to Algorithm $\mathcal{W}$ with the usual occurrence-based let-generalisation rule, the resulting algorithm fails to infer a type for this term, because polymorphism is lost: y is given the monotype $\mathbb{F}\langle \alpha \rangle \to \mathbb{F}\langle \beta * \alpha^{-1} \rangle$ where α and β are unification metavariables, and α cannot unify with $\mathbf{kg}$ and $\mathbf{s}$. However, if y is given its principal type scheme $\forall \alpha : \mathcal{U}. \mathbb{F}\langle \alpha \rangle \to \mathbb{F}\langle \beta * \alpha^{-1} \rangle$, then the term has type $\mathbb{F}\langle \beta \rangle \to (\mathbb{F}\langle \beta * \mathbf{kg}^{-1} \rangle, \mathbb{F}\langle \beta * \mathbf{s}^{-1} \rangle)$, as described in Section 3.3.

The difficulty is that the algorithm fails to assign principal type schemes to open terms because of the nontrivial equational theory on types. One way around this difficulty is to apply a *generaliser*, "a substitution that 'reveals' the polymorphism available under a given type environment"[1], due to Kennedy (1996a) and Rittri (1995). Such a substitution preserves types in the context (up to the equational theory) but rearranges group variables so that the Algorithm $\mathcal{W}$ generalisation rule can be used. Calculating a generaliser is specific to the equational theory and technically nontrivial. It is not implemented in F#, so Kennedy's example does not type check:

```
> fun x -> let y z = x / z in (y mass, y time) ;;
-----------------------------------------^^^^
error FS0001: Type mismatch.
Expecting a float<kg> but given a float<s>
The unit of measure 'kg' does not match the unit of measure 's'
```

[1]Kennedy (1996a, p. 23)

Term variables	x, y		
Type metavariables	α, β, γ		
Kinds	κ	$::=$	$* \mid \mathcal{U}$
Contexts	Θ	$::=$	$\cdot \mid \Theta, \alpha{:}\kappa \mid \Theta, \alpha{:=}\rho : \kappa \mid \Theta, x{:}\sigma \mid \Theta\fatsemi$
Suffixes	Ξ	$::=$	$\cdot \mid \Xi, \alpha{:}\kappa \mid \Xi, \alpha{:=}\rho : \kappa$
Unit suffix	Υ	$::=$	$\cdot \mid \alpha{:}\mathcal{U}$
Type expressions	ρ	$::=$	$\alpha \mid \rho \to \rho' \mid \mathbb{F}\langle\rho\rangle \mid b \mid 1 \mid \rho * \rho' \mid \rho^{-1}$
Types	$\tau,\ \upsilon$	$::=$	$\alpha \mid \tau \to \upsilon \mid \mathbb{F}\langle\nu\rangle$
Units	ν	$::=$	$\alpha \mid b \mid 1 \mid \nu * \nu' \mid \nu^{-1}$
Base units	b	$::=$	$\mathbf{kg} \mid \mathbf{m} \mid \mathbf{s} \mid \cdots$
Type schemes	σ	$::=$	$\rho \mid \forall\alpha{:}\kappa.\,\sigma$
Terms	$t,\ s$	$::=$	$x \mid \lambda x.t \mid s\,t \mid \mathbf{let}\, x{=}s\,\mathbf{in}\, t$
Statements	J	$::=$	$\mathbf{ctx} \mid \sigma{:}\kappa \mid \rho \equiv \rho'{:}\kappa \mid t{:}\sigma \mid \sigma \succ \sigma' \mid J \ \wedge\ J'$

Figure 3.1: Syntax

3.0.2 Extending the framework

In this chapter I extend the unification algorithm from Chapter 2 (and hence type inference) to the theory of abelian groups. Mistaking occurrence for dependency will show up as the source of the difficulty described above, leading to a straightforward solution. With more structure in the context than just typing assumptions, it is easier to see where generality can be lost, and the loss of polymorphism can be avoided in the first place instead of recovered after the fact.

The syntax of contexts, expressions and statements is given in Figure 3.1. As before, a context is a list of metavariable declarations $\alpha{:}\kappa$, definitions $\alpha{:=}\rho : \kappa$, term variable declarations $x{:}\sigma$ and $\fatsemi$ markers. Now, however, metavariables may have kind $*$ (a type) or $\mathcal{U}$ (a unit). Similarly, type schemes record the kind of quantified variables, and the typing and equality statements include kinds. For example,

$$\alpha{:}*, \beta{:}\mathcal{U}, x{:}(\forall\gamma{:}\mathcal{U}.\,\alpha \to \mathbb{F}\langle\beta * \gamma\rangle)$$

is a valid context. A common syntax of type expressions ρ has subgrammars for types τ and units ν.

Figure 3.2 gives rules to construct a valid context and interpret variables in the context. These are similar to the rules for the Hindley-Milner system from the previous chapter (Figure 2.3, page 12), with the addition of the kind $\mathcal{U}$. Types are extended to include a single new type $\mathbb{F}\langle\nu\rangle$ representing a numeric type indexed by a unit ν. A real implementation would allow user-defined unit-indexed types, but one suffices for illustration.

$\boxed{\Theta \vdash \mathbf{ctx}}$ *(Θ is a valid context)*

$$\frac{}{\cdot \vdash \mathbf{ctx}} \qquad \frac{\alpha \# \Theta \quad \Theta \vdash \mathbf{ctx}}{\Theta, \alpha{:}\kappa \vdash \mathbf{ctx}} \qquad \frac{\alpha \# \Theta \quad \Theta \vdash \rho{:}\kappa}{\Theta, \alpha{:=}\rho : \kappa \vdash \mathbf{ctx}} \qquad \frac{x \# \Theta \quad \Theta \vdash \sigma{:}*}{\Theta, x{:}\sigma \vdash \mathbf{ctx}} \qquad \frac{\Theta \vdash \mathbf{ctx}}{\Theta \fatsemi \vdash \mathbf{ctx}}$$

$\boxed{\Theta \vdash \sigma{:}\kappa}$ *(σ is a well-formed scheme of kind κ in Θ)*

$$\frac{\Theta \ni \alpha{:}\kappa}{\Theta \vdash \alpha{:}\kappa} \qquad \frac{\Theta \vdash \tau{:}* \qquad \Theta \vdash \upsilon{:}*}{\Theta \vdash \tau \to \upsilon{:}*} \qquad \frac{\Theta \vdash \nu{:}\mathcal{U}}{\Theta \vdash \mathbb{F}\langle\nu\rangle{:}*} \qquad \frac{\Theta, \alpha{:}\kappa \vdash \sigma{:}*}{\Theta \vdash \forall \alpha{:}\kappa.\,\sigma{:}*}$$

$$\frac{}{\Theta \vdash b{:}\mathcal{U}} \qquad \frac{\Theta \vdash \nu{:}\mathcal{U} \qquad \Theta \vdash \nu'{:}\mathcal{U}}{\Theta \vdash \nu * \nu'{:}\mathcal{U}} \qquad \frac{\Theta \vdash \nu{:}\mathcal{U}}{\Theta \vdash \nu^{-1}{:}\mathcal{U}} \qquad \frac{}{\Theta \vdash 1{:}\mathcal{U}}$$

Figure 3.2: Rules for context validity and well-formed type schemes

$\boxed{\theta : \Theta_0 \sqsubseteq \Theta_1}$ *(θ is a metasubstitution from Θ_0 to Θ_1)*

$$\frac{}{[\,] : \cdot \sqsubseteq \Xi} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \rho{:}\kappa}{(\theta, \rho/\alpha) : \Theta_0, \alpha{:}\kappa \sqsubseteq \Theta_1} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \rho \equiv \theta\,\rho'{:}\kappa}{(\theta, \rho/\alpha) : \Theta_0, \alpha{:=}\rho' : \kappa \sqsubseteq \Theta_1}$$

$$\frac{\theta : \Theta_0 \sqsubseteq \Theta_1}{\theta : \Theta_0, x{:}\sigma \sqsubseteq \Theta_1, x{:}\theta\,\sigma, \Xi} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1}{\theta : \Theta_0 \fatsemi \sqsubseteq \Theta_1 \fatsemi \Xi}$$

$\boxed{\theta \equiv \theta'{:}\Theta_0 \sqsubseteq \Theta_1}$ *(θ and θ' are equivalent metasubstitutions from Θ_0 to Θ_1)*

$$\frac{}{\cdot \equiv \cdot{:}\cdot \sqsubseteq \Theta_1} \qquad \frac{\theta \equiv \theta'{:}\Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \rho \equiv \rho'{:}\kappa}{(\theta, \rho/\alpha) \equiv (\theta', \rho'/\alpha){:}\Theta_0, \alpha{:}\kappa \sqsubseteq \Theta_1}$$

$$\frac{\theta \equiv \theta'{:}\Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1 \vdash \rho \equiv \rho'{:}\kappa \qquad \Theta_1 \vdash \rho' \equiv \theta\,\rho''{:}\kappa}{(\theta, \rho/\alpha) \equiv (\theta', \rho'/\alpha){:}\Theta_0, \alpha{:=}\rho'' : \kappa \sqsubseteq \Theta_1}$$

$$\frac{\theta \equiv \theta'{:}\Theta_0 \sqsubseteq \Theta_1}{\theta \equiv \theta'{:}\Theta_0, x{:}\sigma \sqsubseteq \Theta_1, x{:}\theta\,\sigma, \Xi} \qquad \frac{\theta \equiv \theta'{:}\Theta_0 \sqsubseteq \Theta_1}{\theta \equiv \theta'{:}\Theta_0 \fatsemi \sqsubseteq \Theta_1 \fatsemi \Xi}$$

Figure 3.3: Rules for metasubstitutions

The updated rules for metasubstitutions are given in Figure 3.3. These are obvious extensions of the rules given in Subsection 2.1.2 (page 14).

Recall that a statement J is an assertion that can be judged in a context. The syntax of statements from the previous chapter is extended with kind information, and the sanity conditions (Lemma 2.1) are updated appropriately:

Lemma 3.1 (Sanity conditions). *If* $\Theta \vdash J$ *then* $\Theta \vdash \mathbf{San}\, J$*, where*

$$\begin{array}{rcl}
\mathbf{San}\,\mathbf{ctx} & \mapsto & \mathbf{ctx} \\
\mathbf{San}\,(\sigma : \kappa) & \mapsto & \mathbf{ctx} \\
\mathbf{San}\,(\tau \equiv \upsilon : \kappa) & \mapsto & \tau : \kappa \;\wedge\; \upsilon : \kappa \\
\mathbf{San}\,(t : \sigma) & \mapsto & \sigma : * \\
\mathbf{San}\,(\sigma \succ \sigma') & \mapsto & \sigma : * \;\wedge\; \sigma' : * \\
\mathbf{San}\,(J \;\wedge\; J') & \mapsto & \mathbf{San}\, J \;\wedge\; \mathbf{San}\, J'
\end{array}$$

Proof. By structural induction on derivations. □

The key results from the previous chapter, stability (Lemma 2.2, page 16), the category structure of contexts (Lemma 2.3, page 16), the Optimist's lemma (Lemma 2.4, page 18) and the isomorphism lemma (Lemma 2.5, page 18) apply to the updated notions of statement and metasubstitution without modification.

3.1 Unification for the theory of abelian groups

I now consider abelian group unification problems in the framework. The syntax of types τ is extended with units of measure ν given by

$$\begin{array}{llll}
\nu & ::= & & \\
& \mid & \alpha & \text{metavariable} \\
& \mid & b & \text{base unit} \\
& \mid & 1 & \text{identity} \\
& \mid & \nu * \nu' & \text{product of units} \\
& \mid & \nu^{-1} & \text{inverse}
\end{array}$$

where b ranges over some set of base units, which would be user-defined in a real system for units of measure. Note that units of measure ν are just type expressions of kind $\mathcal{U}$, but the typing rules ensure they must belong to this grammar.

The rules for equivalence of types and units are given in Figure 3.4: reflexivity, symmetry, transitivity and congruence, plus the four abelian group axioms of commutativity, associativity, inverses and identity.

$\boxed{\Theta \vdash \rho \equiv \rho' : \kappa}$ *(ρ and ρ' are equal expressions of kind κ in Θ)*

$$\frac{\Theta \vdash \rho : \kappa}{\Theta \vdash \rho \equiv \rho : \kappa} \qquad \frac{\Theta \vdash \rho \equiv \rho' : \kappa}{\Theta \vdash \rho' \equiv \rho : \kappa} \qquad \frac{\begin{array}{c}\Theta \vdash \rho_0 \equiv \rho_1 : \kappa \\ \Theta \vdash \rho_1 \equiv \rho_2 : \kappa\end{array}}{\Theta \vdash \rho_0 \equiv \rho_2 : \kappa} \qquad \frac{\begin{array}{c}\Theta \vdash \mathbf{ctx} \\ \Theta \ni \alpha := \rho : \kappa\end{array}}{\Theta \vdash \alpha \equiv \rho : \kappa}$$

$$\frac{\begin{array}{c}\Theta \vdash \tau \equiv \upsilon : * \\ \Theta \vdash \tau' \equiv \upsilon' : *\end{array}}{\Theta \vdash \tau \to \tau' \equiv \upsilon \to \upsilon' : *} \qquad \frac{\Theta \vdash \nu \equiv \nu' : \mathcal{U}}{\Theta \vdash \mathbb{F}\langle \nu \rangle \equiv \mathbb{F}\langle \nu' \rangle : *} \qquad \frac{\begin{array}{c}\Theta \vdash \nu_0 \equiv \nu_2 : \mathcal{U} \\ \Theta \vdash \nu_1 \equiv \nu_3 : \mathcal{U}\end{array}}{\Theta \vdash \nu_0 * \nu_1 \equiv \nu_2 * \nu_3 : \mathcal{U}}$$

$$\frac{\Theta \vdash \nu_0 \equiv \nu_1 : \mathcal{U}}{\Theta \vdash {\nu_0}^{-1} \equiv {\nu_1}^{-1} : \mathcal{U}} \qquad \frac{\Theta \vdash \nu : \mathcal{U}}{\Theta \vdash 1 * \nu \equiv \nu : \mathcal{U}} \qquad \frac{\Theta \vdash \nu : \mathcal{U} \qquad \Theta \vdash \nu' : \mathcal{U}}{\Theta \vdash \nu * \nu' \equiv \nu' * \nu : \mathcal{U}}$$

$$\frac{\Theta \vdash \nu_0 : \mathcal{U} \qquad \Theta \vdash \nu_1 : \mathcal{U} \qquad \Theta \vdash \nu_2 : \mathcal{U}}{\Theta \vdash (\nu_0 * \nu_1) * \nu_2 \equiv \nu_0 * (\nu_1 * \nu_2) : \mathcal{U}} \qquad \frac{\Theta \vdash \nu : \mathcal{U}}{\Theta \vdash \nu * \nu^{-1} \equiv 1 : \mathcal{U}}$$

Figure 3.4: Declarative rules for unit equivalence

Let ν^k mean ν multiplied by itself k times and $\nu^{(-k)}$ mean $(\nu^k)^{-1}$. Units have a normal form $\prod \overline{{\nu_i}^{k_i}}^{i}$ representing the product of some distinct atoms (variables or constants) ν_i, each raised to a nonzero integer power k_i. For example, the expression $\alpha * \alpha * \beta * 1 * \beta * \alpha$ has normal form $\alpha^3 * \beta^2$.

Consider the equation $\alpha^3 * \beta^2 \equiv 1$ in the context $\alpha : \mathcal{U}, \beta : \mathcal{U}$. As 2 does not divide 3, β cannot be defined to solve this equation, but the problem can be simplified by taking $\beta := \gamma * \alpha^{-1}$ where γ is a fresh variable. This leaves $\alpha * \gamma^2 \equiv 1$ in the context $\alpha : \mathcal{U}, \gamma : \mathcal{U}$, which is solved by rearranging and defining $\alpha := \gamma^{-2}$. Thus the solution is $\gamma : \mathcal{U}, \alpha := \gamma^{-2} : \mathcal{U}, \beta := \gamma * \alpha^{-1} : \mathcal{U}$, and indeed $\alpha^3 * \beta^2 \equiv (\gamma^{-2})^3 * (\gamma * \alpha^{-1})^2 \equiv \gamma^{-6} * \gamma^6 \equiv 1$. Along the way, the least common multiple of 2 and 3 has been calculated.

More generally, when solving such an equation, one can ask whether a variable has the largest power, and if not, reduce the other powers by it to simplify the problem. Some notation is in order. Suppose $\nu \equiv \prod \overline{{\nu_i}^{k_i}}^{i}$, and define:

$$\begin{array}{rll}
\mathsf{maxpow}(\nu) & = \mathsf{max}\{\, |\, k_i \,| \; : \nu_i \text{ metavariable}\}, & \text{highest absolute variable power;} \\
\mathsf{Q}_k(\nu) & = \prod \overline{{\nu_i}^{(k_i \,\mathsf{quot}\, k)}}^{i}, & \text{quotient by } k \text{ of every power;} \\
\mathsf{R}_k(\nu) & = \prod \overline{{\nu_i}^{(k_i \,\mathsf{rem}\, k)}}^{i}, & \text{remainder by } k \text{ of every power;}
\end{array}$$

where quot is truncated integer division and rem is the corresponding remainder. The point is that $\nu \equiv (\mathsf{Q}_k(\nu))^k * \mathsf{R}_k(\nu)$ and $\mathsf{maxpow}(\mathsf{R}_k(\nu)) < k$.

$\boxed{\Theta_0 \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1}$ *(unifying ν with 1 in Θ_0, Υ results in Θ_1)*

$$\frac{}{\Theta \,\|\, \cdot \vdash 1 \equiv 1 : \mathcal{U} \dashv \Theta}\ \textsc{u-trivial} \qquad \frac{\Theta_0 \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0 \,\fatsemi\, \|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1 \fatsemi}\ \textsc{u-skip-semi}$$

$$\frac{\alpha \notin \mathsf{fmv}(\nu) \qquad \Theta_0 \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0, \alpha{:}\kappa \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1, \alpha{:}\kappa}\ \textsc{u-skip-ty}$$

$$\frac{\Theta_0 \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0, x{:}\sigma \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1, x{:}\sigma}\ \textsc{u-skip-tm}$$

$$\frac{\Theta_0, \Upsilon \,\|\, \cdot \vdash [\rho/\alpha]\,\nu \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0, \alpha{:=}\rho : \kappa \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1, \alpha{:=}\rho : \kappa}\ \textsc{u-subs}$$

$$\frac{k \neq 0}{\Theta, \alpha{:}\,\mathcal{U} \,\|\, \Upsilon \vdash \alpha^k * \nu^k \equiv 1 : \mathcal{U} \dashv \Theta, \Upsilon, \alpha{:=}\nu^{-1} : \mathcal{U}}\ \textsc{u-define}$$

$$\frac{\begin{array}{c}|\,k\,| \ \leq\ \mathsf{maxpow}(\nu) \qquad \beta \text{ fresh} \\ \Theta_0, \Upsilon \,\|\, \beta{:}\,\mathcal{U} \vdash \beta^k * \mathsf{R}_k(\nu) \equiv 1 : \mathcal{U} \dashv \Theta_1\end{array}}{\Theta_0, \alpha{:}\,\mathcal{U} \,\|\, \Upsilon \vdash \alpha^k * \nu \equiv 1 : \mathcal{U} \dashv \Theta_1, \alpha{:=}\beta * \mathsf{Q}_k(\nu) : \mathcal{U}}\ \textsc{u-reduce}$$

$$\frac{|\,k\,| \ >\ \mathsf{maxpow}(\nu) \qquad \Theta_0 \,\|\, \alpha{:}\,\mathcal{U} \vdash \alpha^k * \nu \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0, \alpha{:}\,\mathcal{U} \,\|\, \cdot \vdash \alpha^k * \nu \equiv 1 : \mathcal{U} \dashv \Theta_1}\ \textsc{u-collect}$$

Figure 3.5: Algorithmic rules for abelian group unification

3.1.1 The abelian group unification algorithm

In this subsection, I give a new algorithm for unification problems $\nu \equiv \nu' {:} \mathcal{U}$. The inverse operation means it suffices to solve problems $\nu \equiv 1 {:} \mathcal{U}$.

Figure 3.5 shows the algorithm presented as a collection of inference rules. Given a context Θ_0, Υ and a unit ν, the judgment $\Theta_0 \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1$ means that the algorithm outputs the context Θ_1 such that $\Theta_1 \vdash \nu \equiv 1 {:} \mathcal{U}$. Note that the rules are entirely syntax-directed (up to the equational theory for units): at most one rule applies for any possible initial context and unit. They lead directly to an implementation, which is given in Appendix B.3 (page 210).

So how does the algorithm work? If the problem is $1 \equiv 1$, then it is solved by u-trivial. Otherwise, the algorithm moves back through the context, skipping over (meta)variables that do not occur in the problem using u-skip-ty or u-skip-tm, and moving through localities using u-skip-semi.

The suffix Υ will either be empty (written $\cdot$) or contain only the unknown variable with the strictly largest power in ν, if any. The U-REDUCE and U-COLLECT rules move this variable back in the context, since there is no useful simplification that can be applied to it. Other rules will insert the variable into the context when it no longer has the largest power.

The interesting cases arise when a metavariable α, that occurs in the problem, in reached. This is written $\alpha^k * \nu \equiv 1$, always meaning that $\alpha \notin \mathsf{fmv}(\nu)$. Suppose the normal form of ν is $\prod \overline{{\nu_i}^{k_i}}^{i}$. There are four possibilities, either:

(1) k divides k_i for all i;

(2) ν has at least one variable and $|\,k\,| \leq \mathsf{maxpow}(\nu)$ but case (1) does not apply;

(3) ν has at least one variable and $|\,k\,| > \mathsf{maxpow}(\nu)$; or

(4) ν has no variables.

Case (1). If k divides k_i for all i, then there is some ν_0 such that $\nu \equiv {\nu_0}^k$. The rule U-DEFINE applies and sets $\alpha := {\nu_0}^{-1} : \mathcal{U}$ to give

$$\alpha^k * \nu \equiv \alpha^k * {\nu_0}^k \equiv ({\nu_0}^{-1})^k * {\nu_0}^k \equiv 1.$$

This is clearly a solution, and it is most general for the free abelian group.

Case (2). If not, and $|\,k\,| \leq \mathsf{maxpow}(\nu)$, then the U-REDUCE rule applies and simplifies the problem by reducing the powers modulo k. Recall that we have $\nu \equiv (\mathsf{Q}_k(\nu))^k * \mathsf{R}_k(\nu)$ where $\mathsf{Q}_k(\nu)$ takes the quotient by k of the powers in ν. Hence, generating a fresh variable β and defining $\alpha := \beta * \mathsf{Q}_k(\nu)^{-1}$ gives

$$\alpha^k * \nu \equiv (\beta * \mathsf{Q}_k(\nu)^{-1})^k * \nu \equiv \beta^k * \mathsf{Q}_k(\nu)^{-k} * \nu \equiv \beta^k * \mathsf{R}_k(\nu).$$

Case (3). Suppose $|\,k\,| > \mathsf{maxpow}(\nu)$, so neither of the two previous cases apply, but there is at least one variable in ν. Now k is the largest power of a variable, so reducing the powers modulo k would leave them unchanged. Instead, the U-COLLECT rule moves α further back in the context. This rule maintains the invariant that Υ contains only the variable with the largest power, if any; the invariant also guarantees that Υ will be empty when the rule applies.

Case (4). If ν has no variables and k does not divide the powers of the constants in ν, then $\alpha^k * \nu \equiv 1$ has no solution in the free abelian group.

3.1.2 Correctness of abelian group unification

The problem-solving apparatus introduced in Subsection 2.1.3 carries over without change to this new setting, where the language of statements is more general. In particular, abelian group unification delivers minimal solutions.

Lemma 3.2 (Soundness and generality of abelian group unification)**.** *If the group unification algorithm succeeds with* $\Theta_0 \parallel \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1$*, then* $\Theta_0, \Upsilon \sqsubseteq \Theta_1$ *is a minimal solution of* $\nu \equiv 1 {:} \mathcal{U}$*.*

Proof. By induction on derivations, using the isomorphism lemma (Lemma 2.5). For details, see Appendix D.2 (page 239). □

Lemma 3.3 (Completeness of abelian group unification)**.** *If* ν *is a well-formed unit of measure in* Θ_0*, and there is some* $\theta : \Theta_0 \sqsubseteq \Theta'$ *such that* $\Theta' \vdash \theta\,\nu \equiv 1 {:} \mathcal{U}$*, then the algorithm produces* Θ_1 *such that* $\Theta_0 \parallel \cdot \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1$*.*

Proof. A suitable metric shows that the algorithm terminates. Completeness is by the fact that the rules cover all solvable cases and preserve solutions: if no rule applies then the original problem can have had no solutions. This occurs if a constant is equated to 1 (e.g. $\mathbf{kg} * \mathbf{s} \equiv 1$) or there is one variable and its power does not divide the power of one of the constants (e.g. $\alpha^2 * \mathbf{kg} \equiv 1$). For details, see Appendix D.2 (page 239). □

3.2 Unification for types with units of measure

Having developed a unification algorithm for abelian groups, I now extend type unification to support units of measure, calling group unification from Section 3.1 as a subroutine to solve constraints on units. As in the type unification algorithm of the previous chapter (Figure 2.5, page 21), there are two kinds of rules:

- 'Unify' steps start the process: given an input context Θ_0 and well-formed types τ and υ, the judgment $\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1$ means that the unification problem $\tau \equiv \upsilon {:} *$ is solved with output context Θ_1.
- 'Instantiate' steps handle flex-rigid unification problems:[2] given a context Θ_0, Ξ, a type metavariable α in Θ_0 and a well-formed non-variable type τ over Θ_0, Ξ, the judgment $\Theta_0 \mid \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$ means that the problem

[2]Recall that a *flex-rigid* problem is to unify a variable and a non-variable expression; a *flex-flex* problem has two variables and a *rigid-rigid* problem has two non-variables.

$\alpha \equiv \tau : *$ is solved with output context Θ_1. The context suffix Ξ collects metavariable declarations that τ depends on but that cannot be used to solve the problem.

Compared to the previous chapter, the language of types (of kind $*$) now includes a single new type $\mathbb{F}\langle\nu\rangle$ of numbers parameterised by units. I therefore add a type unification rule that invokes abelian group unification:

$$\frac{\Theta_0 \,\|\, \cdot \vdash \nu_0 * {\nu_1}^{-1} \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0 \vdash \mathbb{F}\langle\nu_0\rangle \equiv \mathbb{F}\langle\nu_1\rangle : * \dashv \Theta_1}\ \textsc{unit}$$

Now suppose the algorithm is used to solve $\mathbb{F}\langle\beta_0 * \beta_1\rangle \to \alpha \equiv \mathbb{F}\langle\beta_0\rangle \to \mathbb{F}\langle\beta_1\rangle$ in the context $\beta_0 : \mathcal{U}, \alpha : *, \beta_1 : \mathcal{U}$. First the constraint $\mathbb{F}\langle\beta_0 * \beta_1\rangle \equiv \mathbb{F}\langle\beta_0\rangle : *$ is reduced to $\beta_0 * \beta_1 \equiv \beta_0 : \mathcal{U}$ by UNIT, and this is solved by group unification (Section 3.1) to give $\beta_0 : \mathcal{U}, \alpha : *, \beta_1 := 1 : \mathcal{U}$. Then the constraint $\alpha \equiv \mathbb{F}\langle\beta_1\rangle$ is solved to give $\beta_0 : \mathcal{U}, \alpha := \mathbb{F}\langle 1\rangle : *, \beta_1 := 1 : \mathcal{U}$

Do the rules in Figure 2.5 extended with the UNIT rule give a correct unification algorithm for the extended type system? The unification algorithm should be sound and complete, as the new algorithmic rule corresponds directly to the declarative rule, but generality fails. Most general unifiers are needed for completeness of type inference, so something had better be done.

3.2.1 Loss of generality and how to retain it

Suppose the algorithm is used to solve the constraint $\alpha \equiv \mathbb{F}\langle\beta_0 * \beta_1\rangle$ in the context $\alpha : * \fatsemi \beta_0 : \mathcal{U}, \beta_1 : \mathcal{U}$. As the rules stand, this flex-rigid problem is solved by moving β_0 and β_1 into the previous locality, and defining α resulting in the context $\beta_0 : \mathcal{U}, \beta_1 : \mathcal{U}, \alpha := \mathbb{F}\langle\beta_0 * \beta_1\rangle : * \fatsemi$. However, another solution exists, namely $\gamma : \mathcal{U}, \alpha := \mathbb{F}\langle\gamma\rangle : * \fatsemi \beta_0 : \mathcal{U}, \beta_1 := {\beta_0}^{-1} * \gamma : \mathcal{U}$, where γ is a fresh group variable. This solution is more general because β_0 is still local (it has not been moved past the $\fatsemi$ marker). Why did the algorithm fail to find this?

The trouble is that, to solve a flex-rigid constraint, the variable need not be *syntactically* equal to the type: units need be equal only up to the theory of abelian groups. The property that equivalent expressions have the same sets of free variables[3] holds for the syntactic theory and some other useful theories (Rémy, 1992) but does not hold for groups. For example, the equation $\alpha * \alpha^{-1} \equiv 1$

[3]This property is sometimes called *regularity* in the literature, but I avoid this term because it means too many different things in other contexts.

has α free on the left but not the right. Thus variable occurrence does not imply dependency. The occurs check in the unification algorithm is overly syntactic.

To solve this, a flex-rigid constraint can be decomposed into a constraint on types, with fresh variables in place of units, and additional constraints to make the fresh variables equal to the units. A rigid type decomposes into a 'hull', or 'type skeleton', that must match exactly, and a collection of constraints in the richer equational theory. Similar techniques are used for type inference in annotated type systems (Nielson et al., 1999, §5.3.2).

In the example, the constraint $\alpha \equiv \mathbb{F}\langle\beta_0 * \beta_1\rangle$ decomposes into two constraints $\alpha \equiv \mathbb{F}\langle\gamma\rangle : * \ \wedge \ \gamma \equiv \beta_0 * \beta_1 : \mathcal{U}$ in the context $\alpha : * \fatsemi \beta_0 : \mathcal{U}, \beta_1 : \mathcal{U}, \gamma : \mathcal{U}$. Solving the first constraint gives $\gamma : \mathcal{U}, \alpha := \mathbb{F}\langle\gamma\rangle : * \fatsemi \beta_0 : \mathcal{U}, \beta_1 : \mathcal{U}$, and solving the second yields the most general solution $\beta' : \mathcal{U}, \alpha := \mathbb{F}\langle\gamma\rangle : * \fatsemi \beta_0 : \mathcal{U}, \beta_1 := ({\beta_0}^{-1} * \gamma) : \mathcal{U}$.

Committing only to the hull is the minimal commitment entailed by the equation, as far as the equational theory on types goes. One could even go further and solve every flex-rigid equation one constructor layer at a time, so $\alpha \equiv \tau \to \upsilon$ would be solved by $\alpha \equiv \beta_0 \to \beta_1 \ \wedge \ \beta_0 \equiv \tau \ \wedge \ \beta_1 \equiv \upsilon$.

The rules from Figure 2.5 (page 21) can be modified to maintain the invariant that the only unit metavariables a flex-rigid problem depends on (i.e. those in the rigid type τ or suffix Ξ) are fresh unknowns. Unit metavariables are never made less local by collecting them in Ξ as dependencies. Type unification does not prejudice locality of unit metavariables: they must be left for group unification. The rule

$$\frac{\tau \text{ non-variable} \qquad \Theta_0 \,|\, \cdot \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0 \vdash \alpha \equiv \tau : * \dashv \Theta_1}\ \textsc{inst}$$

is replaced by

$$\frac{\begin{array}{l}\tau \text{ non-variable} \qquad \overline{\beta_i}^{\,i} \text{ fresh} \\ \Theta_0 \,|\, \overline{\beta_i : \mathcal{U}}^{\,i} \vdash \alpha \equiv \tau\{\,\overline{\beta_i}^{\,i}\,\} : * \dashv \Theta_1 \\ \Theta_1 \vdash \overline{\beta_i \equiv \nu_i : \mathcal{U}}^{\,i} \dashv \Theta_2\end{array}}{\Theta_0 \vdash \alpha \equiv \tau\{\,\overline{\nu_i}^{\,i}\,\} : * \dashv \Theta_2}\ \textsc{inst}$$

where $\tau\{\,\overline{\nu_i}^{\,i}\,\}$ is the hull of the type τ, parameterised by a vector of units (so $\mathbb{F}\langle\nu_0\rangle \to \mathbb{F}\langle\nu_1\rangle$ has hull $\mathbb{F}\langle\_\rangle \to \mathbb{F}\langle\_\rangle$ and $\tau\{\alpha_0, \alpha_1\} = \mathbb{F}\langle\alpha_0\rangle \to \mathbb{F}\langle\alpha_1\rangle$). Vectors of equations are solved one at a time, threading the context:

$$\frac{\Theta_0 \,\|\, \cdot \vdash \beta_0 * {\nu_0}^{-1} \equiv 1 : \mathcal{U} \dashv \Theta_1 \quad \dots \quad \Theta_{n-1} \,\|\, \cdot \vdash \beta_{n-1} * {\nu_{n-1}}^{-1} \equiv 1 : \mathcal{U} \dashv \Theta_n}{\Theta_0 \vdash \beta_0 \equiv \nu_0 : \mathcal{U} \ \wedge \ \dots \ \wedge \ \beta_{n-1} \equiv \nu_{n-1} : \mathcal{U} \dashv \Theta_n}\ \textsc{conj}$$

The updated rules are given in Figures 3.6 and 3.7. Apart from the addition of the UNIT rule, and the modification to the INST rule, the only changes are minor generalisations, such as changing rules to work with an arbitrary kind κ, rather than just $*$. Again, symmetric variants of the INST and DEFINE rules have been omitted. The implementation is given in Appendix B.4 (page 212).

Similarly to Definition 2.1 (page 20) in the previous chapter, the instantiation part of the algorithm expects a number of conditions to be satisfied:

Definition 3.1. The quadruple $(\Theta_0, \Xi, \alpha, \tau)$ *satisfies the input conditions* if

- $\Theta_0 \vdash \alpha : *$ where α is a metavariable,
- $\Theta_0, \Xi \vdash \tau : *$ where τ is not a metavariable,
- Ξ contains only metavariable declarations $\beta : \kappa$ with $\beta \in \mathsf{fmv}(\tau)$, and
- if $\mathbb{F}\langle \nu \rangle$ is a subterm of τ then $\nu = \beta$ for some β with $\Xi \ni \beta : \mathcal{U}$.

The crucial addition, maintained by the new INST rule, is the last condition. This is necessary for generality, as it ensures that every unit metavariable in Ξ is a true dependency of τ, and completeness, as it ensures that Ξ captures *all* the unit metavariable dependencies of τ, so the algorithm will not encounter an unexpected unit metavariable dependency and get stuck.

3.2.2 Correctness of type unification

With the above refinement, type unification gives most general results.

Lemma 3.4 (Soundness and generality of type unification)**.**

(a) If $\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1$*, then* $\Theta_0 \sqsubseteq \Theta_1$ *is a minimal solution of* $\tau \equiv \upsilon : *$.

(b) If $\Theta_0 \mid \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$*, then* $\Theta_0, \Xi \sqsubseteq \Theta_1$ *is a minimal solution of* $\alpha \equiv \tau : *$.

Proof. Proceed by induction on the structure of derivations, as in Lemma 2.6 (page 22). The majority of the cases are similar to the previous proof, but the UNIT rule is new, the INST rule has been modified. The INST-SKIP-SEMI rule requires a more subtle generality proof, in order to verify that instantiation moves only genuine dependencies. The input conditions ensure that units always occur in the form $\mathbb{F}\langle \alpha \rangle$, so it is obvious that α is a dependency. For details, see Appendix D.2 (page 240). □

$\boxed{\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1}$ *(unifying τ with υ in Θ_0 results in Θ_1)*

$$\frac{\Theta_0 \vdash \tau_0 \equiv \upsilon_0 : * \dashv \Theta_1 \qquad \Theta_1 \vdash \tau_1 \equiv \upsilon_1 : * \dashv \Theta_2}{\Theta_0 \vdash (\tau_0 \to \tau_1) \equiv (\upsilon_0 \to \upsilon_1) : * \dashv \Theta_2}\ \textsc{decompose}$$

$$\frac{\Theta_0 \,\|\, \cdot \vdash \nu_0 * {\nu_1}^{-1} \equiv 1 : \mathcal{U} \dashv \Theta_1}{\Theta_0 \vdash \mathbb{F}\langle \nu_0 \rangle \equiv \mathbb{F}\langle \nu_1 \rangle : * \dashv \Theta_1}\ \textsc{unit}$$

$$\frac{\tau \text{ non-variable} \qquad \overline{\beta_i}^{\,i} \text{ fresh} \qquad \Theta_0 \,|\, \overline{\beta_i{:}\mathcal{U}}^{\,i} \vdash \alpha \equiv \tau\{\,\overline{\beta_i}^{\,i}\,\} : * \dashv \Theta_1 \qquad \Theta_1 \vdash \overline{\beta_i \equiv \nu_i{:}\mathcal{U}}^{\,i} \dashv \Theta_2}{\Theta_0 \vdash \alpha \equiv \tau\{\,\overline{\nu_i}^{\,i}\,\} : * \dashv \Theta_2}\ \textsc{inst}$$

$$\frac{}{\Theta, \alpha{:}* \vdash \alpha \equiv \alpha : * \dashv \Theta, \alpha{:}*}\ \textsc{idle} \qquad \frac{\alpha \neq \beta}{\Theta, \alpha{:}* \vdash \alpha \equiv \beta : * \dashv \Theta, \alpha{:=}\beta : *}\ \textsc{define}$$

$$\frac{\Theta_0 \vdash [\rho/\gamma]\,\alpha \equiv [\rho/\gamma]\,\beta : * \dashv \Theta_1}{\Theta_0, \gamma{:=}\rho : \kappa \vdash \alpha \equiv \beta : * \dashv \Theta_1, \gamma{:=}\rho : \kappa}\ \textsc{subs}$$

$$\frac{\Theta_0 \vdash \alpha \equiv \beta : * \dashv \Theta_1 \qquad \alpha \neq \gamma \qquad \beta \neq \gamma}{\Theta_0, \gamma{:}\kappa \vdash \alpha \equiv \beta : * \dashv \Theta_1, \gamma{:}\kappa}\ \textsc{skip-ty}$$

$$\frac{\Theta_0 \vdash \alpha \equiv \beta : * \dashv \Theta_1}{\Theta_0, x{:}\sigma \vdash \alpha \equiv \beta : * \dashv \Theta_1, x{:}\sigma}\ \textsc{skip-tm} \qquad \frac{\Theta_0 \vdash \alpha \equiv \beta : * \dashv \Theta_1}{\Theta_0 \fatsemi \vdash \alpha \equiv \beta : * \dashv \Theta_1 \fatsemi}\ \textsc{skip-semi}$$

$$\frac{\Theta_0 \,\|\, \cdot \vdash \beta_0 * {\nu_0}^{-1} \equiv 1 : \mathcal{U} \dashv \Theta_1 \quad \ldots \quad \Theta_{n-1} \,\|\, \cdot \vdash \beta_{n-1} * {\nu_{n-1}}^{-1} \equiv 1 : \mathcal{U} \dashv \Theta_n}{\Theta_0 \vdash \beta_0 \equiv \nu_0{:}\mathcal{U} \ \wedge\ \ldots\ \wedge\ \beta_{n-1} \equiv \nu_{n-1}{:}\mathcal{U} \dashv \Theta_n}\ \textsc{conj}$$

Figure 3.6: Algorithmic rules for type unification (part 1)

$\boxed{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}$ *(instantiating α with τ in Θ_0, Ξ results in Θ_1)*

$$\frac{\alpha \notin \mathsf{fmv}(\tau)}{\Theta_0, \alpha{:}* \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_0, \Xi, \alpha{:=}\tau : *}\ \textsc{inst-define}$$

$$\frac{\Theta_0, \Xi \vdash [\rho/\beta]\,\alpha \equiv [\rho/\beta]\,\tau : * \dashv \Theta_1}{\Theta_0, \beta{:=}\rho : \kappa \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1, \beta{:=}\rho : \kappa}\ \textsc{inst-subs}$$

$$\frac{\Theta_0 \,|\, \beta{:}*, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \qquad \alpha \neq \beta \qquad \beta \in \mathsf{fmv}(\tau)}{\Theta_0, \beta{:}* \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}\ \textsc{inst-depend}$$

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \qquad \alpha \neq \beta \qquad \beta \notin \mathsf{fmv}(\tau)}{\Theta_0, \beta{:}\kappa \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1, \beta{:}\kappa}\ \textsc{inst-skip-ty}$$

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0, x{:}\sigma \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1, x{:}\sigma}\ \textsc{inst-skip-tm}$$

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0 \fatsemi \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \fatsemi}\ \textsc{inst-skip-semi}$$

Figure 3.7: Algorithmic rules for type unification (part 2)

Lemma 3.5 (Completeness of type unification)**.**

(a) If the types υ and τ are well-formed in Θ_0 and there is some $\theta : \Theta_0 \sqsubseteq \Theta'$ with $\Theta' \vdash \theta\,\upsilon \equiv \theta\,\tau : *$, then unification produces Θ_1 such that $\Theta_0 \vdash \upsilon \equiv \tau : * \dashv \Theta_1$.

(b) Moreover, if $\theta : \Theta_0, \Xi \sqsubseteq \Theta'$ is such that $\Theta' \vdash \theta\,\alpha \equiv \theta\,\tau : *$ and the input conditions (Definition 3.1) are satisfied, then there is some context Θ_1 such that $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$.

Proof. Termination of the algorithm can be established via an appropriate ordering. Proceed by structural induction on the call graph, observing that each rule preserves solutions, and that all (potentially solvable) cases are covered. Completeness of appeals to group unification follows from Lemma 3.3. For more details, see Appendix D.2 (page 241). □

3.3 Type inference for units of measure

I have given a unification algorithm for types containing units of measure in Section 3.2, and this extends to a type inference algorithm for the corresponding type system. Given the new types, amended unification algorithm and the ability for type schemes to quantify over variables of kind $\mathcal{U}$, no changes to the type inference algorithm from Section 2.3 are required.

Generalisation is easy and there is no need to complicate the type inference algorithm to deal with units of measure. The initial context can be extended with constant terms that use the new types. Moreover, thanks to the refinement of Section 3.2.1, the algorithm copes naturally with the problematic term from Subsection 3.0.1, correctly inferring its most general type. Recall the example:

$$\lambda x. \mathbf{let}\; y = \mathrm{div}\; x \;\mathbf{in}\; (y\,\mathrm{mass}, y\,\mathrm{time}), \qquad \text{where}$$
$$\mathrm{div} : \forall\alpha{:}\mathcal{U}.\,\forall\beta{:}\mathcal{U}.\, \mathbb{F}\langle\alpha * \beta\rangle \to \mathbb{F}\langle\alpha\rangle \to \mathbb{F}\langle\beta\rangle, \qquad \mathrm{mass} : \mathbb{F}\langle\mathbf{kg}\rangle, \qquad \mathrm{time} : \mathbb{F}\langle\mathbf{s}\rangle.$$

At the crucial point when the type of y is being inferred, the situation is

$$\alpha : *, x : \alpha \,⨟\, \beta_0 : \mathcal{U}, \beta_1 : \mathcal{U} \vdash \mathrm{div}\; x : \mathbb{F}\langle\beta_0\rangle \to \mathbb{F}\langle\beta_1\rangle \qquad \text{subject to } \alpha \equiv \mathbb{F}\langle\beta_0 * \beta_1\rangle,$$

where α is an unknown fresh type variable standing in for the type of x. The constraint decomposes into two simpler constraints $\alpha \equiv \mathbb{F}\langle\gamma\rangle : * \;\wedge\; \gamma \equiv \beta_0 * \beta_1 : \mathcal{U}$ with γ a fresh unit metavariable. These can be solved one at a time to give the solution $\gamma : \mathcal{U}, \alpha := \mathbb{F}\langle\gamma\rangle : *, x : \alpha \,⨟\, \beta_0 : \mathcal{U}, \beta_1 := (\gamma * {\beta_0}^{-1}) : \mathcal{U}$. Generalising by ‘skimming off’ type variables in the locality gives the type scheme

$$\gamma : \mathcal{U}, x : \mathbb{F}\langle\gamma\rangle \vdash y : \forall \beta_0 : \mathcal{U}.\, \mathbb{F}\langle\beta_0\rangle \to \mathbb{F}\langle\gamma * {\beta_0}^{-1}\rangle,$$

which is principal. Type inference for the whole term succeeds, giving the type

$$\mathbb{F}\langle\gamma\rangle \to (\mathbb{F}\langle\gamma * \mathbf{kg}^{-1}\rangle, \mathbb{F}\langle\gamma * \mathbf{s}^{-1}\rangle).$$

3.4 Discussion

I have shown how to combine abelian group unification with syntactic unification while carefully tracking dependencies in a structured context, so generalisation is straightforward. Crucially, contexts capture an appropriate notion of locality, so a local solution is more general than a global one. The algorithms presented here solve unification problems by making gradual steps towards a solution, and it is comparatively easy to check that each step is sound and most general. A key point is that flex-rigid equations $\alpha \equiv \tau$ cannot always be solved by substituting τ for α, given a nontrivial equational theory. Instead, τ decomposes into a ‘hull’ (the outer structure that α must match exactly) and a collection of constraints in the equational theory.

This technique can be applied to other equational theories and more advanced type systems. The integers are an abelian group under addition, so the work in this chapter could be combined with the account of elaboration in Chapter 7 to elaborate types indexed by integers.

In this chapter I have been following the trail that Kennedy blazed, in the representation of units of measure using a free abelian group, the observation that unification has unique most general unifiers in this case, and the application of these properties to type inference. To extend the technique to less convenient type systems, I will need to deal with problems that cannot necessarily be solved on the first attempt. In the next chapter, I will examine higher-order unification, which is useful for elaborating higher-rank and dependent types. ‘Pattern unification’ as introduced by Miller (1992) provides a solid starting point, but here an explicit representation of postponed unification problems will be essential, because not all higher-order unification problems fall into the fragment that can be solved immediately.

3.4.1 Related work

Many authors have proposed designs for systems of units of measure. I have followed Kennedy's design, using integer powers, so units form an abelian group. Some authors use rational powers (giving a vector space), including Rittri (1995), who discusses the merits of both approaches. Chen et al. (2003) give a useful overview of work on units, and describe an alternative approach using static analysis.

Several impressive implementations of units of measure use advanced type system features such as GHC Haskell extensions (Buckwalter, n.d.) and C++ templates (Schabel and Watanabe, 2013). However, the difficulty of expressing a nontrivial equational theory at the type level means that they are complex, have limited inference capabilities and tend to expose the internal implementation in unfriendly error messages. Making units a type system extension, as in F#, results in a much more user-friendly system.

Rémy (1992) extends the ML type system with other equational theories for which variable occurrence *does* imply dependency (specifically excluding abelian groups). As discussed in the previous chapter, his unification algorithm achieves easy generalisation by tracking the 'ranks' at which type variables are introduced.

Sulzmann et al. (1999) propose a version of the HM(X) framework for representing type systems in constraint form, which avoids the generalisation problems discussed in this chapter by allowing constraints to be quantified over instead of solving them immediately. This is a very useful technique, although it is practically desirable to solve unification constraints as soon as possible (in the interests of efficiency and good error reporting).

Chapter 4

Miller pattern unification

Higher-order unification is the problem of finding definitions for metavariables in order to solve an equation between two λ-calculus terms. It extends first-order unification, as discussed in Chapter 2, in that

- terms have a binding structure, so unifiers must respect variable scope: e.g. $\lambda x.\alpha \approx \lambda x.x$ can only be solved by $\alpha := x$ if the metavariable α may depend on the bound variable x; and
- terms have a nontrivial equational theory, given by the β- and η-rules:[1] for example, $\lambda x.x \approx \lambda x.\lambda y.\alpha\, x\, y$ can be solved by $\alpha := \lambda z.z$ since

$$\lambda x.\lambda y.(\lambda z.z)\, x\, y \equiv_\beta \lambda x.\lambda y.x\, y \equiv_\eta \lambda x.x.$$

Given these complications it is perhaps unsurprising that full higher-order unification is undecidable (Huet, 1973). Most general unifiers do not necessarily exist and terms may have infinite sets of unifiers, though they can be generated by a semidecision procedure (Huet, 1975). Miller (1992) observed that a useful subproblem, unification in the *pattern fragment*, is decidable and has unique most general unifiers if they exist at all. Here metavariables must be applied to spines of distinct bound variables, so $\lambda x.x \approx \lambda x.\lambda y.\alpha\, x\, y$ is included but $\lambda x.\alpha\, x\, x \approx \lambda x.x$ is not; observe that the latter has two incompatible solutions $\alpha := \lambda x.\lambda y.x$ and $\alpha := \lambda y.\lambda x.x$. Equations that look like definitions, are definitions: $\alpha\, \overline{x_i}^{\,i} \approx t$ can be solved by $\alpha := \lambda\, \overline{x_i}^{\,i}\, .t$. An application to variables determines a metavariable fully, while an application to other terms determines it only in part (for example, $\alpha\,(\lambda x.x) \approx t$ cannot easily be solved).

[1] One can consider β-equality alone, but for the purposes of this chapter I will need both.

Dependently typed programming languages rely on higher-order unification for elaborating source programs, much as Hindley-Milner type inference makes use of first-order unification. Languages with a kernel type theory, such as Coq (Coq Development Team, 2013) and Epigram (McBride and McKinna, 2004), do not need unification in the kernel, but they depend on it to elaborate human-readable syntax. Likewise, Agda (Norell, 2007) uses higher-order unification for pattern matching and implicit argument synthesis. During the elaboration of a source language program, metavariables are inserted to stand for function arguments that the user has omitted, and unification problems arise when types do not match exactly. Elaboration will be considered in more detail in Chapter 7. Dependent types naturally lead to higher-order unification problems, since functions express dependency (for example, consider solving for α and β in $\Pi x : \alpha.\, \beta\, x \approx T$).

Programmers in a dependently typed language need to grasp the capabilities of unification if they are to become productive users of the language. Knowing what to omit, because the machine can reconstruct it for you, is a crucial aspect of writing comprehensible programs.

Languages with simple pairs or Σ-types (pairs in which the type of the second component may depend on the value of the first component) motivate extending the pattern fragment to projections. For example, consider $\alpha\, \textsc{hd}\, x \approx x$ where postfix $\textsc{hd}$ is first projection. This does not fall in the original pattern fragment but has most general solution $[(\lambda x.x, \beta)/\alpha]$ where β is a fresh variable.

For many applications, the static pattern fragment is overly restrictive: one often has multiple constraints, some of which fall into the fragment and some of which do not, but solving one constraint may make bring others into pattern form. This leads to 'dynamic' pattern unification, where non-pattern constraints may be postponed in case they are solvable later. For example, given the constraints $\alpha\, x \approx \beta$ and $\alpha\, y\, y \approx t$, the latter is not in the pattern fragment, but after solving the first constraints via $\alpha := \lambda x.\beta$ the second becomes $\beta\, y \approx t$.

Dynamic treatment of constraints is necessary even in first-order problems, because there is no fixed positional order of constraint solving that will work in all cases. For example, consider the problem $(\alpha + \beta, \alpha) \approx (3, 0)$ where α and β are natural number metavariables. If an algorithm always unifies the components of pairs from left to right, it gets stuck on the constraint $\alpha + \beta \approx 3$. On the other hand, after solving $\alpha \approx 0$, the first constraint computes to the much easier $\beta \approx 3$.[2] The Coq proof assistant, used as a dependently typed programming language, suffers from exactly this problem.

[2] Always unifying from right to left is no better: what if we swap the pair's components?

In this chapter, I present a dynamic pattern unification algorithm for a language with full-spectrum dependent types including Σ-types. It includes:

- the use of heterogeneous equality constraints to maintain typing discipline;
- a novel notion of 'twin variables' used to simplify problems heterogeneously when a variable must be assigned two intensionally distinct types, as in $(\lambda x.s : \Pi x : A.\ B) \approx (\lambda x.t : \Pi x : S.\ T)$;
- an extension of the context structure from previous chapters, suitable for managing dependency and partial progress on unification problems; and
- the demonstration of a minimal-commitment unification algorithm that makes it easy to deliver most general unifiers, when they exist.

In Section 4.1, I describe the type theory in which I will work. I give the algorithm in Section 4.2, with a high-level specification via rewrite rules. Correctness properties of the algorithm are proved in Section 4.3, although termination is problematic. Finally, some concluding remarks form Section 4.4. A Haskell reference implementation of the algorithm is given in Appendix C (page 214).

4.0.1 Related work

Since Huet's seminal work on higher-order unification for simply typed λ-calculus (Huet, 1975), many people have sought to extend it to dependently typed calculi, in particular the Edinburgh Logical Framework (Harper et al., 1993), also known as λ^{Π}-calculus. Elliott (1990) and Pym (1992) both demonstrated semidecision procedures for unification based on Huet's, using the fact that dependencies are erasable in the LF to give notions of 'type similarity' (in Pym's terminology) that relate the types of terms being unified. Brown (1996) studied the metatheory of a variant of λ^{Π}-calculus with type similarity, and used this to re-present unification as a system of reduction rules.

In contrast to Huet-style semidecision procedures, which generate a sequence of unifiers, Miller's pattern unification (Miller, 1992) finds most general unifiers when they exist, but applies only to a fragment. Duggan (1998) generalised the pattern condition to support System F_ω with simple product types. Reed (2009a) described how to apply dynamic pattern unification to LF. He introduced 'typing modulo' (discussed in Subsection 4.0.3) as a neat simplification of type similarity and similar invariants used to handle the complications of type dependency. Abel and Pientka (2011) extended Reed's algorithm to support $\lambda^{\Pi\Sigma}$-calculus (LF with Σ-types) and implemented it for the Beluga language.

Separately, higher-order unification algorithms have been developed for languages based on Martin-Löf Type Theory, such as Agda, or the Calculus of Constructions, such as Coq. Here, unlike in LF, *full-spectrum dependency* means that types may be recursively defined and computed from terms by *large elimination*. Thus term dependencies in types are not erasable to produce simple non-dependent types, and the work on unification for LF is not immediately applicable. Pfenning (1991b) extended pattern unification to the Calculus of Constructions, characterising exactly those terms that fall in the pattern fragment statically; hence the types can always be unified first.

This chapter builds on the work of Reed (2009a) and Abel and Pientka (2011) to describe unification for a full-spectrum dependent type theory, rather than LF.

4.0.2 Intensional vs. extensional equality

Definitional equality in an intensional type theory is the $\beta\delta\eta$-convertibility relation, written $s \equiv t$. For a strongly normalising theory, it is easy to test in a type-directed fashion, by checking that s and t have the same normal form (up to α-equivalence) after computation (β-reduction), expansion of definitions (δ-expansion) and η-expansion. It is intensional in the sense that extensionally equal terms need not be definitionally equal: for example, $s = \lambda x.\mathtt{tt}$ and $t = \lambda x.\mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathtt{tt}$ are equal on all boolean inputs, but $s \not\equiv t$.

Extensional type theories typically add a propositional equality type $\mathsf{Id}_T\, s\, t$ of proofs that s and t are equal, together with the equality reflection rule

$$\frac{\Gamma \vdash u : \mathsf{Id}_T\, s\, t}{\Gamma \vdash s \equiv t : T}$$

that embeds arbitrary proofs into the definitional equality. Extensional equality is undecidable in general: given a description of a Turing machine M, consider the function that maps a natural number n to the boolean indicating whether M halts within n steps. One cannot hope to decide whether this function is extensionally equal to the constantly false function!

The unification algorithm I will describe finds solutions up to the intensional definitional equality, not extensional equality. Finding solutions up to extensional equality involves proof search and most general solutions are not (intensionally) unique. For example, if $\alpha : \mathbb{B} \to \mathbb{B}$ is a metavariable and $x : \mathbb{B}$ is a variable, the problem $\alpha\, x \approx \mathtt{tt}$ has unique solution $\lambda x.\mathtt{tt}$ up to definitional equality, but solutions up to extensional equality include $\lambda x.\mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathtt{tt}$ and other terms.

Most type theories have some internal notion of *propositional equality* in which equations can be proved, such as the identity type in Martin-Löf Type Theory (Martin-Löf, 1984), which reflects the definitional equality as a type, or coercion types in System F_C (Sulzmann et al., 2007), where equality evidence is explicit but in a different syntactic category to terms. Given a type theory with a sufficiently expressive propositional equality, one could represent unification problems as types, and unification could deliver terms (equality proofs) as evidence. However, in this chapter I prefer to make fewer assumptions about the object type theory, emphasising that the work is more widely applicable.

4.0.3 Heterogeneous equality

Given the problem $\Pi x : A.\, B \approx \Pi x : S.\, T$, a reasonable step to take is to simplify it to $A \approx S, B \approx T$. However, at this stage B and T expect different types for x, as the equation between A and S may not be solved immediately. This shows the need for a heterogeneous notion of equality, in an intensional setting: it permits the expression of equations where the two sides belong to provably (extensionally) equal but not definitionally (intensionally) equal types. Such equations would be homogeneous in an extensional setting. In general, unification must formulate and solve equations between vectors of terms in a telescope, where unifying the first $n-1$ terms will make the types of the n^{th} terms equal. The unification algorithm will maintain the *heterogeneity invariant*, that every heterogeneous equation involves types whose equality is implied by preceding equations; thus solutions will always be homogeneous.

Reed (2009a) elegantly dealt with heterogeneity using a weaker invariant on homogeneous equations, *typing modulo*, which requires that the two sides be well typed up to the equational theory of the constraints yet to be solved. However, this means that if there are unsolved constraints left when the algorithm terminates, then some solved metavariables may be ill typed, up to the definitional equality. This is problematic for elaboration of a full-spectrum dependently typed source language, where typechecking is interleaved with unification, so unification must not create ill-typed terms. Norell (2007, Ch. 3) shows how ill-typed solutions to metavariables can lead to non-normalising terms and hence non-terminating elaboration. The algorithm I present avoids this difficulty by ensuring that all outputs are well typed, provided it is given well typed input.

Variables	x, y, z, X, Y, Z		
Metavariables	α, β, γ		
Terms	$s,\ t,\ S,\ T$	$::=$	$\underline{n} \mid \lambda x.t \mid c \mid \Pi x\!:\!S.\ T \mid \Sigma x\!:\!S.\ T \mid (s, t)$
Constructors	c	$::=$	$\mathbf{Set} \mid \mathbf{Type} \mid \mathbb{B} \mid \mathbf{tt} \mid \mathbf{ff}$
Heads	h	$::=$	$x \mid \acute{x} \mid \grave{x} \mid \alpha$
Evaluation contexts	e	$::=$	$\bullet \mid e\, t \mid e\, \textsc{hd} \mid e\, \textsc{tl} \mid \mathbf{if}_{(x.\,T)}\, e\ \ s\, t$
Neutral terms	n	$::=$	$h \cdot e$
Metacontexts	Θ	$::=$	$\cdot \mid \Theta, \alpha\!:T \mid \Theta, \alpha := t\!:T \mid \Theta, ?\, P$
Contexts	$\Gamma,\ \Delta$	$::=$	$\cdot \mid \Gamma, x\!:T \mid \Gamma, \hat{x}\!:\!S \ddagger T$
Substitutions	δ	$::=$	$\cdot \mid \delta, t/x \mid \delta, (s, t)/\hat{x}$
Metasubstitutions	$\theta,\ \zeta$	$::=$	$\cdot \mid \theta, t/\alpha$
Problems	$P,\ Q$	$::=$	$\top \mid \bot \mid P \wedge Q \mid (s\!:\!S) \approx (t\!:T) \mid \forall x\!:\!S.P \mid \forall \hat{x}\!:\!S \ddagger T.\, P$

Figure 4.1: Syntax

4.1 Back to basics

The type theory for which I will describe pattern unification essentially consists of Martin-Löf Type Theory with Π and Σ-types, a type of booleans $\mathbb{B}$ and one small universe **Set**. The only form of dependency is a type-level if-expression, allowing large elimination. It is based on Kipling, a theory described by McBride (2010a) with a model construction in the dependently typed language Agda.

In this section, I introduce the representations of terms and contexts, give the typing rules, discuss the use of 'twins' for representing variables with two provably equal types, explain the role of substitutions, and recall some standard metatheoretic properties. These concepts will be used in Section 4.2, where I specify the unification algorithm.

4.1.1 Term representation

The syntax of terms is given in Figure 4.1. Types and terms live in a single syntactic category, though I will typically write s, t, u or v for terms and S, T, U or V for types. A neutral (stuck) term n is represented as $h \cdot e$ where h is a head and e is an evaluation context, generalising the spine form of Cervesato and Pfenning (2003). This allows easy access to the head, which may be a variable x, y, z or a metavariable α, β. The accents on variables will be used to deal with heterogeneity, as discussed in Subsection 4.1.4. Evaluation contexts include applications, if-expressions and projections from Σ-types (written postfix $\textsc{hd}$ for first projection and $\textsc{tl}$ for second projection). Embedding neutral terms

$\boxed{s \cdot e \Downarrow t}$ *(redex $s \cdot e$ reduces to normal form t)*

$$\frac{}{t \cdot \bullet \Downarrow t} \qquad \frac{s \cdot e \Downarrow \lambda x.u \qquad [t/x]\, u \Downarrow v}{s \cdot (e\, t) \Downarrow v} \qquad \frac{s \cdot e \Downarrow (t_0, t_1)}{s \cdot (e\, \textsc{hd}) \Downarrow t_0} \qquad \frac{s \cdot e \Downarrow (t_0, t_1)}{s \cdot (e\, \textsc{tl}) \Downarrow t_1}$$

$$\frac{s \cdot e \Downarrow \mathbf{tt}}{s \cdot (\mathbf{if}_{(x.T)}\, e \;\; t_0\, t_1) \Downarrow t_0} \qquad \frac{s \cdot e \Downarrow \mathbf{ff}}{s \cdot (\mathbf{if}_{(x.T)}\, e \;\; t_0\, t_1) \Downarrow t_1} \qquad \frac{s \cdot e \Downarrow \underline{n}}{s \cdot (e \cdot e') \Downarrow \underline{n \cdot e'}}$$

$\boxed{\delta\, t \Downarrow t'}$ *(applying substitution δ to normal form t reduces to t')*

$$\frac{\delta\,(h) = s \qquad \delta\, e \Downarrow e' \qquad s \cdot e' \Downarrow t}{\delta\,(\underline{h \cdot e}) \Downarrow t} \qquad \frac{}{\delta\, c \Downarrow c} \qquad \frac{\delta\, t \Downarrow t'}{\delta\,(\lambda y.t) \Downarrow \lambda y.t'}$$

$$\frac{\delta\, S \Downarrow S' \qquad \delta\, T \Downarrow T'}{\delta\,(\Pi y\!:\!S.\ T) \Downarrow \Pi y\!:\!S'.\ T'} \qquad \frac{\delta\, S \Downarrow S' \qquad \delta\, T \Downarrow T'}{\delta\,(\Sigma y\!:\!S.\ T) \Downarrow \Sigma y\!:\!S'.\ T'} \qquad \frac{\delta\, t \Downarrow t' \qquad \delta\, u \Downarrow u'}{\delta\,(t, u) \Downarrow (t', u')}$$

$\boxed{\delta\, e \Downarrow e'}$ *(applying substitution δ to evaluation context e reduces to e')*

$$\frac{}{\delta\, \bullet \Downarrow \bullet} \qquad \frac{\delta\, e \Downarrow e' \qquad \delta\, t \Downarrow t'}{\delta\,(e\, t) \Downarrow e'\, t'} \qquad \frac{\delta\, e \Downarrow e'}{\delta\,(e\, \textsc{hd}) \Downarrow e'\, \textsc{hd}} \qquad \frac{\delta\, e \Downarrow e'}{\delta\,(e\, \textsc{tl}) \Downarrow e'\, \textsc{tl}}$$

$$\frac{\delta\, e \Downarrow e' \qquad \delta\, T \Downarrow T' \qquad \delta\, t \Downarrow t' \qquad \delta\, u \Downarrow u'}{\delta\,(\mathbf{if}_{(y.T)}\, e \;\; t\, u) \Downarrow \mathbf{if}_{(y.T')}\, e' \;\; t'\, u'}$$

$$\begin{array}{llll}
\delta\,(x) & \mapsto & t & \text{where } t/x \in \delta \\
\delta\,(\acute{x}) & \mapsto & s & \text{where } (s, t)/\hat{x} \in \delta \\
\delta\,(\grave{x}) & \mapsto & t & \text{where } (s, t)/\hat{x} \in \delta \\
\delta\,(\alpha) & \mapsto & \alpha & \\
 & & & \\
\theta\,(x) & \mapsto & x & \\
\theta\,(\alpha) & \mapsto & t & \text{where } t/\alpha \in \theta
\end{array}$$

Figure 4.2: Hereditary substitution

into normal forms is written $\underline{n}$, though the underline will sometimes be omitted. Evaluation contexts can be composed in the obvious way, written $e \cdot e'$.

In this representation, terms t are always β-normal but not necessarily η-long. This is possible thanks to hereditary substitution (Watkins et al., 2003), defined in Figure 4.2. Here $s \cdot e \Downarrow t$ means $s \cdot e$ reduces to t, while $\delta\, t \Downarrow t'$ or $\delta\, e \Downarrow e'$ means applying the substitution δ to the term t or evaluation context e results in the normal form t' or e' respectively. These reduction relations are all functional, and are decidable for well-typed inputs. Moreover, projecting from any term (even if it is ill typed), or applying any term to a variable, will terminate; I make use of this in the definitional equality rules for functions and pairs. In the rules, I sometimes write redexes in terms, where formally there should be additional premises referring to the reduction relations.

A *telescope* $\Delta = (\overline{x_i : T_i}^{\,i})$ is a vector of name bindings with corresponding types, where each type T_i may depend on the variables $x_0, \ldots, x_{i-1}$. The single binding notation $\Pi x : S.\, T$ or $\lambda x.t$ generalises in the obvious way to bind a telescope $\Pi\Delta.\, T$ or $\lambda\Delta.t$. Similarly $h\,\Delta$ is the application of the head h to the variables bound in Δ. The non-dependent Π and Σ, where x does not occur in the codomain T, are written $S \to T$ and $S \times T$ respectively.

4.1.2 Contexts and unification problems

In the style of contextual type theory (Nanevski et al., 2008), I separate the metacontext Θ, which contains metavariables and unification problems, from the context Γ, which binds variables. In terms of mixed quantifier prefixes, this amounts to maintaining an $\exists\forall$-prefix, a normalised representation of contexts in which the existential quantifiers (metavariables) appear before the universal quantifiers (variables). This avoids the need for Miller's explicit 'raising' step.

Unlike contextual type theory, however, I do not represent metavariable contexts explicitly: metavariables simply have Π-types. This identification of the object language function space with parametrisation in the metalanguage is convenient, when the object type theory is sufficiently expressive, but is not essential. In Chapter 7, where the type language lacks first-class higher-order functions, I will make use of parametrised metavariables instead.

A *context* Γ is a telescope that may also include a novel form of binding, to deal with heterogeneous hypotheses for unification problems (see Subsection 4.1.4). The set of variables bound by a context is written $\mathsf{vars}(\Gamma)$.

A *metacontext* Θ is a list of metavariables α, each carrying a type and possibly a definition, and unification problems P. Scope is managed according to the invariant that each entry depends only on those that precede it, and in terms, metavariables are explicitly applied to all the variables they may depend upon.

Unification problems include heterogeneous equations $(s : S) \approx (t : T)$, universally quantified variables, truth, falsehood and conjunctions. For brevity, I will sometimes omit the types in equations, writing $s \approx t$. In Subsection 4.0.3 I remarked on the need for the terms being unified to have different types.

For example,

$$\alpha : \mathbb{B} \to \mathbb{B}, \beta : \mathbb{B}, ?\, \forall x : \mathbb{B} \to \mathbb{B}.\, (\alpha\,(x\,\beta) : \mathbb{B}) \approx (x\,\beta : \mathbb{B})$$

is a valid metacontext, which declares metavariables α and β and has a single unification problem with parameter x.

A substitution δ or metasubstitution θ contains terms with which to replace variables or metavariables from a context or metacontext. The identity (meta)substitution is written ι, and a substitution written as a finite map (such as $[s/x]$) implicitly acts as the identity on all other variables. I will sometimes write $t\{s\}$ instead of $[s/x]\,t$ where the choice of free variable x is obvious, or to represent a term that includes s as a subterm. Typing rules for (meta)substitutions are given in Subsection 4.1.5.

4.1.3 Typing rules

The typing rules are given in the following figures. They define judgments for well-formed metacontexts, contexts and problems; for definitionally equal $\beta\delta$-normal terms; and for true propositions of the unification logic. In the usual bidirectional style (Pierce and Turner, 2000), the definitional equality judgment is split in two: there is one judgment for normal terms, where a type is given as input, and one for neutral terms, where a type is produced as output. In the interest of brevity, definitional equality is treated as a partial equivalence relation, with the typing judgment being the diagonal of equality. Crucially, the definitional equality, and hence typechecking, is decidable[3] using standard techniques (Coquand, 1996; Chapman et al., 2005; Löh et al., 2010). Appendix C.3 (page 221) gives a Haskell implementation of the typechecking algorithm.

In particular, the theory is well-founded because there is only one universe **Set** : **Type**, and **Type** itself is not a well-typed term. (Formally, T : **Type** should be considered a separate judgment to $t : T$.)

[3]Strictly speaking, it is not possible to decide well-formedness because it depends on the truth of propositions in the unification logic, which is not decidable. In practice, this does not matter, because I can always assume that the algorithms are given well-formed inputs.

The judgments $\Theta \vdash \mathbf{mctx}$, $\Theta \,|\, \Gamma \vdash \mathbf{ctx}$ and $\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}$, defined in Figure 4.3, mean respectively that Θ, Γ and P are well-formed. Contexts and metacontexts must bind distinct variables, as $x\#\Gamma$ means x is fresh for Γ and $\alpha\#\Theta$ means α is fresh for Θ. Note that well-formed problems are required to satisfy the heterogeneity invariant: for $(s : S) \approx (t : T)$ to be well-formed, $(S : \mathbf{Type}) \approx (T : \mathbf{Type})$ must be true in the unification logic.

The judgment $\Theta \,|\, \Gamma \vdash T \ni s \mathrel{\equiv\!\!\![} u \mathrel{]\!\!\!\equiv} t$, defined in Figure 4.4, means that s and t are definitionally equal terms checked at type T, with η-long standard form u (regarded as an output). This ternary presentation of equality is novel, to my knowledge. It is often useful to pick a canonical representative when working up to an equivalence; for example, it makes the admissibility of symmetry easy to prove. As in the work on Kipling by McBride (2010a), this judgment really expresses the fact that s and t are equivalent syntactic presentations of u.

The definitional equality includes type-directed rules that compare functions by applying them to a fresh variable, and compare pairs by computing their projections, thereby covering both the η-laws and congruence for functions and pairs. A type is *atomic* if it is not a Π- or Σ-type: this is used in the change of direction rule to ensure that the equality judgment is syntax-directed, as otherwise it would overlap with the rules for functions and pairs.

The judgment $\Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\!\!\![} h'' \cdot e'' \mathrel{]\!\!\!\equiv} h' \cdot e' \in T$, defined in Figure 4.5, means that the neutral terms $h \cdot e$ and $h' \cdot e'$ are definitionally equal with inferred type T and standard form $h'' \cdot e''$. Note that there is no rule for inferring the type of a defined metavariable $\alpha := t : T$; rather, definitions must be immediately substituted out, which simplifies the presentation of the algorithm.

The judgment $\Theta \,|\, \Gamma \vdash P$, defined in Figures 4.6 and 4.7, means that P is *true*, i.e. it follows from hypotheses in the metacontext. This defines a *unification logic* in the sense of Pfenning (1991a), where the separation of the metacontext from the context amounts to keeping all existential quantifiers outermost. In terms of the analysis by Martin-Löf (1996), this judgment says that P 'is true', whereas the judgment $\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}$ says that P 'is a proposition'.

I will sometimes omit the standard form, writing $\Theta \,|\, \Gamma \vdash T \ni s \equiv t$ instead of $\Theta \,|\, \Gamma \vdash T \ni s \mathrel{\equiv\!\!\![} u \mathrel{]\!\!\!\equiv} t$. The typing judgment $\Theta \,|\, \Gamma \vdash T \ni t$ is defined as $\Theta \,|\, \Gamma \vdash T \ni t \equiv t$, meaning the equivalence relation is reflexive on well-typed terms by definition. Similarly, I will sometimes write $\Theta \,|\, \Gamma \vdash h \cdot e \in T$ instead of $\Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\!\!\![} h' \cdot e' \mathrel{]\!\!\!\equiv} h \cdot e \in T$.

$\boxed{\Theta \vdash \mathbf{mctx}}$ *(Θ is a valid metacontext)*

$$\frac{}{\cdot \vdash \mathbf{mctx}} \qquad \frac{\Theta \,|\, \cdot \vdash P\,\mathbf{wf}}{\Theta, ?\,P \vdash \mathbf{mctx}} \qquad \frac{\alpha\#\Theta \quad \Theta \,|\, \cdot \vdash \mathbf{Type} \ni T}{\Theta, \alpha : T \vdash \mathbf{mctx}} \qquad \frac{\alpha\#\Theta \qquad \Theta \,|\, \cdot \vdash T \ni t}{\Theta, \alpha := t : T \vdash \mathbf{mctx}}$$

$\boxed{\Theta \,|\, \Gamma \vdash \mathbf{ctx}}$ *(Γ is a valid context in metacontext Θ)*

$$\frac{\Theta \vdash \mathbf{mctx}}{\Theta \,|\, \cdot \vdash \mathbf{ctx}} \qquad \frac{x\#\Gamma \quad \Theta \,|\, \Gamma \vdash \mathbf{Type} \ni T}{\Theta \,|\, \Gamma, x : T \vdash \mathbf{ctx}} \qquad \frac{x\#\Gamma \quad \Theta \,|\, \Gamma \vdash (S : \mathbf{Type}) \approx (T : \mathbf{Type})}{\Theta \,|\, \Gamma, \hat{x} : S \ddagger T \vdash \mathbf{ctx}}$$

$\boxed{\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}}$ *(P is a well-formed problem in Θ and Γ)*

$$\frac{\Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash \top\,\mathbf{wf}} \qquad \frac{\Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash \bot\,\mathbf{wf}} \qquad \frac{\Theta \,|\, \Gamma \vdash P\,\mathbf{wf} \quad \Theta, ?\forall\Gamma.\, P \,|\, \Gamma \vdash Q\,\mathbf{wf}}{\Theta \,|\, \Gamma \vdash P \wedge Q\,\mathbf{wf}}$$

$$\frac{\Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash (\mathbf{Set} : \mathbf{Type}) \approx (\mathbf{Set} : \mathbf{Type})\,\mathbf{wf}} \qquad \frac{\Theta \,|\, \Gamma \vdash S \ni s \qquad \Theta \,|\, \Gamma \vdash T \ni t \quad \Theta \,|\, \Gamma \vdash (S : \mathbf{Type}) \approx (T : \mathbf{Type})}{\Theta \,|\, \Gamma \vdash (s : S) \approx (t : T)\,\mathbf{wf}}$$

$$\frac{\Theta \,|\, \Gamma, x : S \vdash P\,\mathbf{wf}}{\Theta \,|\, \Gamma \vdash \forall x : S.\, P\,\mathbf{wf}} \qquad \frac{\Theta \,|\, \Gamma, \hat{x} : S \ddagger T \vdash P\,\mathbf{wf}}{\Theta \,|\, \Gamma \vdash \forall \hat{x} : S \ddagger T.\, P\,\mathbf{wf}}$$

Figure 4.3: Well-formed contexts

$\boxed{\Theta \mid \Gamma \vdash T \ni s \mathrel{\equiv\!\![}\, u \,\mathrel{]\!\!\models} t}$ *(type T accepts s equal to t with standard form u)*

$$\frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \mathbf{Type} \ni \mathbf{Set} \mathrel{\equiv\!\![}\, \mathbf{Set} \,\mathrel{]\!\!\models} \mathbf{Set}} \qquad \frac{\Theta \mid \Gamma \vdash \mathbf{Set} \ni S \mathrel{\equiv\!\![}\, U \,\mathrel{]\!\!\models} T}{\Theta \mid \Gamma \vdash \mathbf{Type} \ni S \mathrel{\equiv\!\![}\, U \,\mathrel{]\!\!\models} T}$$

$$\frac{\Theta \mid \Gamma \vdash \mathbf{Set} \ni S_0 \mathrel{\equiv\!\![}\, U \,\mathrel{]\!\!\models} S_1 \qquad \Theta \mid \Gamma, x\colon U \vdash \mathbf{Set} \ni T_0 \mathrel{\equiv\!\![}\, V \,\mathrel{]\!\!\models} T_1}{\Theta \mid \Gamma \vdash \mathbf{Set} \ni \Pi x\colon S_0.\ T_0 \mathrel{\equiv\!\![}\, \Pi x\colon U.\ V \,\mathrel{]\!\!\models} \Pi x\colon S_1.\ T_1}$$

$$\frac{s\,x \Downarrow s' \qquad t\,x \Downarrow t' \qquad \Theta \mid \Gamma, x\colon U \vdash V \ni s' \mathrel{\equiv\!\![}\, u \,\mathrel{]\!\!\models} t'}{\Theta \mid \Gamma \vdash \Pi x\colon U.\ V \ni s \mathrel{\equiv\!\![}\, \lambda x.u \,\mathrel{]\!\!\models} t}$$

$$\frac{\Theta \mid \Gamma \vdash \mathbf{Set} \ni S_0 \mathrel{\equiv\!\![}\, U \,\mathrel{]\!\!\models} S_1 \qquad \Theta \mid \Gamma, x\colon U \vdash \mathbf{Set} \ni T_0 \mathrel{\equiv\!\![}\, V \,\mathrel{]\!\!\models} T_1}{\Theta \mid \Gamma \vdash \mathbf{Set} \ni \Sigma x\colon S_0.\ T_0 \mathrel{\equiv\!\![}\, \Sigma x\colon U.\ V \,\mathrel{]\!\!\models} \Sigma x\colon S_1.\ T_1}$$

$$\frac{\begin{array}{l} s\ \textsc{hd} \Downarrow s_0 \qquad s\ \textsc{tl} \Downarrow s_1 \\ t\ \textsc{hd} \Downarrow t_0 \qquad t\ \textsc{tl} \Downarrow t_1 \\ \Theta \mid \Gamma \vdash U \ni s_0 \mathrel{\equiv\!\![}\, u_0 \,\mathrel{]\!\!\models} t_0 \\ \Theta \mid \Gamma \vdash V\{u_0\} \ni s_1 \mathrel{\equiv\!\![}\, u_1 \,\mathrel{]\!\!\models} t_1 \end{array}}{\Theta \mid \Gamma \vdash \Sigma x\colon U.\ V \ni s \mathrel{\equiv\!\![}\, (u_0, u_1) \,\mathrel{]\!\!\models} t} \qquad \frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \mathbf{Set} \ni \mathbb{B} \mathrel{\equiv\!\![}\, \mathbb{B} \,\mathrel{]\!\!\models} \mathbb{B}}$$

$$\frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \mathbb{B} \ni \mathbf{tt} \mathrel{\equiv\!\![}\, \mathbf{tt} \,\mathrel{]\!\!\models} \mathbf{tt}} \qquad \frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \mathbb{B} \ni \mathbf{ff} \mathrel{\equiv\!\![}\, \mathbf{ff} \,\mathrel{]\!\!\models} \mathbf{ff}}$$

$$\frac{\begin{array}{l} S \text{ atomic} \\ \Theta \mid \Gamma \vdash h \cdot e \mathrel{\equiv\!\![}\, h'' \cdot e'' \,\mathrel{]\!\!\models} h' \cdot e' \in T \qquad \Theta \mid \Gamma \vdash \mathbf{Type} \ni S \mathrel{\equiv\!\![}\, U \,\mathrel{]\!\!\models} T \end{array}}{\Theta \mid \Gamma \vdash S \ni \underline{h \cdot e} \mathrel{\equiv\!\![}\, \underline{h'' \cdot e''} \,\mathrel{]\!\!\models} \underline{h' \cdot e'}}$$

Figure 4.4: Definitional equality: normal terms

$\boxed{\Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\mkern-6mu[} h'' \cdot e'' \mathrel{]\mkern-6mu\equiv} h' \cdot e' \in T}$ *($h \cdot e$ equals $h' \cdot e'$ with inferred type T)*

$$\frac{\Theta \ni \alpha : T \qquad \Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash \alpha \cdot \bullet \mathrel{\equiv\mkern-6mu[} \alpha \cdot \bullet \mathrel{]\mkern-6mu\equiv} \alpha \cdot \bullet \in T} \qquad \frac{\Gamma \ni x : T \qquad \Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash x \cdot \bullet \mathrel{\equiv\mkern-6mu[} x \cdot \bullet \mathrel{]\mkern-6mu\equiv} x \cdot \bullet \in T}$$

$$\frac{\Gamma \ni \hat{x} : S \ddagger T \qquad \Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash \acute{x} \cdot \bullet \mathrel{\equiv\mkern-6mu[} \acute{x} \cdot \bullet \mathrel{]\mkern-6mu\equiv} \acute{x} \cdot \bullet \in S} \qquad \frac{\Gamma \ni \hat{x} : S \ddagger T \qquad \Theta \,|\, \Gamma \vdash \mathbf{ctx}}{\Theta \,|\, \Gamma \vdash \grave{x} \cdot \bullet \mathrel{\equiv\mkern-6mu[} \grave{x} \cdot \bullet \mathrel{]\mkern-6mu\equiv} \grave{x} \cdot \bullet \in T}$$

$$\frac{\Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\mkern-6mu[} h'' \cdot e'' \mathrel{]\mkern-6mu\equiv} h' \cdot e' \in \Pi x : U.\, V \qquad \Theta \,|\, \Gamma \vdash U \ni u \mathrel{\equiv\mkern-6mu[} u'' \mathrel{]\mkern-6mu\equiv} u'}{\Theta \,|\, \Gamma \vdash h \cdot e\, u \mathrel{\equiv\mkern-6mu[} h'' \cdot e''\, u'' \mathrel{]\mkern-6mu\equiv} h' \cdot e'\, u' \in V\{u''\}}$$

$$\frac{\Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\mkern-6mu[} h'' \cdot e'' \mathrel{]\mkern-6mu\equiv} h' \cdot e' \in \Sigma x : U.\, V}{\Theta \,|\, \Gamma \vdash h \cdot e\, \textsc{hd} \mathrel{\equiv\mkern-6mu[} h \cdot e''\, \textsc{hd} \mathrel{]\mkern-6mu\equiv} h \cdot e'\, \textsc{hd} \in U}$$

$$\frac{\Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\mkern-6mu[} h'' \cdot e'' \mathrel{]\mkern-6mu\equiv} h' \cdot e' \in \Sigma x : U.\, V}{\Theta \,|\, \Gamma \vdash h \cdot e\, \textsc{tl} \mathrel{\equiv\mkern-6mu[} h'' \cdot e''\, \textsc{tl} \mathrel{]\mkern-6mu\equiv} h' \cdot e'\, \textsc{tl} \in V\{h'' \cdot e''\, \textsc{hd}\}}$$

$$\frac{\begin{array}{cc} \Theta \,|\, \Gamma \vdash h \cdot e \mathrel{\equiv\mkern-6mu[} h'' \cdot e'' \mathrel{]\mkern-6mu\equiv} h' \cdot e' \in \mathbb{B} & \Theta \,|\, \Gamma, x {:} \mathbb{B} \vdash \mathbf{Type} \ni S \mathrel{\equiv\mkern-6mu[} U \mathrel{]\mkern-6mu\equiv} T \\ \Theta \,|\, \Gamma \vdash U\{\mathbf{tt}\} \ni u \mathrel{\equiv\mkern-6mu[} u'' \mathrel{]\mkern-6mu\equiv} u' & \Theta \,|\, \Gamma \vdash U\{\mathbf{ff}\} \ni v \mathrel{\equiv\mkern-6mu[} v'' \mathrel{]\mkern-6mu\equiv} v' \end{array}}{\Theta \,|\, \Gamma \vdash h \cdot \mathbf{if}_{(x.S)}\, e\;\; u\, v \mathrel{\equiv\mkern-6mu[} h'' \cdot \mathbf{if}_{(x.U)}\, e\;\; u''\, v'' \mathrel{]\mkern-6mu\equiv} h' \cdot \mathbf{if}_{(x.T)}\, e'\;\; u'\, v' \in U\{h'' \cdot e''\}}$$

Figure 4.5: Definitional equality: neutral terms

$\boxed{\Theta \mid \Gamma \vdash P}$ *(P is true in the unification logic)*

$$\frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \top} \qquad \frac{\Theta \mid \Gamma \vdash \bot \qquad \Theta \mid \Gamma \vdash P\ \mathbf{wf}}{\Theta \mid \Gamma \vdash P} \qquad \frac{\Theta \mid \Gamma, x : S \vdash P}{\Theta \mid \Gamma \vdash \forall x : S.\, P}$$

$$\frac{\begin{array}{l}\Theta \mid \Gamma \vdash (S : \mathbf{Type}) \approx (T : \mathbf{Type}) \\ \Theta \mid \Gamma, \hat{x} : S \ddagger T \vdash P\end{array}}{\Theta \mid \Gamma \vdash \forall \hat{x} : S \ddagger T.\, P} \qquad \frac{\begin{array}{l}\Theta \mid \Gamma \vdash \mathbf{Type} \ni S \equiv\!\![\ U\ ]\!\!\equiv T \\ \Theta \mid \Gamma, x : U \vdash P\{x, x\}\end{array}}{\Theta \mid \Gamma \vdash \forall \hat{x} : S \ddagger T.\, P}$$

$$\frac{\begin{array}{l}\Theta \mid \Gamma \vdash \forall x : S.\, P \\ \Theta \mid \Gamma \vdash S \ni s\end{array}}{\Theta \mid \Gamma \vdash P\{s\}} \qquad \frac{\begin{array}{l}\Theta \mid \Gamma \vdash \forall \hat{x} : S \ddagger T.\, P \\ \Theta \mid \Gamma \vdash \mathbf{Type} \ni S \equiv\!\![\ U\ ]\!\!\equiv T \\ \Theta \mid \Gamma \vdash U \ni u\end{array}}{\Theta \mid \Gamma \vdash P\{u, u\}} \qquad \frac{\begin{array}{l}\Theta \ni ?\, P \\ \Theta \mid \Gamma \vdash \mathbf{ctx}\end{array}}{\Theta \mid \Gamma \vdash P}$$

$$\frac{\Theta \mid \Gamma \vdash P \qquad \Theta \mid \Gamma \vdash Q}{\Theta \mid \Gamma \vdash P \wedge Q} \qquad \frac{\Theta \mid \Gamma \vdash P \wedge Q}{\Theta \mid \Gamma \vdash P} \qquad \frac{\Theta \mid \Gamma \vdash P \wedge Q}{\Theta \mid \Gamma \vdash Q}$$

$$\frac{\Theta \mid \Gamma \vdash \Pi x : A.\, B \approx \Pi x : S.\, T}{\Theta \mid \Gamma \vdash A \approx S \wedge \forall \hat{x} : A \ddagger S.\, B\{\acute{x}\} \approx T\{\grave{x}\}}$$

$$\frac{\Theta \mid \Gamma \vdash \Sigma x : A.\, B \approx \Sigma x : S.\, T}{\Theta \mid \Gamma \vdash A \approx S \wedge \forall \hat{x} : A \ddagger S.\, B\{\acute{x}\} \approx T\{\grave{x}\}} \qquad \frac{\begin{array}{l}\Theta \mid \Gamma \vdash \mathbf{Type} \ni S \equiv\!\![\ U\ ]\!\!\equiv T \\ \Theta \mid \Gamma \vdash U \ni s \equiv t\end{array}}{\Theta \mid \Gamma \vdash (s : S) \approx (t : T)}$$

$$\frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash (\mathbf{Set} : \mathbf{Type}) \approx (\mathbf{Set} : \mathbf{Type})} \qquad \frac{\Theta \mid \Gamma \vdash (t : T) \approx (s : S)}{\Theta \mid \Gamma \vdash (s : S) \approx (t : T)}$$

$$\frac{\begin{array}{l}\Theta \mid \Gamma \vdash (t_0 : T_0) \approx (t_1 : T_1) \\ \Theta \mid \Gamma \vdash (t_1 : T_1) \approx (t_2 : T_2)\end{array}}{\Theta \mid \Gamma \vdash (t_0 : T_0) \approx (t_2 : T_2)} \qquad \frac{\Gamma \ni \hat{x} : S \ddagger T \qquad \Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash (\acute{x} : S) \approx (\grave{x} : T)}$$

Figure 4.6: Unification logic

$\boxed{\Theta \,|\, \Gamma \vdash P}$

$$\frac{\Theta \,|\, \Gamma \vdash (S\!:\!\mathbf{Set}) \approx (U\!:\!\mathbf{Set}) \qquad \Theta \,|\, \Gamma, \hat{x}\!:\!S{\ddagger}U \vdash (T\{\acute{x}\}\!:\!\mathbf{Set}) \approx (V\{\grave{x}\}\!:\!\mathbf{Set})}{\Theta \,|\, \Gamma \vdash (\Pi x\!:\!S.\ T\!:\!\mathbf{Set}) \approx (\Pi x\!:\!U.\ V\!:\!\mathbf{Set})}$$

$$\frac{\Theta \,|\, \Gamma \vdash (S\!:\!\mathbf{Set}) \approx (U\!:\!\mathbf{Set}) \qquad \Theta \,|\, \Gamma, \hat{x}\!:\!S{\ddagger}U \vdash (T\{\acute{x}\}\!:\!\mathbf{Set}) \approx (V\{\grave{x}\}\!:\!\mathbf{Set})}{\Theta \,|\, \Gamma \vdash (\Sigma x\!:\!S.\ T\!:\!\mathbf{Set}) \approx (\Sigma x\!:\!U.\ V\!:\!\mathbf{Set})}$$

$$\frac{\Theta \,|\, \Gamma, \hat{x}\!:\!S{\ddagger}U \vdash (s\,\acute{x}\!:\!T\{\acute{x}\}) \approx (t\,\grave{x}\!:\!V\{\grave{x}\})}{\Theta \,|\, \Gamma \vdash (s\!:\!\Pi x\!:\!S.\ T) \approx (t\!:\!\Pi x\!:\!U.\ V)}$$

$$\frac{\Theta \,|\, \Gamma \vdash (s\,{\scriptstyle\mathrm{HD}}\!:\!S) \approx (t\,{\scriptstyle\mathrm{HD}}\!:\!U) \qquad \Theta \,|\, \Gamma \vdash (s\,{\scriptstyle\mathrm{TL}}\!:\!T\{s\,{\scriptstyle\mathrm{HD}}\}) \approx (t\,{\scriptstyle\mathrm{TL}}\!:\!V\{t\,{\scriptstyle\mathrm{HD}}\})}{\Theta \,|\, \Gamma \vdash (s\!:\!\Sigma x\!:\!S.\ T) \approx (t\!:\!\Sigma x\!:\!U.\ V)}$$

$$\frac{\Theta \,|\, \Gamma \vdash (\underline{n}\!:\!\Pi x\!:\!S.\ T) \approx (\underline{n'}\!:\!\Pi x\!:\!U.\ V) \qquad \Theta \,|\, \Gamma \vdash (s\!:\!S) \approx (t\!:\!U)}{\Theta \,|\, \Gamma \vdash (\underline{n\,s}\!:\!T\{s\}) \approx (\underline{n'\,t}\!:\!V\{t\})}$$

$$\frac{\Theta \,|\, \Gamma \vdash (\underline{n}\!:\!\Sigma x\!:\!S.\ T) \approx (\underline{n'}\!:\!\Sigma x\!:\!U.\ V)}{\Theta \,|\, \Gamma \vdash (\underline{n\,{\scriptstyle\mathrm{HD}}}\!:\!S) \approx (\underline{n'\,{\scriptstyle\mathrm{HD}}}\!:\!U)}$$

$$\frac{\Theta \,|\, \Gamma \vdash (\underline{n}\!:\!\Sigma x\!:\!S.\ T) \approx (\underline{n'}\!:\!\Sigma x\!:\!U.\ V)}{\Theta \,|\, \Gamma \vdash (\underline{n\,{\scriptstyle\mathrm{TL}}}\!:\!T\{\underline{n\,{\scriptstyle\mathrm{HD}}}\}) \approx (\underline{n'\,{\scriptstyle\mathrm{TL}}}\!:\!V\{\underline{n'\,{\scriptstyle\mathrm{HD}}}\})}$$

$$\frac{\begin{array}{c}\Theta \,|\, \Gamma, x\!:\!\mathbb{B} \vdash (T\!:\!\mathbf{Type}) \approx (T'\!:\!\mathbf{Type}) \qquad \Theta \,|\, \Gamma \vdash (\underline{n}\!:\!\mathbb{B}) \approx (\underline{n'}\!:\!\mathbb{B}) \\ \Theta \,|\, \Gamma \vdash (t_0\!:\!T\{\mathbf{tt}\}) \approx (t'_0\!:\!T'\{\mathbf{tt}\}) \qquad \Theta \,|\, \Gamma \vdash (t_1\!:\!T\{\mathbf{ff}\}) \approx (t'_1\!:\!T'\{\mathbf{ff}\})\end{array}}{\Theta \,|\, \Gamma \vdash (\underline{\mathbf{if}_{(x.T)}\,n\ \ t_0\,t_1}\!:\!T\{s\}) \approx (\underline{\mathbf{if}_{(x.T')}\,n'\ \ t'_0\,t'_1}\!:\!T'\{s'\})}$$

Figure 4.7: Unification logic: congruence rules

4.1.4 Twins

Unification will require the incremental simplification of unification problems. In a heterogeneous setting, an immediate question is how to simplify the problem

$$(s : \Pi x : S.\ T) \approx (t : \Pi x : U.\ V),$$

since it would not be type-correct (absent typing modulo) to produce

$$\forall x : S.\,(s\,x : T) \approx (t\,x : V).$$

We need $x : S$ on the left and $x : U$ on the right, and we need to know that they are the same x. This motivates the introduction of *twin variables*, allowing the problem to be simplified to

$$\forall \hat{x} : S \ddagger U.\,(s\,\acute{x} : T\{\acute{x}\}) \approx (t\,\grave{x} : V\{\grave{x}\})$$

where $\acute{x}$ and $\grave{x}$ represent the same variable at two different types, bound by $\hat{x} : S \ddagger U$. The heterogeneity invariant (Subsection 4.0.3) means that S and U will be constrained to be equal by problems in the metacontext, but have not yet necessarily been unified. If the types become definitionally equal, the twins can be replaced with a single variable. On the other hand, the fact that they are different might not prevent the problem from being solved (if at least one of s and t is a constant function, for example).

Twins bind a single name, but occurrences of the variable mark which twin they refer to. Thus they can be distinguished when typechecking, and substitution must replace them with a pair of terms that are provably equal. Of course, twins are bound as parameters of unification problems, not in terms, so β-reduction never substitutes for twins. I write $x \sim y$ if x and y are identical or twins.

If unification problems were represented as types, twins could be distinct variables with a proof of their (propositional) equality; replacing them with a single variable would exploit the elimination principle for propositional equality.

Definitional equality is tested in the algorithm when typechecking a candidate solution for a metavariable, but it treats twins as distinct, so the presence of twins may prevent a metavariable being instantiated with a purported solution, as indeed it should. When calculating the free variables of a term, the twin annotations are ignored, so $\mathsf{fv}(\acute{x}) = \mathsf{fv}(\grave{x}) = \mathsf{fv}(x) = \{x\}$.

$\boxed{\Theta \mid \Gamma \vdash \delta : \Delta}$ *(δ is a substitution from Δ to Γ)*

$$\frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \cdot : \cdot} \qquad \frac{\Theta \mid \Gamma \vdash \delta : \Delta \quad \Theta \mid \Gamma \vdash \delta\, T \ni t}{\Theta \mid \Gamma \vdash (\delta, t/x) : \Delta, x : T} \qquad \frac{\Theta \mid \Gamma \vdash \delta : \Delta \quad \Theta \mid \Gamma \vdash (s : \delta\, S) \approx (t : \delta\, T)}{\Theta \mid \Gamma \vdash (\delta, (s,t)/\hat{x}) : \Delta, \hat{x} : S \ddagger T}$$

$\boxed{\theta : \Theta \sqsubseteq \Theta'}$ *(θ is a metasubstitution from Θ to Θ')*

$$\frac{\Theta' \vdash \mathbf{mctx}}{\cdot : \cdot \sqsubseteq \Theta'} \qquad \frac{\theta : \Theta \sqsubseteq \Theta' \quad \Theta' \mid \cdot \vdash \theta\, T \ni t \equiv \theta\, s}{(\theta, t/\alpha) : \Theta, \alpha := s : T \sqsubseteq \Theta'}$$

$$\frac{\theta : \Theta \sqsubseteq \Theta' \quad \Theta' \mid \cdot \vdash \theta\, T \ni t}{(\theta, t/\alpha) : \Theta, \alpha : T \sqsubseteq \Theta'} \qquad \frac{\theta : \Theta \sqsubseteq \Theta' \quad \Theta' \mid \cdot \vdash \theta\, P}{\theta : \Theta, ?\, P \sqsubseteq \Theta'}$$

Figure 4.8: Typing rules for substitutions and metasubstitutions

4.1.5 Substitutions and metasubstitutions

Figure 4.8 defines well-typed substitutions δ and metasubstitutions θ. They are applied to terms as defined in Figure 4.2, and extended homomorphically to syntax containing terms in the usual way.

The judgment $\Theta \mid \Gamma \vdash \delta : \Delta$ means that δ substitutes a well-typed term in Γ for every variable in Δ. Note that two provably equal terms may be substituted for twins, since twins are not required to be definitionally equal.

The judgment $\theta : \Theta \sqsubseteq \Theta'$ means that θ substitutes a well-typed term in Θ' for every metavariable in Θ. Moreover, any problem hypothesised in the original metacontext must be true somehow in the new metacontext. This allows metasubstitutions to be lifted to apply on derivations, as shown by Lemma 4.2 below. Thus they give rise to an appropriate notion of stability, as in Subsection 2.1.2 (page 14). Two metasubstitutions are equivalent if they assign definitionally equal terms to each metavariable, as defined in Figure 4.9.

The identity substitution ι on Δ includes x/x for each $x : T$ and $(\acute{x}, \grave{x})/\hat{x}$ for each $\hat{x} : S \ddagger T$ in Δ. Weakening is silent, so $\Theta \mid \Gamma \vdash \iota : \Delta$ holds whenever Γ binds all the variables bound in Δ.

I will also use $\iota : \Theta \sqsubseteq \Theta'$ for metacontexts, to represent an identity or inclusion metasubstitution. If Θ' contains definitions for some of the metavariables in Θ then these definitions will be expanded by ι, to maintain the invariant that well-typed terms are always $\beta\delta$-normal.

$\boxed{\theta \equiv \theta' \!:\! \Theta \sqsubseteq \Theta'}$ *(θ and θ' are equivalent metasubstitutions from Θ to Θ')*

$$\frac{\Theta' \vdash \mathbf{mctx}}{\cdot \equiv \cdot \!:\! \cdot \sqsubseteq \Theta'} \qquad \frac{\begin{array}{c}\theta \equiv \theta' \!:\! \Theta \sqsubseteq \Theta' \\ \Theta' \,|\, \cdot \vdash \theta\, T \ni t \equiv t' \qquad \Theta' \,|\, \cdot \vdash \theta\, T \ni t' \equiv \theta\, s\end{array}}{(\theta, t/\alpha) \equiv (\theta', t'/\alpha) \!:\! \Theta, \alpha := s \!:\! T \sqsubseteq \Theta'}$$

$$\frac{\begin{array}{c}\theta \equiv \theta' \!:\! \Theta \sqsubseteq \Theta' \\ \Theta' \,|\, \cdot \vdash \theta\, T \ni t \equiv t'\end{array}}{(\theta, t/\alpha) \equiv (\theta', t'/\alpha) \!:\! \Theta, \alpha \!:\! T \sqsubseteq \Theta'} \qquad \frac{\begin{array}{c}\theta \equiv \theta' \!:\! \Theta \sqsubseteq \Theta' \\ \Theta' \,|\, \cdot \vdash \theta\, P\end{array}}{\theta \equiv \theta' \!:\! \Theta, ?\, P \sqsubseteq \Theta'}$$

Figure 4.9: Equivalence of metasubstitutions

4.1.6 Properties

All the usual metatheoretic properties hold. Where proofs have been omitted, they are by structural induction on derivations.

Lemma 4.1 (Substitution)**.** *Suppose* $\Theta \,|\, \Gamma \vdash \delta \!:\! \Delta$. *Then*

(a) If $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash \mathbf{ctx}$ *then* $\Theta \,|\, \Gamma, \delta\, \Gamma' \vdash \mathbf{ctx}$.

(b) If $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash P\, \mathbf{wf}$ *then* $\Theta \,|\, \Gamma, \delta\, \Gamma' \vdash \delta\, P\, \mathbf{wf}$.

(c) If $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash T \ni s \equiv\![\, u \,]\!\equiv t$ *then* $\Theta \,|\, \Gamma, \delta\, \Gamma' \vdash \delta\, T \ni \delta\, s \equiv\![\, v \,]\!\equiv \delta\, t$.

(d) If $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash h_0 \cdot e_0 \equiv\![\, h_2 \cdot e_2 \,]\!\equiv h_1 \cdot e_1 \in T$ *then*
$\Theta \,|\, \Gamma, \delta\, \Gamma' \vdash \delta\, T \ni \delta\, (h_0 \cdot e_0) \equiv\![\, u \,]\!\equiv \delta\, (h_1 \cdot e_1)$.

(e) If $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash P$ *then* $\Theta \,|\, \Gamma, \delta\, \Gamma' \vdash \delta\, P$.

(f) If $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash \delta' \!:\! \Gamma''$ *then* $\Theta \,|\, \Gamma, \delta\, \Gamma' \vdash \delta \cdot \delta' \!:\! \delta\, \Gamma''$.

Lemma 4.2 (Metasubstitution)**.** *Suppose* $\theta \!:\! \Theta \sqsubseteq \Theta'$.

(a) If $\Theta \,|\, \Gamma \vdash \mathbf{ctx}$ *then* $\Theta' \,|\, \theta\Gamma \vdash \mathbf{ctx}$.

(b) If $\Theta \,|\, \Gamma \vdash P\, \mathbf{wf}$ *then* $\Theta' \,|\, \theta\Gamma \vdash \theta\, P\, \mathbf{wf}$.

(c) If $\Theta \,|\, \Gamma \vdash T \ni s \equiv\![\, u \,]\!\equiv t$ *then* $\Theta' \,|\, \theta\Gamma \vdash \theta\, T \ni \theta\, s \equiv\![\, v \,]\!\equiv \theta\, t$.

(d) If $\Theta \,|\, \Gamma \vdash h \cdot e \equiv\![\, h'' \cdot e'' \,]\!\equiv h' \cdot e' \in T$ *then* $\Theta' \,|\, \theta\Gamma \vdash \theta\, T \ni \theta\, (h \cdot e) \equiv \theta\, (h' \cdot e')$.

(e) If $\Theta \,|\, \Gamma \vdash P$ *then* $\Theta' \,|\, \theta\Gamma \vdash \theta\, P$.

(f) If $\Theta \,|\, \Gamma \vdash \delta \!:\! \Delta$ *then* $\Theta' \,|\, \theta\Gamma \vdash \theta\, \delta \!:\! \theta\Delta$.

Lemma 4.3 (Sanity conditions)**.**
(a) If $\Theta \,|\, \Gamma \vdash \mathbf{ctx}$ *then* $\Theta \vdash \mathbf{mctx}$*.*

(b) If $\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}$ *then* $\Theta \,|\, \Gamma \vdash \mathbf{ctx}$*.*

(c) If $\Theta \,|\, \Gamma \vdash T \ni s \equiv\!\![\, u \,]\!\!\equiv t$ *then* $\Theta \,|\, \Gamma \vdash \mathbf{ctx}$*.*

(d) If $\Theta \,|\, \Gamma \vdash h \cdot e \equiv\!\![\, h'' \cdot e'' \,]\!\!\equiv h' \cdot e' \in T$ *then* $\Theta \,|\, \Gamma \vdash \mathbf{Type} \ni T$*.*

(e) If $\Theta \,|\, \Gamma \vdash P$ *then* $\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}$*.*

Lemma 4.4 (Definitional equality is an equivalence relation)**.**
(a) If $\Theta \,|\, \Gamma \vdash T \ni t$ *then* $\Theta \,|\, \Gamma \vdash T \ni t \equiv t$*.*

(b) If $\Theta \,|\, \Gamma \vdash T \ni s \equiv\!\![\, v \,]\!\!\equiv t$ *then* $\Theta \,|\, \Gamma \vdash T \ni t \equiv\!\![\, v \,]\!\!\equiv s$*.*

(c) If $\Theta \,|\, \Gamma \vdash T \ni t \equiv\!\![\, u \,]\!\!\equiv t'$ *and* $\Theta \,|\, \Gamma \vdash T \ni t' \equiv\!\![\, v \,]\!\!\equiv t''$ *then* $u = v$ *and* $\Theta \,|\, \Gamma \vdash T \ni t \equiv\!\![\, v \,]\!\!\equiv t''$*.*

Proof. Reflexivity, part (a), is precisely the definition of the typing judgment.

Symmetry, part (b), is by structural induction on the derivation. Since the standard form is preserved, it is easy to establish symmetry, because the rules use the standard form rather than choosing one side arbitrarily (and asymmetrically).

Transitivity, part (c), is by structural induction on the first derivation and inversion on the second. The rules are syntax-directed, so in each case, the last rule of the second derivation must be the same as the last rule of the first. □

Lemma 4.5 (Context conversion)**.** *Suppose* $\Theta \,|\, \Gamma \vdash \mathbf{Type} \ni S \equiv T$*. Then*

(a) $\Theta \,|\, \Gamma, x{:}S, \Delta \vdash \mathbf{ctx}$ *implies* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash \mathbf{ctx}$*;*

(b) $\Theta \,|\, \Gamma, x{:}S, \Gamma' \vdash P\,\mathbf{wf}$ *implies* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash P\,\mathbf{wf}$*;*

(c) $\Theta \,|\, \Gamma, x{:}S, \Gamma' \vdash U \ni s \equiv\!\![\, u \,]\!\!\equiv t$ *implies* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash U \ni s \equiv\!\![\, u \,]\!\!\equiv t$*;*

(d) $\Theta \,|\, \Gamma, x{:}S, \Gamma' \vdash h \cdot e \equiv\!\![\, h'' \cdot e'' \,]\!\!\equiv h' \cdot e' \in U$ *implies there is some* V *such that* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash h \cdot e \equiv\!\![\, h'' \cdot e'' \,]\!\!\equiv h' \cdot e' \in V$ *and* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash \mathbf{Type} \ni U \equiv V$*;*

(e) $\Theta \,|\, \Gamma, x{:}S, \Gamma' \vdash P$ *implies* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash P$*.*

(f) $\Theta \,|\, \Gamma, x{:}S, \Gamma' \vdash \delta{:}\Delta$ *implies* $\Theta \,|\, \Gamma, x{:}\,T, \Gamma' \vdash \delta{:}\Delta$*.*

A similar result applies to twins.

Lemma 4.6 (Conversion)**.** *If* $\Theta \,|\, \Gamma \vdash \mathbf{Type} \ni S \equiv T$ *and* $\Theta \,|\, \Gamma \vdash S \ni s \equiv\!\![\, u \,]\!\!\equiv t$ *then* $\Theta \,|\, \Gamma \vdash T \ni s \equiv\!\![\, u \,]\!\!\equiv t$*.*

4.2 Specification of unification

Having shown how to represent unification problems in context, let me address the question of how to solve them. Following the approach of the previous chapters, the idea is always to make small, local changes to the metacontext, each of which is type-correct, makes the problem simpler and makes no unforced intensional choices. This ensures that any solution found is most general.

In the following subsections, I will examine the range of problems one might encounter, and discuss the step to take in each case. Then I will summarise all the steps of the algorithm. The steps are divided into five main groups:

- solving equations of the form $\alpha\,\overline{x_i}^{\,i} \approx t$ by $\alpha := \lambda\,\overline{x_i}^{\,i}\,.t$ (Subsection 4.2.1);
- solving equations $\alpha\,\overline{x_i}^{\,i} \approx \alpha\,\overline{y_i}^{\,i}$ by limiting the domain of α (Subsection 4.2.2);
- gaining information via pruning (Subsection 4.2.3);
- simplifying metavariables by removing Σ-types (Subsection 4.2.4); and
- simplifying problems locally (Subsection 4.2.5).

The rules are not deterministic, as they permit working on problems in any order, but the nondeterminism does not matter: every step is most general, so the order will not affect the final result. A deterministic algorithm can be obtained from the rules by choosing a suitable order (such as leftmost problem first).

Since definitions must be immediately substituted out, in order to keep everything δ-normal, I write $\Theta, \alpha :=^{*} t\colon T, \Xi$ to represent $\Theta, \alpha := t\colon T, [t/\alpha]\Xi$.

4.2.1 Solving problems by inversion

Given the metacontext

$$\Theta, \alpha\colon T \to T, ?\,\forall x\colon T.\, \alpha\, x \approx x,$$

where the equation looks like a definition, it should be unsurprising that

$$\Theta, \alpha := \lambda x.x\colon T \to T, ?\,\forall x\colon T.\, x \approx x$$

is a most general solution. Miller (1992) observed that, in general, the problem $\forall\Gamma.\, \alpha\,\overline{x_i}^{\,i} \approx t$ has unique solution $\alpha := \lambda\,\overline{x_i}^{\,i}\,.t$ provided that the evaluation context of α is a list of distinct variables containing all the free variables of t, and α does not occur in t.

On the other hand, an equation like $\alpha\,\mathbf{tt} \approx t$ is not a good definition for α: taking $\alpha := \lambda x.t$ is a solution but is not most general, because another equation might require $\alpha\,\mathbf{ff} \approx s$ for some $s \neq t$. Defining α by case analysis is not most general as it makes an unforced intensional choice: a later equation might demand $\alpha \approx \lambda x.\mathbf{ff}$. Intuitively, Miller's *pattern condition* says that only an application to variables 'captures the whole nature' of the metavariable; an application to non-variables only determines it for those specific arguments.

Linearity

It is crucial that variables occurring in t appear linearly (exactly once) in $\overline{x_i}^{\,i}$. The equation $\beta\,x\,x \approx x$ cannot be solved immediately, as β could project either its first or second argument, so there is no unique most general solution. On the other hand, $\gamma\,y\,x\,y \approx x$ can be solved unambiguously by $\gamma := \lambda y_0\,x\,y_1.x$ despite the repetition of y. The $\overline{x_i}^{\,i}$ may include twins, which are treated as equal for the purposes of this check.

Occurs check

If α occurs in t, then it is obviously unsound to use t as the definition for α. However, the question of whether the problem can have a solution *at all* is more subtle, and depends on the exact form of the occurrence.

A subterm occurs *flexibly* if it is in the evaluation context of a metavariable, and *rigidly* if not. In the term $\alpha\,x \to y\,z$, α, y and z occur rigidly while x occurs flexibly. Miller (1992, p. 26) describes rigid occurrences as 'permanent' and flexible occurrences as 'possible', because flexible occurrences might be removed by substituting for metavariables but rigid occurrences cannot. A rigid occurrence is *strong* if it is not in the evaluation context of a variable, so no substitution for variables can remove it. In the example, y occurs strong rigidly but z does not.

I write $\mathsf{fmv}(t)$ for the set of free metavariables and $\mathsf{fv}(t)$ for the set of free variables of t. Either may have a $\cdot^{\mathsf{rig}}$ or $\cdot^{\mathsf{srig}}$ superscript to include only those that occur rigidly or strong rigidly (respectively).

Reed (2009b, §5.1.5) observes that when performing the occurs check before solving a metavariable, a problem is definitely unsolvable if

- the metavariable occurs strong rigidly in its own candidate solution, such as in $\alpha\,x \approx \alpha\,\mathbf{tt} \to \alpha\,\mathbf{ff}$; or
- an application of the metavariable *to variables* occurs rigidly in its own candidate solution, such as in $\alpha\,x \approx x\,(\alpha\,x)$.

If a weak rigid occurrence of a metavariable is applied to non-variables, the problem may have solutions, for example $\beta\, y \approx y\,(\beta\,(\lambda x.x))$ is solvable (by taking $\beta := \lambda y.y\,\mathbf{tt}$, amongst other things). Again, Miller's pattern condition appears: only an application to variables determines the whole nature of a metavariable.

Permuting the metacontext

If t depends on some metavariables declared after α, these must be moved prior to α for the definition to be well-scoped. However, other metavariables may depend on α, so they must remain after it. For example, given the context

$$\Theta, \alpha : \mathbf{Set}, \beta : \mathbf{Set}, \gamma : \alpha, ?\,\alpha \approx \beta \to \beta$$

an appropriate solution is

$$\Theta, \beta : \mathbf{Set}, \alpha := \beta \to \beta : \mathbf{Set}, \gamma : \beta \to \beta.$$

In general, solving

$$\Theta, \alpha : T, \Xi, ?\,\forall \Gamma.\, \alpha\, \overline{x_i}^{\,i} \approx t$$

requires finding a dependency-respecting permutation of Ξ into two segments Ξ_0 and Ξ_1 (written $\Xi \cong \Xi_0, \Xi_1$), where Ξ_0 contains all the metavariables that occur in t and its type, and does not depend on α. If the necessary permutation does not exist, then α cannot be solved immediately, though solving other metavariables may remove the dependency cycle. The existence of such a permutation can be determined in a small-step fashion by scanning dependencies from right to left, as in the instantiation judgment for first-order unification (Figure 2.5, page 21).

Typechecking

Once the algorithm has a candidate solution $\lambda\, \overline{x_i}^{\,i}.t$ for α, it must check that the solution is well typed, as heterogeneity means that this is not guaranteed. In particular, the type of t might not be definitionally equal to the type of $\alpha\, \overline{x_i}^{\,i}$, or if some twin variable $\acute{y}$ occurs in $\overline{x_i}^{\,i}$ and $\grave{y}$ occurs in t, then the solution will not be valid until the types of $\acute{y}$ and $\grave{y}$ become definitionally equal. Strictly speaking it is not necessary to fully recheck the solution: it is enough to test these conditions directly and rely on the fact that the original problem was well-typed. A real implementation would record the desired solution for α and the constraints that must be solved before it can be applied, as in Agda (Norell, 2007, Ch. 3).

$$
\begin{array}{lcl}
\mathsf{intersect}\ \cdot\ \cdots & \mapsto & \cdot \\
\mathsf{intersect}\,(\Delta, z\!:\!S)\,(\overline{x_i}^{\,i}, x)\,(\overline{y_i}^{\,i}, y) & \mapsto & \begin{cases} \mathsf{intersect}\,\Delta\,\overline{x_i}^{\,i}\,\overline{y_i}^{\,i}, z\!:\!S & \text{if } x \sim y \\ \mathsf{intersect}\,\Delta\,\overline{x_i}^{\,i}\,\overline{y_i}^{\,i} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4.10: Intersection

4.2.2 Solving flex-flex problems by intersection

As well as equations between an eliminated metavariable and an arbitrary term, some equations have the form $\alpha \cdot e \approx \alpha \cdot e'$, with the same metavariable on both sides but different evaluation contexts. If both contexts are applications of lists of variables, then a most general solution is given by restricting α to those arguments on which the two lists of variables agree. For example, a solution of

$$\Theta, \alpha\!: T \rightarrow T \rightarrow T, ?\,\forall x\!: T.\,\forall y\!: T.\,\alpha\,x\,x \approx \alpha\,y\,x$$

is possible only if α does not depend on its first argument, giving

$$\Theta, \beta\!: T \rightarrow T, \alpha := \lambda\_.\beta\!: T \rightarrow T \rightarrow T, ?\,\forall x\!: T.\,(\beta\,x\!: T) \approx (\beta\,x\!: T)$$

where β is a fresh metavariable.

Figure 4.10 defines the operation $\mathsf{intersect}\,\Delta\,\overline{x_i}^{\,i}\,\overline{y_i}^{\,i}$, which takes a telescope Δ and two lists of variables to fit it, and produces the telescope on which they agree. Twin variables are considered equal for the purposes of intersection, though in any case, twins could be replaced with a single variable since they must share a common type. Given the context

$$\Theta, \alpha\!:\!\Pi\Delta.\ T, \Xi, ?\,\forall\Gamma.\,\alpha\,\overline{x_i}^{\,i} \approx \alpha\,\overline{y_i}^{\,i}$$

the problem is solved by creating a fresh metavariable β and defining

$$\Theta, \beta\!:\!\Pi\Delta'.\ T, \alpha :=^{*} \lambda\Delta.\beta\,\Delta'\!:\!\Pi\Delta.\ T, \Xi \quad \text{where } \Delta' = \mathsf{intersect}\,\Delta\,\overline{x_i}^{\,i}\,\overline{y_i}^{\,i}$$

provided the free variables of the codomain T are retained in the telescope Δ'.

In LF, one can define intersection for arbitrary argument lists that contain no metavariables, but this is not possible in a type theory with large elimination. For example, $\alpha\,\mathbf{tt}\,x \approx \alpha\,\mathbf{tt}\,y$ does not imply that α is independent of its second argument, as it might be defined by case analysis on its first argument.

4.2.3 Pruning

The problem in

$$\Theta, \alpha : (T \to T) \to T, ?\, \forall x : (T \to T).\, \forall y : T.\, \alpha\, x \approx x\, y$$

is unsolvable, because there is no way for α to depend on y, since it does not occur as an argument on the left-hand side. On the other hand,

$$\Theta, \beta : T \to T, \alpha : (T \to T) \to T, ?\, \forall x : (T \to T).\, \forall y : T.\, \alpha\, x \approx x\, (\beta\, y)$$

can be solved by observing that β may not depend on its argument, so it must be of the form $\lambda \_ .\gamma$ for some fresh metavariable γ. This gives

$$\Theta, \gamma : T, \beta := \lambda \_ .\gamma : T \to T, \alpha : (T \to T) \to T, ?\, \forall x : (T \to T).\, \forall y : T.\, \alpha\, x \approx x\, \gamma$$

which can be solved by

$$\Theta, \gamma : T, \beta := \lambda \_ .\gamma : T \to T, \alpha := \lambda x.x\, \gamma : (T \to T) \to T.$$

For a problem of the form $\forall \Gamma.\, \alpha \cdot e \approx t$ to be solvable, all the free variables of t must occur in e; otherwise, they will be out of scope for solutions of α. If any out-of-scope variables occur rigidly in t, then the equation can never be solved. If an out-of-scope variable occurs flexibly, in the evaluation context of a metavariable, then it might be possible to remove the occurrence by *pruning* the metavariable, restricting its telescope of arguments.

Pruning cannot always remove occurrences of out-of-scope variables. For example, pruning the equation $\forall x : T.\, \alpha \approx \beta\, (\gamma\, x)$ fails because it is not clear which metavariable ignores its argument: either β or γ could be constant, so there is no most general solution. In this situation, the unification algorithm will have to tackle other constraints, which may result in the problem becoming easier.

Moreover, knowing $\beta\, \mathbf{tt}\, x$ cannot depend on x does not mean that β cannot depend on its second argument, because it might be defined by case analysis on the first argument (so removing other arguments might lose solutions). Pruning therefore retains arguments only if they are variables, failing otherwise. Once again, Miller's pattern condition appears: a constraint captures the entire behaviour of a metavariable only if the metavariable is applied to a list of variables.

$\boxed{\mathsf{pruneTm}\,\mathcal{V}\,t \;\mapsto\; (\beta, \Delta)}$ *(pruning t to $\mathcal{V}$ requires β to have telescope Δ)*

$$\frac{\begin{array}{ll}\Theta \ni \beta : \Pi\Delta.\, T & \mathsf{prune}\,\mathcal{V}\,\Delta\,\overline{t_i}^{i} \mapsto \Delta' \\ \mathsf{fv}(T) \subset \mathsf{vars}(\Delta') & \Delta \neq \Delta'\end{array}}{\mathsf{pruneTm}\,\mathcal{V}\,(\underline{\beta\,\overline{t_i}^{i}}) \;\mapsto\; (\beta, \Delta')} \qquad \frac{\mathsf{pruneTm}\,\mathcal{V}\,S \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(\Pi x : S.\, T) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,(\mathcal{V} \cup \{x\})\,T \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(\Pi x : S.\, T) \;\mapsto\; (\beta, \Delta)} \qquad \frac{\mathsf{pruneTm}\,\mathcal{V}\,S \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(\Sigma x : S.\, T) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,(\mathcal{V} \cup \{x\})\,T \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(\Sigma x : S.\, T) \;\mapsto\; (\beta, \Delta)} \qquad \frac{\mathsf{pruneTm}\,\mathcal{V}\,s \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(s, t) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,\mathcal{V}\,t \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(s, t) \;\mapsto\; (\beta, \Delta)} \qquad \frac{\mathsf{pruneTm}\,(\mathcal{V} \cup \{x\})\,t \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(\lambda x.t) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,\mathcal{V}\,s \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(x \cdot (e\,s \cdot e')) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,(\mathcal{V} \cup \{y\})\,T \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(x \cdot (\mathbf{if}_{(y.T)}\,e\;\;s\,t \cdot e')) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,\mathcal{V}\,s \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(x \cdot (\mathbf{if}_{(y.T)}\,e\;\;s\,t \cdot e')) \;\mapsto\; (\beta, \Delta)}$$

$$\frac{\mathsf{pruneTm}\,\mathcal{V}\,t \;\mapsto\; (\beta, \Delta)}{\mathsf{pruneTm}\,\mathcal{V}\,(x \cdot (\mathbf{if}_{(y.T)}\,e\;\;s\,t \cdot e')) \;\mapsto\; (\beta, \Delta)}$$

$\boxed{\mathsf{prune}\,\mathcal{V}\,\Delta\,\overline{t_i}^{i} \mapsto \Delta'}$ *(pruning arguments $\overline{t_i}^{i}$ in Δ to $\mathcal{V}$ gives telescope Δ')*

$$\frac{}{\mathsf{prune}\,\mathcal{V}\;\cdot\;\;\cdot \mapsto \cdot} \qquad \frac{\mathsf{prune}\,\mathcal{V}\,\Delta\,\overline{t_i}^{i} \mapsto \Delta' \qquad y \in \mathcal{V} \qquad \mathsf{fv}(S) \subset \mathsf{vars}(\Delta')}{\mathsf{prune}\,\mathcal{V}\,(\Delta, x : S)\,(\overline{t_i}^{i}, y) \mapsto \Delta', x : S}$$

$$\frac{\mathsf{prune}\,\mathcal{V}\,\Delta\,\overline{t_i}^{i} \mapsto \Delta' \qquad \mathsf{fv}^{\mathsf{rig}}(s) \not\subset \mathcal{V}}{\mathsf{prune}\,\mathcal{V}\,(\Delta, x : S)\,(\overline{t_i}^{i}, s) \mapsto \Delta'}$$

Figure 4.11: Pruning

Pruning uses two auxiliary relations defined in Figure 4.11. Both depend on a set $\mathcal{V}$ of variables that may occur in arguments, which will initially be $\mathsf{fv}(e)$ and will accumulate locally bound variables.

- The relation $\mathsf{pruneTm}\,\mathcal{V}\,t \mapsto (\beta, \Delta')$ means that t has an occurrence of β, whose telescope has been pruned to Δ'. This works by searching t for a subterm $\beta\,\overline{t_i}^{\,i}$ then using the following function.
- The relation $\mathsf{prune}\,\mathcal{V}\,\Delta\,\overline{t_i}^{\,i} \mapsto \Delta'$ computes the pruned telescope Δ' for β, where Δ is its original telescope and $\overline{t_i}^{\,i}$ is the list of its arguments.

These relations are partial, as pruning may fail, and the former is nondeterministic, as there may be multiple ways to prune a term. The nondeterminism does not matter, however, as pruning is always a most general step and can be applied repeatedly if necessary.

To prune a telescope $\Delta, x : S$ corresponding to the list of arguments $\overline{t_i}^{\,i}, s$, the preceding telescope Δ is pruned with the list of arguments $\overline{t_i}^{\,i}$. If this succeeds, producing Δ', then there are three possible cases:

- if s is a variable $y \in \mathcal{V}$, whose type depends only on variables that remain in the pruned telescope Δ', then the binding $x : S$ can be left in the telescope;
- if s has a rigid occurrence of a variable not in $\mathcal{V}$, then the binding must be removed from the telescope;
- otherwise, pruning fails.

If s has a flexible occurrence of a variable not in $\mathcal{V}$, pruning fails because while the whole term cannot depend on the variable, it is not clear which metavariable projects it away, as in the $\alpha \approx \beta\,(\gamma\,x)$ example.

Note that the potential presence of type dependencies means pruning must check the well-formedness of types. For example, if $\beta : \Pi x : S.\ T$ where x occurs free in T, then the first argument of β cannot be pruned.

For the earlier example

$$\Theta, \beta : T \to T, \alpha : (T \to T) \to T, ?\forall x : (T \to T). \forall y : T.\, \alpha\, x \approx x\,(\beta\, y)$$

we have $\mathsf{pruneTm}\,\{x\}\,(x\,(\beta\,y)) \mapsto (\beta, \cdot\,)$, because y does not occur in the set of allowed variables $\{x\}$, so $\mathsf{prune}\,\{x\}\,(z : T)\,y \mapsto (\cdot)$, i.e. the telescope $z : T$ of β is pruned to the empty telescope.

In the metacontext

$$\Theta, \beta : \Pi\Delta.\ T, \Xi, ?\forall\Gamma.\ \alpha \cdot e \approx t,$$

if $\mathsf{pruneTm}\,(\mathsf{fv}(e))\ t \mapsto (\beta, \Delta')$ and all the variables in T are retained in Δ' then pruning β results in the metacontext

$$\Theta, \gamma : \Pi\Delta'.\ T, \beta :=^{*} \lambda\Delta.\gamma\,\Delta' : \Pi\Delta.\ T, \Xi, ?\forall\Gamma.\ \alpha \cdot e \approx t$$

where γ is a fresh variable. This restricts the telescope in a similar way to intersection, though it does not apply to α but a different metavariable.

4.2.4 Metavariable simplification

Suppose $\alpha : \Sigma x : S.\ T$; how might the constraint $\alpha\ \textsc{hd} \approx s$ be solved? One option is to extend the pattern fragment to cover projections, as Duggan (1998) does for System F_ω, but I take the simpler option of aggressively lowering metavariables to eliminate projections. In this case, replacing α with the pair (β_0, β_1) of fresh metavariables $\beta_0 : S, \beta_1 : T\{\beta_0\}$ simplifies the constraint to $\beta_0 \approx s$.

In general, the metavariable α might be under a telescope of parameters, so $\alpha : \Pi\Delta.\ \Sigma x : S.\ T$ can be replaced with

$$\alpha_0 : \Pi\Delta.\ S, \alpha_1 : \Pi\Delta.\ T\{\alpha_0\,\Delta\}, \alpha := \lambda\Delta.(\alpha_0\,\Delta, \alpha_1\,\Delta).$$

Similarly, a metavariable $\alpha : \Pi x : (\Sigma y : S.\ T).\ U$ can be uncurried to produce $\beta : \Pi y : S.\ \Pi z : T.\ [(y, z)/x]\ U$, which will transform the non-pattern constraint $\alpha\,(y, z) \approx t$ into the pattern $\alpha\,y\,z \approx t$. The general case is even worse here, as α might have a telescope of parameters and the type of x might have parameters preceding the Σ. Thus $\alpha : \Pi\Delta.\ \Pi x : (\Pi\Delta'.\ \Sigma z : S.\ T).\ U$ can be replaced with

$$\begin{gathered}\Theta, \beta : \Pi\Delta.\ \Pi y : (\Pi\Delta'.\ S).\ \Pi z : (\Pi\Delta'.\ T\{y\,\Delta'\}).\ U\{\lambda\Delta'.(y\,\Delta', z\,\Delta')\},\\ \alpha := \lambda\Delta.\lambda x.\beta\,\Delta\,(\lambda\Delta'.x\,\Delta'\ \textsc{hd})\,(\lambda\Delta'.x\,\Delta'\ \textsc{tl}).\end{gathered}$$

These transformations maintain the same set of solutions thanks to the η-rule for Σ-types, otherwise known as surjective pairing, $(n\ \textsc{hd}, n\ \textsc{tl}) \equiv_\eta n$. This is built into the definitional equality by the rule for pairs, which always η-expands the terms being compared.

4.2.5 Problem simplification

The problem decomposition operation $P \Mapsto Q$ locally replaces a problem with a simpler problem without changing the rest of the metacontext. Each decomposition step can be applied in an arbitrary context. Thus $P \Mapsto Q$ means that $\Theta, ?\,\forall\Gamma.\, P$ can be replaced with $\Theta, ?\,\forall\Gamma.\, Q$. Additionally, conjunctions can be split into their components, replacing $\Theta, ?\,\forall\Gamma.\, P \wedge Q$ by $\Theta, ?\,\forall\Gamma.\, P, ?\,\forall\Gamma.\, Q$, and trivial problems can be removed, replacing $\Theta, ?\,\top$ with Θ. First I will discuss the decomposition steps, then later summarise them in Figure 4.14. Steps are numbered for ease of reference.

Perhaps the most basic simplification step is the removal of equations that are reflexive up to the definitional equality, and hence trivial:

$$(s\,{:}\,S) \approx (t\,{:}\,T) \quad \Mapsto \quad \top \qquad \text{if } \Theta \mid \Gamma \vdash \mathbf{Type} \ni S \mathrel{\equiv\!\!\![} U \mathrel{]\!\!\!\equiv} T \text{ and } \Theta \mid \Gamma \vdash U \ni s \equiv t \tag{4.1}$$

η-expansion

Given an equation between two functions, we saw in Subsection 4.1.4 that both sides can be η-expanded, even if the domains are not definitionally equal, by introducing twin variables. Thus $\alpha \approx \lambda x.t$ becomes $\alpha\, \acute{x} \approx t\{\grave{x}\}$. Similarly, pairs can be η-expanded, for example turning $(\alpha, \beta) \approx s$ into $\alpha \approx s\,\textsc{hd}$ and $\beta \approx s\,\textsc{tl}$.

$$(f\,{:}\,\Pi x\,{:}\,S.\ T) \approx (g\,{:}\,\Pi x\,{:}\,U.\ V) \quad \Mapsto \quad \forall \hat{x}\,{:}\,S \ddagger U.\, (f\,\acute{x}\,{:}\, T\{\acute{x}\}) \approx (g\,\grave{x}\,{:}\, V\{\grave{x}\}) \tag{4.2}$$

$$(s\,{:}\,\Sigma x\,{:}\,S.\ T) \approx (t\,{:}\,\Sigma x\,{:}\,U.\ V) \quad \Mapsto \quad (s\,\textsc{hd}\,{:}\,S) \approx (t\,\textsc{hd}\,{:}\,U) \wedge (s\,\textsc{tl}\,{:}\,T\{s\,\textsc{hd}\}) \approx (t\,\textsc{tl}\,{:}\,V\{t\,\textsc{hd}\}) \tag{4.3}$$

Rigid-rigid decomposition

A rigid-rigid equation is one where neither side is a metavariable in an evaluation context, so either the same head symbol appears on both sides, or the equation is unsolvable. For example, $\Pi x : S.\, T \approx \Pi x : U.\, V$ can be decomposed into $S \approx U \wedge T \approx V$, though twins must be used because S and U might not be definitionally equal. A similar decomposition applies to Σ-types.

$$\Pi x\,{:}\,S.\ T \approx \Pi x\,{:}\,U.\ V \quad \Mapsto \quad S \approx U \wedge \forall \hat{x}\,{:}\,S \ddagger U.\ T\{\acute{x}\} \approx V\{\grave{x}\} \tag{4.4}$$

$$\Sigma x\,{:}\,S.\ T \approx \Sigma x\,{:}\,U.\ V \quad \Mapsto \quad S \approx U \wedge \forall \hat{x}\,{:}\,S \ddagger U.\ T\{\acute{x}\} \approx V\{\grave{x}\} \tag{4.5}$$

If the equation is between two eliminated variables, $x \cdot e \approx x' \cdot e'$, it can be decomposed into equations between the arguments contained in the evaluation

$$\begin{array}{rcl}
x \cdot \bullet \bowtie x' \cdot \bullet & \mapsto & \top \text{ if } x \sim x' \\
x \cdot (e\, s) \bowtie x' \cdot (e'\, t) & \mapsto & x \cdot e \bowtie x' \cdot e' \wedge s \approx t \\
x \cdot (e\,\textsc{hd}) \bowtie x' \cdot (e'\,\textsc{hd}) & \mapsto & x \cdot e \bowtie x' \cdot e' \\
x \cdot (e\,\textsc{tl}) \bowtie x' \cdot (e'\,\textsc{tl}) & \mapsto & x \cdot e \bowtie x' \cdot e' \\
x \cdot (\mathbf{if}_{(y.T)}\, e\ \ s\, t) \bowtie x' \cdot (\mathbf{if}_{(y.T')}\, e'\ \ s'\, t') & \mapsto & x \cdot e \bowtie x' \cdot e' \wedge (\forall y{:}\mathbb{B}.\ T \approx T') \\
& & \quad \wedge\ s \approx s' \wedge t \approx t'
\end{array}$$

Figure 4.12: Evaluation context decomposition

$\boxed{s \perp\!\!\!\perp t}$ *(s and t are rigidly incompatible)*

$$\frac{}{\Pi x{:}S.\ T \perp\!\!\!\perp \Sigma y{:}U.\ V} \qquad \frac{}{\Pi x{:}S.\ T \perp\!\!\!\perp c} \qquad \frac{}{\Sigma x{:}S.\ T \perp\!\!\!\perp c} \qquad \frac{c \neq c'}{c \perp\!\!\!\perp c'}$$

$$\frac{}{x \cdot e \perp\!\!\!\perp \Pi y{:}S.\ T} \qquad \frac{}{x \cdot e \perp\!\!\!\perp \Sigma y{:}S.\ T} \qquad \frac{}{x \cdot e \perp\!\!\!\perp c} \qquad \frac{x \not\sim x'}{x \cdot \bullet \perp\!\!\!\perp x' \cdot \bullet}$$

$$\frac{}{x \cdot \bullet \perp\!\!\!\perp x' \cdot e\, s} \qquad \frac{}{x \cdot \bullet \perp\!\!\!\perp x' \cdot e\,\textsc{hd}} \qquad \frac{}{x \cdot \bullet \perp\!\!\!\perp x' \cdot e\,\textsc{tl}} \qquad \frac{}{x \cdot \bullet \perp\!\!\!\perp \mathbf{if}_{(y.T)}\, x' \cdot e\ \ s\, t}$$

$$\frac{}{x \cdot e\, s \perp\!\!\!\perp x' \cdot e\,\textsc{hd}} \qquad \frac{}{x \cdot e\, s \perp\!\!\!\perp x' \cdot e\,\textsc{tl}} \qquad \frac{}{x \cdot e\, s \perp\!\!\!\perp \mathbf{if}_{(y.T)}\, x' \cdot e\ \ s\, t}$$

$$\frac{}{x \cdot e\,\textsc{hd} \perp\!\!\!\perp x' \cdot e\,\textsc{tl}} \qquad \frac{}{x \cdot e\,\textsc{hd} \perp\!\!\!\perp \mathbf{if}_{(y.T)}\, x' \cdot e\ \ s\, t} \qquad \frac{}{x \cdot e\,\textsc{tl} \perp\!\!\!\perp \mathbf{if}_{(y.T)}\, x' \cdot e\ \ s\, t}$$

$$\frac{x \cdot e_0 \perp\!\!\!\perp x' \cdot e'_0}{x \cdot e_0 \cdot e_1 \perp\!\!\!\perp x' \cdot e'_0 \cdot e'_1} \qquad \frac{s \perp\!\!\!\perp t}{t \perp\!\!\!\perp s}$$

Figure 4.13: Impossible constraints

contexts, provided they match. For example, the problem

$$\forall \hat{x} : (S \to U \times U) \ddagger (T \to V \times V).\, (\acute{x}\, s\, \textsc{hd} : U) \approx (\grave{x}\, t\, \textsc{hd} : V)$$

decomposes into the equation $(s : S) \approx (t : T)$. On the other hand, $y\, \textsc{hd} \approx y\, \textsc{tl}$ has no solutions, because the projections do not match.

The evaluation context decomposition function $x \cdot e \bowtie x' \cdot e'$, which is defined in Figure 4.12, computes the conjunction of problems required to make $x \cdot e \approx x' \cdot e'$. It is made available via the step

$$x \cdot e \approx x' \cdot e' \qquad \Mapsto \qquad x \cdot e \bowtie x' \cdot e' \tag{4.6}$$

The outermost eliminator in the evaluation context is decomposed first, with the equality of the variables (ignoring twin annotations) being checked last, to allow for extension to handle proof-irrelevant types.[4]

The evaluation context decomposition function is partial because a mismatched equation like $x \approx y$ for distinct x and y, or $y\, \textsc{hd} \approx y\, \textsc{tl}$, has no solutions. Similarly, equations between dissimilar canonical constructors (such as $\mathbf{tt} \approx \mathbf{ff}$) are unsolvable. To capture this, Figure 4.13 defines the relation $s \perp\!\!\!\perp t$, meaning that s and t are rigidly incompatible, so $s \approx t$ can never be solved. The step

$$s \approx t \qquad \Mapsto \qquad \bot \text{ if } s \perp\!\!\!\perp t \tag{4.7}$$

allows $\bot$ to be derived from such a contradiction. This definition depends on the fact that equations are being solved up to the intensional definitional equality: $(x : S) \approx (y : S)$ can be solved up to extensionality if S has only one inhabitant.[5]

η-contraction of subterms

Miller's pattern condition requires that a metavariable should be applied to a list of variables. As the definitional equality includes η-conversion, however, it is enough for the arguments to be η-contractible to variables. For example, $\alpha\, (\lambda x.y\, x) \approx t$ can be η-contracted to $\alpha\, y \approx t$, potentially allowing the solution $\alpha := \lambda y.t$. This motivates the steps

$$P\{\lambda x.n\, x\} \qquad \Mapsto \qquad P\{n\} \tag{4.8}$$

$$P\{(n\, \textsc{hd}, n\, \textsc{tl})\} \qquad \Mapsto \qquad P\{n\} \tag{4.9}$$

that permit η-contraction anywhere inside problems. In practice, these are useful only to make steps that depend on the pattern condition apply, so an implementation would perform η-contraction only when testing the pattern condition.

[4]Eliminations of an empty type can be equal even if the eliminated terms are not equal.

[5]Also, given proof-irrelevant types, the definition of $s \perp\!\!\!\perp t$ would need to check that the types were not proof-irrelevant (and could not become so after instantiation of metavariables).

Parameter simplification

Parameters that do not occur in the problem can be discarded by the four steps

$$\forall x{:}\,T.\,P \quad \mapsto\!\!\!> \quad P \text{ if } x \notin \mathsf{fv}(P) \tag{4.10}$$

$$\forall \hat{x}{:}\,S\ddagger T.\,P \quad \mapsto\!\!\!> \quad P \text{ if } x \notin \mathsf{fv}(P) \tag{4.11}$$

$$\forall \hat{x}{:}\,S\ddagger T.\,P\{\acute{x}\} \quad \mapsto\!\!\!> \quad \forall x{:}\,S.\,P \text{ if } \Theta \mid \Gamma, x{:}\,S \vdash P\ \mathbf{wf} \tag{4.12}$$

$$\forall \hat{x}{:}\,S\ddagger T.\,P\{\grave{x}\} \quad \mapsto\!\!\!> \quad \forall x{:}\,T.\,P \text{ if } \Theta \mid \Gamma, x{:}\,T \vdash P\ \mathbf{wf} \tag{4.13}$$

The point of these steps is to remove unnecessary dependencies, making it easier to compute the dependency-respecting permutation required when solving a metavariable by inversion. Again, they depend on intensionality, because extensionally a problem that quantifies over an empty type is trivially solvable. Here Θ and Γ are implicitly parameters to the decomposition relation $\mapsto\!\!\!>$, used in steps (4.12) and (4.13) to emphasise that P depends only on one of the twins.

Given a pair of twins whose types are definitionally equal, they can be replaced with a single variable, potentially allowing further progress. For example, the problem $\forall \hat{x}{:}\,S\ddagger S.\,s\{\acute{x}\} \approx t\{\grave{x}\}$ becomes $\forall x{:}\,S.\,s\{x\} \approx t\{x\}$.

$$\forall \hat{x}{:}\,S\ddagger T.\,P \quad \mapsto\!\!\!> \quad \forall x{:}\,U.\,P\{x, x\} \qquad \text{if } \Theta \mid \Gamma \vdash \mathbf{Set} \ni S \mathrel{\equiv\!\![} U \mathrel{]\!\!\equiv} T \tag{4.14}$$

If a parameter has a Σ-type, it can be replaced with two parameters in order to eliminate projections from equations, as in metavariable simplification (Subsection 4.2.4). For example, the problem $\forall x : (\Sigma y : S.\ T).\,\alpha\,(x\,_{\mathrm{TL}}) \approx t\{x\}$ can simplify to $\forall y{:}\,S, z{:}\,T.\,\alpha\,z \approx t\{(y, z)\}$. This simplification happens by the step

$$\forall x{:}\,(\Pi\Delta.\,\Sigma x_0{:}\,S.\ T).\,P \quad \mapsto\!\!\!> \quad \forall y{:}\,(\Pi\Delta.\,S), z{:}\,(\Pi\Delta.\ T\{y\,\Delta\}).\,P\{\lambda\Delta.(y\,\Delta, z\,\Delta)\} \tag{4.15}$$

4.2.6 Summary of the algorithm

Figure 4.14 summarises the problem decomposition steps, and Figure 4.15 summarises the steps for transforming the metacontext, discussed in the previous subsections. In addition to the steps already discussed, the latter figure includes the symmetry step (4.26), which saves writing out symmetrical variants of all the other steps, and the suffix step (4.27), which allows other steps to be applied at an arbitrary point in the metacontext.

Any variables that appear on the right but not on the left are implicitly assumed to be freshly generated, so they do not conflict with any existing names.

Reflexivity

$$(s\!:\!S) \approx (t\!:T) \quad \Mapsto \quad \top \qquad \text{if } \Theta \mid \Gamma \vdash \mathbf{Type} \ni S \mathrel{\equiv\!\![} U \mathrel{]\!\!\equiv} T \text{ and } \Theta \mid \Gamma \vdash U \ni s \equiv t \tag{4.1}$$

η-expansion

$$(f\!:\!\Pi x\!:\!S.\,T) \approx (g\!:\!\Pi x\!:U.\,V) \quad \Mapsto \quad \forall \hat{x}\!:\!S{\ddagger}U.\,(f\,\acute{x}\!:T\{\acute{x}\}) \approx (g\,\grave{x}\!:V\{\grave{x}\}) \tag{4.2}$$

$$(s\!:\!\Sigma x\!:\!S.\,T) \approx (t\!:\!\Sigma x\!:U.\,V) \quad \Mapsto \quad (s_{\,\mathrm{HD}}\!:\!S) \approx (t_{\,\mathrm{HD}}\!:U) \wedge (s_{\,\mathrm{TL}}\!:T\{s_{\,\mathrm{HD}}\}) \approx (t_{\,\mathrm{TL}}\!:V\{t_{\,\mathrm{HD}}\}) \tag{4.3}$$

Rigid-rigid decomposition

$$\Pi x\!:\!S.\,T \approx \Pi x\!:U.\,V \quad \Mapsto \quad S \approx U \wedge \forall \hat{x}\!:\!S{\ddagger}U.\,T\{\acute{x}\} \approx V\{\grave{x}\} \tag{4.4}$$

$$\Sigma x\!:\!S.\,T \approx \Sigma x\!:U.\,V \quad \Mapsto \quad S \approx U \wedge \forall \hat{x}\!:\!S{\ddagger}U.\,T\{\acute{x}\} \approx V\{\grave{x}\} \tag{4.5}$$

$$x \cdot e \approx x' \cdot e' \quad \Mapsto \quad x \cdot e \bowtie x' \cdot e' \tag{4.6}$$

$$s \approx t \quad \Mapsto \quad \bot \text{ if } s \perp\!\!\!\perp t \tag{4.7}$$

η-contraction of subterms

$$P\{\lambda x.n\,x\} \quad \Mapsto \quad P\{n\} \tag{4.8}$$

$$P\{(n_{\,\mathrm{HD}}, n_{\,\mathrm{TL}})\} \quad \Mapsto \quad P\{n\} \tag{4.9}$$

Parameter simplification

$$\forall x\!:T.\,P \quad \Mapsto \quad P \text{ if } x \notin \mathsf{fv}(P) \tag{4.10}$$

$$\forall \hat{x}\!:\!S{\ddagger}T.\,P \quad \Mapsto \quad P \text{ if } x \notin \mathsf{fv}(P) \tag{4.11}$$

$$\forall \hat{x}\!:\!S{\ddagger}T.\,P\{\acute{x}\} \quad \Mapsto \quad \forall x\!:\!S.\,P \text{ if } \Theta \mid \Gamma, x\!:\!S \vdash P\ \mathbf{wf} \tag{4.12}$$

$$\forall \hat{x}\!:\!S{\ddagger}T.\,P\{\grave{x}\} \quad \Mapsto \quad \forall x\!:T.\,P \text{ if } \Theta \mid \Gamma, x\!:T \vdash P\ \mathbf{wf} \tag{4.13}$$

$$\forall \hat{x}\!:\!S{\ddagger}T.\,P \quad \Mapsto \quad \forall x\!:U.\,P\{x, x\} \qquad \text{if } \Theta \mid \Gamma \vdash \mathbf{Set} \ni S \mathrel{\equiv\!\![} U \mathrel{]\!\!\equiv} T \tag{4.14}$$

$$\forall x\!:\!(\Pi\Delta.\,\Sigma x_0\!:\!S.\,T).\,P \quad \Mapsto \quad \forall y\!:\!(\Pi\Delta.\,S), z\!:\!(\Pi\Delta.\,T\{y\,\Delta\}).\,P\{\lambda\Delta.(y\,\Delta, z\,\Delta)\} \tag{4.15}$$

Figure 4.14: Problem decomposition steps

Solving equations by inversion (4.2.1)

$$\Theta, \alpha : T, \Xi, ?\,\forall\Gamma.\, \alpha\,\overline{x_i}^{\,i} \approx t \quad\mapsto\quad \Theta, \Xi_0, \alpha :=^* \lambda\,\overline{x_i}^{\,i}.t : T, \Xi_1 \tag{4.16}$$

if $\Xi \cong \Xi_0, \Xi_1$; $\overline{x_i}^{\,i}$ is linear on $\mathsf{fv}(t)$ and $\Theta, \Xi_0 \mid \cdot \vdash T \ni \lambda\,\overline{x_i}^{\,i}.t$

$$\Theta, ?\,\forall\Gamma.\, \alpha\,\overline{x_i}^{\,i} \approx t \quad\mapsto\quad \Theta, ?\,\bot \tag{4.17}$$

if $t \neq \alpha \cdot e'$ and either $\alpha \in \mathsf{fmv}^{\mathsf{srig}}(t)$ or $\underline{\alpha\,\overline{y_i}^{\,i}}$ occurs rigidly in t

Solving flex-flex equations by intersection (4.2.2)

$$\Theta, \alpha : \Pi\Delta.\, T, \Xi, ?\,\forall\Gamma.\, \alpha\,\overline{x_i}^{\,i} \approx \alpha\,\overline{y_i}^{\,i} \quad\mapsto\quad \Theta, \beta : \Pi\Delta'.\, T, \alpha :=^* \lambda\Delta.\beta\,\Delta', \Xi \tag{4.18}$$

if $\Delta' = \mathsf{intersect}\,\Delta\,\overline{x_i}^{\,i}\,\overline{y_i}^{\,i}$ and $\mathsf{fv}(T) \subset \mathsf{vars}(\Delta')$

Pruning (4.2.3)

$$\Theta, \beta : \Pi\Delta.\, T, \Xi, ?\,\forall\Gamma.\, \alpha \cdot e \approx t \quad\mapsto \tag{4.19}$$

$$\Theta, \gamma : \Pi\Delta'.\, T, \beta :=^* \lambda\Delta.\gamma\,\Delta', \Xi, ?\,\forall\Gamma.\, \alpha \cdot e \approx t$$

if $\mathsf{pruneTm}\,(\mathsf{fv}(e))\,t \mapsto (\beta, \Delta')$

$$\Theta, ?\,\forall\Gamma.\, \alpha \cdot e \approx t \quad\mapsto\quad \Theta, ?\,\bot \text{ if } \mathsf{fv}^{\mathsf{rig}}(t) \not\subset \mathsf{fv}(e) \tag{4.20}$$

Metavariable simplification (4.2.4)

$$\Theta, \alpha : \Pi\Delta.\, \Sigma x : S.\, T \quad\mapsto \tag{4.21}$$

$$\Theta, \alpha_0 : \Pi\Delta.\, S, \alpha_1 : \Pi\Delta.\, T\{\alpha_0\,\Delta\}, \alpha := \lambda\Delta.(\alpha_0\,\Delta, \alpha_1\,\Delta)$$

$$\Theta, \alpha : \Pi\Delta.\, \Pi x : (\Pi\Delta'.\, \Sigma z : S.\, T).\, U \quad\mapsto \tag{4.22}$$

$$\Theta, \beta : \Pi\Delta.\, \Pi y : (\Pi\Delta'.\, S).\, \Pi z : (\Pi\Delta'.\, T\{y\,\Delta'\}).\, U\{\lambda\Delta'.(y\,\Delta', z\,\Delta')\},$$
$$\alpha := \lambda\Delta.\lambda x.\beta\,\Delta\,(\lambda\Delta'.x\,\Delta'\,\textsc{hd})\,(\lambda\Delta'.x\,\Delta'\,\textsc{tl})$$

Problem simplification (4.2.5)

$$\Theta, ?\,\forall\Gamma.\, P \quad\mapsto\quad \Theta, ?\,\forall\Gamma.\, Q \text{ if } P \Mapsto Q \tag{4.23}$$

$$\Theta, ?\,\forall\Gamma.\, P \wedge Q \quad\mapsto\quad \Theta, ?\,\forall\Gamma.\, P, ?\,\forall\Gamma.\, Q \tag{4.24}$$

$$\Theta, ?\,\top \quad\mapsto\quad \Theta \tag{4.25}$$

Symmetry and metacontext suffix

$$\Theta, ?\,\forall\Gamma.\, s \approx t \quad\mapsto\quad \Theta' \text{ if } \Theta, ?\,\forall\Gamma.\, t \approx s \mapsto \Theta' \tag{4.26}$$

$$\Theta, \Xi \quad\mapsto\quad \Theta', \iota\Xi \text{ if } \Theta \mapsto \Theta' \tag{4.27}$$

Figure 4.15: Constraint solving steps

4.3 Correctness

In order to prove that unification correctly solves equational problems, I must first explain what it means for a problem to be solved. I will show that the unification logic is consistent, and that the steps of the unification algorithm are sound for the logic. Moreover, I will prove that every step is most general (in an appropriate sense). Total completeness cannot be expected, but I will show a partial completeness result for the pattern fragment under the assumption of termination. However, it is difficult to prove termination and I conclude this section with a discussion of the problems involved.

4.3.1 Solved problems and logical consistency

An equation $(s : S) \approx (t : T)$ is *solved* if it is true according to the definitional equality, i.e. $\Theta \,|\, \Gamma \vdash \mathbf{Type} \ni S \mathrel{\equiv[} U \mathrel{]\equiv} T$ and $\Theta \,|\, \Gamma \vdash U \ni s \equiv t$. More generally, a problem is solved if the equations it contains are true in the definitional equality. This is captured by the judgment $\Theta \,|\, \Gamma \vdash P\ \mathbf{is}$, defined in Figure 4.16. This requires twins to have equal types, so they can be replaced with a single variable.

Solved problems satisfy the expected substitution properties, proved by structural induction on derivations using Lemma 4.1 and Lemma 4.2:

Lemma 4.7. *If* $\Theta \,|\, \Gamma \vdash \delta : \Delta$ *and* $\Theta \,|\, \Gamma, \Delta, \Gamma' \vdash P\ \mathbf{is}$ *then* $\Theta \,|\, \Gamma, \delta\,\Gamma' \vdash \delta\, P\ \mathbf{is}$.

Lemma 4.8. *If* $\theta : \Theta \sqsubseteq \Theta'$ *and* $\Theta \,|\, \Gamma \vdash P\ \mathbf{is}$ *then* $\Theta' \,|\, \theta\Gamma \vdash \theta\, P\ \mathbf{is}$.

A metacontext is solved if all its hypothesised problems are solved. If a problem is solved, it is true, that is, if $\Theta \,|\, \Gamma \vdash P\ \mathbf{is}$ then $\Theta \,|\, \Gamma \vdash P$. I will show that the converse holds provided Θ is solved: problems assuming only solved hypotheses are themselves solved. This is essentially a cut elimination or normalisation result, as it says that any proof of a problem can be reduced to a normal form, with the normal form proofs of equations being definitional equalities.

In Subsection 4.3.2, I will show that unification steps are sound in the sense that they preserve provability of problems. Hence, if the algorithm steps to a solved metacontext, then the problems it started from must be solved.

The potential presence of twins forces me to prove a slightly more general result, which allows any twins in the context to be replaced with definitionally equal terms. The desired result for the empty context is then an immediate corollary. Say that a substitution $\Theta \,|\, \Delta \vdash \delta : \Gamma$ *identifies twins* if for all $\hat{x} : S \ddagger T \in \Gamma$ we have $\Theta \,|\, \Delta \vdash \mathbf{Type} \ni \delta\, S \mathrel{\equiv[} U \mathrel{]\equiv} \delta\, T$ and $\Theta \,|\, \Delta \vdash U \ni \delta\, s \equiv \delta\, t$.

$\boxed{\Theta \mid \Gamma \vdash P\ \mathbf{is}}$ *(P is solved in Θ and Γ)*

$$\frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash \top\ \mathbf{is}} \qquad \frac{\Theta \mid \Gamma, x{:}S \vdash P\ \mathbf{is}}{\Theta \mid \Gamma \vdash \forall x{:}S.\, P\ \mathbf{is}} \qquad \frac{\begin{array}{l}\Theta \mid \Gamma \vdash \mathbf{Type} \ni S \equiv\!\![\ U\ ]\!\!\equiv T \\ \Theta \mid \Gamma, x{:}\,U \vdash P\{x,x\}\ \mathbf{is}\end{array}}{\Theta \mid \Gamma \vdash \forall \hat{x}{:}\,S \ddagger T.\, P\ \mathbf{is}}$$

$$\frac{\begin{array}{l}\Theta \mid \Gamma \vdash \mathbf{Type} \ni S \equiv\!\![\ U\ ]\!\!\equiv T \\ \Theta \mid \Gamma \vdash U \ni s \equiv t\end{array}}{\Theta \mid \Gamma \vdash (s{:}S) \approx (t{:}\,T)\ \mathbf{is}} \qquad \frac{\Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash (\mathbf{Set}{:}\mathbf{Type}) \approx (\mathbf{Set}{:}\mathbf{Type})\ \mathbf{is}}$$

$$\frac{\Theta \mid \Gamma \vdash P\ \mathbf{is} \qquad \Theta \mid \Gamma \vdash Q\ \mathbf{is}}{\Theta \mid \Gamma \vdash P \wedge Q\ \mathbf{is}}$$

Figure 4.16: Solved problems

Lemma 4.9. *If Θ is solved, $\Theta \mid \Gamma \vdash P$ and δ is a substitution from Γ to Δ that identifies twins, then $\Theta \mid \Delta \vdash \delta\, P$ $\mathbf{is}$.*

Proof. By induction on the derivation of $\Theta \mid \Gamma \vdash P$. The absence of hypothetical problems or first-class quantification over problems makes it easy to show that the rules of the unification logic (Figure 4.6) correspond to solved problems (Figure 4.16). For details, see Appendix D.3.1 (page 242). □

Corollary 4.10. *If Θ is solved and $\Theta \mid \cdot \vdash P$ then $\Theta \mid \cdot \vdash P$ $\mathbf{is}$.*

Corollary 4.11 (Consistency)**.** *If Θ is solved, there is no derivation of $\Theta \mid \cdot \vdash \bot$.*

A metasubstitution $\theta : \Theta \sqsubseteq \Theta'$ is a *solution of* Θ if Θ' is solved. Now if $?\,P \in \Theta$, then $\Theta' \mid \cdot \vdash \theta\, P$ by Lemma 4.2, and hence $\Theta' \mid \cdot \vdash \theta\, P$ $\mathbf{is}$ by Corollary 4.10.

4.3.2 Soundness

Since the algorithm works in small steps, it is easy to verify that each is type safe. All permutations of the metacontext respect dependency. Whenever the algorithm instantiates a metavariable, it does so with a term of the appropriate type. Moreover, every unification problem is replaced with an equivalent conjunction of unification problems. Crucially, the algorithm uses heterogeneous equality to make it easy to represent the telescopes of equations that arise from dependent arguments, potentially allowing progress on some equations even if the equation that makes their types equal is initially blocked. Despite this, and unlike typing modulo, every solution is well typed up to the definitional equality, making the algorithm useful when mixing typechecking with elaboration.

Lemma 4.12. *If* $\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}$ *and* $P \mapsto Q$ *then*

(a) $\Theta \,|\, \Gamma \vdash Q\,\mathbf{wf}$*, and*

(b) $\Theta \,|\, \Gamma \vdash Q$ *implies* $\Theta \,|\, \Gamma \vdash P$*.*

Proof. By case analysis on the decomposition step. I must show that the truth of Q implies the truth of P, so that replacing a hypothesis $?\,P$ with $?\,Q$ leads to a valid metasubstitution. For details, see Appendix D.3.2 (page 245). □

Lemma 4.13. *If* $\Theta \vdash \mathbf{mctx}$ *and* $\Theta \mapsto \Theta'$ *then* $\iota : \Theta \sqsubseteq \Theta'$*.*

Proof. By induction on the step taken, using Lemma 4.12 for problem decomposition. For details, see Appendix D.3.2 (page 246). □

Theorem 4.14 (Soundness)**.** *If* $\Theta \vdash \mathbf{mctx}$ *and* $\Theta \mapsto^* \Theta'$ *where* Θ' *is solved, then* $\iota : \Theta \sqsubseteq \Theta'$ *is a solution of* Θ*.*

Proof. Follows from Lemma 4.13 by induction on the number of steps. □

4.3.3 Generality

The algorithm is carefully designed to make no unforced intensional choices: that is, metavariables are instantiated only if the value is unique up to definitional equality. This corresponds to finding most general unifiers. The particular strategy for tackling constraints is unimportant, as the order in which constraints are solved does not make a difference to the result. Implementations are free to make alternative choices, provided all constraints are eventually dealt with. Of course, since vectors of equations arise from telescopes, it will usually make sense to solve the leftmost equations first so that later equations become homogeneous. Indeed, the reference implementation always works on the leftmost problem for which progress can be made (see Appendix C.4.6, page 235).

Lemma 4.15 (Generality of problem decomposition)**.** *If* $\Theta \,|\, \Gamma \vdash P\,\mathbf{wf}$*, the metasubstitution* $\theta : \Theta, ?\,\forall\Gamma.\, P \sqsubseteq \Theta'$ *is a solution and* $P \mapsto Q$*, then* $\theta : \Theta, ?\,\forall\Gamma.\, Q \sqsubseteq \Theta'$*.*

Proof. By case analysis on $P \mapsto Q$, supposing that $\theta\,(\forall\Gamma.\, P)$ is solved and showing that $\theta\,(\forall\Gamma.\, Q)$ is solved. For details, see Appendix D.3.3 (page 247). □

Theorem 4.16 (Generality)**.** *If* $\Theta_0 \vdash \mathbf{mctx}$*, the metasubstitution* $\theta : \Theta_0 \sqsubseteq \Theta'$ *is a solution and* $\Theta_0 \mapsto \Theta_1$ *then there exists a cofactor* $\zeta : \Theta_1 \sqsubseteq \Theta'$ *such that* $\theta \equiv \zeta \cdot \iota$*.*

Proof. By induction on the step taken, using Lemma 4.15 for problem decomposition. For details, see Appendix D.3.3 (page 248). □

$\boxed{t\ \mathbf{pat}}$

$$\frac{}{\underline{h}\ \mathbf{pat}} \qquad \frac{}{c\ \mathbf{pat}} \qquad \frac{S\ \mathbf{pat} \qquad T\ \mathbf{pat}}{\Pi x{:}S.\,T\ \mathbf{pat}} \qquad \frac{S\ \mathbf{pat} \qquad T\ \mathbf{pat}}{\Sigma x{:}S.\,T\ \mathbf{pat}} \qquad \frac{t\ \mathbf{pat}}{\lambda x.t\ \mathbf{pat}}$$

$$\frac{s\ \mathbf{pat} \qquad t\ \mathbf{pat}}{(s,t)\ \mathbf{pat}} \qquad \frac{\underline{\alpha\cdot e}\ \mathbf{pat} \qquad x\notin \mathsf{fv}(e)}{\underline{\alpha\cdot e\,x}\ \mathbf{pat}} \qquad \frac{\underline{x\cdot e}\ \mathbf{pat} \qquad t\ \mathbf{pat}}{\underline{x\cdot e\,t}\ \mathbf{pat}} \qquad \frac{\underline{n}\ \mathbf{pat}}{\underline{n\,\textsc{hd}}\ \mathbf{pat}}$$

$$\frac{\underline{n}\ \mathbf{pat}}{\underline{n\,\textsc{tl}}\ \mathbf{pat}} \qquad \frac{\mathsf{fmv}(T)=\emptyset \qquad \underline{x\cdot e}\ \mathbf{pat} \qquad s\ \mathbf{pat} \qquad t\ \mathbf{pat}}{\underline{\mathbf{if}_{(y.T)}\,x\cdot e}\ \ s\,t\ \mathbf{pat}}$$

Figure 4.17: Pattern fragment

4.3.4 Partial completeness

As I observed in the introduction, full higher-order unification is undecidable, so the algorithm is incomplete in general. I will show that it is complete for the *static* Miller pattern fragment, where all metavariables are applied to distinct bound variables, assuming it terminates. It goes beyond the pattern fragment in handling Σ-types, and postponing non-pattern problems in case they become solvable later. I believe that it handles a sufficiently broad class of problems to be useful for elaboration of a dependently typed language.

A term t is in the *pattern fragment* if, for every evaluation context of a metavariable $\alpha\cdot e$ in t, e consists solely of projections and applications to distinct variables. This is captured by the judgment t **pat** defined in Figure 4.17. The definition could be extended to allow projections of variables, provided they are distinct in an appropriate sense. For technical reasons in the completeness proof, the result type of an if-expression cannot contain metavariables. A problem is in the pattern fragment if all the terms it equates are in the pattern fragment. A metacontext is in the fragment if all its hypothesised problems are.

To show partial completeness, I will prove that the algorithm can always take a step unless the metacontext is already solved or it contains a contradiction. A metacontext is *failed* if it contains $\bot$ as a hypothesised problem.

Lemma 4.17. *Suppose Θ is a well-formed metacontext in the pattern fragment that is not solved or failed. Then $\Theta \mapsto \Theta'$ for some Θ' in the pattern fragment.*

Proof. By considering the structure of the first unsolved problem in Θ, demonstrating that at least one step of the algorithm must apply. The heterogeneity invariant means that twins or heterogeneous problems must have provably equal types, and for the first unsolved problem, Corollary 4.10 implies that they must be definitionally equal. Hence heterogeneity will not prevent progress. For details, see Appendix D.3.4 (page 249). □

Theorem 4.18. *If Θ is a well-formed metacontext in the pattern fragment, and $\Theta \mapsto^* \Theta'$ such that no more steps apply, then Θ' is solved or failed.*

Proof. Follows immediately from Lemma 4.17: if Θ' were not solved or failed, then the algorithm could take a step. □

4.3.5 Towards a proof of termination

Intuitively, it seems obvious that the algorithm terminates: each step makes the metacontext simpler, either by decomposing a unification problem into smaller components, by solving a metavariable, or by replacing a metavariable with one or more metavariables of smaller type.

However, it is difficult to construct a termination ordering. The conventional approach is to define a measure on the sizes of terms and types in the context, then show that each step of the algorithm reduces the measure. Abel and Pientka (2011) exhibit a suitable ordinal-based measure to show termination of their algorithm for LF.

The picture is more complex for the full-spectrum dependent type theory I have outlined, thanks to the presence of large elimination and metavariables standing for types. Defining a metavariable that occurs in a type can result in types becoming larger, which is not the case in LF. It is thus not clear how to calculate the size of a metavariable. If one takes the supremum over all possible instantiations of a metavariable when calculating its size, then splitting up inhabitants of Σ-types by step (4.21) does not strictly decrease the measure in the resulting ordering.

Any proof of termination will need to take account of the stratification of the type theory. Obviously, if the underlying theory is not strongly normalising then encoding a divergent term can result in non-termination of unification. However, in an inconsistent system even simpler non-termination is possible. Suppose our type theory included the axiom that there is a type of all types, sometimes written **Set** : **Set**. Martin-Löf (1975) had to abandon this axiom after Girard demonstrated its inconsistency. Now consider the context

$$\alpha : \Sigma X : \mathbf{Set}.\, X, ?\,\alpha \approx (\Sigma X : \mathbf{Set}.\, X, \alpha).$$

As α has a Σ-type, a reasonable step is to split it into its components, giving

$$\beta : \mathbf{Set}, \gamma : \beta, ?\,(\beta, \gamma) \approx (\Sigma X : \mathbf{Set}.\, X, (\beta, \gamma)).$$

Now the equation can be decomposed into

$$\beta : \mathbf{Set}, \gamma : \beta, ?\,\beta \approx \Sigma X : \mathbf{Set}.\, X, ?\,\gamma \approx (\beta, \gamma)$$

and solving $\beta := \Sigma X : \mathbf{Set}.\, X$ yields

$$\gamma : \Sigma X : \mathbf{Set}.\, X, ?\,\gamma \approx (\Sigma X : \mathbf{Set}.\, X, \gamma)$$

which is the original problem. Applying the unification algorithm is therefore not guaranteed to terminate, in the presence of the **Set** : **Set** axiom.

The lack of a termination proof for the unification algorithm (applied to the correctly stratified version of the theory) is rather unsatisfactory, and it is left as an open issue for future work.[6] It should be possible to stratify the proof in the same manner as the theory, demonstrating termination for small problems, then extending the result to the full theory with large eliminations.

4.4 Discussion

I have presented an algorithm for higher-order dynamic pattern unification in a full-spectrum dependent type theory. The approach to problem solving in this thesis, based on representing metavariables and problems in an ordered context, allows careful control over dependency and makes it easy to suspend work on one problem while the algorithm tries to solve another.

The algorithm is optimised for clarity rather than performance, and I have not considered its algorithmic complexity. A 'real' implementation would probably need to use a representation of terms with more control over depth of evaluation, rather than working solely with $\beta\delta$-normal forms. Some care is also necessary to determine when to attempt each step: the reference implementation uses a fairly naïve approach, recording the fact that no more steps apply to a given problem, but not the conditions under which this will change. Thus every problem must be examined again whenever a substitution changes its type. Similarly, rather than repeatedly checking to see if the types of metavariables can be simplified,

[6]Termination of higher-order unification can be surprisingly subtle: Dowek et al. (1996) describe a pattern unification algorithm for which termination can fail, as Reed (2009b, §5.1.1) explains. The algorithm I have described is at least not vulnerable to the same counterexample!

as in the reference implementation, projections could be eliminated only as they arise in unification problems.

In this chapter, I described unification for a very restricted type theory, but the algorithm can be extended to support inductive types, proof irrelevance and other advanced features. It therefore forms the base on which to build an elaborator for a full-spectrum dependently typed language, in the style of Agda or Epigram.

However, it is now time to take a different tack. In the second part of this thesis I will describe an extension of Haskell with dependent types. Underlying the elaboration algorithm for this language, as described in Chapter 7, is a constraint solver that makes use of the techniques for unification and type inference described in this chapter and those that preceded it.

Part II

Haskell with dependent types

Chapter 5

The *inch* language: adding dependent types to Haskell

Modern Haskell's poorly-concealed support for dependent types is increasingly being used to obtain correctness guarantees for Haskell programs (McBride, 2002). From the ubiquitous vectors, to well-scoped λ-terms and more exotic examples, dependent types allow programmers to express their intentions more precisely. However, many of these experiments are testaments to the versatility of generalised algebraic datatypes, multi-parameter type classes, functional dependencies and type families, rather than practical programming techniques. In particular, working with type-level numbers and teaching arithmetic to a compiler is a complex, inefficient business; the syntax is ugly, error messages are convoluted and typechecking is sometimes difficult to predict.

Wouldn't it be nicer if we could write programs like the following?

```
data Vec :: * → ℕ → * where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Suc n)

append :: Vec a m → Vec a n → Vec a (m + n)
append Nil           ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

replicate :: Π (n :: ℕ) → a → Vec a n
replicate Zero    _ = Nil
replicate (Suc n) x = Cons x (replicate n x)
```

The *inch* language presented in this part extends Haskell with dependent functions (Π-types), promoted datatypes (including the integers), type-level arithmetic operations and integer constraints. This is not just an attempt to turn

Haskell into Agda or similar full-spectrum dependently typed languages. A clear account of the phase distinction and the operational behaviour of programs is needed. Working in a weaker system enables more powerful type inference. Moreover, the equational theory of arithmetic is not just β-reduction: programming with dependent types can be made easier by automatically solving constraints that depend on algebraic properties (such as the commutativity of addition).

This chapter consists of an overview of related systems (including those based on current features of GHC) and an informal introduction to the syntax and features of *inch* by means of examples. Following this introduction to the high-level language, I will define a corresponding language of *evidence* in Chapter 6. Typechecking the *evidence* language is straightforward, and it is suitable as an intermediate language during compilation. It is very explicit (for example, all type abstractions and applications must be present in the syntax), so information omitted from *inch* programs must be inferred when producing the corresponding *evidence* program. This translation, called elaboration, is the focus of Chapter 7. I will demonstrate larger examples of the use of *inch* in Chapter 8.

The description of elaboration develops the approach to the Hindley-Milner system studied in Chapter 2. I will not study constraint solving in detail, but the unification algorithm for abelian groups in Chapter 3 and the higher-order unification algorithm in Chapter 4 demonstrate the basic ideas.

5.1 Related work

No idea exists in a vacuum. In this section, I will summarise the ideas and predecessor systems on which *inch* is based, including the current state of Haskell as implemented in GHC, and more distantly related work. In the following section, I will lay out the key features of *inch*, comparing it to these systems as I do so.

5.1.1 Full-spectrum dependently typed languages

In full-spectrum dependently typed languages such as Agda (Norell, 2007), based on Martin-Löf Type Theory (Nordström et al., 1990), arbitrary terms can be used to index types. Numbers can be modelled as an inductive datatype and mathematical operations defined on them by recursion. The type theory can be used to prove equations needed to make a program type check. There are no limitations on the form of numeric expressions (to linear functions or polynomials, for example), since the only automatic constraint solving arises from computation (β-reduction) when checking definitional equality.

Suppose we have the following standard definitions (in Agda syntax):

```
data ℕ : Set  where
   Zero : ℕ
   Suc  : ℕ → ℕ
_+_ : ℕ → ℕ → ℕ
Zero   + n = n
Suc m + n = Suc (m + n)
data Vec (A : Set) : ℕ → Set  where
   Nil   : Vec A Zero
   Cons : ∀{n} → A → Vec A n → Vec A (Suc n)
```

Vector concatenation is easily defined by recursion on the first argument, because the $+$ function is also recursive on its first argument:

```
_++_ : ∀{A m n} → Vec A m → Vec A n → Vec A (m + n)
Nil           ++ ys = ys
Cons x xs ++ ys = Cons x (xs ++ ys)
```

However, defining vector reverse is trickier, because $+$ does not reduce if its first argument is neutral and its second is canonical. Consider the following:

```
reverse : ∀{A m} → Vec A m → Vec A m
reverse xs = help xs Nil
   where
     help : ∀{A m n} → Vec A m → Vec A n → Vec A (m + n)
     help Nil             ys = ys
     help (Cons x xs) ys = help xs (Cons x ys)
```

The definition of reverse is not accepted, because $m + \mathsf{Zero} \not\equiv m$, and the second line of help is not accepted, because $m + \mathsf{Suc}\ n \not\equiv \mathsf{Suc}\ (m + n)$. Instead, the user must insert explicit appeals to a proof of the commutativity of $+$. The equational theory of addition is not merely given by a recursive definition!

In general, the user may need to prove many properties of the mathematical operators they have defined. There has been some work on automating this, particularly via tactics in the interactive theorem prover Coq (Gregoire and Mahboubi, 2005), but integrating this with programming can be difficult.

5.1.2 Dependent ML

Xi (1998, 2007) describes *Dependent ML* (DML), a conservative extension of ML that supports "a restricted form of dependent types." Formally, DML is a language schema parameterised on a constraint domain $\mathcal{L}$ from which type indices are drawn. Type checking is reduced to constraint solving in $\mathcal{L}$. Instantiating $\mathcal{L}$ with a language of arithmetic expressions results in a system for type-level numbers, but other choices are possible, such as the theory of free algebraic terms. Xi and Pfenning (1998) demonstrate one application of dependent numeric types: the safe elimination of runtime array-bounds checks.

The development of DML lead Xi and coworkers to design the *Applied Type System* ($\mathcal{ATS}$) framework (Xi, 2004) and the ATS language (Chen, 2006).

Dependent ML is a major inspiration for this work, but extending Haskell with dependent types and type-level numbers requires more than adapting Xi's work to another syntax. While DML extends ML with a fixed domain of indices and constraints, I show how extensions to the Haskell kind system allow indexing by arbitrary type-level expressions, and I focus on the introduction on Π-types, which are not supported by DML.

One feature of DML that is absent from *inch* is support for effects. Since Haskell is more-or-less a pure language, with effects encapsulated in the IO monad, there is no need for specific consideration of effects in the type system, nor for the value restriction. I will not discuss effects further in this thesis.

5.1.3 Generalised algebraic datatypes

Unlike normal algebraic datatypes, *generalised algebraic datatypes* (GADTs) allow the return types of data constructors to specialise the indices of the datatype, by imposing additional equality constraints that must be satisfied on construction and become available through pattern-matching. Thus they are a kind of inductive family indexed by type-level expressions. By defining suitable type constructors, numerically indexed types can be approximated, for example:

```
data ZeroType
data SucType :: * → *
data VecGADT :: * → * → * where
  NilGADT  :: VecGADT a ZeroType
  ConsGADT :: a → VecGADT a n → VecGADT a (SucType n)
```

The *GADT translation* replaces each expression in an index of the result type with a variable, and imposes an equality constraint between the variable and the original expression. This gives:

$$\begin{aligned}
&\mathsf{NilGADT} &&:: \forall\, a\ m\,.\, m \sim \mathsf{ZeroType} \Rightarrow \mathsf{VecGADT}\ a\ m \\
&\mathsf{ConsGADT} &&:: \forall\, a\ m\ n\,.\, m \sim \mathsf{SucType}\ n \Rightarrow \\
& && \qquad\qquad a \to \mathsf{VecGADT}\ a\ n \to \mathsf{VecGADT}\ a\ m
\end{aligned}$$

The idea for GADTs dates back to a draft by Augustsson and Petersson (1994), although the closely related *inductive families* (Dybjer, 1994) have a much longer history in the dependent types community. A variety of names have arisen for essentially the same concept. Early theoretical treatments of GADTs were given by Xi et al. (2003) (under the name *guarded recursive datatypes*) and Cheney and Hinze (2003) (as *first-class phantom types*). They were later studied by Sheard and Pasalic (2008) (as *equality-qualified types*), Simonet and Pottier (2007) (as *guarded algebraic datatypes*) and Peyton Jones et al. (2006) who christened them GADTs. Much subsequent work has gone in to finding good type inference algorithms, especially in the presence of other advanced type system features, and they are well-supported in recent versions of GHC.

Moving beyond the free algebraic equational theory of type constructors, the *associated type families* (Chakravarty et al., 2005) extension to GHC allows type-level functions to be defined. For example, addition can be given thus:

$$\begin{aligned}
&\textbf{type family}\ m + n \\
&\textbf{type instance}\ \mathsf{ZeroType} \quad + n = n \\
&\textbf{type instance}\ \mathsf{SucType}\ m + n = \mathsf{SucType}\ (m + n)
\end{aligned}$$

A similar approach is possible using multi-parameter type classes and functional dependencies (Jones, 2000). In both cases, however, the type-level programming is effectively untyped (as all types have kind $*$). There is nothing to stop one forming the type $\mathbb{Z} + \mathsf{Bool}$, or even declaring such nonsense as

$$\textbf{type instance}\ \mathbb{Z} + \mathsf{Bool} = \mathbb{Z}$$

5.1.4 Haskell libraries

McBride (2002) showed that ‘faking it’ is a viable technique for simulating numeric dependent types in Haskell, including type-safe vector operations and matrix multiplication. Subsequently, there have been numerous implementations of type-level numbers as libraries using existing features of Haskell. The Hackage repository includes the packages `sized-types`, `type-level`, `type-level-numbers`, `type-level-natural-number`, `numtype` and undoubtedly some with more equivocal names! Kiselyov (2005) discusses several possible encodings including a particularly ingenious decimal representation, and manages to get a long way without using any extensions to Haskell 98.

Many of these libraries have arisen in response to the need for type-level numbers in a particular application. For example, the `sized-types` library is part of Kansas Lava (Gill et al., 2009), a DSL for hardware description. The ForSyDe project (Acosta, 2008) uses fixed-size vectors as part of a DSL for modelling computation using signals and processes. Eaton (2006) describes a linear algebra library that provides static guarantees about the dimensions of vectors and matrices, ensuring compatibility when they are multiplied.

Impressive as these libraries are, they are all hampered by the limitations imposed by the language, in such areas as syntax, type inference and clarity of error messages. Better language support for type-level data would make it possible to move beyond these limitations and produce a more user-friendly system.

5.1.5 GHC TypeNats

Recently, Diatchki (n.d.) has developed an extension to GHC that supports type-level natural numbers, adding a new kind `Nat`. The choice of natural numbers rather than integers is motivated by the intended applications, such as measuring the sizes of datatypes, but there is no fundamental reason why the alternative choice could not be made. Of course, natural numbers are easily recovered from integers and an inequality constraint, but the reverse is not so easy.

Work is underway to support arithmetic operations on natural numbers, including addition, multiplication and exponentiation. The plan is for them to be described by type families that trigger special behaviour in GHC's constraint solver (Vytiniotis et al., 2011).

5.2 Features of *inch*

Having described the giants on whose shoulders I am standing, I now give an overview of the *inch* language and compare it to its predecessors. The reader may wish to consult Chapter 8 alongside this section, for more extensive examples of the use of some of these features.

5.2.1 Down with kinds

The Haskell kind system has been expanding for some time. From including just the kind $*$ and function spaces, it has grown to encompass 'promoted' datatypes and kind polymorphism, as described by Yorgey et al. (2012). Promotion allows arbitrary algebraic datatypes to be used as kinds. For example,

```
data Nat = Zero | Suc Nat
```

allows Nat to be used as a kind, and its constructors to be used as types, as in:

```
data Vec :: * → Nat → * where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Suc n)
```

This is a significant improvement on the essentially untyped type-level programming that is otherwise required with GADTs. However, the class of types that can be promoted is somewhat limited. In particular, GADTs cannot themselves be promoted. This prevents indexing a type by a GADT, which is necessary for more advanced dependently typed programming. For example, it is not straightforward to extend the traditional GADT example of well-typed terms in the simply-typed λ-calculus so that contexts are represented by vectors. The following is rejected, because Vec cannot be promoted:

```
data Elem :: Vec k n → k → * where
  Top :: Elem (Cons a v) a
  Pop :: Elem v a → Elem (Cons b v) a
data Tm :: Vec * n → * → * where
  Var :: Elem v a → Tm v a
  Lam :: Tm (Cons a v) b → Tm v (a → b)
  App :: Tm v (a → b) → Tm v a → Tm v b
```

Now the kind system has algebraic datatypes, function spaces and polymorphism, so it increasingly resembles the type system, or at least the type system without recent extensions. Why not simplify matters by removing the distinction between types and kinds? This eliminates the boundary between 'promotable' and 'non-promotable' datatypes. It is a conceptual simplification, because users do not need to learn two slightly different type systems, and it means that type and kind checking become the same operation, which may reduce the burden of specifying and implementing the compiler.

Weirich et al. (2013) show that this identification of types and kinds gives a perfectly good intermediate language. They adopt the typing rule $* : *$, rather than a hierarchy of universes, because *logical* soundness of the system is not a concern. Haskell has general recursion anyway! On the other hand, *type* soundness (progress and preservation) is retained. The *inch* system follows their approach.

ML lacks higher kinds (all types are of kind $*$), so DML distinguishes between types and indices, with the latter having a sort drawn from the underlying constraint language $\mathcal{L}$. It adds a single index to each datatype, and ensures types appear only applied to an index value. Multiple indices can be supplied as pairs.

Integrating type-level data into a single type and kind system, as in *inch*, gives a great deal of extra expressivity. For example, the type of reflexive transitive closures of binary relations on a can be defined in general, then specialised:

```
data RTC :: (a → a → *) → a → a → * where
  Embed      :: r m n → RTC r m n
  Reflexive  :: RTC r n n
  Transitive :: RTC r l m → RTC r m n → RTC r l n
```

DML supports polymorphism over type indices, but since parametric polymorphism in ML is restricted to types of kind $*$, a separate quantifier is needed. It uses $\Pi\, a{:}\gamma\,.\,\tau$ for the universally quantified type of elements of τ polymorphic in an index of sort γ. Since this is a type, it can appear on the left of an arrow, effectively permitting a limited form of higher-rank polymorphism. Unlike the usual notion of a dependent function space (Π-type) in type theory, this construct is parametric: the value a is not available at runtime and the function cannot eliminate it by case analysis. It thus corresponds to $\forall$ in *inch*.

5.2.2 Dependent functions

How might the replicate function, which repeats a value n times to produce a list, be extended to return vectors? The type of the resulting vector depends on the integer argument, so the argument must be known statically (available during typechecking), but the operational behaviour of the function also requires it, so it must be available at runtime. It really requires a dependent Π-type:

```
replicate :: Π (n :: ℕ) → a → Vec a n
replicate Zero     _ = Nil
replicate (Suc n) x = Cons x (replicate n x)
```

The variable n is bound in the range of the Π-type, just as for a universally quantified type scheme, but it also appears in the patterns defining the function.

An alternative to introducing explicit Π-types is connecting the term and type levels using singleton types. In this approach, a family of types is indexed by type-level representations of term-level data, so that each type has a single inhabitant. In Haskell with GADTs and datatype promotion, the example can be expressed using a singleton type SingNat thus:

```
data SingNat :: Nat → ∗ where
  SingZero :: SingNat Zero
  SingSuc  :: SingNat n → SingNat (Suc n)
replicateSing :: SingNat n → a → Vec a n
replicateSing SingZero     _ = Nil
replicateSing (SingSuc n) x = Cons x (replicateSing n x)
```

Converting between the representations requires additional functions. Here a higher-rank function has been used to convert the runtime Nat into the singleton SingNat; an alternative approach is to use existential types (see Subsection 5.2.3).

```
forget :: SingNat n → Nat
forget SingZero     = Zero
forget (SingSuc n) = Suc (forget n)
remember :: Nat → (∀ n . SingNat n → t) → t
remember Zero     f = f SingZero
remember (Suc n) f = remember n (f ∘ SingSuc)
```

There is some duplication and redundancy inherent in this approach, since term-level data must be re-expressed at the type level, but some of this can be taken care of by the compiler. Monnier and Haguenauer (2010) show how to convert from the Calculus of Constructions into a non-dependent language with singleton types. The Strathclyde Haskell Enhancement (McBride, 2010b) supports defining the type-level copy and singleton GADT for an algebraic datatype automatically, and the `singletons` library of Eisenberg and Weirich (2012) goes even further than this, using Template Haskell to automatically convert sufficiently simple term-level functions into type families.

Dependent ML uses singletons, rather than Π-types in the sense above.

5.2.3 Dependent existential types

A key feature of DML is its support for dependent existential types, allowing (for example) the type of lists to be replaced by vectors of existentially quantified length. This is useful for abstraction purposes, or when the invariants being maintained are difficult to express at the type level. For example, the length of the list returned by filter depends on how many elements satisfy the predicate, and rather than building this into the type, another option is to return a vector of existentially quantified length, with a type like

$$\mathsf{filter} :: (a \to \mathsf{Bool}) \to \mathsf{Vec}\ a\ n \to \exists\, m\,.\, \mathsf{Vec}\ a\ m\,.$$

This is a powerful but complex feature, as the combination of parametric polymorphism and existential dependent types significantly complicates type inference. An alternative, introduced by Läufer and Odersky (1992) and used in Haskell, is to associate existential values with data constructors, closing the existential package when data is constructed and opening it when pattern-matching. A variable is existentially quantified if it does not appear in the parameters associated with its constructor. I use this option for *inch*. It is less flexible than genuine existential types, as in DML, but it is also significantly simpler for type inference purposes and is familiar to Haskell programmers through its support in GHC.

Xi (2007) argues that connecting existential types with data constructors leads to a need for too many datatypes with slightly different constraints, and Chen (2006, p. 23) further suggests that "indirect support to existential types is simply impractical in the presence of dependent types", using the example of the singleton family of integers in DML. However, higher-kinded and higher-rank polymorphism ameliorate the problem to an extent, as does native support for Π-types rather than using the singleton encoding. For example, the datatype

$$\begin{array}{l}\textbf{data}\ \mathsf{Ex} :: (k \to *) \to *\ \textbf{where} \\ \quad \mathsf{MkEx} :: f\ x \to \mathsf{Ex}\ f\end{array}$$

allows any singly-indexed type to be converted into an existential. It can safely be eliminated via rank-2 polymorphism:

$$\begin{array}{l}\mathsf{unEx} :: \forall\, a\ f\,.\,(\forall\, x\,.\, f\ x \to a) \to \mathsf{Ex}\ f \to a \\ \mathsf{unEx}\ g\ (\mathsf{MkEx}\ x) = g\ x\end{array}$$

Admittedly, the usual problems with argument order for higher-kinded types will arise: $\mathsf{Ex}\ (\mathsf{Vec}\ a)$ is conveniently the type of vectors of existentially quantified length, but if its arguments were reversed, $\mathsf{Ex}\ (\mathsf{Vec}\ n)$ would be the rather less useful type of vectors of fixed length but unknown element type. In general, a small amount of bureaucratic constructor shuffling may be necessary, but this seems reasonable given the complications of type inference for existentials.

5.2.4 Implicit and explicit arguments

When should it be possible to omit the argument of a function? Milner (1978) achieved a remarkable coincidence, as Lindley and McBride (2013) observe:

Syntactic category	Types	Terms
In source language	Implicit	Explicit
Abstraction	Dependent ($\forall$)	Non-dependent ($\rightarrow$)
Runtime	Erased	Present

So neat is this coincidence that one may forget to distinguish these concepts.

However, as more advanced type systems have been developed, Milner's coincidence has been stretched. On the positive side, Wadler and Blott (1989) introduced typeclasses, a system of implicit term-level arguments that are not erased at runtime. More negatively, current GHC sometimes insists on playing a frustrating guessing game, where it does not allow a type-level argument to be specified but tries to reconstruct it by unification, which is not always possible. That is, there are implicit static arguments that would be better made explicit.

For example, consider the following definitions:

$\textbf{type family}\ \mathsf{F}\ a$

$f :: \mathsf{F}\ a \rightarrow \mathsf{F}\ a$
$f\ x = x$

$g :: \mathsf{F}\ a \rightarrow \mathsf{F}\ a$
$g = f$

The definition of g is rejected by GHC even though its type is syntactically identical to that of f, because it helpfully freshens a to a_0, then fails to solve for the original a since F might not be injective.[1]

A folklore trick often used to solve this problem is to declare a 'proxy type' with a single phantom parameter. This allows an extra argument to be added to each function where the type should be passed explicitly, annotating the proxy constructor appropriately:

$\textbf{data}\ \mathsf{Proxy}\ (a :: k) = \mathsf{Proxy}$

$f' :: \mathsf{Proxy}\ a \rightarrow \mathsf{F}\ a \rightarrow \mathsf{F}\ a$
$f'\ \_\ x = x$

$g' :: \forall\, b\,.\, \mathsf{F}\ b \rightarrow \mathsf{F}\ b$
$g' = f'\ (\mathsf{Proxy} :: \mathsf{Proxy}\ b)$

[1] In fact, GHC even rejects g without a type signature, presumably because it tries to recheck the type it has inferred and hits the same problem.

While this provides a workaround for the problem, it is quite invasive, as the original definition of the function needs to be changed, and it is syntactically noisy. The ability to write type application explicitly in source Haskell is long overdue; the only major stumbling block is deciding upon the concrete syntax.

On the other hand, it is often desirable to omit arguments that can be reconstructed mechanically. This does not necessarily correspond to the type-level/term-level or compile-time/runtime distinctions: runtime terms may well be inferred if the types determine them. This issue has been studied extensively in the setting of dependently typed programming languages, in particular by Pollack (1990). A common approach is to allow certain arguments of functions to be designated *implicit*,[2] with the idea that they will be found automatically during type inference (typically by unification). For example, in Agda the $\mathsf{replicate}$ function can be written

$$\begin{array}{l}\mathsf{replicate} : \{a : \mathsf{Set}\}\ \{n : \mathbb{N}\} \to a \to \mathsf{Vec}\ a\ n \\ \mathsf{replicate}\ \{n = \mathsf{Zero}\}\quad \_ = \mathsf{Nil} \\ \mathsf{replicate}\ \{n = \mathsf{Suc}\ n\}\ x = \mathsf{Cons}\ x\ (\mathsf{replicate}\ x)\end{array}$$

Now n is implicit by default at use sites, since it can usually be inferred from the context, even though it is critical for the runtime behaviour of the function. This is a big win: the compiler is writing operationally relevant parts of the program!

Implicit arguments are written in curly braces in the type, and may be omitted by default in patterns and expressions, or specified by wrapping them in curly braces. Both positional and named variants on the notation are available. In $\{n = \mathsf{Suc}\ n\}$, the first n specifies the implicit argument to match, and the second is a binding occurrence. The fact that an implicit argument can always be specified explicitly if necessary avoids the problems discussed above. Agda-style notation would allow a much neater solution to the problem discussed above:

$$\begin{array}{l}g'' :: \forall\, b\,.\, \mathsf{F}\ b \to \mathsf{F}\ b \\ g'' = f\ \{a = b\}\end{array}$$

In Section 7.1, I will show how *inch* supports implicit argument notation. It adopts a slight generalisation of the Haskell syntax for quantifiers in types: a dot following the binder means the quantification is implicit, while an arrow means the quantification is explicit. Thus $\forall\, a\,.\,\tau$ and $\Pi\ (n :: \mathbb{N})\,.\,\upsilon$ are implicitly quantified, while $\forall\,(a :: *) \to \tau$ and $\Pi\ n \to \upsilon$ are explicitly quantified. For applications, the Agda-style named implicit argument notation is used, as in $f\ \{a = b\}$.

[2]Implicit arguments are not the same as the 'implicit parameters' of Lewis et al. (2000), which are a construct for dynamic scoping.

Implicit Π-types

Typeclasses provide a form of term-level implicit arguments for Haskell. Along with the singleton encoding, this allows an approximation of an implicit Π-type:

```
class ImplicitNat (n :: Nat) where
  sing :: SingNat n
instance ImplicitNat Zero where
  sing = SingZero
instance ImplicitNat n ⇒ ImplicitNat (Suc n) where
  sing = SingSuc sing
```

A class context containing ImplicitNat n means that n is passed implicitly. It is meaningful even though an obvious induction shows that the predicate holds for every canonical Nat. An implicit version of replicate can be defined thus:

```
replicateImplicit :: ImplicitNat n ⇒ a → Vec a n
replicateImplicit = replicateSing sing
```

However, there are now three variations on a single type (Nat, SingNat and ImplicitNat), all of which must be understood by the programmer. Moreover, switching between explicit and implicit arguments is clumsy: sing :: Sing x must be used in place of a simple x.

Implicit Π-types are often useful in class instances. For example, in order to make Flip Vec n a monad, the length n must be supplied at runtime so that replicate can be used in the implementation of return:

```
newtype Flip f x y = Flip {unFlip :: f y x}
instance Π (n :: ℕ) . Monad (Flip Vec n) where
  return        = Flip ∘ replicate n
  Flip xs ≫= f = Flip (help xs (unFlip ∘ f))
    where
      help :: Vec a m → (a → Vec b m) → Vec b m
      help Nil         g = Nil
      help (Cons x xs) g = Cons (vhead (g x)) (help xs (vtail ∘ g))
```

5.2.5 Type-level numbers

I have already shown several examples of the use of type-level natural numbers in measuring the lengths of vectors. For many applications involving measuring the sizes of datatypes, natural numbers suffice, and they have an obvious inductive definition from zero and successor constructors, as shown above. Most Haskell libraries for type-level numbers use naturals, as does the TypeNats extension to GHC (Diatchki, n.d.).

However, another choice is available: the integers. This increases expressivity as natural numbers can be recovered from integers using inequality constraints (see Subsection 5.2.7). DML (Xi, 2007) takes this choice.

The design considerations for a language extension are different to those of a library. There is no restriction to ad-hoc type-level programming techniques. Type inference may be easier for integers, because they form an abelian group, allowing the unification algorithm from Chapter 3 to be used.

Moreover, there are some use cases that rely on negative as well as positive integers, such as implementing a library for units of measure. Given a fixed set of base units, a derived unit can be represented by its integer exponents: for example, metres per second ($\mathbf{m}/\mathbf{s}$) could be represented by 1 as the exponent of metres, -1 as the exponent of seconds and 0 as the exponent of other base units. The NumType library of Buckwalter (2009) is one of the few libraries to support negative numbers for exactly this reason. In Chapter 8, units of measure are developed using the *inch* system.

Zenger (1997) describes a Haskell-like language with types indexed by polynomials over the complex numbers. Gröbner basis techniques can then be used to solve constraints. This is an interesting choice of constraint domain, but does not quite match most of the examples, which expect integers or natural numbers. This may lead to overly permissive type-checking (if constraints with no integer solution can be solved in $\mathbb{C}$) or failures to deduce desired properties (for example, $n > 0$ does not imply $n \geq 1$).

The prototype implementation of the *inch* language supports a kind $\mathbb{Z}$ of integers, plus a kind $\mathbb{N}$ of natural numbers that is treated as syntactic sugar for $\mathbb{Z}$ with an inequality constraint. I will focus on the addition of Π-types, rather than numeric constraint solving, however.

5.2.6 Supported operations

Closely connected to the choice of numbers to represent is the signature of operations that are available on them. Addition is a must for any nontrivial use of type-level numbers, even just appending vectors. If negative integers are permitted, then subtraction is also useful. If not, it is less clear what meaning (if any) to give subtraction; though there are several options (Runciman, 1989), perhaps it is easiest to require types to be rewritten to avoid it.

With just addition (and perhaps subtraction) one can express multiplication by constants and many useful linear properties, while remaining within the theory of Presburger arithmetic. This theory is decidable (Presburger, 1930, translated by Stansifer (1984)), so complete constraint solving is feasible. It should not be dismissed out of hand, as many useful examples can be expressed in this fragment.

Diatchki's TypeNats extension includes addition, multiplication and exponentiation on natural numbers, but omits their (partial) inverses. This leads to interesting challenges in designing a suitable constraint solver that is powerful enough to handle common constraints but also allows the user to supply proofs.

Xi's constraint solver for DML handles only linear constraints, though his formalism allows for more complex numeric expressions, and he mentions the possibility of postponing nonlinear constraints in the hope that they will become linear and hence solvable. In his subsequent work on the ATS programming language (Xi, 2004), he argues for the combination of programming and theorem proving to allow the user to supply proofs of difficult constraints.

5.2.7 Constraints

When working with GADTs or type families, it is frequently useful to add equality constraints to qualified types; indeed GADTs are implemented using equality constraints on constructors that are made available by pattern-matching. Similarly, equality and inequality constraints are useful for type-level numbers.

The encoding of propositional equality in type theory (Nordström et al., 1990) can be translated into Haskell thus (writing $(\sim)$ for built-in equality constraints):

$$\begin{array}{l}
\textbf{data}\ \mathsf{Id}\ m\ n\ \ \textbf{where} \\
\quad \mathsf{Refl} :: m \sim n \Rightarrow \mathsf{Id}\ m\ n \\
\mathsf{elimEq} :: \forall a\ m\ n\,.\,\mathsf{Id}\ m\ n \to (m \sim n \Rightarrow a) \to a \\
\mathsf{elimEq}\ \mathsf{Refl}\ x = x
\end{array}$$

One could abstract a over an index in the definition of elimEq, giving

$$\begin{array}{l}
\mathsf{elimEq}' :: \forall (a :: t \to *)\ m\ n\,.\,\mathsf{Id}\ m\ n \to a\ m \to a\ n \\
\mathsf{elimEq}'\ \mathsf{Refl}\ x = x
\end{array}$$

but since Haskell's type-level function space lacks first-class λ-abstraction, it is easier to work in the former style, using equality rather than abstraction.

A decision procedure that produces a witness to the equality can be given by

$$\mathsf{decideEq} :: \Pi\ (m\ n :: \mathbb{Z}) \to \mathsf{Maybe}\ (\mathsf{Id}\ m\ n)$$

or even

$$\mathsf{decideEq'} :: \Pi\ (m\ n :: \mathbb{Z}) \to \mathsf{Either}\ (\mathsf{Id}\ m\ n)\ (\mathsf{Id}\ m\ n \to \mathsf{Void})$$

where negation is expressed as a function to the type Void with no constructors. This encoding of negation is not entirely satisfactory in a non-total language, however, since all types are inhabited.

Alternatively, a function to compare two integers can be given a rank-2 type:

$$\mathsf{ifEq} :: \Pi\ (m\ n :: \mathbb{Z}) \to (m \sim n \Rightarrow a) \to a \to a$$

In the third argument, the assumption that m and n are equal is available to the typechecker. The kind of continuation-passing style demonstrated by ifEq is frequently useful to introduce additional hypotheses or eliminate existential type variables, showing the need for a system that integrates type-level data with arbitrary-rank polymorphism. Of course, it also makes use of Haskell's laziness and the corresponding ease of writing control operators.

Going beyond equalities, inequality constraints ($<, \leq, >, \geq$) are useful in order to express weak bounds. For example, they allow safe projection from a vector:

$$\begin{array}{l}
\mathsf{index} :: \forall (m :: \mathbb{N}) \,.\, \Pi\ (n :: \mathbb{N}) \to n < m \Rightarrow \mathsf{Vec}\ a\ m \to a \\
\mathsf{index}\ \mathsf{Zero}\qquad (\mathsf{Cons}\ x\ xs) = x \\
\mathsf{index}\ (\mathsf{Suc}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{index}\ n\ xs
\end{array}$$

Similar techniques can be used to create a safe array library that eliminates runtime bounds checks, as Xi and Pfenning (1998) taught us.

When used in a quantifier, the natural number kind imposes a constraint on the bound variable: $\Pi\ (n :: \mathbb{N}) \,.\, t$ translates to $\Pi\ (n :: \mathbb{Z}) \,.\, 0 \leqslant n \Rightarrow t$. This is similar to (though simpler and less expressive than) DML's notion of a 'subset sort' (Xi, 1998), which allows a new sort to be formed by restricting an existing sort with some constraints.[3]

Learning by testing

A crucial feature for working with type-level data is the ability to perform type-refining dynamic tests, enabling "learning by testing" (Altenkirch et al., 2005). Dependently typed programming languages typically exploit dependent pattern

[3] A DML sort is similar to a Haskell kind, but restricted to terms in the index language $\mathcal{L}$.

matching and techniques such as views (McBride and McKinna, 2004). Dependent pattern matching is supported by *inch*, as in the replicate example.

A small extension to Haskell's notation for guards is useful. I use curly braces to mark a guard, written in the constraint language, that refines the type of the corresponding branch. For example, the ifEq function can be implemented as:[4]

$$
\begin{aligned}
&\mathsf{ifEq} :: \Pi\,(m\ n :: \mathbb{Z}) \to (m \sim n \Rightarrow a) \to a \to a\\
&\mathsf{ifEq}\ m\ n\ x\ y \mid \{m \equiv n\} = x\\
&\qquad\qquad\quad\ \mid \mathsf{otherwise} = y
\end{aligned}
$$

The runtime behaviour of such expressions is straightforward: drop the curly braces to obtain the usual guard. If-expressions can be handled in a similar way.

Helping the constraint solver

Given an incomplete constraint solver, what can the user do if a program is rejected because a true constraint was not solved by the system? Sometimes it may be possible to extend the type signature by quantifying over the additional constraint, requiring callers to prove it; eventually a caller may be reached that supplies concrete values for variables, so the constraint is easily checked. However, in some cases it may not be possible to quantify over the required hypothesis, for example if the function pattern-matches on a GADT introducing local constraints.

One possibility is to supply additional information to the typechecker using a higher-rank function. For example, a term for commutativity of multiplication

$$\mathsf{commutes} :: \forall\,(m\ n :: \mathbb{Z}) \to (m * n \sim n * m \Rightarrow a) \to a$$

would allow the user to write commutes $m\ n\ x$ in place of an expression x that depends on the assumption $m * n \sim n * m$. The quantification over m and n is explicit, even though they are erased at runtime. This is necessary because the typechecker will not be able to choose appropriate arguments.

A trusted library of properties could be implemented as 'unsafe' coercions. If the variables were available at runtime (quantified over by Π rather than $\forall$), such properties could be 'proved' by writing a recursive function to perform the necessary induction, but in a partial language this function must be executed at runtime in order to ensure type safety, which is likely to be undesirable.

[4]Here $\sim$ is the equality type constraint, $\equiv$ the runtime equality test and $=$ the Haskell syntax for a definition!

Chapter 6

A language of *evidence*

In this chapter, I describe the *evidence* language, suitable as an intermediate language for a Haskell compiler. The next chapter will describe how to elaborate *inch* terms into *evidence* terms. The language presented here is based on System F_C, the core language of GHC[1], with modifications inspired by dependent type theory to support the new features of *inch* and make the presentation uniform.

One reason for compiling via an intermediate language, rather than directly to a low-level language, is to ensure correctness. It is analogous to the use of an easily checked kernel type theory in a proof assistant such as Coq. Typechecking intermediate language code is straightforward, as expressions encode their own typing derivations, and everything is fully explicit. Terms can be checked after elaboration and during optimisation, leading to early detection of compiler bugs.

A key inspiration for this chapter is the work of Weirich, Hsu, and Eisenberg (2013). Like them, I adopt the dangerous-sounding rule $* : *$, so the kind of types classifies itself. To a dependent type theorist, this instantly suggests paradox,[2] but the system will permit general recursion at the type level in any case, so the potential paradox is irrelevant. There is no hope of proving strong normalisation in general, but the usual subject reduction and progress properties are maintained. The system does include a logic of equality, and this must be kept consistent, which can be achieved by keeping it weak. Coercions encode the exact amount of computation to be done, so there is no risk that typechecking an evidence term will fail to terminate. Moreover, "the point of writing a proof in a strongly normalizing calculus is that you don't need to normalize it".[3] There is no need to compute coercions, whereas if coercions could be bogus, they would need to be normalised before being relied upon to coerce values.

[1]System F_C has developed over time; the main versions are discussed in Subsection 6.7.3.

[2]The 1971 type theory of Martin-Löf (1975, 1998) was inconsistent for this reason.

[3]A saying of Randy Pollack, quoted by Altenkirch et al. (2005).

The main feature that the *evidence* language adds to previous versions of System F_C is Π-types, allowing types to depend on a limited fragment of 'shared' runtime expressions. To enable a compact presentation of the system, I abstract over the possible 'phases' of quantification and typing judgments, and write a single set of typing rules covering both types and terms. This highlights the common structure and avoids repetition. For example, a single application rule replaces a multitude of rules for applying one sort of expression to another.

Moreover, a single syntax and type system for type and term-level constructs allows them to have a common operational semantics, in the usual style of dependent type theory. This is a fundamental difference in perspective from System F_C. It leads to the replacement of type families (that are axiomatically defined and lacking operational behaviour) with honest-to-goodness case analysis. Type-level functions are then mere recursive definitions. There is no λ-abstraction at the type level, and type-level functions must be saturated (fully applied), so the language of types is essentially first-order and elaboration is as simple as possible.

Unlike type families, type-level functions as I define them do not support case analysis on types or the open world assumption. The two are not necessarily mutually exclusive. One could certainly imagine a system in which type families and true type-level (or shared type- and term-level) functions are both available.

In Subsection 5.2.2 (page 96), I gave the example of the replicate function:

$$
\begin{array}{l}
\mathsf{replicate} :: \Pi\ (n :: \mathbb{N}) \to a \to \mathsf{Vec}\ a\ n \\
\mathsf{replicate\ Zero}\qquad \_ = \mathsf{Nil} \\
\mathsf{replicate\ (Suc}\ n)\ x = \mathsf{Cons}\ x\ (\mathsf{replicate}\ n\ x)
\end{array}
$$

This uses its natural number argument both statically (as it occurs in the type) and dynamically (for pattern-matching at runtime). It can be seen as a single shared function that makes sense at the type level and the term level.

For comparison, here is the same thing implemented using a type family and term-level singletons, the alternative to Π-types discussed in Subsection 5.2.2.[4]

$$
\begin{array}{l}
\textbf{type family}\ \mathsf{Replicate}\ (n :: \mathbb{N})\ (x :: a) :: \mathsf{Vec}\ a\ n \\
\textbf{type instance}\ \mathsf{Replicate\ Zero}\qquad \_ = \mathsf{Nil} \\
\textbf{type instance}\ \mathsf{Replicate\ (Suc}\ n)\ x = \mathsf{Cons}\ x\ (\mathsf{Replicate}\ n\ x) \\
\\
\mathsf{replicateSing} :: \mathsf{SingNat}\ n \to a \to \mathsf{Vec}\ a\ n \\
\mathsf{replicateSing\ SingZero}\qquad \_ = \mathsf{Nil} \\
\mathsf{replicateSing\ (SingSuc}\ n)\ x = \mathsf{Cons}\ x\ (\mathsf{replicateSing}\ n\ x)
\end{array}
$$

[4]The Replicate type family is rejected by GHC 7.6, because it involves a promoted GADT. It is forbidden by the system of Weirich et al. (2013), which does not permit the result kind of a type family to depend on its arguments, but this may not be a fundamental restriction.

The type family version can be defined directly on the kind of natural numbers, but the term-level version must use a singleton copy to pattern-match at runtime. The connection between the term and type-level functions has been lost.

In the sequel, I introduce the syntax of the *evidence* language (6.1), discuss the key role that phase distinctions play (6.2) and give the type system for the language (6.3). I then define its operational semantics and prove subject reduction (6.4). Proving progress takes a little more work (6.5). Finally, I define a runtime erasure operation that removes types and coercions (6.6) and conclude with a discussion of possible extensions, related systems and future work (6.7).

6.1 Syntax

In this section, I present the syntax for the *evidence* language. It may be worth skipping quickly through this on first reading, and returning to clarify details of the syntax. Figure 6.1 shows the naming conventions in use in this chapter.

Figure 6.2 gives the syntax of signatures and contexts. The signature Σ contains global top-level symbols that may appear in expressions, including type constructors D, data constructors K, functions f and axioms C. The context or telescope Γ, Δ contains variables bound locally. Contexts will later be generalised to metacontexts Θ, which include metavariables for use in elaboration (discussed in Chapter 7).

The common syntax of expressions is shown in Figure 6.3. Unifying the syntax avoids redundancy, as there are unique forms for abstraction, application and quantification, and it simplifies the operational semantics. In Section 6.2, I will explain the use of phases Φ, Ψ to distinguish the different roles of types, coercions and terms. Saturated function applications $f(\delta)$ are syntactically distinguished from normal application.

For the sake of familiarity, Figure 6.4 gives subgrammars of ρ for type expressions τ, υ, κ, coercions γ, η, runtime terms e and shared terms ε. Variables are accounted for by a single production but I will frequently write a, b for type variables, c for coercion variables and x, y, z for term variables. Propositions φ are a subgrammar of types that represent quantified equations.

De Bruijn (1991) showed that working with telescopes of bindings Δ, and vectors of expressions δ corresponding to them, rather than single bindings and single substitution, is often a significant simplification. I write ψ for a vector containing type expressions.

a, b	type variable	κ	kind
c	coercion variable	λ	abstraction
e	expression	ξ	value type
f	function	ρ	expression
i, j, k, l, m, n	integer	τ, υ	type
r	erased runtime term	φ	proposition
v	value expression	ψ	vector of type expressions
x, y, z	term variable	ω	telescoped coercion
C	coercion axiom	Γ, Δ	context (telescope)
D	type constructor	Λ	abstraction
H	rigid constructor	Π	dependent function space
K	data constructor	Σ	signature
γ, η	coercion	Υ	type phase
δ	vector of expressions	Φ, Ψ	phase
ε	shared term	Ω	non-type phase
ι	identity substitution		

Figure 6.1: Naming conventions

$$\begin{array}{rcl}
\Sigma & ::= & \cdot \mid \Sigma, \mathsf{D} :^{\Phi} \kappa \mid \Sigma, \mathsf{K} :^{\Phi} \kappa \mid \Sigma, C :^{\Box} \varphi \mid \Sigma, f\,[\Delta] :^{\Phi} \kappa \mid \Sigma, f\,[\Delta] = \rho :^{\Phi} \kappa \\
\Gamma,\ \Delta & ::= & \cdot \mid \Gamma, a :^{\Phi} \tau \\
\Phi,\ \Psi & ::= & \forall \mid \Pi \mid \Box \mid \text{人} \\
\Upsilon & ::= & \forall \mid \Pi \\
\Omega & ::= & \Box \mid \text{人}
\end{array}$$

Figure 6.2: Grammar of signatures, contexts and phases

ρ	$::=$		expression
	$\vert$	a	variable
	$\vert$	$\rho^{\Phi}\,\rho'$	application
	$\vert$	$(a :^{\Phi} \rho) \to \rho'$	quantification
	$\vert$	H	constructor
	$\vert$	$f(\delta)$	saturated function
	$\vert$	$\rho \rhd \gamma$	type cast
	$\vert$	q	coercion evidence
	$\vert$	$(\mathbf{d})\mathbf{case}\,\rho\,\mathbf{of}\,\overline{br_i}^{\,i}$	case expression
	$\vert$	$\Lambda a :^{\Phi} \kappa\,.\,\rho$	abstraction
H	$::=$		rigid constructor
	$\vert$	D	type constructor
	$\vert$	K	data constructor
	$\vert$	$*$	kind of types
	$\vert$	$(\sim)$	equality type
q	$::=$		coercion evidence
	$\vert$	C	axiom
	$\vert$	$\mathbf{resp}\,\omega\,\Delta\,\tau$	congruence
	$\vert$	$\mathbf{left}\,\gamma$	left injectivity
	$\vert$	$\mathbf{right}\,\gamma$	right injectivity
	$\vert$	$\mathbf{conga}^{\Upsilon}\,\gamma\,\eta$	congruence of Υ application
	$\vert$	$\mathbf{conga}^{\Box}\,\gamma\,(\eta_1,\,\eta_2)$	congruence of $\Box$ application
	$\vert$	$\mathbf{cong}\,\Phi\,\eta\,\gamma$	congruence of quantification
	$\vert$	$\gamma@\eta$	congruence of Υ instantiation
	$\vert$	$\gamma@(\eta_1, \eta_2)$	congruence of $\Box$ instantiation
	$\vert$	$\mathbf{coh}\,\gamma\,\eta$	coherence
	$\vert$	$\mathbf{kind}\,\gamma$	equality of kinds
	$\vert$	$\mathbf{step}\,\rho$	computation step

$$\begin{array}{rcl}
\delta & ::= & \cdot \mid \delta, \rho \\
\omega & ::= & \cdot \mid \omega, (\tau, \tau', \gamma) \mid \omega, (\rho, \rho') \\
(\mathbf{d})\mathbf{case} & ::= & \mathbf{dcase} \mid \mathbf{case} \\
br & ::= & \mathsf{K}\,\Delta \to \rho
\end{array}$$

Figure 6.3: Grammar of expressions

$\tau,\ \upsilon,\ \kappa$	$::=$		type expression (phase $\forall$)
	$\vert$	a	variable
	$\vert$	$\tau^{\Phi}\rho$	application at phase Φ
	$\vert$	$(a{:}^{\Phi}\kappa) \to \tau$	quantification
	$\vert$	H	constructor
	$\vert$	$f(\delta)$	saturated function
	$\vert$	$\tau \triangleright \gamma$	type cast
	$\vert$	$(\mathbf{d})\mathbf{case}\,\tau\,\mathbf{of}\,\overline{br_i}^{\,i}$	case expression
$\gamma,\ \eta$	$::=$		coercion (phase $\square$)
	$\vert$	c	variable
	$\vert$	$\gamma \triangleright \eta$	cast
	$\vert$	q	coercion evidence
	$\vert$	$\Lambda a{:}^{\Phi}\kappa\,.\,\gamma$	proof abstraction
e	$::=$		runtime term (phase 人)
	$\vert$	x	variable
	$\vert$	$e^{\Phi}\rho$	application at phase Φ
	$\vert$	K	data constructor
	$\vert$	$f(\delta)$	saturated function
	$\vert$	$e \triangleright \gamma$	type cast
	$\vert$	$(\mathbf{d})\mathbf{case}\,e\,\mathbf{of}\,\overline{br_i}^{\,i}$	case expression
	$\vert$	$\Lambda a{:}^{\Phi}\kappa\,.\,e$	abstraction
ε	$::=$		shared term (phase Π)
	$\vert$	x	variable
	$\vert$	$\varepsilon^{\Phi}\rho$	application at phase Φ
	$\vert$	K	data constructor
	$\vert$	$f(\delta)$	saturated function
	$\vert$	$\varepsilon \triangleright \gamma$	type cast
	$\vert$	$(\mathbf{d})\mathbf{case}\,\varepsilon\,\mathbf{of}\,\overline{br_i}^{\,i}$	case expression

$$\begin{array}{lll} \varphi & ::= & (\sim)\kappa_1\,\kappa_2\,\tau_1\,\tau_2 \mid (a{:}^{\Phi}\kappa) \to \varphi \\ \psi & ::= & \cdot \mid \psi, \tau \end{array}$$

Figure 6.4: Subgrammars of type expressions, coercions and terms

6.2 Phase distinctions and promotion

Existing work by Yorgey et al. (2012) on System $\mathrm{F}_\mathrm{C}^{\uparrow}$, which extends System F_C with type-level data, is based around the idea of 'promoting' datatypes to the kind level and data constructors to the type level. By a fortuitous coincidence, some terms turn out to be well-kinded type expressions, but there is no formal relationship between well-typed terms and well-kinded types. Not all datatypes can be promoted, since the kind system is more restrictive than the type system, although work is underway to change this (Weirich et al., 2013).

Adding Π-types to a system with $\mathrm{F}_\mathrm{C}^{\uparrow}$-like promotion is possible, adding yet more abstraction and application forms, and another typing judgment. However, factoring out the common structure makes the relationships between the phases clear. This is particularly true when it comes to the operational semantics: rather than trying to juggle separate rules for runtime terms, shared terms and type expressions, I can instead give a single system that covers them all. Of course, the purpose of the phase distinction is maintained: type expressions and coercions are erased at runtime, as discussed in Section 6.6.

The *evidence* language distinguishes between *phases* given by

$\Phi,\ \Psi$	$::=$		phase
	$\mid$	$\forall$	static phase (universal quantification)
	$\mid$	Π	shared phase (dependent product)
	$\mid$	$\square$	proof phase (coercion quantification)
	$\mid$	人	runtime phase (function space)

There is a single typing judgment, annotated by the phase at which it holds. Phases occur on quantifiers, λ-abstractions and context bindings, to indicate the phase at which variables are bound, and on applications, to indicate the phase of the quantifier. This means that the typing rules have a single rule for each construct, rather than a whole host of similar rules. It is not essential to unify these concepts; one might choose to present the phases separately. The $\square$ phase must sometimes be distinguished, in order to ensure it remains consistent. In particular, it cannot admit case analysis or recursive functions.

The phase annotations on typing judgments will justify the subgrammars given in Figure 6.4, as whenever an expression ρ is well-typed at phase Φ, it will belong to the subgrammar corresponding to Φ.

The single syntax for quantifiers $(a :^{\Phi} \tau) \to \upsilon$ subsumes universal quantification and the runtime function space: $\forall\, a : \tau \,.\, \upsilon$ becomes $(a :^{\forall} \tau) \to \upsilon$ and $\tau \to \upsilon$

becomes $(x{:}^{\text{⅄}}\tau) \to \upsilon$. The latter is never dependent, however, as the typing rules ensure x cannot be used in υ, so I will often write the familiar syntax instead.

Similarly, the single syntax for abstractions $\Lambda a{:}^{\Phi}\tau \,.\, \rho$ subsumes λ-abstraction over terms and Λ-abstraction over type expressions. Again, I will write the more familar $\lambda x{:}\tau \,.\, e$ instead of $\Lambda x{:}^{\text{⅄}}\tau \,.\, e$, but this is merely syntactic sugar. Abstractions may occur only at phase ⅄ or $\Box$: there is no type-level λ-abstraction.

6.2.1 The access policy

The fortuitous coincidence that some terms are also well-kinded type expressions now turns into a solid metatheoretic property: all well-typed shared terms are both well-typed runtime terms and well-kinded type expressions. The 'access policy' relation $\Phi \hookrightarrow \Psi$ expresses when things at one phase can be used at another. This is a partial order, defined by the following Hasse diagram:[5]

$$\begin{tikzcd}
 & \forall \arrow[dr, no head] & & \text{⅄} \arrow[dl, no head] \\
\Box & & \Pi &
\end{tikzcd}$$

The typing rule for variables (see Figure 6.6)

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad \Gamma \ni a :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi}{\Gamma \vdash a \;:^{\Psi}\; \kappa}$$

uses this relation: any variable bound at phase Φ is accessible at phase Ψ. A key result (Lemma 6.4) extends this to show that if a typing judgment holds at phase Φ, and $\Phi \hookrightarrow \Psi$, then it holds at phase Ψ.

6.2.2 Promoted data constructors

Where does promotion fit in to this system? The constructor Just has type $(a \;:^{\forall}\; *, x \;:^{\text{⅄}}\; a) \to \mathsf{Maybe}\, a$, so it seemingly expects a static and a runtime argument. We want to be able to use it at the type level with static arguments, so that $\mathsf{Just}\; *\; \mathsf{Bool}$ has type $\mathsf{Maybe}\, *$. Thus the application rule

$$\frac{\Gamma \vdash \rho \;:^{\Psi}\; (a{:}^{\Phi}\kappa_1) \to \kappa_2 \qquad \Gamma \vdash \rho' \;:^{\Phi \,/\!\!/\, \Psi}\; \kappa_1}{\Gamma \vdash \rho^{\Phi}\rho' \;:^{\Psi}\; [\rho'/a]\,\kappa_2}$$

[5]So $\Pi \hookrightarrow \forall$ and $\Pi \hookrightarrow$ ⅄, while $\Box$ is lonely.

calculates the phase $\Phi \mathbin{/\!\!/} \Psi$ at which to check the argument from the phase Φ of the quantification and the phase Ψ at which the expression is being checked. The *relativisation* operator $\Phi \mathbin{/\!\!/} \Psi$, pronounced '$\Phi$ for Ψ', is defined by

$/\!\!/$	$\curlywedge$	Π	$\forall$	$\Box$
$\curlywedge$	$\curlywedge$	Π	$\forall$	$\forall$
Π	Π	Π	$\forall$	$\forall$
$\forall$	$\forall$	$\forall$	$\forall$	$\forall$
$\Box$	$\Box$	$\Box$	$\Box$	$\Box$

When checking a runtime term, the phase of the typing judgment is $\curlywedge$, and $\Phi \mathbin{/\!\!/} \curlywedge$ is just Φ, so arguments to runtime functions must be of the phase stated in their type. However, when checking in a static context, the argument must be known statically. This causes implicit promotion: $\curlywedge \mathbin{/\!\!/} \forall = \forall$ means that $\mathsf{Just} * :^{\forall} (x :^{\curlywedge} *) \to \mathsf{Maybe} *$ can be applied to $\mathsf{Bool} :^{\forall} *$. Since $\curlywedge \not\rightarrowtriangle \forall$ and $\curlywedge \not\rightarrowtriangle \Phi \mathbin{/\!\!/} \forall$ for all Φ, there is no way that a variable at phase $\curlywedge$ can be used in a type expression at phase $\forall$.

I will usually omit the annotation on applications, writing $\rho\, \rho'$ instead of $\rho^{\Phi} \rho'$, since it is easily recovered from the type of ρ. It is useful for defining erasure as an operation on syntax rather than on typing derivations in Section 6.6.

6.2.3 Promoted functions

The $(+)$ function is useful in terms, but appears also in the type of append for vectors. Therefore, the *evidence* language introduces a new style of 'shared' functions, which may occur in types and terms.

Shared functions may appear as arguments at phase Π, so type safety will require that reduction (in the operational semantics for shared terms) implies propositional equality (in the language of coercion proofs). An easy way to achieve this is to give a consistent operational semantics at all phases, rather than the different semantics of term-level functions and type families possible in Haskell. The operational semantics will be given in Section 6.4.

Crucially, shared functions applications $f(\delta)$ must be saturated, to distinguish function application from normal application. This retains the injectivity of type-level application from System $\mathrm{F_C}$, and avoids introducing type-level λ-abstractions, which would complicate type inference.

The signature Σ contains function declarations $f\,[\Delta] :^{\Phi} \kappa$ that record the phase of the function, the telescope Δ of arguments, and the resulting type κ, which may depend on Δ. For example, the $(+)$ function is declared at phase Π (because it can be used in runtime terms and in type expressions) with telescope $x :^{\curlywedge} \mathbb{N}, y :^{\curlywedge} \mathbb{N}$ and result type $\mathbb{N}$. I will write $x + y$ instead of $(+)\,(x\ , y)$.

Function definitions $f\,[\Delta] = \rho :^{\Phi} \kappa$ are separate from declarations, because the body ρ may call f recursively. They are expanded eagerly, with a call-by-name semantics, and any pattern matching must be performed by explicit case expressions (as discussed in the Subsection 6.2.4).

Since functions are not guaranteed to terminate, they may not appear at phase $\Box$, which needs to be kept consistent. This means that type safety will not depend on strong normalisation of functions used in types, although they might lead to non-termination of type inference for the source language, just as with type families in Haskell. Of course, it is possible to impose conditions that guarantee termination for a class of programs, as in Agda.

Consider the type of the function

```
vsplitAt :: ∀ a (n :: ℕ) . Π (m :: ℕ) → Vec (m + n) a → (Vec m a, Vec n a)
vsplitAt Zero       xs          = (Nil, xs)
vsplitAt (Suc m) (Cons x xs) = (Cons x ys, zs)
   where (ys, zs) = vsplitAt m xs
```

This type applies the function $(+)$ to arguments at phases $\forall$ and Π respectively, building a result at phase $\forall$. As with promoted constructors, this is possible due to the relativisation operator, applied to the function's telescope by the rule

$$\frac{\Sigma \ni f\,[\Delta] :^{\Phi} \kappa \qquad \Gamma \vdash \delta \ :\ \Delta \mathbin{/\!\!/} \Psi \qquad \Phi \hookrightarrow \Psi}{\Gamma \vdash f(\delta)\ :^{\Psi}\ [\delta/\Delta]\,\kappa}$$

Phases act on telescopes, written $\Delta \mathbin{/\!\!/} \Psi$, thus:

$$\begin{array}{rcl} \cdot \mathbin{/\!\!/} \Psi & \mapsto & \cdot \\ (\Delta, a :^{\Phi} \kappa) \mathbin{/\!\!/} \Psi & \mapsto & (\Delta \mathbin{/\!\!/} \Psi), a :^{(\Phi \mathbin{/\!\!/} \Psi)} \kappa \end{array}$$

This operation will also be used in the typing rule for dependent case branches, so the arguments to the constructor will be available statically.

6.2.4 Dependent case analysis

The replicate and vsplitAt functions rely on dependent pattern matching: case analysis on the $\mathbb{N}$ argument establishes that the result is type-correct. That is, it allows 'learning by testing' (Altenkirch et al., 2005). Recall the replicate example, reformulated to use a dependent case expression:

$$\begin{array}{l}
\mathsf{replicate} :: \forall\, a :: * \,.\, \Pi\; n :: \mathbb{N} \to a \to \mathsf{Vec}\; a\; n \\
\mathsf{replicate}\; n\; x = \textbf{dcase}\; n\; \textbf{of} \\
\quad \mathsf{Zero} \quad\;\; \to \mathsf{Nil} \\
\quad \mathsf{Suc}\; m \to \mathsf{Cons}\; x\; (\mathsf{replicate}\; m\; x)
\end{array}$$

In the Zero branch, Nil needs to have type Vec a n; this is possible because a local constraint $n \sim$ Zero is brought into scope. Similarly, in the Suc branch, a local constraint $n \sim$ Suc m is available. In general, each branch can make use of the information that the scrutinee is equal to the matched constructor.

This resembles a GADT pattern match (see Subsection 5.1.3, page 92). Indeed the singleton construction makes use of GADTs to encode dependent pattern matching. The crucial difference is that here the constraint is not an implicit argument to the data constructor, as with GADTs, but is separately brought into scope by the dependent case expression.[6]

The **dcase** construct of the *evidence* language supports dependent case analysis. In the typing rule for dependent case branches, an additional variable is brought into scope: a proof that the scrutinee is equal to the matched constructor. The scrutinee must be well-typed at phase $\forall$, since it will appear in an equality type. This is ensured by checking it at phase $\Pi \mathbin{/\!\!/} \Psi$ where Ψ is the phase of the case expression; the access policy gives $\Pi \mathbin{/\!\!/} \Psi \hookrightarrow \forall$. A non-dependent **case** construct is also available, allowing runtime expressions to appear as scrutinees.

Thus the body of replicate could be translated into the *evidence* term

$$\begin{array}{l}
\textbf{dcase}\; n\; \textbf{of} \\
\quad \mathsf{Zero}\; (c :^{\Box}\; n \sim \mathsf{Zero}) \qquad\qquad\quad\; \to \mathsf{Nil}\; a\; n\; c \\
\quad \mathsf{Suc}\; (m :^{\Pi}\; \mathbb{N}, c :^{\Box}\; n \sim \mathsf{Suc}\; m) \to \mathsf{Cons}\; a\; n\; m\; x\; (\mathsf{replicate}\; (a, m, x))\; c
\end{array}$$

where the types of the constructors, after the GADT translation, are:

$$\begin{array}{l}
\mathsf{Nil} \quad\; : (a :^{\forall} *, n :^{\forall} \mathbb{N}, c :^{\Box}\; n \sim \mathsf{Zero}) \to \mathsf{Vec}\; a\; n \\
\mathsf{Cons} : (a :^{\forall} *, n :^{\forall} \mathbb{N}, m :^{\forall} \mathbb{N}, \\
\qquad\qquad x :^{\curlywedge}\; a, xs :^{\curlywedge}\; \mathsf{Vec}\; a\; m, c :^{\Box}\; n \sim \mathsf{Suc}\; m) \to \mathsf{Vec}\; a\; n
\end{array}$$

The mechanism for reconstructing the implicit arguments will be discussed in Chapter 7. Note that the recursive call to replicate uses an alternative syntax, with a comma-separated vector of arguments, to emphasise the fact that it is a fully-applied shared function (see Subsection 6.2.3).

[6]Of course, a GADT may appear as the scrutinee type in a dependent case expression.

6.3 Type system

The *evidence* type system consists of the following judgments:

$\Sigma \vdash \mathbf{sig}$	Σ is a valid signature
$\Gamma \vdash \mathbf{ctx}$	Γ is a valid context
$\Gamma \vdash \rho :^{\Psi} \kappa$	ρ has type κ at phase Ψ in context Γ
$\Gamma \vdash \mathit{br} :^{\Psi} \upsilon \blacktriangleright \tau$	br is a case branch with scrutinee type υ, result type τ
$\Gamma \vdash \mathit{br} :^{\Psi} (\varepsilon : \upsilon) \blacktriangleright \tau$	br is a dependent case branch, scrutinee $\varepsilon : \upsilon$, result τ
$\Gamma \vdash \delta : \Delta$	δ is a vector in Δ
$\Gamma \vdash^{\mathrm{tc}} \omega : \Delta$	ω is a telescoped coercion with domain Δ

All judgments except $\Sigma \vdash \mathbf{sig}$ are implicitly parameterised by a signature Σ.

6.3.1 Well-formed signatures and contexts

Figure 6.5 defines the signature and context well-formedness judgments. These check that each declared name is fresh (written #) and well-typed in the appropriate sense, and that it is introduced at suitable phase. The signature Σ contains global top-level definitions: type constructors D, data constructors K, functions f and axioms C. The context Γ binds variables.

Type constructors are always static, whereas data constructors may be static, dynamic or shared (but not proofs). A Haskell-style datatype declaration corresponds to a single type constructor and a number of data constructors. System $\mathrm{F_C}$ encodes datatypes in the same way, although my use of telescopes Δ to collect type and term bindings represents a slight simplification. For GADT data constructors, the return type is an application of the type constructor to variables, but the telescope will include constraints on the variables.

As discussed in Subsection 6.2.3, functions f are separated into declarations $f\,[\Delta] :^{\Phi} \kappa$ and definitions $f\,[\Delta] = \rho :^{\Phi} \kappa$, with the declaration appearing before the definition in the signature, in order to permit general recursion. They have a telescope Δ of parameters, which the result type κ may depend on. Function applications will always be saturated (written $f(\delta)$ where δ is a vector in Δ).

Axioms $C :^{\square} \varphi$ assert that all closed instances of the proposition φ hold. For example, the proposition $(a :^{\forall} \mathbb{N}, b :^{\forall} \mathbb{N}) \to (a + b) \sim (b + a)$ asserts that addition is commutative, but this fact is not otherwise derivable as a coercion (because the proof language does not permit induction). Adding this as an axiom makes it available when generating evidence for equalities. Since the exact form of proofs is unimportant, much like in Observational Type Theory (Altenkirch et al., 2007), any consistent axiom may be added without affecting computation.

$\boxed{\Sigma \vdash \mathbf{sig}}$ *(Σ is a valid signature)*

$$\frac{}{\cdot \vdash \mathbf{sig}} \qquad \frac{\Sigma \vdash \mathbf{sig} \qquad \mathsf{D}\#\Sigma \qquad \overline{a_i :^{\forall} \kappa_i}^{\,i} \vdash \mathbf{ctx}}{\Sigma, \mathsf{D} :^{\forall} (\overline{a_i :^{\forall} \kappa_i}^{\,i}) \to * \vdash \mathbf{sig}} \qquad \frac{\Sigma \vdash \mathbf{sig} \qquad \mathsf{K}\#\Sigma \qquad \Phi \neq \Box \qquad \overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta \vdash \mathsf{D}\,\overline{a_i}^{\,i} :^{\forall} *}{\Sigma, \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i} \vdash \mathbf{sig}}$$

$$\frac{f\#\Sigma \qquad \Phi \neq \Box \qquad \Sigma \vdash \mathbf{sig} \qquad \Delta \vdash \kappa :^{\forall} *}{\Sigma, f\,[\Delta] :^{\Phi} \kappa \vdash \mathbf{sig}} \qquad \frac{\Sigma \ni f\,[\Delta] :^{\Phi} \kappa \qquad \Sigma \vdash \mathbf{sig} \qquad \Delta \vdash \rho :^{\Phi} \kappa}{\Sigma, f\,[\Delta] = \rho :^{\Phi} \kappa \vdash \mathbf{sig}}$$

$$\frac{\Sigma \vdash \mathbf{sig} \qquad C\#\Sigma \qquad \cdot \vdash \varphi :^{\forall} *}{\Sigma, C :^{\Box} \varphi \vdash \mathbf{sig}}$$

$\boxed{\Gamma \vdash \mathbf{ctx}}$ *(Γ is a valid context)*

$$\frac{\Sigma \vdash \mathbf{sig}}{\cdot \vdash \mathbf{ctx}} \qquad \frac{a\#\Gamma \qquad \Gamma \vdash \kappa :^{\forall} * \qquad \Phi \neq \Box}{\Gamma, a :^{\Phi} \kappa \vdash \mathbf{ctx}} \qquad \frac{c\#\Gamma \qquad \Gamma \vdash \varphi :^{\forall} *}{\Gamma, c :^{\Box} \varphi \vdash \mathbf{ctx}}$$

Figure 6.5: Validity of signatures and contexts

In contexts, the validity rules require that the type of each variable is well-kinded. They distinguish between coercion variables c and other variables a, because the type of a coercion variable must be syntactically a proposition φ rather than an arbitrary type κ, for technical reasons in the consistency proof.

6.3.2 Well-typed terms

Figure 6.6 defines the expression typing judgment $\Gamma \vdash \rho :^{\Psi} \kappa$, meaning that ρ is an expression of type κ when checked at phase Ψ. The same judgment is given additional rules in Figure 6.7, as discussed in the next subsection. The variable, application and function rules were introduced in Section 6.2.

Type constructors D and data constructors K are available as declared in the signature. In addition, there are two built-in constants: $*$ (the kind of types) and heterogeneous equality ($\sim$). I will write the equality relation infix, using the syntactic sugar introduced in Subsection 6.3.5.

The rule for casts $\rho \triangleright \gamma$ explicitly changes the type of ρ using the coercion γ. This replaces the conversion rule, which would prevent decidability of typechecking since type expressions are not strongly normalising. Casting a proof uses a separate rule, described in the next subsection.

$\boxed{\Gamma \vdash \rho :^{\Psi} \kappa}$ *(ρ has type κ at phase Ψ)*

$$\frac{\begin{array}{l}\Gamma \vdash \mathbf{ctx} \\ \Gamma \ni a :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash a :^{\Psi} \kappa} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathbf{ctx} \\ \Sigma \ni \mathsf{D} :^{\forall} \kappa\end{array}}{\Gamma \vdash \mathsf{D} :^{\forall} \kappa} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathbf{ctx} \\ \Sigma \ni \mathsf{K} :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash \mathsf{K} :^{\Psi} \kappa}$$

$$\frac{\begin{array}{l}\Sigma \ni f\,[\Delta] :^{\Phi} \kappa \\ \Gamma \vdash \delta : \Delta \mathbin{/\!\!/} \Psi \\ \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash f(\delta) :^{\Psi} [\delta/\Delta]\,\kappa} \qquad \frac{\begin{array}{l}\Gamma \vdash \rho :^{\Psi} (a :^{\Phi} \kappa_1) \to \kappa_2 \\ \Gamma \vdash \rho' :^{\Phi \mathbin{/\!\!/} \Psi} \kappa_1\end{array}}{\Gamma \vdash \rho^{\Phi} \rho' :^{\Psi} [\rho'/a]\,\kappa_2} \qquad \frac{\begin{array}{l}\Gamma \vdash \kappa :^{\forall} * \\ \Gamma, a :^{\Phi} \kappa \vdash \tau :^{\forall} *\end{array}}{\Gamma \vdash (a :^{\Phi} \kappa) \to \tau :^{\forall} *}$$

$$\frac{\begin{array}{l}\Gamma \vdash \rho :^{\Psi} \kappa \\ \Gamma \vdash \gamma :^{\square} \kappa \sim \kappa' \\ \Psi \neq \square\end{array}}{\Gamma \vdash \rho \triangleright \gamma :^{\Psi} \kappa'} \qquad \frac{\Gamma \vdash \mathbf{ctx}}{\Gamma \vdash * :^{\forall} *} \qquad \frac{\Gamma \vdash \mathbf{ctx}}{\Gamma \vdash (\sim) :^{\forall} (a :^{\forall} *) \to (b :^{\forall} *) \to a \to b \to *}$$

$$\frac{\Gamma, a :^{\Phi} \kappa \vdash \rho :^{\Omega} \tau}{\Gamma \vdash \Lambda a :^{\Phi} \kappa \,.\, \rho :^{\Omega} (a :^{\Phi} \kappa) \to \tau} \qquad \frac{\begin{array}{l}\Gamma \vdash \rho :^{\Psi} \upsilon \qquad \Psi \neq \square \\ \Gamma \vdash br_0 :^{\Psi} \upsilon \blacktriangleright \tau \quad \ldots \quad \Gamma \vdash br_n :^{\Psi} \upsilon \blacktriangleright \tau\end{array}}{\Gamma \vdash \mathbf{case}\,\rho\,\mathbf{of}\,br_0 \ldots br_n :^{\Psi} \tau}$$

$$\frac{\begin{array}{l}\Gamma \vdash \varepsilon :^{\Pi \mathbin{/\!\!/} \Psi} \upsilon \qquad \Psi \neq \square \\ \Gamma \vdash br_0 :^{\Psi} (\varepsilon : \upsilon) \blacktriangleright \tau \quad \ldots \quad \Gamma \vdash br_n :^{\Psi} (\varepsilon : \upsilon) \blacktriangleright \tau\end{array}}{\Gamma \vdash \mathbf{dcase}\,\varepsilon\,\mathbf{of}\,br_0 \ldots br_n :^{\Psi} \tau}$$

$\boxed{\Gamma \vdash br :^{\Psi} \upsilon \blacktriangleright \tau}$ *(br is a well-typed case branch)*

$$\frac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i} \\ \Gamma, [\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\Delta \vdash \rho :^{\Psi} \tau \\ \Gamma \vdash \tau :^{\forall} * \qquad \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash \mathsf{K}\,([\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\Delta) \to \rho :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau}$$

$\boxed{\Gamma \vdash br :^{\Psi} (\varepsilon : \upsilon) \blacktriangleright \tau}$ *(br is a well-typed dependent case branch)*

$$\frac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i} \\ \Delta' = [\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\Delta \mathbin{/\!\!/} \Pi, c :^{\square} \varepsilon \sim (\mathsf{K}\,\overline{\upsilon_i}^{\,i}\,\Delta) \\ \Gamma, \Delta' \vdash \rho :^{\Psi} \tau \qquad \Gamma \vdash \tau :^{\forall} * \qquad \Phi \hookrightarrow \Pi \mathbin{/\!\!/} \Psi\end{array}}{\Gamma \vdash \mathsf{K}\,\Delta' \to \rho :^{\Psi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau}$$

Figure 6.6: Typing rules

Abstractions $\Lambda a :^{\Phi} \kappa \,.\, \rho$ are available at phases $\Omega \in \{⅄, \Box\}$, but may not appear directly in types (at phase $\forall$ or Π).

Case expressions were discussed in Subsection 6.2.4. They may not occur in proofs, since nontermination might result. Case branches are checked using two auxiliary judgments, $\Gamma \vdash \mathit{br} \;\; :^{\Phi} \upsilon \blacktriangleright \tau$ and $\Gamma \vdash \mathit{br} \;\; :^{\Phi} (\varepsilon : \upsilon) \blacktriangleright \tau$, meaning that the branch br matches a scrutinee of type υ and returns an expression of type τ at phase Φ. The second judgment makes an extra assumption, that the scrutinee is equal to ε, available in the branch.

6.3.3 Well-typed coercions

Figure 6.7 adds rules for well-typed coercions to the typing judgment of Figure 6.6. Thus variables, applications and abstractions are available for coercions as well as other expressions. Coercions have a specialised version of the cast rule $\gamma \triangleright \eta$, which ensures that the result of the cast is syntactically a proposition φ'.

The coercion syntax includes the general-purpose congruence rule $\mathbf{resp}\, \omega\, \Delta\, \tau$,[7] making various structural rules derivable by asserting that $[\overleftarrow{\omega}/\Delta]\, \tau \sim [\overrightarrow{\omega}/\Delta]\, \tau$. In particular, it means that reflexivity, symmetry, transitivity and congruence for dynamic functions are all derivable rules, as shown in Figure 6.9.

Making congruence an explicit coercion form avoids the need to prove its admissibility (called the 'lifting theorem' in previous work on System $\mathrm{F_C}$) and reduces the number of structural rules required. The system is proof-irrelevant so the exact form of the coercion language is unimportant. The formulation given here is not general enough to prove the congruence rules for application ($\mathbf{conga}^{\Upsilon}\, \gamma\, \eta$ and $\mathbf{conga}^{\Box}\, \gamma\, (\eta_1,\, \eta_2)$), quantification ($\mathbf{cong}\, \Phi\, \eta\, \gamma$) and case analysis ($\mathbf{cong}\, (\mathbf{d})\mathbf{case}\, \gamma\, \overline{\eta_i}^{\,i}$), so these must be present explicitly.[8]

The congruence rule for case analysis relies on the auxiliary definitions in Figure 6.8 for computing the equality proposition between two case branches. The operation $\Delta \bowtie \Delta'$ combines two telescopes that bind corresponding variables, but may assign them types that are only propositionally equal. It produces a single telescope that quantifies over variables of both types and a proof of their equality. Equality between two case branches $\mathit{br} \approx \mathit{br}'$ takes the proposition that the branch results are equal and quantifies over the combined telescope.

Just like in System $\mathrm{F_C}$, injectivity rules $\mathbf{left}\, \gamma$ and $\mathbf{right}\, \gamma$ allow decomposition

[7]It is sometimes useful to optimise coercions (such as replacing a coercion whose subterms are all reflexive with a direct appeal to reflexivity). This is possible with the **resp** formulation, but may be easier if all the structural rules are introduced separately.

[8]A more general congruence rule, allowing local parameterisation in the telescope, could be used to remove these.

$\boxed{\Gamma \vdash \gamma :^{\Box} \varphi}$ *(γ has type φ at phase $\Box$)*

$$\frac{\Gamma \vdash^{\mathsf{tc}} \omega : \Delta \qquad \Gamma, \Delta \vdash \tau :^{\forall} \kappa}{\Gamma \vdash \mathbf{resp}\, \omega\, \Delta\, \tau :^{\Box} [\overleftarrow{\omega}/\Delta]\tau \sim [\overrightarrow{\omega}/\Delta]\tau} \qquad \frac{\Gamma \vdash \gamma :^{\Box} \tau\,\tau' \sim \upsilon\,\upsilon'}{\Gamma \vdash \mathbf{left}\,\gamma :^{\Box} \tau \sim \upsilon}$$

$$\frac{\Gamma \vdash \gamma :^{\Box} \tau\,\tau' \sim \upsilon\,\upsilon'}{\Gamma \vdash \mathbf{right}\,\gamma :^{\Box} \tau' \sim \upsilon'} \qquad \frac{\Gamma \vdash \gamma :^{\Box} ((a_1 :^{\Phi} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Phi} \kappa_2) \to \tau_2)}{\Gamma \vdash \mathbf{left}\,\gamma :^{\Box} \kappa_1 \sim \kappa_2}$$

$$\frac{\Gamma \vdash \gamma :^{\Box} (\kappa_1 \to \tau_1) \sim (\kappa_2 \to \tau_2)}{\Gamma \vdash \mathbf{right}\,\gamma :^{\Box} \tau_1 \sim \tau_2} \qquad \frac{\Gamma \vdash \mathbf{ctx} \qquad \Sigma \ni C :^{\Box} \varphi}{\Gamma \vdash C :^{\Box} \varphi}$$

$$\frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} (\tau_1 : (a_1 :^{\Upsilon} \kappa_1) \to \kappa_1') \sim (\tau_2 : (a_2 :^{\Upsilon} \kappa_2) \to \kappa_2') \\ \Gamma \vdash \eta :^{\Box} (\upsilon_1 : \kappa_1) \sim (\upsilon_2 : \kappa_2)\end{array}}{\Gamma \vdash \mathbf{conga}^{\Upsilon}\,\gamma\,\eta :^{\Box} (\tau_1\,\upsilon_1) \sim (\tau_2\,\upsilon_2)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} (\tau_1 : (c_1 :^{\Box} \varphi_1) \to \kappa_1) \sim (\tau_2 : (c_2 :^{\Box} \varphi_2) \to \kappa_2) \\ \Gamma \vdash \eta_1 :^{\Box} \varphi_1 \qquad \Gamma \vdash \eta_2 :^{\Box} \varphi_2\end{array}}{\Gamma \vdash \mathbf{conga}^{\Box}\,\gamma\,(\eta_1, \eta_2) :^{\Box} (\tau_1\,\eta_1) \sim (\tau_2\,\eta_2)}$$

$$\frac{\begin{array}{l}\Gamma, a_1 :^{\Upsilon} \kappa_1 \vdash \tau_1 :^{\forall} * \qquad \Gamma, a_2 :^{\Upsilon} \kappa_2 \vdash \tau_2 :^{\forall} * \qquad \Gamma \vdash \eta :^{\Box} \kappa_1 \sim \kappa_2 \\ \Gamma \vdash \gamma :^{\Box} (a_1 :^{\Upsilon} \kappa_1, a_2 :^{\Upsilon} \kappa_2, c :^{\Box} a_1 \sim a_2) \to \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash \mathbf{cong}\,\Upsilon\,\eta\,\gamma :^{\Box} ((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2)}$$

$$\frac{\begin{array}{l}\Gamma, c_1 :^{\Box} \varphi_1 \vdash \tau_1 :^{\forall} * \qquad \Gamma, c_2 :^{\Box} \varphi_2 \vdash \tau_2 :^{\forall} * \\ \Gamma \vdash \eta :^{\Box} \varphi_1 \sim \varphi_2 \qquad \Gamma \vdash \gamma :^{\Box} (c_1 :^{\Box} \varphi_1, c_2 :^{\Box} \varphi_2) \to \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash \mathbf{cong}\,\Box\,\eta\,\gamma :^{\Box} ((c_1 :^{\Box} \varphi_1) \to \tau_1) \sim ((c_2 :^{\Box} \varphi_2) \to \tau_2)}$$

$$\frac{\Gamma \vdash \gamma :^{\Box} \varepsilon \sim \varepsilon' \qquad \Gamma \vdash \eta_0 :^{\Box} br_0 \approx br_0' \;\ldots\; \Gamma \vdash \eta_n :^{\Box} br_n \approx br_n'}{\Gamma \vdash (\mathbf{cong}\,(\mathbf{d})\mathbf{case}\,\gamma\,\overline{\eta_i}^{\,i}) :^{\Box} ((\mathbf{d})\mathbf{case}\,\varepsilon\,\mathbf{of}\,\overline{br_i}^{\,i}) \sim ((\mathbf{d})\mathbf{case}\,\varepsilon'\,\mathbf{of}\,\overline{br_i'}^{\,i})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} \varphi \\ \Gamma \vdash \eta :^{\Box} \varphi \sim \varphi'\end{array}}{\Gamma \vdash \gamma \rhd \eta :^{\Box} \varphi'} \qquad \frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} ((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2) \\ \Gamma \vdash \eta :^{\Box} (\upsilon_1 : \kappa_1) \sim (\upsilon_2 : \kappa_2)\end{array}}{\Gamma \vdash \gamma @ \eta :^{\Box} [\upsilon_1/a_1]\tau_1 \sim [\upsilon_2/a_2]\tau_2}$$

$$\frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} ((c_1 :^{\Box} \varphi_1) \to \tau_1) \sim ((c_2 :^{\Box} \varphi_2) \to \tau_2) \\ \Gamma \vdash \eta_1 :^{\Box} \varphi_1 \qquad \Gamma \vdash \eta_2 :^{\Box} \varphi_2\end{array}}{\Gamma \vdash \gamma @ (\eta_1, \eta_2) :^{\Box} [\eta_1/c_1]\tau_1 \sim [\eta_2/c_2]\tau_2} \qquad \frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} (\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2) \\ \Gamma \vdash \eta :^{\Box} \kappa_1 \sim \upsilon\end{array}}{\Gamma \vdash \mathbf{coh}\,\gamma\,\eta :^{\Box} \tau_1 \rhd \eta \sim \tau_2}$$

$$\frac{\Gamma \vdash \tau :^{\forall} \kappa \qquad \Gamma \vdash \tau' :^{\forall} \kappa \qquad \tau \xrightarrow{\mathsf{kpush}} \tau'}{\Gamma \vdash \mathbf{step}\,\tau :^{\Box} \tau \sim \tau'} \qquad \frac{\Gamma \vdash \gamma :^{\Box} (\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2)}{\Gamma \vdash \mathbf{kind}\,\gamma :^{\Box} \kappa_1 \sim \kappa_2}$$

Figure 6.7: Well-typed coercions

$$(\mathsf{K}\,\Delta \to \tau) \approx (\mathsf{K}\,\Delta' \to \tau') \quad \mapsto \quad ((\Delta \mathbin{\wedge\!\!\wedge} \Delta')) \to (\tau \sim \tau')$$

$$\begin{array}{rcl}
\cdot \mathbin{\wedge\!\!\wedge} \cdot & \mapsto & \cdot \\
\Gamma, a :^{\Upsilon} \kappa \mathbin{\wedge\!\!\wedge} \Gamma', a' :^{\Upsilon} \kappa' & \mapsto & \Gamma \mathbin{\wedge\!\!\wedge} \Gamma', a :^{\Upsilon} \kappa, a' :^{\Upsilon} \kappa', c :^{\square} a \sim a' \\
\Gamma, x :^{\Omega} \tau \mathbin{\wedge\!\!\wedge} \Gamma', x' :^{\Omega} \tau' & \mapsto & \Gamma \mathbin{\wedge\!\!\wedge} \Gamma', x :^{\Omega} \tau, x' :^{\Omega} \tau'
\end{array}$$

Figure 6.8: Evidence for equality of case branches

$\boxed{\Gamma \vdash \gamma :^{\square} \varphi}$ *(derivable rules: γ has type φ at phase $\square$)*

$$\frac{\Gamma \vdash \tau :^{\forall} \kappa}{\Gamma \vdash \langle \tau \rangle :^{\square} \tau \sim \tau} \qquad \frac{\Gamma \vdash \gamma :^{\square} (\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2)}{\Gamma \vdash \mathbf{sym}\,\gamma :^{\square} (\tau_2 : \kappa_2) \sim (\tau_1 : \kappa_1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma_1 :^{\square} (\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2) \\ \Gamma \vdash \gamma_2 :^{\square} (\tau_2 : \kappa_2) \sim (\tau_3 : \kappa_3)\end{array}}{\Gamma \vdash \gamma_1 ; \gamma_2 :^{\square} (\tau_1 : \kappa_1) \sim (\tau_3 : \kappa_3)} \qquad \frac{\Gamma \vdash \eta :^{\square} \tau_1 \sim \tau_2 \qquad \Gamma \vdash \gamma :^{\square} \upsilon_1 \sim \upsilon_2}{\Gamma \vdash \mathbf{cong}\ \text{⅄}\ \eta\,\gamma :^{\square} (\tau_1 \to \upsilon_1) \sim (\tau_2 \to \upsilon_2)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \gamma :^{\square} (\tau_1 : \kappa_1 \to \upsilon_1) \sim (\tau_2 : \kappa_2 \to \upsilon_2) \\ \Gamma \vdash \eta :^{\square} (\tau_1' : \kappa_1) \sim (\tau_2' : \kappa_2)\end{array}}{\Gamma \vdash \mathbf{conga}^{\text{⅄}}\,\gamma\,\eta :^{\square} (\tau_1\,\tau_1') \sim (\tau_2\,\tau_2')} \qquad \frac{\Gamma \vdash \gamma :^{\square} \mathsf{H}\,\overline{\tau_i}^{\,i} \sim \mathsf{H}\,\overline{\upsilon_i}^{\,i}}{\Gamma \vdash \mathbf{nth}^{i}\,\gamma :^{\square} \tau_i \sim \upsilon_i}$$

$$\begin{array}{rcl}
\langle \tau \rangle & \mapsto & \mathbf{resp} \cdot \cdot\ \tau \\
\mathbf{sym}\,\gamma & \mapsto & \langle \tau_1 \rangle \triangleright \mathbf{resp}\,((\kappa_1, \kappa_2, \mathbf{kind}\,\gamma), (\tau_1, \tau_2, \gamma))\,(a :^{\forall} *, b :^{\forall} a)\,(b \sim \tau_1) \\
\gamma_1 ; \gamma_2 & \mapsto & \gamma_1 \triangleright \mathbf{resp}\,((\kappa_2, \kappa_3, \mathbf{kind}\,\gamma_2), (\tau_2, \tau_3, \gamma_2))\,(a :^{\forall} *, b :^{\forall} a)\,(\tau_1 \sim b) \\
\mathbf{cong}\ \text{⅄}\ \eta\,\gamma & \mapsto & \mathbf{resp}\,((\tau_1, \tau_2, \eta), (\upsilon_1, \upsilon_2, \gamma))\,(a :^{\forall} *, b :^{\forall} *)\,(a \to b) \\
\mathbf{conga}^{\text{⅄}}\,\gamma\,\eta & \mapsto & \mathbf{resp}\,((\kappa_1, \kappa_2, \mathbf{left}\,(\mathbf{kind}\,\gamma)), (\upsilon_1, \upsilon_2, \mathbf{right}\,(\mathbf{kind}\,\gamma)), \\
 & & \qquad (\tau_1, \tau_2, \gamma), (\tau_1', \tau_2', \eta)) \\
 & & \qquad (a :^{\forall} *, b :^{\forall} *, x :^{\forall} (a \to b), y :^{\forall} a)\ (x\,y) \\
\mathbf{nth}^{i}\,\gamma & \mapsto & \mathbf{right}(\underbrace{\mathbf{left} \ldots \mathbf{left}}_{n-i \text{ times}}\ \gamma) \quad \text{where } \overline{\tau_i}^{\,i} \text{ and } \overline{\upsilon_i}^{\,i} \text{ have } n \text{ elements}
\end{array}$$

Figure 6.9: Derivable rules for coercions

of an equation between applications or non-dependent function spaces. Instantiation rules $\gamma @ \eta$ and $\gamma @ (\eta_1, \eta_2)$ play a similar role for dependent quantifications.

As in the work of Weirich et al. (2013), heterogeneous equality uses the 'Σ-interpretation' in which an equation between expressions of different kinds implies that the kinds themselves are equal. This is witnessed by the $\mathbf{kind}\,\gamma$ coercion. Also present in their work and Observational Type Theory is the coherence rule $\mathbf{coh}\,\gamma\,\eta$, which states that casts do not change the identity of an expression.

New in the *evidence* language is the $\mathbf{step}\,\tau$ rule, making a redex equal to its reduct. Thus the operational semantics, given by the $\xrightarrow{\text{kpush}}$ relation to be defined in Section 6.4, is embedded in the propositional equality. The presence of **step** constructors means that the computation necessary to typecheck a term is finite.

6.3.4 Vectors and telescoped coercions

Figure 6.10 gives the rules for vectors and telescoped coercions. A *vector* δ contains expressions that can be substituted for a telescope Δ. Thus each expression in the vector must be checked at the appropriate phase, with the type determined by substituting for the preceding telescope.

Equality of types ($\sim$) extends to equality of vectors. A *telescoped coercion* ω represents two vectors ($\overleftarrow{\omega}$ and $\overrightarrow{\omega}$) along with proofs of equality for the type expressions they contain. Thus it consists of pairs of type expressions plus a coercion between them ($\tau\ ,\ \upsilon\ ,\ \gamma$), and pairs of terms ($e\ ,\ e'$) or coercions ($\gamma\ ,\ \gamma'$). No proof of equality is needed for runtime terms because they cannot appear in types; no proof is needed for coercions because the system is proof-irrelevant.

6.3.5 Syntactic sugar

Some convenient abbreviations are given in Figure 6.11. Just as System $\mathrm{F_C}$ formally distinguishes between type, term and coercion application, so applications $\rho^{\Phi}\rho'$ carry a phase, but this is easily recoverable from the type of ρ, so I will usually omit it. The presence of phase annotations on applications allows the erasure operation (Section 6.6) to be defined on the syntax of terms, rather than on typing derivations, but it is otherwise harmless to omit the annotations. I will write the application of an expression to a vector $\rho\,\delta$.

Since dynamic variables cannot occur in type expressions, thanks to the phase distinction, the function space $(x :^{\lambda} \tau) \to \upsilon$ may be written $\tau \to \upsilon$, as there is no possibility of x occurring in υ. The familiar notation $\lambda x : \tau \,.\, e$ is used for inhabitants of this function space.

$\boxed{\Gamma \vdash \delta : \Delta}$ *(δ is a vector in Δ)*

$$\frac{\Gamma \vdash \mathbf{ctx}}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash \delta : \Delta \quad \Gamma \vdash \rho :^{\Phi} [\delta/\Delta]\,\kappa}{\Gamma \vdash (\delta, \rho) : (\Delta, a :^{\Phi} \kappa)}$$

$\boxed{\Gamma \vdash^{\mathrm{tc}} \omega : \Delta}$ *(ω is a telescoped coercion in Δ)*

$$\frac{\Gamma \vdash \mathbf{ctx}}{\Gamma \vdash^{\mathrm{tc}} \cdot : \cdot} \qquad \frac{\Gamma \vdash^{\mathrm{tc}} \omega : \Delta \quad \Gamma \vdash \gamma :^{\square} \tau \sim \upsilon \quad \Gamma \vdash \tau :^{\Upsilon} [\overleftarrow{\omega}/\Delta]\,\kappa \quad \Gamma \vdash \upsilon :^{\Upsilon} [\overrightarrow{\omega}/\Delta]\,\kappa}{\Gamma \vdash^{\mathrm{tc}} (\omega, (\tau, \upsilon, \gamma)) : (\Delta, a :^{\Upsilon} \kappa)}$$

$$\frac{\Gamma \vdash^{\mathrm{tc}} \omega : \Delta \quad \Gamma \vdash \eta :^{\square} [\overleftarrow{\omega}/\Delta]\,\varphi \quad \Gamma \vdash \eta' :^{\square} [\overrightarrow{\omega}/\Delta]\,\varphi}{\Gamma \vdash^{\mathrm{tc}} (\omega, (\eta, \eta')) : (\Delta, c :^{\square} \varphi)} \qquad \frac{\Gamma \vdash^{\mathrm{tc}} \omega : \Delta \quad \Gamma \vdash e :^{\curlywedge} [\overleftarrow{\omega}/\Delta]\,\tau \quad \Gamma \vdash e' :^{\curlywedge} [\overrightarrow{\omega}/\Delta]\,\tau}{\Gamma \vdash^{\mathrm{tc}} (\omega, (e, e')) : (\Delta, x :^{\curlywedge} \tau)}$$

$$\begin{array}{rcl}
\overleftarrow{\cdot} & \mapsto & \cdot \\
\overleftarrow{\omega, (\tau, \upsilon, \gamma)} & \mapsto & \overleftarrow{\omega}, \tau \\
\overleftarrow{\omega, (\rho, \rho')} & \mapsto & \overleftarrow{\omega}, \rho \\
\\
\overrightarrow{\cdot} & \mapsto & \cdot \\
\overrightarrow{\omega, (\tau, \upsilon, \gamma)} & \mapsto & \overrightarrow{\omega}, \upsilon \\
\overrightarrow{\omega, (\rho, \rho')} & \mapsto & \overrightarrow{\omega}, \rho'
\end{array}$$

Figure 6.10: Vectors and telescoped coercions

$$\begin{array}{rcl}
\rho\,\rho' & \mapsto & \rho^{\Phi}\rho' \quad \text{where } \Gamma \vdash \rho :^{\Psi} (a{:}^{\Phi}\tau) \to \upsilon \\
\rho\,\delta & \mapsto & \begin{cases} \rho & \text{if } \delta = \cdot \\ (\rho\,\delta')\,\rho' & \text{if } \delta = \delta', \rho' \end{cases} \\
\tau \to \upsilon & \mapsto & (x{:}^{\curlywedge}\tau) \to \upsilon \\
\lambda x{:}\tau\,.\,e & \mapsto & \Lambda x{:}^{\curlywedge}\tau\,.\,e \\
(\tau_1{:}\kappa_1) \sim (\tau_2{:}\kappa_2) & \mapsto & (\sim)\,\kappa_1\,\kappa_2\,\tau_1\,\tau_2 \\
\tau_1 \sim \tau_2 & \mapsto & (\sim)\,\kappa_1\,\kappa_2\,\tau_1\,\tau_2 \quad \text{where } \Gamma \vdash \tau_1 :^{\forall} \kappa_1 \text{ and } \Gamma \vdash \tau_2 :^{\forall} \kappa_2
\end{array}$$

Figure 6.11: Syntactic sugar

6.3.6 Meta-theoretic properties

I will now prove some results for working with telescopes, the usual weakening and substitution lemmas, and a more liberal form of the substitution lemma required for subject reduction. Where proofs have been omitted, they are by induction on derivations. Writing a vector of arguments instead of a single argument for an application is justified by the first lemma, which I will often use implicitly.

Lemma 6.1. *Suppose* $\Gamma \vdash \rho :^{\Phi} (\Delta) \to \tau$. *Then* $\Gamma \vdash \rho\, \delta :^{\Phi} [\delta/\Delta]\, \tau$ *if and only if* $\Gamma \vdash \delta : \Delta \mathbin{/\!\!/} \Phi$.

Lemma 6.2. *If* $\Gamma \vdash^{\mathrm{tc}} \omega : \Delta$ *then* $\Gamma \vdash \overleftarrow{\omega} : \Delta$ *and* $\Gamma \vdash \overrightarrow{\omega} : \Delta$.

Lemma 6.3 (Weakening). *Let* J *be an arbitrary judgment. If* $\Gamma, \Gamma' \vdash J$ *and* $\Gamma, \Delta \vdash \mathbf{ctx}$ *where the variables in* Δ *and* Γ' *are distinct, then* $\Gamma, \Delta, \Gamma' \vdash J$.

To prove the substitution lemma, I must show that judgments are preserved under phase increases following the access policy, as described in Subsection 6.2.1.

Lemma 6.4 (Phase inclusion). *Suppose* $\Phi \hookrightarrow \Psi$.

(a) If $\Gamma \vdash \rho :^{\Phi} \kappa$ *then* $\Gamma \vdash \rho :^{\Psi} \kappa$.

(b) If $\Gamma \vdash \delta : \Delta \mathbin{/\!\!/} \Phi$ *then* $\Gamma \vdash \delta : \Delta \mathbin{/\!\!/} \Psi$.

(c) If $\Gamma \vdash^{\mathrm{tc}} \omega : \Delta \mathbin{/\!\!/} \Phi$ *then* $\Gamma \vdash^{\mathrm{tc}} \omega : \Delta \mathbin{/\!\!/} \Psi$.

Proof. By induction on derivations, following from the use of the access policy $\Phi \hookrightarrow \Psi$ for the variable rule, the right-monotonicity of $/\!\!/$ for application, and the transitivity of $\Phi \hookrightarrow \Psi$ for case analysis. □

Lemma 6.5 (Substitution). *Suppose* $\Gamma \vdash \delta : \Delta$ *and let* Γ' *be a telescope.*

(a) If $\Gamma, \Delta, \Gamma' \vdash \mathbf{ctx}$ *then* $\Gamma, [\delta/\Delta]\, \Gamma' \vdash \mathbf{ctx}$.

(b) If $\Gamma, \Delta, \Gamma' \vdash \rho :^{\Phi} \kappa$ *then* $\Gamma, [\delta/\Delta]\, \Gamma' \vdash [\delta/\Delta]\, \rho :^{\Phi} [\delta/\Delta]\, \kappa$.

(c) If $\Gamma, \Delta, \Gamma' \vdash \delta' : \Delta'$ *then* $\Gamma, [\delta/\Delta]\, \Gamma' \vdash [\delta/\Delta]\, \delta' : [\delta/\Delta]\, \Delta'$.

(d) If $\Gamma, \Delta, \Gamma' \vdash^{\mathrm{tc}} \omega : \Delta'$ *then* $\Gamma, [\delta/\Delta]\, \Gamma' \vdash^{\mathrm{tc}} [\delta/\Delta]\, \omega : [\delta/\Delta]\, \Delta'$.

Proof. By induction on derivations. The interesting case is for variables in Δ. Here δ contains an expression that is well-typed at the phase of the variable, and Lemma 6.4 means it is well-typed at the phase at which the variable is used. □

$\boxed{\Phi \propto \Psi}$ *(checking types at phase Φ may involve checking types at phase Ψ)*

$$\frac{}{\Phi \propto \Phi} \qquad \frac{}{\Phi \propto \forall} \qquad \frac{\Phi \propto \Psi}{\Phi \propto (\Phi' \mathbin{/\!/} \Psi)}$$

Figure 6.12: Relevance relation

To prove subject reduction in the presence of promotion, I will need a more liberal substitution lemma (Lemma 6.8), where the vector being substituted inhabits $\Delta \mathbin{/\!/} \Phi$ rather than Δ. This depends on the fact that if a typing judgment holds at phase Φ, then it still holds when the $/\!/\Phi$ operator is applied to part of the context. However, a straightforward inductive proof of this property fails, due to the phase change in the application rule. Instead, I must prove a more general property relating the phases involved (Lemma 6.7), using the 'relevance' relation $\Phi \propto \Psi$ defined in Figure 6.12.

Lemma 6.6. *If $\Phi \propto \Psi$ and $\Phi' \hookrightarrow \Psi$ then $\Phi' \mathbin{/\!/} \Phi \hookrightarrow \Psi$.*

Lemma 6.7 (Context for phase)**.** *Suppose $\Phi \propto \Psi$.*

(a) If $\Gamma, \Delta, \Gamma' \vdash \mathbf{ctx}$ then $\Gamma, \Delta \mathbin{/\!/} \Phi, \Gamma' \vdash \mathbf{ctx}$.

(b) If $\Gamma, \Delta, \Gamma' \vdash \rho :^{\Psi} \kappa$ then $\Gamma, \Delta \mathbin{/\!/} \Phi, \Gamma' \vdash \rho :^{\Psi} \kappa$.

(c) If $\Gamma, \Delta, \Gamma' \vdash \delta : \Delta' \mathbin{/\!/} \Psi$ then $\Gamma, \Delta \mathbin{/\!/} \Phi, \Gamma' \vdash \delta : \Delta' \mathbin{/\!/} \Psi$.

(d) If $\Gamma, \Delta, \Gamma' \vdash^{\mathrm{tc}} \omega : \Delta' \mathbin{/\!/} \Psi$ then $\Gamma, \Delta \mathbin{/\!/} \Phi, \Gamma' \vdash^{\mathrm{tc}} \omega : \Delta' \mathbin{/\!/} \Psi$.

Proof. By induction on derivations. In the variable case, if $x :^{\Phi'} \kappa \in \Delta$ and $\Phi' \hookrightarrow \Psi$, then $\Phi' \mathbin{/\!/} \Phi \hookrightarrow \Psi$ by Lemma 6.6. Thus the variable rule still applies. For application at phase Φ', the argument is well-typed at phase $\Phi' \mathbin{/\!/} \Psi$, and $\Phi \propto \Phi' \mathbin{/\!/} \Psi$ by definition, so the result follows by induction. □

Lemma 6.8 (Substitution at phase)**.** *Suppose $\Gamma \vdash \delta : \Delta \mathbin{/\!/} \Phi$ and fix Γ'.*

(a) If $\Gamma, \Delta, \Gamma' \vdash \mathbf{ctx}$ then $\Gamma, [\delta/\Delta]\,\Gamma' \vdash \mathbf{ctx}$.

(b) If $\Gamma, \Delta, \Gamma' \vdash \rho :^{\Phi} \kappa$ then $\Gamma, [\delta/\Delta]\,\Gamma' \vdash [\delta/\Delta]\,\rho :^{\Phi} [\delta/\Delta]\,\kappa$.

(c) If $\Gamma, \Delta, \Gamma' \vdash \delta' : \Delta' \mathbin{/\!/} \Phi$ then $\Gamma, [\delta/\Delta]\,\Gamma' \vdash [\delta/\Delta]\,\delta' : [\delta/\Delta]\,\Delta' \mathbin{/\!/} \Phi$.

(d) If $\Gamma, \Delta, \Gamma' \vdash^{\mathrm{tc}} \omega : \Delta' \mathbin{/\!/} \Phi$ then $\Gamma, [\delta/\Delta]\,\Gamma' \vdash^{\mathrm{tc}} [\delta/\Delta]\,\omega : [\delta/\Delta]\,\Delta' \mathbin{/\!/} \Phi$.

Proof. In each case, Lemma 6.7 gives that $\Gamma, \Delta, \Gamma' \vdash J$ implies $\Gamma, \Delta \mathbin{/\!/} \Phi, \Gamma' \vdash J$ (by reflexivity of $\propto$). Then the result follows from Lemma 6.5. □

Each judgment has associated sanity conditions, giving admissible rules:

Lemma 6.9 (Sanity conditions)**.** *Let Σ be the implicit signature.*

$$\begin{array}{rll}
\Gamma \vdash \mathbf{ctx} & \textit{implies} & \Sigma \vdash \mathbf{sig} \\
\Gamma \vdash \rho :^{\Phi} \tau & \textit{implies} & \Gamma \vdash \tau :^{\forall} * \textit{ and } \Gamma \vdash \mathbf{ctx} \\
\Gamma \vdash \delta : \Delta & \textit{implies} & \Gamma \vdash \mathbf{ctx} \\
\Gamma \vdash^{\mathrm{tc}} \omega : \Delta & \textit{implies} & \Gamma \vdash \mathbf{ctx}
\end{array}$$

Proof. By induction on derivations, using the preceding results. Consider the application rule as an example:

$$\frac{\begin{array}{l}\Gamma \vdash \rho :^{\Psi} (a :^{\Phi} \kappa_1) \to \kappa_2 \\ \Gamma \vdash \rho' :^{\Phi /\!\!/ \Psi} \kappa_1\end{array}}{\Gamma \vdash \rho^{\Phi} \rho' :^{\Psi} [\rho'/a]\, \kappa_2}$$

Induction on the first premise gives $\Gamma \vdash (a :^{\Phi} \kappa_1) \to \kappa_2 :^{\forall} *$, so $\Gamma, a :^{\Phi} \kappa_1 \vdash \kappa_2 :^{\forall} *$ by inversion. Then Lemma 6.8 gives $\Gamma \vdash [\rho'/a]\, \kappa_2 :^{\forall} *$. $\square$

6.4 Operational semantics

In this section, I will give a small-step operational semantics for expressions. The reduction rules are given in Figure 6.13. These are essentially the rules of System $\mathrm{F_C}$ (Sulzmann et al., 2007), with the addition of function definitions and dependent case analysis. The other novelty is that the rules apply to type expressions as well as terms.

The syntax of *values* v and *value types* ξ is:

$$\begin{array}{lll}
v & ::= & \mathsf{H}\,\delta \mid (a :^{\Phi} \kappa) \to \tau \mid \Lambda a :^{\Phi} \kappa \,.\, e \\
\xi & ::= & \mathsf{H}\,\psi \mid (a :^{\Phi} \kappa) \to \tau
\end{array}$$

A value type is a value that has kind $*$ at phase $\forall$ (so λ-abstraction is excluded). In the usual System $\mathrm{F_C}$ fashion, expressions reduce to values that may be wrapped in a coercion, so there are rules to push coercions out of the way when they would otherwise prevent reduction. Of particular note is the push rule for the scrutinee of a case expression, described in Subsection 6.4.1.

The same rules apply to phases $\forall$, Π and $\curlywedge$, but coercions (at phase $\square$) are not evaluated. For type expressions, evidence that the redex is equal to the reduct may be required. The usual practice in dependent type theory is to build reduction into the definitional equality, but here there is no guarantee that

$\boxed{\rho \longrightarrow \rho'}$ *(ρ reduces to ρ' in one step)*

$$\frac{\rho \longrightarrow \rho'}{\rho \triangleright \eta \longrightarrow \rho' \triangleright \eta} \qquad \frac{\rho \longrightarrow \rho'}{\rho\, \rho'' \longrightarrow \rho'\, \rho''} \qquad \frac{\rho \xrightarrow{\text{kpush}} \rho'}{\mathbf{case}\, \rho\, \mathbf{of}\, \overline{br_j}^{\,j} \longrightarrow \mathbf{case}\, \rho'\, \mathbf{of}\, \overline{br_j}^{\,j}}$$

$$\frac{\begin{array}{l} \varepsilon \xrightarrow{\text{kpush}} \varepsilon' \\ br'_0 = br_0 \triangleright \mathbf{step}\, \varepsilon \quad ... \quad br'_n = br_n \triangleright \mathbf{step}\, \varepsilon \end{array}}{\mathbf{dcase}\, \varepsilon\, \mathbf{of}\, br_0 \ldots br_n \longrightarrow \mathbf{dcase}\, \varepsilon'\, \mathbf{of}\, br'_0 \ldots br'_n} \qquad \frac{\mathsf{K}\, \Delta \to \rho \in \overline{br_i}^{\,i}}{\mathbf{case}\, \mathsf{K}\, \psi\, \delta\, \mathbf{of}\, \overline{br_i}^{\,i} \longrightarrow [\delta/\Delta]\, \rho}$$

$$\frac{\mathsf{K}\, \Delta \to \rho \in \overline{br_i}^{\,i}}{\mathbf{dcase}\, \mathsf{K}\, \psi\, \delta\, \mathbf{of}\, \overline{br_i}^{\,i} \longrightarrow [(\delta, \langle \mathsf{K}\, \psi\, \delta \rangle)/\Delta]\, \rho} \qquad \frac{\Sigma \ni f\, [\Delta] = \rho :^{\Phi} \kappa}{f(\delta) \longrightarrow [\delta/\Delta]\, \rho}$$

$$\frac{}{(\Lambda a :^{\Phi} \kappa\, .\, e)^{\Phi} \rho \longrightarrow [\rho/a]\, e} \qquad \frac{\begin{array}{l} \Gamma \vdash \gamma\ :^{\Box}\ ((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2) \\ \gamma_0 = \mathbf{sym}\, (\mathbf{left}\, \gamma) \qquad\quad \gamma_1 = \gamma@(\mathbf{coh}\, \langle \tau \rangle\, \gamma_0) \end{array}}{(\mathrm{v} \triangleright \gamma)^{\Upsilon} \tau \longrightarrow \mathrm{v}^{\Upsilon} (\tau \triangleright \gamma_0) \triangleright \gamma_1}$$

$$\frac{\begin{array}{l} \Gamma \vdash \gamma\ :^{\Box}\ ((c_1 :^{\Box} \varphi_1) \to \tau_1) \sim ((c_2 :^{\Box} \varphi_2) \to \tau_2) \\ \gamma_0 = \mathbf{sym}\, (\mathbf{left}\, \gamma) \qquad\quad \gamma_1 = \gamma@(\eta \triangleright \gamma_0, \eta) \end{array}}{(\mathrm{v} \triangleright \gamma)^{\Box} \eta \longrightarrow \mathrm{v}^{\Box} (\eta \triangleright \gamma_0) \triangleright \gamma_1}$$

$$\frac{\begin{array}{l} \Gamma \vdash \gamma\ :^{\Box}\ ((a_1 :^{\curlywedge} \kappa_1) \to \tau_1) \sim ((a_2 :^{\curlywedge} \kappa_2) \to \tau_2) \\ \gamma_0 = \mathbf{sym}\, (\mathbf{left}\, \gamma) \qquad\quad \gamma_1 = \mathbf{right}\, \gamma \end{array}}{(\mathrm{v} \triangleright \gamma)^{\curlywedge} \rho \longrightarrow \mathrm{v}^{\curlywedge} (\rho \triangleright \gamma_0) \triangleright \gamma_1} \qquad \frac{}{(\mathrm{v} \triangleright \gamma) \triangleright \gamma' \longrightarrow \mathrm{v} \triangleright (\gamma; \gamma')}$$

$\boxed{\rho \xrightarrow{\text{kpush}} \rho'}$ *(ρ reduces to ρ' as the scrutinee of a case expression)*

$$\frac{\begin{array}{l} \Gamma \vdash \gamma\ :^{\Box}\ \mathsf{D}\, \overline{\tau_i}^{\,i} \sim \mathsf{D}\, \overline{\upsilon_i}^{\,i} \\ \Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\, \overline{a_i}^{\,i} \\ \omega = \overline{(\tau_i, \upsilon_i, \mathbf{nth}^i\, \gamma)}^{\,i}\ :\ \overline{a_i :^{\forall} \kappa_i}^{\,i}\ \prec\ \delta : \Delta \end{array}}{(\mathsf{K}\, \overline{\tau_i}^{\,i}\, \delta) \triangleright \gamma \xrightarrow{\text{kpush}} \mathsf{K}\, \overline{\upsilon_i}^{\,i}\, \overrightarrow{\omega}} \qquad \frac{\rho \longrightarrow \rho'}{\rho \xrightarrow{\text{kpush}} \rho'}$$

Figure 6.13: Operational semantics for shared terms

reduction will terminate, so explicit coercions are required to retain decidability of typechecking. The **step** coercion provides the necessary evidence:

$$\frac{\Gamma \vdash \tau \ :^{\forall}\ \kappa \qquad \Gamma \vdash \tau' \ :^{\forall}\ \kappa \qquad \tau \xrightarrow{\text{kpush}} \tau'}{\Gamma \vdash \mathbf{step}\,\tau \ :^{\Box}\ \tau \sim \tau'}$$

The second premise is only to ensure that the relevant sanity property, that $\tau \sim \tau'$ is well-kinded, does not depend on subject reduction.

Computing an expression can change the type of a surrounding construction to something provably equal but not syntactically identical.[9] For example, suppose $f : (a :^{\Pi} \tau) \to \upsilon$ and $\rho \longrightarrow \varepsilon$, then $f(\rho) : [\rho/a]\,\upsilon$ but $f(\varepsilon) : [\varepsilon/a]\,\upsilon$. It is not straightforward to construct the coercion manipulations required to preserve the type, especially where there is a telescope of arguments, though the **resp** congruence can be used to prove the required equations. I take a simpler approach. By giving a call-by-name semantics to shared functions and lifting case analysis to the type level, I avoid the need for reduction in an argument position.

In the rule for dependent case analysis, when the scrutinee takes a step, the branches must be coerced so that they remain type correct, since their types depend on a proof that the scrutinee is equal to the relevant constructor. Define coercion of a branch $br \rhd \gamma$ by

$$(\mathsf{K}\,(\Delta, c :^{\Box} \varepsilon \sim \mathsf{K}\,\delta) \to \rho) \rhd \gamma \qquad \mapsto \qquad \mathsf{K}\,(\Delta, c' :^{\Box} \varepsilon' \sim \mathsf{K}\,\delta) \to [\gamma; c'/c]\,\rho$$

so that $\Gamma \vdash br \ :^{\Phi}\ (\varepsilon : \upsilon) \blacktriangleright \tau$ and $\Gamma \vdash \gamma \ :^{\Box}\ \varepsilon \sim \varepsilon'$ implies $\Gamma \vdash br \rhd \gamma \ :^{\Phi}\ (\varepsilon' : \upsilon) \blacktriangleright \tau$.

6.4.1 The push rule for scrutinees

Each push rule has a similar form: given an expression with a coerced value that blocks reduction, push the coercion deeper into the term. Coerced data constructors may block reduction if they appear as the scrutinee of a case expression, so a rule is needed to push the coercion inside the arguments of the data constructor. For example, suppose $\Gamma \vdash \gamma \ :^{\Box}\ \mathsf{Maybe\,Bool} \sim \mathsf{Maybe}\,a$ and consider the scrutinee $(\mathsf{Just\,Bool\,True}) \rhd \gamma$. Pushing the coercion inside the arguments produces $\mathsf{Just}\,a\,(\mathsf{True} \rhd \mathbf{right}\,\gamma)$, an applied constructor, so the case expression can reduce.

The push rule for scrutinees is formulated as an extra reduction step, available when evaluating the scrutinee of a case expression, as shown in Figure 6.13. It is not available elsewhere as this would lead to nondeterminism: in particular, terms like $(\mathsf{K} \rhd \gamma)\,\rho$ could reduce in two different ways.

[9]In Type Theory, definitional equality includes computation, so this problem does not arise.

Given a coerced data constructor $\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta \rhd \gamma$, where $\Gamma \vdash \gamma :^{\Box} \mathsf{D}\,\overline{\tau_i}^{\,i} \sim \mathsf{D}\,\overline{\upsilon_i}^{\,i}$ and $\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i}$, each τ_i needs to be replaced with υ_i and the elements of the vector δ coerced appropriately. The telescoped coercion $\overline{(\tau_i\ ,\ \upsilon_i\ ,\ \mathbf{nth}^{i}\,\gamma)}^{\,i}$ is formed for $\overline{a_i :^{\forall} \kappa_i}^{\,i}$, then extended by δ in Δ to produce a telescoped coercion $\overline{(\tau_i, \upsilon_i, \mathbf{nth}^{i}\,\gamma)}^{\,i}, \omega$ in $\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta$ such that $\overline{\upsilon_i}^{\,i}, \overrightarrow{\omega}$ is the new vector of arguments for K.

Recall that a telescoped coercion ω represents two vectors in some telescope Γ, given by $\overleftarrow{\omega}$ and $\overrightarrow{\omega}$, plus proofs that they are equal. If Δ is a telescope extending Γ and δ is a vector in $[\overleftarrow{\omega}/\Gamma]\,\Delta$, then $\omega' = \omega : \Gamma \ \prec\ \delta : \Delta$ is such that ω, ω' is a telescoped coercion in Γ, Δ, and $\overleftarrow{\omega'} = \delta$. The telescoped coercion extension operation is defined thus:

$$\begin{array}{rcl}
\omega : \Gamma \ \prec\ \cdot : \cdot & \mapsto & \cdot \\
\omega : \Gamma \ \prec\ (\delta, \tau) : (\Delta, a :^{\Upsilon} \kappa) & \mapsto & \omega', (\tau\ ,\ \tau \rhd \gamma, \mathbf{sym}\,(\mathbf{coh}\,\langle\tau\rangle\,\gamma)) \\
 & & \quad \text{where } \omega' = \omega : \Gamma \ \prec\ \delta : \Delta \\
 & & \quad \text{and } \gamma = \mathbf{resp}\,(\omega, \omega')\,(\Gamma, \Delta)\,\kappa \\
\omega : \Gamma \ \prec\ (\delta, \rho) : (\Delta, x :^{\Omega} \tau) & \mapsto & \omega', (\rho, \rho \rhd \mathbf{resp}\,(\omega, \omega')\,(\Gamma, \Delta)\,\tau) \\
 & & \quad \text{where } \omega' = \omega : \Gamma \ \prec\ \delta : \Delta
\end{array}$$

The point of this definition, upon which subject reduction will depend, is:

Lemma 6.10 (Telescoped coercion extension)**.** *Suppose that* $\Gamma \vdash^{\mathrm{tc}} \omega_0 : \Delta_0$*,* $\Gamma \vdash \overleftarrow{\omega_0}, \delta : \Delta_0, \Delta_1$ *and* $\omega_1 = \omega_0 : \Delta_0 \ \prec\ \delta : \Delta_1$*. Then* $\Gamma \vdash^{\mathrm{tc}} \omega_0, \omega_1 : \Delta_0, \Delta_1$*.*

Proof. By induction on the definition of telescoped coercion extension. □

6.4.2 Subject reduction

The point of all the work pushing coercions around is that subject reduction is easy to prove. It is enough to inspect the reduction steps and verify that each one preserves the type up to syntactic equality.

Theorem 6.11 (Subject reduction)**.** *The operational semantics preserves types: if* $\Gamma \vdash \rho :^{\Phi} \tau$ *and* $\rho \xrightarrow{\text{kpush}} \rho'$ *then* $\Gamma \vdash \rho' :^{\Phi} \tau$*.*

Proof. By induction on the reduction step. I consider some illustrative cases.

For the β-reduction step

$$\frac{}{(\Lambda a{:}^{\Phi}\kappa\,.\,e)^{\Phi}\,\rho \longrightarrow [\rho/a]\,e}$$

inversion gives $\Gamma \vdash (\Lambda a{:}^{\Phi}\kappa\,.\,e)\,\rho :^{\curlywedge} [\rho/a]\,\tau$, so $\Gamma \vdash \Lambda a{:}^{\Phi}\kappa\,.\,e :^{\curlywedge} (a{:}^{\Phi}\,\kappa) \to \tau$ and $\Gamma \vdash \rho :^{\Phi /\!\!/ \curlywedge} \kappa$. Then inversion on the first premise gives $\Gamma, a :^{\Phi} \kappa \vdash e :^{\curlywedge} \tau$ and substitution (Lemma 6.5) gives $\Gamma \vdash [\rho/a]\,e :^{\curlywedge} [\rho/a]\,\tau$ as required.

If the scrutinee of a dependent case expression takes a step, its type is preserved by induction, and the definition of coercion for case branches ensures that the whole expression is well-typed (by the substitution lemma).

For the dependent case analysis step

$$\frac{\mathsf{K}\,\Delta \to \rho \in \overline{br_i}^{\,i}}{\mathbf{dcase}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \longrightarrow [(\delta, \langle \mathsf{K}\,\psi\,\delta\rangle)/\Delta]\,\rho}$$

from $\Gamma \vdash \mathbf{dcase}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \;:^{\Phi}\; \tau$ inversion gives that $\Gamma \vdash \mathsf{K}\,\psi\,\delta \;:^{\Pi\,/\!\!/\,\Phi}\; \mathsf{D}\,\psi$ and $\Gamma \vdash br_i \;:^{\Phi}\; (\mathsf{K}\,\psi\,\delta : \mathsf{D}\,\psi) \blacktriangleright \tau$. Suppose K has type $(\overline{a_j :^{\forall} \kappa_j}^{\,j}, \Delta) \to \mathsf{D}\,\overline{a_j}^{\,j}$, then $\Gamma \vdash \psi, \delta \;:\; (\overline{a_j :^{\forall} \kappa_j}^{\,j}, \Delta) \,/\!\!/\, (\Pi \,/\!\!/\, \Phi)$ by Lemma 6.1. Now substitution gives that $\Gamma \vdash \delta, \langle \mathsf{K}\,\psi\,\delta\rangle \;:\; [\psi/\overline{a_j :^{\forall} \kappa_j}^{\,j}]\,\Delta \,/\!\!/\, \Pi \,/\!\!/\, \Phi, c :^{\Box} \mathsf{K}\,\psi\,\delta \sim \mathsf{K}\,\psi\Delta$. Inversion on the rule for case branches gives $\Gamma, [\psi/\overline{a_j :^{\forall} \kappa_j}^{\,j}]\,\Delta \,/\!\!/\, \Pi, c :^{\Box} \mathsf{K}\,\psi\,\delta \sim \mathsf{K}\,\psi\Delta \vdash \rho \;:^{\Phi}\; \tau$ and applying Lemma 6.8 gives $\Gamma \vdash [(\delta, \langle \mathsf{K}\,\psi\,\delta\rangle)/\Delta]\,\rho \;:^{\Phi}\; \tau$.

For the scrutinee reduction step

$$\frac{\begin{array}{l}\Gamma \vdash \gamma :^{\Box} \mathsf{D}\,\overline{\tau_i}^{\,i} \sim \mathsf{D}\,\overline{\upsilon_i}^{\,i} \\ \Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i} \\ \omega = \overline{(\tau_i, \upsilon_i, \mathbf{nth}^{i}\,\gamma)}^{\,i} : \overline{a_i :^{\forall} \kappa_i}^{\,i} \prec \delta : \Delta\end{array}}{(\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta) \rhd \gamma \xrightarrow{\text{kpush}} \mathsf{K}\,\overline{\upsilon_i}^{\,i}\,\overrightarrow{\omega}}$$

from $\Gamma \vdash^{\mathrm{tc}} \overline{(\tau_i, \upsilon_i, \mathbf{nth}^{i}\,\gamma)}^{\,i} \;:\; \overline{a_i :^{\forall} \kappa_i}^{\,i}$ and $\Gamma \vdash \overline{\tau_i}^{\,i}, \delta \;:\; \overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta \,/\!\!/\, \Phi$, Lemma 6.10 gives $\Gamma \vdash^{\mathrm{tc}} \overline{(\tau_i, \upsilon_i, \mathbf{nth}^{i}\,\gamma)}^{\,i}, \omega \;:\; \overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta \,/\!\!/\, \Phi$. Hence Lemma 6.2 implies that $\Gamma \vdash \overline{\upsilon_i}^{\,i}, \overrightarrow{\omega} \;:\; \overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta \,/\!\!/\, \Phi$ and so $\Gamma \vdash \mathsf{K}\,\overline{\upsilon_i}^{\,i}\,\overrightarrow{\omega} \;:^{\Phi}\; \mathsf{D}\,\overline{\upsilon_i}^{\,i}$. □

6.5 Consistency and progress

To prove progress, I must demonstrate the consistency of closed terms in the $\Box$ fragment, as the existence of a coercion between dissimilar types would lead to stuck terms. For example, if $\cdot \vdash \gamma :^{\Box} \mathsf{Bool} \sim (\mathsf{Bool} \to \mathsf{Bool})$ then $(\mathsf{True} \rhd \gamma)\,\mathsf{False}$ is well-typed but stuck. I will prove consistency as a corollary of a more general theorem, by defining a compatibility relation between type expressions that implies they have the same head constructor, and showing that provably equal expressions are compatible. Compatibility will require that closed expressions reduce to head-normal forms with identical outermost constructors and compatible subcomponents, if they terminate at all.

Consistency and progress depend on the usual canonical forms lemma, which is easy to prove thanks to the very restricted definitional equality.

Lemma 6.12 (Canonical forms)**.** *If* v *is a value and* $\Gamma \vdash \mathrm{v} :^{\Phi} \tau$ *then* τ *is a value type. Moreover,*

(a) If $\tau = (a{:}^{\Phi}\,\kappa) \to \upsilon$ *then* v *is of the form* $\Lambda a{:}^{\Phi}\kappa \,.\, e$ *or* $\mathsf{K}\,\delta$*.*

(b) If $\tau = \mathsf{D}\,\psi$ *then* v *is of the form* $\mathsf{K}\,\delta$*.*

(c) If $\tau = *$ *then* v *is a value type.*

Proof. By induction on the typing derivation. □

6.5.1 The definition of compatibility

Given the reduction relation on types, obvious choices for a type equivalence relation include joinability or the equivalence closure of reduction. These ensure that equivalent types have the same head constructors, so would guarantee consistency. However, they are too strong: for example, there is a coercion between $(c :^{\Box} (\mathsf{D}_1 \sim \mathsf{D}_2)) \to \mathsf{D}_1$ and $(c :^{\Box} (\mathsf{D}_1 \sim \mathsf{D}_2)) \to \mathsf{D}_2$ given by $\mathbf{cong}\ \Box\ \langle \mathsf{D}_1 \sim \mathsf{D}_2 \rangle\, (\Lambda c_1 :^{\Box} \mathsf{D}_1 \sim \mathsf{D}_2, c_2 :^{\Box} \mathsf{D}_1 \sim \mathsf{D}_2, c' :^{\Box} c_1 \sim c_2 . c_1)$, but these types are clearly not joinable if D_1 and D_2 are distinct constructors.

Weirich et al. (2013) get round this problem by restricting the well-typed coercions so that they cannot use potentially inconsistent assumptions. This is necessarily over-restrictive, because there can be no decision procedure for consistency of a set of assumptions. A coercion between distinct types can exist in an inconsistent context, and this does not endanger consistency of the whole system. Instead, I define compatibility on closed types to ensure they have the same head constructors, and extend it to open types by considering closed instances. All types are equivalent in an inconsistent context, since there are no closed instances. Thus the existence of a coercion between two types can imply their compatibility. This novel approach works well for the *evidence* language, where types have a well-defined operational semantics; it would be interesting to see if it can be applied to System $\mathrm{F_C}$ with type families.

The definitions and proofs in this section are rather technical, and can safely be skipped by the casual reader. The payoff comes in Subsection 6.5.4: the *evidence* language has the progress and type safety properties. I will present the structure of the argument here, and defer the details of proofs to Appendix D.4.

I will define $\mathbf{A}_k(\varphi)$ where φ is a proposition and k is a natural number, to mean that φ cannot be falsified within k steps. A proposition 'really' holds if the relation holds for all k. This indexing ensures that the relation is well-founded, and facilitates proof by induction on the index, like a step-indexed logical relation.

Definition 6.1 (Computational, coerced and structural type expressions)**.** A type expression is *computational* if it is a function application $f(\delta)$ or a case expression $(\mathbf{d})\mathbf{case}\,\tau\,\mathbf{of}\,\overline{br_i}^{\,i}$; *coerced* if it is a coercion $\tau \rhd \gamma$; otherwise it is *structural.*

Roughly speaking, $\mathbf{A}_k(\tau \sim \upsilon)$ means that if τ and υ are computational, they can both take a step and remain related, whereas if they are structural, they both have the same structure and the substructures are compatible. Any coercions are ignored (but must be between compatible types). Moreover, the kinds of the expressions must be compatible.

Definition 6.2 (Compatibility)**.** Define $\mathbf{A}_k(\varphi)$ inductively on k , provided there exists γ such that $\cdot \vdash \gamma :^{\Box} \varphi$. For such a γ, I write $\mathbf{A}_k(\gamma : \varphi)$ to mean that $\mathbf{A}_k(\varphi)$ holds. The index k represents the depth of comparison to perform. $\mathbf{A}_0(\varphi)$ holds for any well-typed coercion. For $k > 0$, $\mathbf{A}_k(\varphi)$ is defined based on φ.

If φ equates two computational expressions, their reducts must be compatible:

- $\mathbf{A}_k((\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2))$ for τ_1 and τ_2 computational if $\mathbf{A}_{k-1}(\kappa_1 \sim \kappa_2)$, $\tau_1 \longrightarrow \upsilon_1$, $\tau_2 \longrightarrow \upsilon_2$ and $\mathbf{A}_{k-1}(\upsilon_1 \sim \upsilon_2)$.

If φ equates two structural expressions, these must be the same structure and the subcomponents must be compatible:

- $\mathbf{A}_k((\mathsf{H}{:}\kappa) \sim (\mathsf{H}{:}\kappa))$ if $\mathbf{A}_{k-1}(\kappa \sim \kappa)$;
- $\mathbf{A}_k((\tau_1{}^{\Phi}\upsilon_1 : \kappa_1) \sim (\tau_2{}^{\Phi}\upsilon_2 : \kappa_2))$ for $\Phi \neq \Box$ if $\mathbf{A}_{k-1}(\kappa_1 \sim \kappa_2)$, $\mathbf{A}_k(\tau_1 \sim \tau_2)$ and $\mathbf{A}_k(\upsilon_1 \sim \upsilon_2)$;
- $\mathbf{A}_k((\tau_1{}^{\Box}\eta_1 : \kappa_1) \sim (\tau_2{}^{\Box}\eta_2 : \kappa_2))$ if $\mathbf{A}_{k-1}(\kappa_1 \sim \kappa_2)$, $\mathbf{A}_k(\tau_1 \sim \tau_2)$, $\mathbf{A}_{k-1}(\eta_1 : \varphi_1)$ and $\mathbf{A}_{k-1}(\eta_2 : \varphi_2)$;
- $\mathbf{A}_k(\tau_1 \to \upsilon_1 \sim \tau_2 \to \upsilon_2)$ if $\mathbf{A}_k(\tau_1 \sim \tau_2)$ and $\mathbf{A}_k(\upsilon_1 \sim \upsilon_2)$;
- $\mathbf{A}_k((a_1 :^{\Upsilon} \kappa_1) \to \tau_1 \sim (a_2 :^{\Upsilon} \kappa_2) \to \tau_2)$ if $\mathbf{A}_k(\kappa_1 \sim \kappa_2)$ and for all $l < k$, $\mathbf{A}_l((\upsilon_1 : \kappa_1) \sim (\upsilon_2 : \kappa_2))$ implies $\mathbf{A}_l([\upsilon_1/a_1]\,\tau_1 \sim [\upsilon_2/a_2]\,\tau_2)$.
- $\mathbf{A}_k((c_1 :^{\Box} \varphi_1) \to \tau_1 \sim (c_2 :^{\Box} \varphi_2) \to \tau_2)$ if $\mathbf{A}_k(\varphi_1 \sim \varphi_2)$ and for all $l < k$, $\mathbf{A}_l(\gamma_1 : \varphi_1)$ and $\mathbf{A}_l(\gamma_2 : \varphi_2)$ imply $\mathbf{A}_l([\gamma_1/c_1]\,\tau_1 \sim [\gamma_2/c_2]\,\tau_2)$.

If one side is structural and the other is computational, the computational expression must reduce to a compatible structure:

- $\mathbf{A}_k(\tau_1 \sim \tau_2)$ where τ_1 is structural and τ_2 is computational if $\tau_2 \longrightarrow^* \upsilon$ where υ is structural or coerced and $\mathbf{A}_k(\tau_1 \sim \upsilon)$;

- $\mathbf{A}_k(\tau_1 \sim \tau_2)$ where τ_1 is computational and τ_2 is structural if $\tau_1 \longrightarrow^* \upsilon$ where υ is structural or coerced and $\mathbf{A}_k(\upsilon \sim \tau_2)$.

If either side is coerced, the coercion must be between compatible types and the underlying expressions must be compatible:

- $\mathbf{A}_k(\tau_1 \triangleright \eta \sim \tau_2)$ if $\mathbf{A}_k(\tau_1 \sim \tau_2)$ and $\mathbf{A}_{k-1}(\eta : \kappa_1 \sim \kappa_2)$;

- $\mathbf{A}_k(\tau_1 \sim \tau_2 \triangleright \eta)$ where τ_1 is not coerced if $\mathbf{A}_k(\tau_1 \sim \tau_2)$ and $\mathbf{A}_{k-1}(\eta : \kappa_1 \sim \kappa_2)$.

Now the definition of compatibility is extended to quantified equations, by taking closed instances:

- $\mathbf{A}_k((a :^{\Upsilon} \kappa) \rightarrow \varphi)$ if for all $l < k$, $\mathbf{A}_l((\tau : \kappa) \sim (\tau : \kappa))$ implies $\mathbf{A}_l([\tau/a]\, \varphi)$;

- $\mathbf{A}_k((c :^{\square} \varphi') \rightarrow \varphi)$ if for all $l < k$, $\mathbf{A}_l(\eta : \varphi')$ implies $\mathbf{A}_l([\eta/c]\, \varphi)$;

- $\mathbf{A}_k((x :^{\curlywedge} \tau) \rightarrow \varphi)$ if $\mathbf{A}_k(\varphi)$.

This definition extends naturally to closed telescoped coercions, requiring that all the coercions are between compatible types.

Definition 6.3. Define $\mathbf{A}_k(\omega : \Delta)$ where $\cdot \vdash^{\mathrm{tc}} \omega : \Delta$ by

- $\mathbf{A}_k(\cdot : \cdot)$ always;

- $\mathbf{A}_k(\omega, (\tau, \upsilon, \gamma) : \Delta, a :^{\Upsilon} \kappa)$ if $\mathbf{A}_k(\omega : \Delta)$ and $\mathbf{A}_k(\gamma : \tau \sim \upsilon)$;

- $\mathbf{A}_k(\omega, (\eta, \eta') : \Delta, c :^{\square} \varphi)$ if $\mathbf{A}_k(\omega : \Delta)$ and both $\mathbf{A}_{k-1}(\eta : [\overleftarrow{\omega}/\Delta]\, \varphi)$ and $\mathbf{A}_{k-1}(\eta' : [\overrightarrow{\omega}/\Delta]\, \varphi)$;

- $\mathbf{A}_k(\omega, (e, e') : \Delta, x :^{\curlywedge} \kappa)$ if $\mathbf{A}_k(\omega : \Delta)$.

For consistency and progress, the signature Σ must not contain any inconsistent axioms or malformed types (as they would invalidate consistency), or any undefined runtime functions (as they would invalidate progress). These conditions are encapsulated in the following definition.

Definition 6.4 (Good declaration and signature)**.** A signature Σ is *good* if all the entries in Σ are good, where:

- An axiom $C :^{\square} \varphi$ is good if $\mathbf{A}_k(\varphi)$ and $\mathbf{A}_k(\varphi \sim \varphi)$ for all k.

- A constructor $\mathsf{H} :^{\Phi} \tau$ is good if $\mathbf{A}_k(\tau \sim \tau)$ for all k.

- A static function declaration $f\,[\Delta] :^{\Upsilon} \kappa$ is good if it has a unique corresponding definition $f\,[\Delta] = \tau :^{\Upsilon} \kappa$ in Σ, such that $\mathbf{A}_k(\omega : \Delta)$ implies $\mathbf{A}_k([\overleftarrow{\omega}/\Delta]\,\tau \sim [\overrightarrow{\omega}/\Delta]\,\tau)$.
- A dynamic function declaration $f\,[\Delta] :^{⅄} \kappa$ is always good, since it cannot occur in types.

From now on I will implicitly assume that the signature Σ is good.

6.5.2 Properties of compatibility

I now prove that compatibility is a partial equivalence relation on types, that it respects computation and is a congruence. It is reflexive on well-typed expressions, but to prove this I must show that all well-typed coercions are compatible, which is the main result in the following section.

Lemma 6.13 (Symmetry)**.** *If* $\mathbf{A}_k(\tau \sim \upsilon)$ *then* $\mathbf{A}_k(\upsilon \sim \tau)$.

Proof. By inversion on the definition. It is clear that every case is symmetric. □

Lemma 6.14 (Transitivity)**.** *If* $\mathbf{A}_k(\tau \sim \upsilon)$ *and* $\mathbf{A}_k(\upsilon \sim \kappa)$ *then* $\mathbf{A}_k(\tau \sim \kappa)$.

Proof. By induction on k and inversion on $\mathbf{A}_k(\varphi)$. For details, see Appendix D.4 (page 250). □

In the usual step-indexed fashion, decreasing the step index preserves the relation, because strictly less of the expressions' structures are compared.

Lemma 6.15 (Downward closure)**.**

(a) If $\mathbf{A}_{k+1}(\varphi)$ *then* $\mathbf{A}_k(\varphi)$.

(b) If $\mathbf{A}_{k+1}(\omega : \Delta)$ *then* $\mathbf{A}_k(\omega : \Delta)$.

Proof. Part (a) is by induction on k and inversion on $\mathbf{A}_{k+1}(\varphi)$. Part (b) follows from part (a) by structural induction on ω. □

To show that the **step** coercion preserves compatibility, use the following:

Lemma 6.16 (Reduction preserves compatibility)**.** *If* $\tau \xrightarrow{\text{kpush}} \upsilon$ *and* $\mathbf{A}_k(\tau \sim \tau)$ *then* $\mathbf{A}_{k-1}(\tau \sim \upsilon)$.

Proof. By induction on k and the reduction step $\tau \xrightarrow{\text{kpush}} \upsilon$. For details, see Appendix D.4 (page 252). □

The definition of compatibility makes it a congruence for structural expressions and coercions. I must prove that it is a congruence for case analysis.

Lemma 6.17 (Congruence for case analysis)**.** *If $\mathbf{A}_k(\varepsilon \sim \varepsilon')$ and $\mathbf{A}_k(br_i \approx br'_i)$ for all i, then $\mathbf{A}_k((\mathbf{d})\mathbf{case}\,\varepsilon\,\mathbf{of}\,\overline{br_i}^{\,i} \sim (\mathbf{d})\mathbf{case}\,\varepsilon'\,\mathbf{of}\,\overline{br'_i}^{\,i})$.*

Proof. By induction on k and case analysis on ε and ε', using Lemma 6.16. For details, see Appendix D.4 (page 255). □

To show compatibility of the $\mathbf{kind}\,\gamma$ coercion, which extracts a proof that the kinds are equal from a proof that two types are equal, I will need the following:

Lemma 6.18 (Compatibility of kinds)**.** *If $\mathbf{A}_k((\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2))$ holds, then $\mathbf{A}_{k-1}((\kappa_1 : *) \sim (\kappa_2 : *))$.*

Proof. By induction on k and inversion on $\mathbf{A}_k(\varphi)$. □

6.5.3 Well-typed coercions are compatible

Finally, I can show that the existence of a coercion between types implies their compatibility. Consistency is then an immediate corollary. Crucially, the logical unsoundness of the type language, due to the presence of general recursion and the paradoxical $* : *$, does not affect the $\Box$ fragment. General recursion is not available in coercions, and they may perform only a finite amount of computation.

Lemma 6.19 (Basic Lemma)**.**

(a) If $\Gamma \vdash \tau :^{\forall} \kappa$ then for all k, $\mathbf{A}_k(\omega_0 : \Gamma)$ implies $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\overrightarrow{\omega_0}/\Gamma]\,\tau)$.

(b) If $\Gamma \vdash br :^{\forall} \upsilon \blacktriangleright \tau$ or $\Gamma \vdash br :^{\forall} (\varepsilon : \upsilon) \blacktriangleright \tau$ then for all k, $\mathbf{A}_k(\omega_0 : \Gamma)$ implies $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,br \approx [\overrightarrow{\omega_0}/\Gamma]\,br)$.

(c) If $\Gamma \vdash \gamma :^{\Box} \varphi$ then for all k, $\mathbf{A}_k(\omega_0 : \Gamma)$ implies $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\varphi)$ and $\mathbf{A}_k([\overrightarrow{\omega_0}/\Gamma]\,\varphi)$.

(d) If $\Gamma \vdash^{\mathrm{tc}} \omega : \Delta$ then for all k, $\mathbf{A}_k(\omega_0 : \Gamma)$ implies $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$.

Proof. By structural induction on typing derivations. Note that k is quantified inside the inductive hypothesis. For details, see Appendix D.4 (page 256). □

Theorem 6.20 (Consistency)**.** *If $\cdot \vdash \gamma :^{\Box} (\xi_0 : *) \sim (\xi_1 : *)$ then ξ_0 and ξ_1 have the same head constructor (that is, either $\xi_i = (a_i :^{\Phi} \kappa_i) \to \tau_i$ or $\xi_i = \mathsf{H}\,\psi_i$).*

Proof. This follows from the special case of Lemma 6.19(c) where Γ is empty, since $\mathbf{A}_1(\xi_1 \sim \xi_2)$ implies ξ_1 and ξ_2 have the same head constructor, by definition. □

6.5.4 Progress

Thanks to the consistency proof, progress is straightforward, as in previous work. Type safety is an immediate corollary.

Theorem 6.21 (Progress)**.** *If* $\cdot \vdash e :^{\curlywedge} \tau$ *then either* e *is a value,* e *is a coerced value or there is some* e' *such that* $e \longrightarrow e'$*.*

Proof. By structural induction on the typing derivation. When a coerced value prevents reduction, Theorem 6.20 ensures that the relevant push rule applies. □

Corollary 6.22 (Type safety)**.** *If* $\cdot \vdash e :^{\curlywedge} \tau$ *and* $e \longrightarrow^* e'$ *then either* e' *is a value,* e' *is a coerced value or there is some* e'' *such that* $e' \longrightarrow e''$*.*

6.6 Erasure

The erasure operation, defined in Figure 6.14, produces a runtime version $\|e\|$ of an evidence term e (phase $\curlywedge$) by removing static subterms (phases $\forall$ and $\Box$). Similarly, an erasure operation $\|\delta : \Delta\|$ is defined for a vector δ in telescope Δ.

Runtime terms r are a subgrammar of evidence terms e, except that λ-abstractions do not record their types and an additional marker _ is used to indicate where a subterm has been erased. This could be implemented with a unit type. No casts are present in runtime terms, and dependent case analysis has been turned into normal case analysis. The grammar of runtime terms is:

$$r \quad ::= \quad a \mid \lambda x.r \mid r\,r' \mid K \mid f(\overline{r_k}^{k}) \mid \textbf{case}\,r\,\textbf{of}\,\overline{\mathsf{K}_i\,\Delta_i \to r_i}^{i} \mid \_$$

Erasure uses the phase annotations on applications to avoid reconstructing the type of the term, but if they were not present it would be easy to define erasure in a typed fashion, since the evidence term encodes its typing derivation. Saturated function applications $f(\delta)$ are not annotated with phases, so erasure for vectors $\|\delta : \Delta\|$ uses the telescope Δ from the declaration of the function.

The operational semantics of runtime terms is given in Figure 6.15. It is a subset of the rules from Figure 6.13, omitting those related to coercions, and erasing the bodies of functions defined in the signature.

The motivation for replacing erased subterms with the _ marker, rather than removing them (and the corresponding λ-abstractions) altogether,[10] is that it simplifies the correspondence between the original and erased operational semantics. This correspondence is shown by the following lemma.

[10]Of course, subsequent optimisation of runtime terms could remove the unnecessary redexes.

$$\begin{array}{rcl}
\|x\| & \mapsto & x \\
\|\Lambda x{:}^{\Phi}\kappa\,.\,e\| & \mapsto & \lambda x.\|e\| \\
\|e^{\forall}\tau\| & \mapsto & \|e\|\,\_ \\
\|e^{\Box}\gamma\| & \mapsto & \|e\|\,\_ \\
\|e^{\Pi}\varepsilon\| & \mapsto & \|e\|\,\|\varepsilon\| \\
\|e^{\curlywedge}e'\| & \mapsto & \|e\|\,\|e'\| \\
\|\mathsf{K}\| & \mapsto & \mathsf{K} \\
\|f(\delta)\| & \mapsto & f(\|\delta : \Delta\|) \text{ where } \Sigma \ni f\,[\Delta] :^{\Phi} \kappa \\
\|(\mathbf{d})\mathbf{case}\, e\, \mathbf{of}\, \overline{\mathsf{K}_i\,\Delta_i \to e_i}^{\,i}\| & \mapsto & \mathbf{case}\, \|e\|\, \mathbf{of}\, \overline{\mathsf{K}_i\,\Delta_i \to \|e_i\|}^{\,i} \\
\|e \rhd \gamma\| & \mapsto & \|e\|
\end{array}$$

$$\begin{array}{rcl}
\|\cdot : \cdot\,\| & \mapsto & \cdot \\
\|\delta, \tau : \Delta, a :^{\forall} \kappa\| & \mapsto & \|\delta : \Delta\|, \_ \\
\|\delta, \gamma : \Delta, c :^{\Box} \varphi\| & \mapsto & \|\delta : \Delta\|, \_ \\
\|\delta, \varepsilon : \Delta, x :^{\Pi} \tau\| & \mapsto & \|\delta : \Delta\|, \|\varepsilon\| \\
\|\delta, e : \Delta, x :^{\curlywedge} \tau\| & \mapsto & \|\delta : \Delta\|, \|e\|
\end{array}$$

Figure 6.14: Erasure of terms and vectors

$\boxed{r \longrightarrow r'}$ *(r reduces to r′)*

$$\frac{r \longrightarrow r'}{r\,r'' \longrightarrow r'\,r''} \qquad \frac{r \longrightarrow r'}{\mathbf{case}\, r\, \mathbf{of}\, \overline{\mathsf{K}_i\,\Delta_i \to r_i}^{\,i} \longrightarrow \mathbf{case}\, r'\, \mathbf{of}\, \overline{\mathsf{K}_i\,\Delta_i \to r_i}^{\,i}}$$

$$\frac{}{\mathbf{case}\, \mathsf{K}_j\, \overline{r_k}^{\,k}\, \mathbf{of}\, \overline{\mathsf{K}_i\,\Delta_i \to r_i}^{\,i} \longrightarrow [\overline{r_k}^{\,k}/\Delta_j]r_j} \qquad \frac{\Sigma \ni f\,[\Delta] = e :^{\Phi} \kappa}{f(\overline{r_k}^{\,k}) \longrightarrow [\overline{r_k}^{\,k}/\Delta]\|e\|}$$

$$\frac{}{(\lambda x.r)\,r' \longrightarrow [r'/x]r}$$

Figure 6.15: Operational semantics of erased terms

Lemma 6.23. *If* $\cdot \vdash e :^{\curlywedge} \tau$*, then either*

- *e is a coerced value and $\|e\|$ is a value; or*
- *$e \longrightarrow e'$ and either $\|e\| = \|e'\|$ or $\|e\| \longrightarrow \|e'\|$.*

Proof. If e is a coerced value, it is easy to see that $\|e\|$ is a value. If not, Theorem 6.21 means that e can take a step to some e'; proceed by induction on the step taken. For most steps, the result follows immediately by induction or the fact that both e and e' are identical after erasure. The cases for β-reduction and definitional expansion make use of the fact that erasure commutes with substitution, i.e. $\|[\delta/\Delta]\, e\| = [\|\delta : \Delta\|/\Delta]\|e\|$. When the scrutinee of a case expression takes a push step, this does not change its erasure. $\square$

The erasure operation described above removes all static information from *evidence* terms. In some cases it is also possible to erase dependencies without erasing types entirely: datatype indices are removed and Π-types become non-dependent functions. For example, in *inch* syntax,

```
data Vec :: * → ℕ → * where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Suc n)
```

would be erased to

```
data Vec :: * → * where
  Nil  :: Vec a
  Cons :: a → Vec a → Vec a
```

otherwise known as the type of lists, and

```
replicate :: Π (n :: ℕ) → a → Vec a n
```

would be erased to

```
replicate :: ℕ → a → Vec a
```

Thus an *inch* term can sometimes be converted into a Haskell term, or an *evidence* term can be converted into a System F_C-like term. However, this is not possible for terms containing large eliminations, where a type is computed from a shared term by type-level case analysis. This approach is used in the prototype implementation, as discussed in Chapter 8.

6.7 Discussion

I conclude this chapter with comments on possible extensions to the *evidence* language, and a comparison to its predecessors. In the following chapter, I will show how high-level *inch* source code can be translated to the *evidence* language discussed in this chapter, by a process of elaboration.

6.7.1 Representing numbers

So far I have said a great deal about how to manage Π-types, and indeed phases more generally, but I have not said much about numbers. How might the evidence language described here be extended to support them?

One option is to adopt the traditional algebraic datatype presentation of natural numbers and integers:

$$\textbf{data}\ \mathbb{N} = \mathsf{Zero} \mid \mathsf{Suc}\ \mathbb{N}$$
$$\textbf{data}\ \mathbb{Z} = \mathsf{NonNegative}\ \mathbb{N} \mid \mathsf{StrictlyNegative}\ \mathbb{N}$$

Mathematical operations such as addition can be defined on these representations as pattern-matching functions, and the machinery in this chapter will allow them to be used on the type level. This is rather inefficient, though perhaps the compiler could replace the representation with a native version after typechecking.

However, the equational theory desired for these operations is more than the behaviour delivered by computation. By adding axioms to the signature, properties such as the commutativity of addition can be made available as coercions, and hence used by the elaborator. Consistency of the system, and hence type safety, are ensured provided the conditions of Definition 6.4 are satisfied.

In particular, any new axioms must be compatible, i.e. true on closed instances. The commutativity of addition axiom

$$(a :^{\forall} \mathbb{N}, b :^{\forall} \mathbb{N}) \rightarrow (a + b) \sim (b + a)$$

is fine, because $(a + b) \sim (b + a)$ holds by computation whenever a and b are replaced with closed values, but a bogus axiom such as

$$(a :^{\forall} \mathbb{N}) \rightarrow a \sim \mathsf{Suc}\ a$$

will not be compatible.

One problem with this approach is that some valid axioms do not satisfy the compatibility relation, because they change termination properties. For example,

$$(a :^{\forall} \mathbb{Z}) \to (a \; - \; a) \sim \mathsf{Zero}$$

is not accepted, because if a is instantiated with a closed divergent term, then the left-hand side diverges but the right does not. This could be resolved by extending the definition of compatibility, so that rather than considering reduction alone, numeric expressions could be simplified via axioms. Consistency would then depend on a global property of the axioms, that they could not be used to derive $\mathsf{Zero} \sim \mathsf{Suc}\,\mathsf{Zero}$.

There is also more work to do on the evidence for inequality constraints. These can be encoded using algebraic datatypes, for example

```
**data** $m \leqslant n$  **where**
   $\mathsf{Z} :: \mathsf{Zero} \leqslant n$
   $\mathsf{S} :: m \leqslant n \to \mathsf{Suc}\ m \leqslant \mathsf{Suc}\ n$
```

but it might be preferable to make use of the $\Box$ fragment to record known-consistent (and hence erasable) inequality proofs, just as coercions are known-consistent equality proofs.

6.7.2 Adding η-laws

Another desirable extension of the compatibility relation is support for η-conversion of single-constructor (record) datatypes. For example, the usual fst and snd projections from pairs are perfectly good shared definitions, so they can be used at the type level. It would be useful to support the η-axiom

$$(a :^{\forall} *, b :^{\forall} *, x :^{\forall} (a \;, b)) \to x \sim (\mathsf{fst}(x) \;,\; \mathsf{snd}(x))$$

which says that any inhabitant of a pair type is equal to the pair of its projections.

For example, this is needed to show that the type of paths in a binary relation

```
**data** Path :: $((a, a) \to *) \to ((a, a) \to *)$  **where**
   Stop :: Path $r$ $(x, x)$
   Step :: $r$ $(x, y) \to$ Path $r$ $(y, z) \to$ Path $r$ $(x, z)$
```

forms an indexed monad. The following definition is accepted

```
returnIx :: $r$ $(x, y) \to$ Path $r$ $(x, y)$
returnIx $v$ = Step $v$ Stop
```

but its type is insufficiently general; it should have the type

$$\mathsf{returnIx} :: r\ c \to \mathsf{Path}\ r\ c$$

which requires η-expansion.

As in the previous section, η-axioms change termination properties, because x might diverge, so they do not satisfy the existing definition of compatibility. However, as with numeric axioms, compatibility could be modified to build in η-expansion, by defining $\mathbf{A}_k((\tau\ ,\ \tau') \sim \upsilon)$ for computational expressions υ to mean $\mathbf{A}_k(\tau \sim \mathsf{fst}(\upsilon))$ and $\mathbf{A}_k(\tau' \sim \mathsf{snd}(\upsilon))$.

6.7.3 Related work

System $\mathrm{F_C}(X)$ was introduced by Sulzmann et al. (2007) as a new core language for GHC. It is based on System F, the second-order polymorphic λ-calculus (Reynolds, 1974; Girard et al., 1989), but adds algebraic datatypes, higher kinds and explicit coercions (proofs of type equality). It was motivated by the need to elaborate GADTs and type families, both of which can be understood as extensions to the equational theory of types: case analysis on GADTs introduces new equational hypotheses, which may be used to show the body is well-typed, while type families add axiomatically-defined type-level functions. This was a major advance on the previous approach used in GHC, of adding GADTs to System F directly. The (X) parameterisation in the system represents its dependence on an unspecified decision procedure for checking that a context is consistent, i.e. that the axioms and equational hypotheses do not entail a contradiction. The system was subsequently revised by the authors in the light of implementation experience (Sulzmann et al., 2009).

Weirich et al. (2011a) developed System $\mathrm{F_{C2}}$ to rectify a consistency problem discovered in the implementation of GHC. This resulted from the combination of newtypes, which introduce axioms asserting their equality with the underlying representation type, and type families, which can distinguish between a newtype and its representation. They proved that their system is consistent if type family declarations are non-overlapping, using an approach based on rewriting.

Development continued with System $\mathrm{F_C^{\uparrow}}$ (Yorgey et al., 2012), which adds datatype promotion and kind polymorphism. This allows algebraic datatypes to be used as kinds, so type-level programming need not be entirely untyped: for example, a datatype of Peano numerals can be promoted to the kind level and used to index a GADT of vectors. However, kind equality in $\mathrm{F_C^{\uparrow}}$ is purely syntactic, so it is not possible to promote GADTs. Vytiniotis et al. (2012) tweaked

System $F_C^{\uparrow}$ to support deferred type errors, by distinguishing between an 'unlifted' type of known-good equality proofs and a 'lifted' type of potentially bogus proofs that must be evaluated before use.

Weirich et al. (2013) took the datatype promotion and kind polymorphism ideas to their logical conclusion, by eliminating the distinction between types and kinds. The *evidence* language described in this chapter continues in this direction, as it makes no distinction between types and kinds. It goes further in that terms and types share a common syntax and typing rules, though the phase restrictions mean not every expression form is available at every phase. Moreover, it adds Π-types, allowing real dependency without the need for the singleton construction.

6.7.4 Future work

A key idea of this chapter is the use of a common syntax for terms, types and kinds, while the phase distinction is maintained by indexing typing judgments with the phase at which they apply. Variables in the context carry a phase, and application allows for promotion implicitly, as described in Section 6.2. An ordering on phases makes it possible for data at one phase to be used at another, thereby streamlining the presentation of Π-types.

Phases need not be confined to this system, however: they can be defined for any Pure Type System. The set of phases need not be $\{\forall, \square, \Pi, \curlywedge\}$ as in this chapter, but could be any partially ordered set with a suitable relativisation operator $\Phi \mathbin{/\!\!/} \Psi$. For example, a system with two phases could model a dependent type theory that distinguishes between runtime and compile-time data. The results of this chapter illustrate the properties required for a system of phases. Work is ongoing to develop the theory of phases and investigate its applications.

The novel consistency proof for coercions given in Section 6.5 takes a different approach to previous work, and thereby lifts a technical restriction on the use of potentially inconsistent assumptions in coercions between $\square$-quantified types. However, this approach relies on the common operational semantics for types and terms, and in particular the treatment of type functions via case analysis. It remains to be seen whether the method can be extended to support the notion of type families in System F_C, which are defined axiomatically.

Chapter 7

Producing the evidence: elaborating *inch*

Broadly construed, *elaboration* is a type-directed process of translating a high-level source language into a more explicit intermediate language, inferring details that were originally left implicit. Section 2.4 (page 27) showed how to elaborate λ-calculus with let-expressions into explicitly-typed System F. GHC elaborates Haskell programs into System $\mathrm{F_C}$, which adds algebraic datatypes, higher kinds and type equality constraints to System F. Dependently typed languages such as Epigram are explained by elaboration into a type theory, with the elaborator synthesising implicit arguments and solving higher-order unification problems.

Following the Curry-Howard correspondence, elaboration of programming languages is closely connected to generating proof objects from proof scripts in interactive theorem provers (such as Coq with its core language Gallina). Here, the primary motivation is ensuring correctness through the de Bruijn criterion. A well-understood kernel theory, with simple typechecking, allows the output from complex tactics and decision procedures to be independently rechecked. Likewise, GHC is an extremely complex program, and the ability to easily typecheck programs in the intermediate language is crucial to debugging the compiler. Moreover, the intermediate language provides a good basis for implementing optimisations, as all the typing information is available explicitly.

In this chapter, I will describe the process for elaborating *inch* programs into the *evidence* language defined in the last chapter. I begin by introducing 'type schemes', which decorate *evidence* language types with information on implicit arguments (7.1), inspired by the work of Pollack (1990). The formal syntax of the *inch* language (7.2) includes a large fragment of the informally presented syntax. Instead of giving this a type system directly, I supply a non-deterministic elaboration system that relates *inch* terms to *evidence* terms (7.3).

I then explain how partial knowledge and progress can be represented (7.4), and describe a definite (and necessarily incomplete) algorithm for elaboration (7.5). This is based on the work on type inference in Part I, where unification variables and unsolved constraints are explicitly represented using metacontexts. The algorithm reduces elaboration to constraint solving in the underlying evidence language. Designing a constraint solving algorithm is a complex task in itself. I will specify its required properties and describe it at a high level, but I will not describe constraint solving in detail.

Elaboration of case expressions, which is the basis for the treatment of pattern-matching definitions, is somewhat involved and is therefore postponed (7.6). The chapter concludes with some contextualising remarks (7.7).

7.1 Type schemes

As discussed in Subsection 5.2.4 (page 99), it is desirable to have finer-grained control over which arguments are automatically inferred than the current Haskell policy of forcing ∀-bound arguments to be implicit and other arguments to be explicit. Instead, constants and variables in the context will be assigned a *type scheme* σ, consisting of a quantified type with annotations indicating whether each argument is implicit ($:_i$) or explicit ($:_e$). The grammar of schemes is given in Figure 7.1. For example, the type scheme of the equality constructor is

$$(\sim) : (a :_i^{\forall} *, b :_i^{\forall} *, x :_e^{\curlywedge} a, y :_e^{\curlywedge} b) \to *$$

meaning that the first two arguments are implicit and the last two are explicit, thereby justifying the usual use of $(\sim)$ as a binary operator. The usual definition of vectors gives rise to the following type schemes:

$$\begin{aligned}
&\mathsf{Vec} &&: (a :_e^{\forall} *, n :_e^{\forall} \mathbb{N}) \to * \\
&\mathsf{Nil} &&: (a :_i^{\forall} *, n :_i^{\forall} \mathbb{N}, c :_i^{\Box} n \sim \mathsf{Zero}) \to \mathsf{Vec}\ a\ n \\
&\mathsf{Cons} &&: (a :_i^{\forall} *, n :_i^{\forall} \mathbb{N}, m :_i^{\forall} \mathbb{N}, \\
& && \qquad x :_e^{\curlywedge} a, xs :_e^{\curlywedge} \mathsf{Vec}\ a\ m, c :_i^{\Box} n \sim \mathsf{Suc}\ m) \to \mathsf{Vec}\ a\ n
\end{aligned}$$

I will not give formal rules for elaborating source language datatype declarations into constructors with the appropriate type schemes.

Quantification over proofs (at phase $\Box$) will always be implicit, because coercions are not written in the source language. On the other hand, dynamically quantified variables will be explicit, as they cannot be determined by unification

$$\begin{array}{rcl} \sigma & ::= & \tau \mid (a :_e^\Phi \sigma') \to \sigma \mid (a :_i^\Phi \tau) \to \sigma \\ \mathbf{\Gamma},\ \mathbf{\Delta} & ::= & \cdot \mid \mathbf{\Gamma}, a :_e^\Phi \sigma \mid \mathbf{\Gamma}, a :_i^\Phi \tau \end{array}$$

$$\begin{array}{rcl} \lfloor \tau \rfloor & \mapsto & \tau \\ \lfloor (x :_e^\Phi \sigma') \to \sigma \rfloor & \mapsto & (x :^\Phi \lfloor \sigma' \rfloor) \to \lfloor \sigma \rfloor \\ \lfloor (a :_i^\Phi \tau) \to \sigma \rfloor & \mapsto & (a :^\Phi \tau) \to \lfloor \sigma \rfloor \end{array}$$

$$\begin{array}{rcl} \lfloor \cdot \rfloor & \mapsto & \cdot \\ \lfloor x :_e^\Phi \sigma, \mathbf{\Delta} \rfloor & \mapsto & x :^\Phi \lfloor \sigma \rfloor, \lfloor \mathbf{\Delta} \rfloor \\ \lfloor a :_i^\Phi \tau, \mathbf{\Delta} \rfloor & \mapsto & a :^\Phi \tau, \lfloor \mathbf{\Delta} \rfloor \end{array}$$

Figure 7.1: Grammar and erasure of schemes and annotated telescopes

constraints. Typeclasses can be seen as a form of implicit dynamic quantification, with an alternative strategy for finding the corresponding arguments, based on instance search rather than unification. This idea underlies Agda's support for *instance arguments* (Devriese and Piessens, 2011). I will not consider typeclasses further, but it is straightforward to handle them using the elaboration framework.

Like schemes, telescopes can be annotated to indicate whether the argument is implicit or explicit, writing $\mathbf{\Delta}$ instead of Δ. Erasing the annotations produces a type or telescope in the *evidence* language, written $\lfloor \sigma \rfloor$ or $\lfloor \mathbf{\Delta} \rfloor$ and defined in Figure 7.1. I will assume that the signature Σ assigns type schemes to constructors H and annotated telescopes to shared functions f. In general, I will elide the distinction between a quantified *type* $(a :^\Phi \kappa) \to \tau$ and an explicitly-quantified type *scheme* $(a :_e^\Phi \kappa) \to \tau$.

Quantifying a scheme over a telescope $(\mathbf{\Delta}) \to \sigma$ and the relativisation operator $\mathbf{\Delta} \mathbin{/\!\!/} \Phi$ extend the definitions on *evidence* expressions in the obvious way.

I do not extend the type system of the *evidence* language itself. This avoids complicating the metatheory with details of implicit arguments. Rather, schemes are a tool for explaining how elaboration should generate explicit *evidence* terms.

To obtain good inference behaviour, the elaboration algorithm should never attempt to 'guess' type schemes, only propagate them through bidirectional type inference. This avoids questions of how to unify type schemes. For this reason, the domain of an implicit quantification is always a type rather than a scheme.

Following the Agda convention, the application syntax $\mathsf{\rho}\{a = \mathsf{\rho}'\}$ is used to supply explicitly an argument that is usually implicit, with name a. This means type schemes cannot always be treated as equivalent up to α-conversion, as names may appear outside the scope in which they are bound.

$\mathrm{\rho}$	$::=$		*inch* expression
	$\mid$	a	variable
	$\mid$	$\mathrm{\rho}\,\mathrm{\rho}'$	explicit application
	$\mid$	$\mathrm{\rho}\{a = \mathrm{\rho}'\}$	implicit application
	$\mid$	$\forall(a : \mathrm{\kappa}) \to \mathrm{\tau}$	explicit $\forall$ quantification
	$\mid$	$\Pi(a : \mathrm{\tau}) \to \mathrm{\upsilon}$	explicit Π quantification
	$\mid$	$\mathrm{\tau} \to \mathrm{\upsilon}$	function type (explicit ⅄ quantification)
	$\mid$	H	constructor
	$\mid$	$f(\mathrm{\delta})$	saturated function
	$\mid$	$\mathrm{\rho} : \mathrm{\sigma}$	type ascription
	$\mid$	$\lambda x\,.\,\mathrm{\rho}$	abstraction
	$\mid$	$\mathbf{let}\; x = \mathrm{\rho}\; \mathbf{in}\; \mathrm{\rho}'$	let binding
	$\mid$	_	unknown

Figure 7.2: Grammar of *inch* expressions

7.2 Formal syntax of *inch*

The grammar of *inch* is presented in Figures 7.2 and 7.3. Like the *evidence* language, there is a common syntax for expressions $\mathrm{\rho}$, but I will usually use $\mathrm{\tau}$, $\mathrm{\upsilon}$ or $\mathrm{\kappa}$ for types and s or t for runtime terms (according to the respective subgrammars). While the presentation using a common syntax is compact, it is inessential and one may use different syntaxes for the term and type levels.

The main additions, compared to the *evidence* language, are: let-expressions; the ability to ascribe a type scheme to an expression, written $\mathrm{\rho} : \mathrm{\sigma}$; and the 'unknown' marker _, which asks for a value to be inferred by the elaborator. All coercion proofs are omitted (as they will be generated by constraint solving, not supplied by the user). The *inch* syntax uses upright Greek letters such as $\mathrm{\rho}$, where the *evidence* syntax would use the italic ρ.

The syntax of *inch* type schemes $\mathrm{\sigma}$ is deliberately chosen to resemble Haskell syntax. It will be translated by elaboration into the *evidence* language type schemes of Section 7.1. There is no explicit quantifier at phase $\square$, and the implicit quantifier does not bind a variable, because proofs are invisible in the source language. There is no implicit quantifier at phase ⅄, because no constraints would be able to determine the value of a dynamic argument (absent typeclasses).

The source syntax should allow type ascriptions on quantifiers to be omitted, but this can be dealt with by inserting _ markers as necessary. For example, the universal quantifier $\forall\, a.\mathrm{\sigma}$ can be desugared into $\forall(a : \_).\ \mathrm{\sigma}$ before being elaborated into the *evidence* type scheme $(a :^{\forall}_{i} \kappa) \to \sigma$.

The treatment of (dependent) case analysis is postponed to Section 7.6.

σ	$::=$		*inch* type scheme
	$\mid$	$\forall(a{:}\kappa).\,\sigma$	implicit $\forall$ quantification
	$\mid$	$\forall(a{:}\kappa) \to \sigma$	explicit $\forall$ quantification
	$\mid$	$\Pi(a{:}\tau) \to \sigma$	explicit Π quantification
	$\mid$	$\Pi(a{:}\tau).\,\sigma$	implicit Π quantification
	$\mid$	$\tau \Rightarrow \sigma$	constraint (implicit $\Box$ quantification)
	$\mid$	$\sigma' \to \sigma$	function type (explicit $\curlywedge$ quantification)
	$\mid$	τ	type
$\tau,\ \upsilon,\ \kappa$	$::=$		*inch* type
	$\mid$	a	variable
	$\mid$	$\tau\,\upsilon$	explicit application
	$\mid$	$\tau\{a{=}\upsilon\}$	implicit application
	$\mid$	$\forall(a{:}\kappa) \to \tau$	explicit $\forall$ quantification
	$\mid$	$\Pi(a{:}\tau) \to \upsilon$	explicit Π quantification
	$\mid$	$\tau \to \upsilon$	function type (explicit $\curlywedge$ quantification)
	$\mid$	H	rigid constructor
	$\mid$	$f(\delta)$	saturated function
	$\mid$	$\tau{:}\sigma$	type ascription
	$\mid$	$\_$	unknown
$\mathrm{t},\ \mathrm{s}$	$::=$		*inch* term
	$\mid$	a	variable
	$\mid$	$\mathrm{t}\,\rho$	explicit application
	$\mid$	$\mathrm{t}\{a{=}\rho\}$	implicit application
	$\mid$	K	data constructor
	$\mid$	$f(\delta)$	saturated function
	$\mid$	$\mathrm{t}{:}\sigma$	type ascription
	$\mid$	$\lambda x\,.\,\mathrm{t}$	abstraction
	$\mid$	$\mathbf{let}\,x{=}\mathrm{s}\,\mathbf{in}\,\mathrm{t}$	let binding
	$\mid$	$\_$	unknown

$$\delta \quad ::= \quad \cdot \mid \delta, \rho \mid \delta, \{a{=}\rho\}$$

Figure 7.3: Grammar of *inch* type schemes, types, terms and vectors

7.3 Non-deterministic elaboration

I start by giving a non-deterministic presentation of elaboration that relates *inch* syntax to well-typed *evidence* terms, following the account of elaboration for implicit argument synthesis in the Calculus of Constructions by Luther (2003). The non-deterministic presentation resembles a type system for *inch*, as it allows types and *evidence* terms to be assigned, but does not indicate how they are to be discovered. I will then show how missing information can be reconstructed and give a deterministic algorithm.

The non-deterministic elaboration rules are presented in the Figures 7.4–7.6. Intuitively, elaboration is built out of structural rules, which preserve the structure of the input term, and wrapping rules, which add information missing from the input. It is a kind of 'embedding' of *inch* terms into *evidence* terms. The judgments defined are:

- $\Gamma \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Phi} \sigma$, meaning that the *inch* expression $\mathsf{\rho}$ can be elaborated into the *evidence* expression ρ with scheme σ;
- $\mathbf{\Gamma} \vdash \mathsf{\delta} \rightsquigarrow \delta : \mathbf{\Delta}$, meaning that the *inch* vector $\mathsf{\delta}$ elaborates to δ in the telescope $\mathbf{\Delta}$, by inserting implicit arguments;
- $\mathbf{\Gamma} \vdash \mathsf{\sigma} \rightsquigarrow \sigma$, meaning that the *inch* type scheme $\mathsf{\sigma}$ elaborates to the *evidence* type scheme σ; and
- $\Gamma \vdash e : \sigma \prec e' : \sigma'$, meaning that the type scheme σ is subsumed by σ' and if $e : \sigma$ then $e' : \sigma'$.

7.3.1 Non-deterministic elaboration of expressions

The judgment $\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Phi} \sigma$, defined in Figure 7.4, means that in a context $\mathbf{\Gamma}$, the *inch* expression $\mathsf{\rho}$ interpreted at phase Φ can be elaborated to the *evidence* term ρ with type scheme σ. This judgment is not defined at phase $\Box$ because coercions do not appear in the source language. It uses annotated contexts $\mathbf{\Gamma}$ so that variables record whether they were explicitly bound, and hence in scope for the source language, or implicitly bound, and hence inaccessible.

Most of the elaboration rules simply preserve the structure of the source language expression in the target language. An important exception is the 'magic' rule for implicit λ-abstraction

$$\frac{\mathbf{\Gamma}, a :_i^{\Phi} \tau \vdash \mathsf{t} \rightsquigarrow e :^{\curlywedge} \sigma}{\mathbf{\Gamma} \vdash \mathsf{t} \rightsquigarrow \Lambda a{:}^{\Phi}\tau \,.\, e :^{\curlywedge} (a :_i^{\Phi} \tau) \to \sigma}$$

$\boxed{\mathbf{\Gamma} \vdash \uprho \rightsquigarrow \rho :^{\Psi} \sigma}$ *($\uprho$ can elaborate to ρ with scheme σ at phase $\Psi \in \{\forall, \Pi, \curlywedge\}$)*

$$\frac{\begin{array}{c}\lfloor\mathbf{\Gamma}\rfloor \vdash \mathbf{ctx} \\ \mathbf{\Gamma} \ni a :^{\Phi}_{e} \sigma \qquad \Phi \hookrightarrow \Psi\end{array}}{\mathbf{\Gamma} \vdash a \rightsquigarrow a :^{\Psi} \sigma} \qquad \frac{\begin{array}{c}\lfloor\mathbf{\Gamma}\rfloor \vdash \mathbf{ctx} \\ \Sigma \ni \mathsf{H} :^{\Phi} \sigma \qquad \Phi \hookrightarrow \Psi\end{array}}{\mathbf{\Gamma} \vdash \mathsf{H} \rightsquigarrow \mathsf{H} :^{\Psi} \sigma}$$

$$\frac{\begin{array}{c}\Sigma \ni f\,[\mathbf{\Delta}] :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi \\ \mathbf{\Gamma} \vdash \updelta \rightsquigarrow \delta : \mathbf{\Delta} \mathbin{/\!\!/} \Psi\end{array}}{\mathbf{\Gamma} \vdash f(\updelta) \rightsquigarrow f(\delta) :^{\Psi} [\delta/\mathbf{\Delta}]\,\kappa} \qquad \frac{\begin{array}{c}\mathbf{\Gamma} \vdash \uprho \rightsquigarrow \rho :^{\Psi} (\mathbf{\Delta}) \to \sigma \\ \mathbf{\Gamma} \vdash \updelta \rightsquigarrow \delta : \mathbf{\Delta} \mathbin{/\!\!/} \Psi\end{array}}{\mathbf{\Gamma} \vdash \uprho\,\updelta \rightsquigarrow \rho\,\delta :^{\Psi} [\delta/\mathbf{\Delta}]\,\sigma}$$

$$\frac{\begin{array}{c}\mathbf{\Gamma} \vdash \upkappa \rightsquigarrow \kappa :^{\forall} * \\ \mathbf{\Gamma}, a :^{\forall}_{e} \kappa \vdash \uptau \rightsquigarrow \tau :^{\forall} *\end{array}}{\mathbf{\Gamma} \vdash \forall(a{:}\upkappa) \to \uptau \rightsquigarrow (a{:}^{\forall}\kappa) \to \tau :^{\forall} *} \qquad \frac{\begin{array}{c}\mathbf{\Gamma} \vdash \uptau \rightsquigarrow \tau :^{\forall} * \\ \mathbf{\Gamma}, x :^{\Pi}_{e} \tau \vdash \upupsilon \rightsquigarrow \upsilon :^{\forall} *\end{array}}{\mathbf{\Gamma} \vdash \Pi(x{:}\uptau) \to \upupsilon \rightsquigarrow (x{:}^{\Pi}\tau) \to \upsilon :^{\forall} *}$$

$$\frac{\mathbf{\Gamma} \vdash \uptau \rightsquigarrow \tau :^{\forall} * \qquad \mathbf{\Gamma} \vdash \upupsilon \rightsquigarrow \upsilon :^{\forall} *}{\mathbf{\Gamma} \vdash \uptau \to \upupsilon \rightsquigarrow \tau \to \upsilon :^{\forall} *} \qquad \frac{\lfloor\mathbf{\Gamma}\rfloor \vdash \mathbf{ctx}}{\mathbf{\Gamma} \vdash * \rightsquigarrow * :^{\forall} *}$$

$$\frac{\lfloor\mathbf{\Gamma}\rfloor \vdash \mathbf{ctx}}{\mathbf{\Gamma} \vdash (\sim) \rightsquigarrow (\sim) :^{\forall} (a :^{\forall}_{i} *) \to (b :^{\forall}_{i} *) \to a \to b \to *}$$

$$\frac{\mathbf{\Gamma}, x :^{\Phi}_{e} \tau \vdash \mathsf{t} \rightsquigarrow e :^{\curlywedge} \sigma}{\mathbf{\Gamma} \vdash \lambda x \,.\, \mathsf{t} \rightsquigarrow \Lambda x{:}^{\Phi}\tau \,.\, e :^{\curlywedge} (x :^{\Phi}_{e} \tau) \to \sigma} \qquad \frac{\mathbf{\Gamma}, a :^{\Phi}_{i} \tau \vdash \mathsf{t} \rightsquigarrow e :^{\curlywedge} \sigma}{\mathbf{\Gamma} \vdash \mathsf{t} \rightsquigarrow \Lambda a{:}^{\Phi}\tau \,.\, e :^{\curlywedge} (a :^{\Phi}_{i} \tau) \to \sigma}$$

$$\frac{\begin{array}{c}\mathbf{\Gamma} \vdash \mathsf{s} \rightsquigarrow e :^{\curlywedge} \sigma \\ \mathbf{\Gamma}, x :^{\curlywedge}_{e} \sigma \vdash \mathsf{t} \rightsquigarrow e' :^{\curlywedge} \sigma'\end{array}}{\mathbf{\Gamma} \vdash \mathbf{let}\, x {=} \mathsf{s}\, \mathbf{in}\, \mathsf{t} \rightsquigarrow (\lambda x{:}\lfloor\sigma\rfloor \,.\, e')\; e :^{\curlywedge} \sigma'} \qquad \frac{\begin{array}{c}\mathbf{\Gamma} \vdash \upsigma \;\rightsquigarrow\; \sigma \\ \mathbf{\Gamma} \vdash \uprho \rightsquigarrow \rho :^{\Psi} \sigma\end{array}}{\mathbf{\Gamma} \vdash (\uprho{:}\upsigma) \rightsquigarrow \rho :^{\Psi} \sigma}$$

$$\frac{\lfloor\mathbf{\Gamma}\rfloor \vdash \rho \;:^{\Psi}\; \tau}{\mathbf{\Gamma} \vdash \_ \rightsquigarrow \rho :^{\Psi} \tau} \qquad \frac{\mathbf{\Gamma} \vdash \uprho \rightsquigarrow \rho :^{\Psi} \tau \qquad \lfloor\mathbf{\Gamma}\rfloor \vdash \gamma \;:^{\square}\; \tau \sim \upsilon}{\mathbf{\Gamma} \vdash \uprho \rightsquigarrow \rho \rhd \gamma :^{\Psi} \upsilon}$$

$$\frac{\mathbf{\Gamma} \vdash \mathsf{t} \rightsquigarrow e :^{\curlywedge} \sigma \qquad \mathbf{\Gamma} \vdash e : \sigma \prec e' : \sigma'}{\mathbf{\Gamma} \vdash \mathsf{t} \rightsquigarrow e' :^{\curlywedge} \sigma'}$$

Figure 7.4: Non-deterministic elaboration of expressions

This rule inserts an abstraction based on the type scheme, leaving the term t unchanged. The variable is *implicitly* bound in the context, so it cannot be used in the source language. Similarly, the rules for _ markers and conversion

$$\frac{\lfloor\mathbf{\Gamma}\rfloor \vdash \rho :^{\Psi} \tau}{\mathbf{\Gamma} \vdash \_ \rightsquigarrow \rho :^{\Psi} \tau} \qquad \frac{\mathbf{\Gamma} \vdash \mathrm{\rho} \rightsquigarrow \rho :^{\Psi} \tau \qquad \lfloor\mathbf{\Gamma}\rfloor \vdash \gamma :^{\square} \tau \sim \upsilon}{\mathbf{\Gamma} \vdash \mathrm{\rho} \rightsquigarrow \rho \rhd \gamma :^{\Psi} \upsilon}$$

invent evidence out of thin air. This shows the non-determinism of the system.

Applications are elaborated using the judgment for vectors of arguments, discussed below. For example, if $f : \mathsf{Bool} \rightarrow \mathsf{Int}$ then the source term $\mathsf{map}\ f$ will be elaborated using the telescope $(a :_{i}^{\forall} *, b :_{i}^{\forall} *, f :_{e}^{\curlywedge} (a \rightarrow b), x :_{e}^{\curlywedge} [a]) \rightarrow [b]$ of map, inserting the two implicit arguments to produce $\mathsf{map}\ \mathsf{Bool}\ \mathsf{Int}\ f$. Note that the vector may be empty, allowing constants (e.g. Nil) to take implicit arguments. Applications need not be saturated, except for applications of shared functions.

Non-deterministic elaboration of vectors

The judgment $\mathbf{\Gamma} \vdash \mathrm{\delta} \rightsquigarrow \delta : \mathbf{\Delta}$, defined in Figure 7.5, means that the vector $\mathrm{\delta}$ can be elaborated to δ in the annotated telescope $\mathbf{\Delta}$. This inserts implicit arguments: for example, the source vector containing the single entry Bool can be elaborated in the telescope $a :_{i}^{\forall} *, b :_{e}^{\forall} a$ to the two-element vector $*\ , \mathsf{Bool}$ where the first component has been inserted. Usually-implicit arguments may also have been explicitly specified by the user: for example, the source vector $\{a{=}\mathbb{Z}\}, 3$ can be elaborated in the telescope $a :_{i}^{\forall} *, b :_{e}^{\forall} a$ to $\mathbb{Z}\ , 3$.

Non-deterministic elaboration of type schemes

The judgment $\mathbf{\Gamma} \vdash \mathrm{\sigma} \rightsquigarrow \sigma$, also defined in Figure 7.5, means that the *inch* type scheme $\mathrm{\sigma}$ can be elaborated to σ. This is entirely structural; the only interesting behaviour is when elaborating types. Elaborating the codomain of a type scheme always takes place with the domain variable bound explicitly, even if the quantification is implicit, since the variable is still in scope for the codomain. As an example, the type scheme for $\mathsf{replicate}$

$$\forall\, a :: * \,.\, \Pi\ n :: \mathbb{N} \rightarrow a \rightarrow \mathsf{Vec}\ a\ n$$

can be elaborated to the *evidence* type scheme

$$(a :_{i}^{\forall} *, n :_{e}^{\Pi} \mathbb{N}, x :_{e}^{\curlywedge} a) \rightarrow \mathsf{Vec}\ a\ n\,.$$

$\boxed{\boldsymbol{\Gamma} \vdash \updelta \rightsquigarrow \delta : \boldsymbol{\Delta}}$ *(vector* $\updelta$ *can elaborate to* δ *in telescope* $\boldsymbol{\Delta}$*)*

$$\frac{}{\boldsymbol{\Gamma} \vdash \cdot \rightsquigarrow \cdot : \cdot} \qquad \frac{\boldsymbol{\Gamma} \vdash \uprho \rightsquigarrow \rho :^{\Phi} \sigma \qquad \boldsymbol{\Gamma} \vdash \updelta \rightsquigarrow \delta : [\rho/x]\,\boldsymbol{\Delta}}{\boldsymbol{\Gamma} \vdash \uprho, \updelta \rightsquigarrow \rho, \delta : x :^{\Phi}_{e} \sigma, \boldsymbol{\Delta}}$$

$$\frac{\begin{array}{c}\boldsymbol{\Gamma} \vdash \uprho \rightsquigarrow \rho :^{\Phi} \kappa \\ \boldsymbol{\Gamma} \vdash \updelta \rightsquigarrow \delta : [\rho/a]\,\boldsymbol{\Delta}\end{array}}{\boldsymbol{\Gamma} \vdash \{a = \uprho\}, \updelta \rightsquigarrow \rho, \delta : a :^{\Phi}_{i} \kappa, \boldsymbol{\Delta}} \qquad \frac{\begin{array}{c}\lfloor \boldsymbol{\Gamma} \rfloor \vdash \rho :^{\Phi} \kappa \\ \boldsymbol{\Gamma} \vdash \updelta \rightsquigarrow \delta : [\rho/a]\,\boldsymbol{\Delta}\end{array}}{\boldsymbol{\Gamma} \vdash \updelta \rightsquigarrow \rho, \delta : a :^{\Phi}_{i} \kappa, \boldsymbol{\Delta}}$$

$\boxed{\boldsymbol{\Gamma} \vdash \upsigma \rightsquigarrow \sigma}$ *(scheme* $\upsigma$ *can elaborate to* σ*)*

$$\frac{\boldsymbol{\Gamma} \vdash \uptau \rightsquigarrow \tau :^{\forall} *}{\boldsymbol{\Gamma} \vdash \uptau \rightsquigarrow \tau} \qquad \frac{\boldsymbol{\Gamma} \vdash \upkappa \rightsquigarrow \kappa :^{\forall} * \qquad \boldsymbol{\Gamma}, a :^{\forall}_{e} \kappa \vdash \upsigma \rightsquigarrow \sigma}{\boldsymbol{\Gamma} \vdash \forall (a{:}\upkappa).\, \upsigma \rightsquigarrow (a :^{\forall}_{i} \kappa) \to \sigma}$$

$$\frac{\begin{array}{c}\boldsymbol{\Gamma} \vdash \upkappa \rightsquigarrow \kappa :^{\forall} * \\ \boldsymbol{\Gamma}, a :^{\forall}_{e} \kappa \vdash \upsigma \rightsquigarrow \sigma\end{array}}{\boldsymbol{\Gamma} \vdash \forall (a{:}\upkappa) \to \upsigma \rightsquigarrow (a :^{\forall}_{e} \kappa) \to \sigma} \qquad \frac{\begin{array}{c}\boldsymbol{\Gamma} \vdash \uptau \rightsquigarrow \tau :^{\forall} * \\ \boldsymbol{\Gamma}, x :^{\Pi}_{e} \tau \vdash \upsigma \rightsquigarrow \sigma\end{array}}{\boldsymbol{\Gamma} \vdash \Pi (x{:}\uptau) \to \upsigma \rightsquigarrow (x :^{\Pi}_{e} \tau) \to \sigma}$$

$$\frac{\begin{array}{c}\boldsymbol{\Gamma} \vdash \uptau \rightsquigarrow \tau :^{\forall} * \\ \boldsymbol{\Gamma}, x :^{\Pi}_{e} \tau \vdash \upsigma \rightsquigarrow \sigma\end{array}}{\boldsymbol{\Gamma} \vdash \Pi (x{:}\uptau).\, \upsigma \rightsquigarrow (x :^{\Pi}_{i} \tau) \to \sigma} \qquad \frac{\begin{array}{c}\boldsymbol{\Gamma} \vdash \uptau \rightsquigarrow \varphi :^{\forall} * \\ \boldsymbol{\Gamma}, c :^{\Box}_{e} \varphi \vdash \upsigma \rightsquigarrow \sigma\end{array}}{\boldsymbol{\Gamma} \vdash \uptau \Rightarrow \upsigma \rightsquigarrow (c :^{\Box}_{i} \varphi) \to \sigma}$$

$$\frac{\boldsymbol{\Gamma} \vdash \upsigma' \rightsquigarrow \sigma' \qquad \boldsymbol{\Gamma} \vdash \upsigma \rightsquigarrow \sigma}{\boldsymbol{\Gamma} \vdash \upsigma' \to \upsigma \rightsquigarrow (x :^{\curlywedge}_{e} \sigma') \to \sigma}$$

Figure 7.5: Non-deterministic elaboration of vectors and type schemes

$\boxed{\mathbf{\Gamma} \vdash e : \sigma \prec e' : \sigma'}$ *(scheme σ is subsumed by σ', converting e to e')*

$$\frac{\lfloor \mathbf{\Gamma} \rfloor \vdash e :^{\text{⅄}} \lfloor \sigma \rfloor}{\mathbf{\Gamma} \vdash e : \sigma \prec e : \sigma} \qquad \frac{\lfloor \mathbf{\Gamma} \rfloor \vdash \gamma :^{\Box} \tau \sim \upsilon}{\mathbf{\Gamma} \vdash e : \tau \prec e \rhd \gamma : \upsilon}$$

$$\frac{\mathbf{\Gamma}, y :^{\text{⅄}}_{e} \sigma'_0 \vdash y : \sigma'_0 \prec e' : \sigma_0 \qquad \mathbf{\Gamma}, y :^{\text{⅄}}_{e} \sigma'_0 \vdash e\, e' : \sigma_1 \prec e'' : \sigma'_1}{\mathbf{\Gamma} \vdash e : (x :^{\text{⅄}}_{e} \sigma_0) \to \sigma_1 \prec \Lambda y{:}^{\text{⅄}} \lfloor \sigma'_0 \rfloor \,.\, e'' : (y :^{\text{⅄}}_{e} \sigma'_0) \to \sigma'_1}$$

$$\frac{\lfloor \mathbf{\Gamma} \rfloor \vdash \gamma :^{\Box} \upsilon \sim \tau \qquad \mathbf{\Gamma}, b :^{\Upsilon}_{e} \upsilon \vdash e\,(b \rhd \gamma) : [b \rhd \gamma / a]\, \sigma \prec e' : \sigma'}{\mathbf{\Gamma} \vdash e : (a :^{\Upsilon}_{e} \tau) \to \sigma \prec \Lambda b{:}^{\Upsilon} \upsilon \,.\, e' : (b :^{\Upsilon}_{e} \upsilon) \to \sigma'}$$

$$\frac{\begin{array}{l}\lfloor \mathbf{\Gamma} \rfloor \vdash \rho :^{\Phi} \tau \\ \mathbf{\Gamma} \vdash e\, \rho : [\rho/a]\, \sigma \prec e' : \sigma'\end{array}}{\mathbf{\Gamma} \vdash e : (a :^{\Phi}_{i} \tau) \to \sigma \prec e' : \sigma'} \qquad \frac{\mathbf{\Gamma}, a :^{\Phi}_{i} \tau \vdash e : \sigma \prec e' : \sigma'}{\mathbf{\Gamma} \vdash e : \sigma \prec \Lambda a{:}^{\Phi} \tau \,.\, e' : (a :^{\Phi}_{i} \tau) \to \sigma'}$$

Figure 7.6: Non-deterministic subsumption

7.3.2 Subsumption

Programs involving higher-rank types may require the elaborator to do more than insert implicit arguments in order to assign the right type. For example, if

$x :: \forall\, a \,.\, \mathsf{Bool} \to a$
$y :: (\forall\, b \,.\, (\forall\, c \,.\, c) \to b) \to \mathsf{Bool}$

then the application $y\ x$ should be well-typed. The elaborator must check that x has the scheme $\forall\, b \,.\, (\forall\, c \,.\, c) \to b$, which is more specific than $\forall\, a \,.\, \mathsf{Bool} \to a$ thanks to the contravariance in the domain, as Bool is more specific than $\forall\, c \,.\, c$.

The conversion rule for terms

$$\frac{\mathbf{\Gamma} \vdash \mathrm{t} \rightsquigarrow e :^{\text{⅄}} \sigma \qquad \mathbf{\Gamma} \vdash e : \sigma \prec e' : \sigma'}{\mathbf{\Gamma} \vdash \mathrm{t} \rightsquigarrow e' :^{\text{⅄}} \sigma'}$$

invokes the subsumption judgment $\mathbf{\Gamma} \vdash e : \sigma \prec e' : \sigma'$ to verify that σ' is more general than σ. This judgment, defined in Figure 7.6, constructs e' corresponding to e but with appropriate (implicit) abstractions and applications so that it has type scheme σ' rather than σ.

In the example given above, $e = x$ with scheme $\sigma = (a :^{\forall}_{i} *, z :^{\text{⅄}}_{e} \mathsf{Bool}) \to a$ and $\sigma' = (b :^{\forall}_{i} *, z :^{\text{⅄}}_{e} ((c :^{\forall}_{i} *) \to c)) \to b$. The variable b is bound in the context, so it can be substituted for a. Then both schemes are explicit ⅄-quantifications, so the contravariance rule applies and checks that $(c :^{\forall}_{i} *) \to c$ is below Bool.

In turn, this instantiates c with Bool and applies reflexivity. Having checked the domains, the contravariance rule checks the codomains, which are identical. The resulting evidence term is $y\,(\Lambda b\!:^{\forall} * \,.\, \lambda z\!:\!((c\!:^{\forall}*) \to c)\,.\, x\,b\,(z\,\mathsf{Bool}))$.

Since subsumption involves inserting implicit λ-abstractions, it is only available for terms (at phase $\curlywedge$). It is not possible at a static phase Υ because there is no type-level λ-abstraction. Instead, the conversion rule for types

$$\frac{\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \tau \qquad \lfloor \mathbf{\Gamma} \rfloor \vdash \gamma :^{\square} \tau \sim \upsilon}{\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \rho \rhd \gamma :^{\Psi} \upsilon}$$

can only appeal to a proof of type equality. This restricts the utility of higher-rank definitions at the type level.

7.3.3 Soundness of non-deterministic elaboration

Obviously, the non-deterministic system should be *sound* in the sense that the resulting *evidence* expression is actually well-typed.

Theorem 7.1 (Soundness of non-deterministic elaboration)**.**

(a) If $\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \sigma$ *for* $\Psi \in \{\forall, \Pi, \curlywedge\}$*, then* $\lfloor \mathbf{\Gamma} \rfloor \vdash \rho :^{\Psi} \lfloor \sigma \rfloor$*.*

(b) If $\mathbf{\Gamma} \vdash \mathsf{\sigma} \rightsquigarrow \sigma$ *then* $\lfloor \mathbf{\Gamma} \rfloor \vdash \lfloor \sigma \rfloor :^{\forall} *$*.*

(c) If $\mathbf{\Gamma} \vdash \mathsf{\delta} \rightsquigarrow \delta : \mathbf{\Delta}$ *then* $\lfloor \mathbf{\Gamma} \rfloor \vdash \delta : \lfloor \mathbf{\Delta} \rfloor$*.*

(d) If $\mathbf{\Gamma} \vdash e : \sigma \prec e' : \sigma'$ *and* $\lfloor \mathbf{\Gamma} \rfloor \vdash e :^{\curlywedge} \lfloor \sigma \rfloor$ *then* $\lfloor \mathbf{\Gamma} \rfloor \vdash e' :^{\curlywedge} \lfloor \sigma' \rfloor$*.*

Proof. Straightforward structural induction on derivations. □

While this system provides a helpful starting point, it does not define an algorithm. The same syntax can be translated in many different ways depending on the placement of implicit applications and quantifications. For example, the *inch* term $\lambda x.\lambda y.x\,y$ could be translated to $\Lambda a\!:^{\forall}\!* \,.\, \Lambda b\!:^{\forall}\!* \,.\, \lambda x\!:\!(a \to b)\,.\,\lambda y\!:\!a\,.\,x\,y$ or $\Lambda b : ^{\forall} * \,.\, \lambda x : ((a :^{\forall} *) \to a \to b)\,.\,\lambda y : \mathsf{Bool}\,.\,x\,\mathsf{Bool}\,y$ or many other mutually-incompatible *evidence* terms, with no principal or canonical choice. Even if the type scheme is known, there are many unspecified choices.

To describe the deterministic algorithm, I must first extend the type system to support metavariables, which will stand for the unknown types and proof obligations (constraints) that arise during elaboration.

7.4 Metavariables and information increase

Just as in the unification and type inference algorithms of Part I, a metacontext Θ contains declarations of metavariables to represent unknowns that arise during elaboration. This includes types, represented by metavariables α and β, and coercion proofs ζ. Each metavariable has a telescope Δ of parameters, and a kind κ that may depend on Δ.

Metacontexts may also bind variables. Like the annotated contexts of Section 7.1, these record whether the binding is implicit or explicit. Source language programs may refer only to explicitly bound variables.

The grammar of metacontexts is given by

$$\begin{array}{lcll}
\Theta & ::= & & \text{metacontext} \\
 & \mid & \cdot & \text{empty} \\
 & \mid & \Theta, \alpha\,[\Delta] :^{\Phi} \kappa & \text{unknown metavariable} \\
 & \mid & \Theta, \alpha\,[\Delta] = \rho :^{\Phi} \kappa & \text{defined metavariable} \\
 & \mid & \Theta, a :^{\Phi}_{e} \sigma & \text{explicitly-bound variable} \\
 & \mid & \Theta, a :^{\Phi}_{i} \tau & \text{implicitly-bound variable}
\end{array}$$

I will use Ξ for a metacontext that contains only metavariables; the telescopes Γ, Δ are metacontexts that contain only variables.

As in previous chapters, metacontexts are ordered by dependency. Figure 7.7 gives the rules for a valid metacontext, generalising the judgment $\Gamma \vdash \textbf{ctx}$ defined in Figure 6.5 (page 118). This ensures that metavariables are defined uniquely and that their types are well-kinded. The sanity condition (Lemma 6.9, page 127) continues to hold: if $\Theta \vdash \textbf{mctx}$ then $\Sigma \vdash \textbf{sig}$. The typing rules in the previous chapter are generalised by replacing Γ with Θ and $\Gamma \vdash \textbf{ctx}$ with $\Theta \vdash \textbf{mctx}$.

The syntax of *evidence* expressions is extended with a new form $\alpha^{[\delta]}$ for metavariable occurrences, where δ is a vector in Δ. I add a typing rule for metavariables to the rules in Figure 6.6 (page 119):

$$\frac{\Theta \ni \alpha\,[\Delta] :^{\Phi} \kappa \qquad \Gamma \vdash \delta : \Delta \qquad \Phi \hookrightarrow \Psi}{\Gamma \vdash \alpha^{[\delta]} :^{\Psi} [\delta/\Delta]\,\kappa}$$

Metasubstitutions

A metasubstitution $\theta : \Theta_0 \sqsubseteq \Theta_1$ gives values for metavariables in the metacontext Θ_0 in terms of the metacontext Θ_1. Since metavariables have parameters, each component of a metasubstitution takes the form $\Delta.\rho \,/\, \alpha$ where Δ is the telescope

$$\boxed{\Theta \vdash \mathbf{mctx}}$$

$$\frac{\Sigma \vdash \mathbf{sig}}{\cdot \vdash \mathbf{mctx}} \qquad \frac{\alpha \# \Theta \qquad \Theta, \Delta \vdash \kappa :^{\forall} *}{\Theta, \alpha\,[\Delta] :^{\Phi} \kappa \vdash \mathbf{mctx}} \qquad \frac{\alpha \# \Theta \qquad \Theta, \Delta \vdash \rho :^{\Phi} \kappa}{\Theta, \alpha\,[\Delta] = \rho :^{\Phi} \kappa \vdash \mathbf{mctx}}$$

$$\frac{a \# \Theta \qquad \Theta \vdash \lfloor \sigma \rfloor :^{\forall} * \qquad \Phi \neq \Box}{\Theta, a :_{e}^{\Phi} \sigma \vdash \mathbf{mctx}} \qquad \frac{a \# \Theta \qquad \Theta \vdash \tau :^{\forall} * \qquad \Phi \neq \Box}{\Theta, a :_{i}^{\Phi} \tau \vdash \mathbf{mctx}}$$

$$\frac{c \# \Theta \qquad \Theta \vdash \varphi :^{\forall} *}{\Theta, c :_{i}^{\Box} \varphi \vdash \mathbf{mctx}}$$

Figure 7.7: Validity of metacontexts

$$\boxed{\theta : \Theta_0 \sqsubseteq \Theta_1}$$

$$\frac{}{\cdot : \cdot \sqsubseteq \Xi} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1, \theta\Delta \vdash \rho :^{\Phi} \theta\,\kappa}{(\theta, \Delta.\rho \,/\, \alpha) : \Theta_0, \alpha\,[\Delta] :^{\Phi} \kappa \sqsubseteq \Theta_1}$$

$$\frac{\theta : \Theta_0 \sqsubseteq \Theta_1 \qquad \Theta_1, \theta\Delta \vdash \rho \equiv \theta\,\rho' :^{\Phi} \theta\,\kappa}{(\theta, \Delta.\rho \,/\, \alpha) : \Theta_0, \alpha\,[\Delta] = \rho' :^{\Phi} \kappa \sqsubseteq \Theta_1} \qquad \frac{\theta : \Theta_0 \sqsubseteq \Theta_1}{\theta : \Theta_0, a :_{e}^{\Phi} \sigma \sqsubseteq \Theta_1, a :_{e}^{\Phi} \theta\,\sigma, \Xi}$$

$$\frac{\theta : \Theta_0 \sqsubseteq \Theta_1}{\theta : \Theta_0, a :_{i}^{\Phi} \tau \sqsubseteq \Theta_1, a :_{i}^{\Phi} \theta\,\tau, \Xi}$$

Figure 7.8: Metasubstitutions

for α, and binds variables in ρ. Valid metasubstitutions are defined in Figure 7.8. The rules ensure that metasubstitutions preserve the structure of variables in the metacontexts, as in Subsection 2.1.2 (page 14).

Metasubstitutions act on syntax by the structural closure of

$$\theta\,(\alpha^{[\delta]}) \mapsto [\delta/\Delta]\,\tau \quad \text{where } \theta \text{ contains } \Delta.\tau \,/\, \alpha.$$

The identity metasubstitution ι is defined in the usual way, replacing each metavariable with itself. I write $\Theta_0 \sqsubseteq \Theta_1$ where the information increase is by the identity metasubstitution.

Lemma 7.2 (Metasubstitution)**.** *If $\theta : \Theta_0 \sqsubseteq \Theta_1$ and $\Theta_0 \vdash J$, then $\Theta_1 \vdash \theta\, J$.*

Proof. By induction on derivations. □

7.5 Deterministic elaboration

The deterministic elaboration algorithm is built from the non-deterministic relation by attaching input and output metavariable contexts, allowing missing information to be replaced with metavariables. It is defined in a bidirectional style, based on the following judgments defined in Figures 7.9 and 7.10:

- $\Theta_0 \vdash^{\Psi} \mathsf{\rho} \rightsquigarrow^{\mathrm{sch}} \rho : \sigma \dashv \Theta_1$, meaning that $\mathsf{\rho}$ elaborates at phase Ψ to ρ with assigned type scheme σ;
- $\Theta_0 \vdash^{\Psi} \mathsf{\rho} \rightsquigarrow \rho : \tau \dashv \Theta_1$, meaning that $\mathsf{\rho}$ elaborates at phase Ψ to ρ with inferred type τ; and
- $\Theta_0 \vdash^{\Psi} \mathsf{\rho} : \sigma \rightsquigarrow \rho \dashv \Theta_1$, meaning that elaborating $\mathsf{\rho}$ with the type scheme σ at phase Ψ produces the *evidence* term ρ.

The following auxiliary judgments are defined in Figures 7.11–7.13:

- $\Theta_0 \vdash \mathsf{\sigma} \rightsquigarrow \sigma \dashv \Theta_1$, meaning that the type scheme $\mathsf{\sigma}$ elaborates to σ;
- $\Theta_0 \vdash^{\Psi} (\rho : \sigma)\, \mathsf{\delta} \rightsquigarrow \rho' : \tau \dashv \Theta_1$, meaning that elaborating the spine of arguments $\mathsf{\delta}$ applied to the elaborated head $\rho : \sigma$ results in $\rho' : \tau$;
- $\Theta_0 \vdash \mathsf{\delta} : \mathbf{\Delta} \rightsquigarrow \delta \dashv \Theta_1$, meaning that elaborating the components of the vector $\mathsf{\delta}$ in the telescope $\mathbf{\Delta}$ results in δ; and
- $\Theta_0 \vdash e : \sigma \prec \sigma' \rightsquigarrow e' \dashv \Theta_1$, meaning that the scheme σ is subsumed by σ' and if e has scheme σ then e' is the corresponding term with scheme σ'.

Finally, the judgment $\Theta_0 \vdash \tau \sim \upsilon \rightsquigarrow \gamma \dashv \Theta_1$, means that τ and υ are unified, witnessed by the coercion γ. This is an invocation of the constraint solver, which I do not specify in detail. I discuss this further in Subsection 7.5.1.

For all these judgments, the parameters before the arrow $\rightsquigarrow$ are inputs, and they determine the outputs (which appear after the arrow). In general, information flows clockwise through each inference rule, with the inputs to the conclusion determining the inputs to the first premise, whose outputs determine the inputs to the next premise, and so forth, until the outputs from all the premises determine the outputs of the conclusion. In this way, the rules yield an algorithm.

The distinction between scheme assignment $\Theta_0 \vdash^{\Psi} \mathsf{\rho} \rightsquigarrow^{\mathrm{sch}} \rho : \sigma \dashv \Theta_1$ and type inference $\Theta_0 \vdash^{\Psi} \mathsf{\rho} \rightsquigarrow \rho : \tau \dashv \Theta_1$ is that schemes are not inferred, only looked up in the context or explicitly annotated by the user. A single application rule allows an expression with a scheme to have its type inferred, by checking the vector of arguments (which may be empty) and completing the scheme to produce a type. Expressions with inferred types are embedded in those with assigned schemes because the head of an application may be a λ-expression (i.e. in a β-redex) or other expression that does not have an assigned scheme.

Example of elaboration

Recall the example *inch* term $\lambda x.\lambda y.x\,y$. This is elaborated by generating fresh metavariables for the domain types, so under the abstractions the context will be $\alpha\,[\cdot\,] :^{\forall} *, x :^{\lambda}_{e} \alpha, \beta\,[\cdot\,] :^{\forall} *, y :^{\lambda}_{e} \beta$. The application $x\,y$ is elaborated by looking up the type α of x in the context, and checking the vector y against it. Since the type does not start with a quantifier, fresh metavariables α_0 and α_1 for the domain and codomain are created, and the constraint $\alpha \sim (\alpha_0 \to \alpha_1)$ passed to the constraint solver. Then y is checked at type α_0, but looking up its type gives β and the subsumption judgment generates another constraint, $\beta \sim \alpha_0$. Assuming no constraint solving, the resulting evidence term is

$$\lambda x{:}\alpha\,.\,\lambda y{:}\beta\,.\,(x \triangleright \zeta)\,(y \triangleright \zeta')$$

in the context

$$\alpha\,[\cdot\,] :^{\forall} *, \beta\,[\cdot\,] :^{\forall} *, \alpha_0\,[\cdot\,] :^{\forall} *, \alpha_1\,[\cdot\,] :^{\forall} *, \zeta\,[\cdot\,] :^{\square} \alpha \sim (\alpha_0 \to \alpha_1), \zeta'\,[\cdot\,] :^{\square} \beta \sim \alpha_0.$$

In practice, the unifier will solve the constraints to give the context

$$\alpha_0\,[\cdot\,] :^{\forall} *, \alpha_1\,[\cdot\,] :^{\forall} *, \alpha\,[\cdot\,] = \alpha_0 \to \alpha_1 :^{\forall} *, \beta\,[\cdot\,] = \alpha_0 :^{\forall} *$$

$\boxed{\Theta_0 \vdash^{\Psi} \mathsf{\rho} : \sigma \rightsquigarrow \rho \dashv \Theta_1}$ *(checking $\mathsf{\rho}$ at scheme σ and phase Ψ delivers ρ)*

$$\frac{\Theta_0, a :_i^{\Phi} \kappa \vdash^{\curlywedge} \mathrm{t} : \sigma \rightsquigarrow e \dashv \Theta_1, a :_i^{\Phi} \kappa, \Xi}{\Theta_0 \vdash^{\curlywedge} \mathrm{t} : (a :_i^{\Phi} \kappa) \to \sigma \rightsquigarrow \Lambda a :^{\Phi} \kappa \,.\, e \dashv \Theta_1, (a :^{\Phi} \kappa) \nearrow \Xi}$$

$$\frac{\Theta_0, x :_e^{\Phi} \sigma' \vdash^{\curlywedge} \mathrm{t} : \sigma \rightsquigarrow e \dashv \Theta_1, x :_e^{\Phi} \sigma', \Xi}{\Theta_0 \vdash^{\curlywedge} \lambda x \,.\, \mathrm{t} : (x :_e^{\Phi} \sigma') \to \sigma \rightsquigarrow (\Lambda x :^{\Phi} \lfloor \sigma' \rfloor \,.\, e) \dashv \Theta_1, (x :^{\Phi} \sigma') \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\curlywedge} \mathrm{s} \rightsquigarrow^{\mathrm{sch}} e : \sigma \dashv \Theta_1 \qquad \Theta_1, x :_e^{\curlywedge} \sigma \vdash^{\curlywedge} \mathrm{t} : \sigma' \rightsquigarrow e' \dashv \Theta_2, x :_e^{\curlywedge} \sigma, \Xi}{\Theta_0 \vdash^{\curlywedge} (\mathbf{let}\, x = \mathrm{s}\, \mathbf{in}\, \mathrm{t}) : \sigma' \rightsquigarrow (\lambda x : \lfloor \sigma \rfloor \,.\, e')\, e \dashv \Theta_2, \Xi}$$

$$\frac{}{\Theta_0 \vdash^{\Psi} \_ : \tau \rightsquigarrow \beta \dashv \Theta_0, \beta\, [\cdot\,] :^{\Psi} \tau} \qquad \frac{\begin{array}{c}\Theta_0 \vdash^{\curlywedge} \mathrm{t} \rightsquigarrow^{\mathrm{sch}} e : \sigma \dashv \Theta_1 \\ \Theta_1 \vdash e : \sigma \prec \sigma' \rightsquigarrow e' \dashv \Theta_2\end{array}}{\Theta_0 \vdash^{\curlywedge} \mathrm{t} : \sigma' \rightsquigarrow e' \dashv \Theta_2}$$

$$\frac{\Theta_0 \vdash^{\Upsilon} \mathsf{\rho} \rightsquigarrow \rho : \tau \dashv \Theta_1 \qquad \Theta_1 \vdash \tau \sim \upsilon \rightsquigarrow \gamma \dashv \Theta_2}{\Theta_0 \vdash^{\Upsilon} \mathsf{\rho} : \upsilon \rightsquigarrow \rho \rhd \gamma \dashv \Theta_2}$$

Figure 7.9: Type-checking elaboration

with reflexive proofs of the coercion metavariables, and the evidence term

$$\lambda x : (\alpha_0 \to \alpha_1) \,.\, \lambda y : \alpha_0 \,.\, (x \rhd \langle \alpha_0 \to \alpha_1 \rangle)\, (y \rhd \langle \alpha_0 \rangle).$$

Parameterisation

The operation $\Delta \nearrow \Xi$ parameterises the metavariables Ξ over a telescope Δ:

$$\begin{array}{lcl} \Delta \nearrow \cdot & \mapsto & \cdot \\ \Delta \nearrow (\alpha\, [\Gamma] :^{\Phi} \kappa, \Xi) & \mapsto & \alpha\, [\Delta, \Gamma] :^{\Phi} \kappa, \Delta \nearrow \Xi \end{array}$$

This allows a telescope of variables to be taken out of scope during elaboration, such that any metavariables introduced retain the appropriate parameters: if $\Theta, \Delta, \Xi \vdash$ **mctx** then $\Theta, \Delta \nearrow \Xi \vdash$ **mctx**. It permutes existential quantifiers from right to left past universal quantifiers, the 'raising' of Miller (1992).

This definition and its uses involve a slight abuse of notation, as formally all occurrences of metavariables from Ξ should be replaced with occurrences in which the parameters are prefixed by the identity substitution: for example, $a :_e^{\forall} \kappa, \beta\, [\cdot\,] :^{\forall} \kappa \vdash \beta^{[\cdot]} :^{\forall} \kappa$ but $\beta\, [a :^{\forall} \kappa] :^{\forall} \kappa, a :_e^{\forall} \kappa \vdash \beta^{[a]} :^{\forall} \kappa$. In practice, I will elide the necessary weakenings.

$\boxed{\Theta_0 \vdash^{\Psi} \mathsf{\rho} \leadsto^{\mathrm{sch}} \rho : \sigma \dashv \Theta_1}$ *($\mathsf{\rho}$ elaborates to ρ with assigned scheme σ)*

$$\frac{\Theta_0 \ni x :^{\Phi}_{e} \sigma \qquad \Phi \hookrightarrow \Psi}{\Theta_0 \vdash^{\Psi} x \leadsto^{\mathrm{sch}} x : \sigma \dashv \Theta_1} \qquad \frac{\Sigma \ni \mathsf{H} :^{\Phi} \sigma \qquad \Phi \hookrightarrow \Psi}{\Theta_0 \vdash^{\Psi} \mathsf{H} \leadsto^{\mathrm{sch}} \mathsf{H} : \sigma \dashv \Theta_1}$$

$$\frac{\begin{array}{c}\Theta_0 \vdash \mathsf{\sigma} \leadsto \sigma \dashv \Theta_1\\ \Theta_1 \vdash^{\Psi} \mathsf{\rho} : \sigma \leadsto \rho \dashv \Theta_2\end{array}}{\Theta_0 \vdash^{\Psi} (\mathsf{\rho} : \mathsf{\sigma}) \leadsto^{\mathrm{sch}} \rho : \sigma \dashv \Theta_2} \qquad \frac{\Theta_0 \vdash^{\Psi} \mathsf{\rho} \leadsto \rho : \tau \dashv \Theta_1}{\Theta_0 \vdash^{\Psi} \mathsf{\rho} \leadsto^{\mathrm{sch}} \rho : \tau \dashv \Theta_1}$$

$\boxed{\Theta_0 \vdash^{\Psi} \mathsf{\rho} \leadsto \rho : \tau \dashv \Theta_1}$ *($\mathsf{\rho}$ elaborates to ρ with inferred type τ)*

$$\frac{\begin{array}{c}\Theta_0 \vdash^{\Psi} \mathsf{\rho} \leadsto^{\mathrm{sch}} \rho : \sigma \dashv \Theta_1\\ \Theta_1 \vdash^{\Psi} (\rho : \sigma)\, \mathsf{\delta} \leadsto \rho' : \tau \dashv \Theta_2\end{array}}{\Theta_0 \vdash^{\Psi} \mathsf{\rho}\, \mathsf{\delta} \leadsto \rho' : \tau \dashv \Theta_2} \qquad \frac{\begin{array}{c}\Sigma \ni f\,[\mathbf{\Delta}] :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi\\ \Theta_0 \vdash \mathsf{\delta} : \mathbf{\Delta} \mathbin{/\!\!/} \Psi \leadsto \delta \dashv \Theta_1\end{array}}{\Theta_0 \vdash^{\Psi} f(\mathsf{\delta}) \leadsto f(\delta) : [\delta/\mathbf{\Delta}]\, \kappa \dashv \Theta_2}$$

$$\frac{\Theta_0 \vdash^{\forall} \mathsf{\kappa} : * \leadsto \kappa \dashv \Theta_1 \qquad \Theta_1, a :^{\forall}_{e} \kappa \vdash^{\forall} \mathsf{\tau} : * \leadsto \tau \dashv \Theta_2, a :^{\forall}_{e} \kappa, \Xi}{\Theta_0 \vdash^{\forall} \forall (a : \mathsf{\kappa}) \to \mathsf{\tau} \leadsto (a :^{\forall} \kappa) \to \tau : * \dashv \Theta_2, (a :^{\forall} \kappa) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\forall} \mathsf{\tau} : * \leadsto \tau \dashv \Theta_1 \qquad \Theta_1, x :^{\Pi}_{e} \tau \vdash^{\forall} \mathsf{\upsilon} : * \leadsto \upsilon \dashv \Theta_2, x :^{\Pi}_{e} \tau, \Xi}{\Theta_0 \vdash^{\forall} \Pi (x : \mathsf{\tau}) \to \mathsf{\upsilon} \leadsto (x :^{\Pi} \tau) \to \upsilon : * \dashv \Theta_2, (x :^{\Pi} \tau) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\forall} \mathsf{\tau} : * \leadsto \tau \dashv \Theta_1 \qquad \Theta_1 \vdash^{\forall} \mathsf{\upsilon} : * \leadsto \upsilon \dashv \Theta_2}{\Theta_0 \vdash^{\forall} \mathsf{\tau} \to \mathsf{\upsilon} \leadsto \tau \to \upsilon : * \dashv \Theta_2}$$

$$\frac{\Theta_0, \alpha\,[\cdot\,] :^{\forall} *, x :^{\curlywedge}_{e} \alpha \vdash^{\curlywedge} \mathrm{t} \leadsto e : \tau \dashv \Theta_1, x :^{\curlywedge}_{e} \alpha, \Xi}{\Theta_0 \vdash^{\curlywedge} \lambda x \,.\, \mathrm{t} \leadsto (\lambda x : \alpha \,.\, e) : (\alpha \to \tau) \dashv \Theta_1, \Xi}$$

$$\frac{\Theta_0 \vdash^{\curlywedge} \mathrm{s} \leadsto^{\mathrm{sch}} e : \sigma \dashv \Theta_1 \qquad \Theta_1, x :^{\curlywedge}_{e} \sigma \vdash^{\curlywedge} \mathrm{t} \leadsto e' : \tau \dashv \Theta_2, x :^{\curlywedge}_{e} \sigma, \Xi}{\Theta_0 \vdash^{\curlywedge} (\mathbf{let}\, x {=} \mathrm{s}\, \mathbf{in}\, \mathrm{t}) \leadsto (\lambda x : \lfloor \sigma \rfloor \,.\, e')\, e : \tau \dashv \Theta_2, \Xi}$$

$$\frac{}{\Theta_0 \vdash^{\Psi} \_ \leadsto \beta : \alpha \dashv \Theta_0, \alpha\,[\cdot\,] :^{\forall} *, \beta\,[\cdot\,] :^{\Psi} \alpha}$$

Figure 7.10: Type-reconstructing elaboration

$\boxed{\Theta_0 \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1}$ *(scheme $\upsigma$ elaborates to σ)*

$$\frac{\Theta_0 \vdash^{\forall} \uptau : * \rightsquigarrow \tau \dashv \Theta_1}{\Theta_0 \vdash \uptau \rightsquigarrow \tau \dashv \Theta_1}$$

$$\frac{\Theta_0 \vdash^{\forall} \upkappa : * \rightsquigarrow \kappa \dashv \Theta_1 \qquad \Theta_1, a :^{\forall}_{e} \kappa \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1, a :^{\forall}_{e} \kappa, \Xi}{\Theta_0 \vdash \forall (a{:}\upkappa).\, \upsigma \rightsquigarrow (a :^{\forall}_{i} \kappa) \to \sigma \dashv \Theta_1, (a :^{\forall} \kappa) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\forall} \upkappa : * \rightsquigarrow \kappa \dashv \Theta_1 \qquad \Theta_1, a :^{\forall}_{e} \kappa \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1, a :^{\forall}_{e} \kappa, \Xi}{\Theta_0 \vdash \forall (a{:}\upkappa) \to \upsigma \rightsquigarrow (a :^{\forall}_{e} \kappa) \to \sigma \dashv \Theta_1, (a :^{\forall} \kappa) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\forall} \uptau : * \rightsquigarrow \tau \dashv \Theta_1 \qquad \Theta_1, x :^{\Pi}_{e} \tau \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1, x :^{\Pi}_{e} \tau, \Xi}{\Theta_0 \vdash \Pi (x{:}\uptau) \to \upsigma \rightsquigarrow (x :^{\Pi}_{e} \tau) \to \sigma \dashv \Theta_1, (x :^{\Pi} \tau) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\forall} \uptau : * \rightsquigarrow \tau \dashv \Theta_1 \qquad \Theta_1, x :^{\Pi}_{e} \tau \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1, x :^{\Pi}_{e} \tau, \Xi}{\Theta_0 \vdash \Pi (x{:}\uptau).\, \upsigma \rightsquigarrow (x :^{\Pi}_{i} \tau) \to \sigma \dashv \Theta_1, (x :^{\Pi} \tau) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash^{\forall} \uptau : * \rightsquigarrow \varphi \dashv \Theta_1 \qquad \Theta_1, c :^{\forall}_{e} \varphi \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1, c :^{\forall}_{e} \varphi, \Xi}{\Theta_0 \vdash \uptau \Rightarrow \upsigma \rightsquigarrow (c :^{\Box}_{i} \varphi) \to \sigma \dashv \Theta_1, (c :^{\Box} \varphi) \nearrow \Xi}$$

$$\frac{\Theta_0 \vdash \upsigma' \rightsquigarrow \sigma' \dashv \Theta_1 \qquad \Theta_1 \vdash \upsigma \rightsquigarrow \sigma \dashv \Theta_1}{\Theta_0 \vdash \upsigma' \to \upsigma \rightsquigarrow (x :^{\curlywedge}_{e} \sigma') \to \sigma \dashv \Theta_1}$$

Figure 7.11: Elaboration of type schemes

$\boxed{\Theta_0 \vdash^{\Psi} (\rho : \sigma)\, \updelta \rightsquigarrow \rho' : \tau \dashv \Theta_1}$ *(applying $\rho : \sigma$ to $\updelta$ results in $\rho' : \tau$)*

$$\frac{}{\Theta \vdash^{\Psi} (\rho : \sigma) \cdot \rightsquigarrow \rho : \lfloor \sigma \rfloor \dashv \Theta} \qquad \frac{\begin{array}{c}\Theta_0 \vdash^{\Phi \,/\!\!/\, \Psi} \uprho' : \sigma' \rightsquigarrow \rho' \dashv \Theta_1 \\ \Theta_1 \vdash^{\Psi} (\rho\,\rho' : [\rho'/a]\,\sigma)\, \updelta \rightsquigarrow \rho'' : \tau \dashv \Theta_2\end{array}}{\Theta_0 \vdash^{\Psi} (\rho : (a :^{\Phi}_{e} \sigma') \to \sigma)\, (\uprho', \updelta) \rightsquigarrow \rho'' : \tau \dashv \Theta_2}$$

$$\frac{\begin{array}{c}\Theta_0 \vdash^{\Phi \,/\!\!/\, \Psi} \uprho' : \kappa \rightsquigarrow \rho' \dashv \Theta_1 \\ \Theta_1 \vdash^{\Psi} (\rho\,\rho' : [\rho'/a]\,\sigma)\, \updelta \rightsquigarrow \rho'' : \tau \dashv \Theta_2\end{array}}{\Theta_0 \vdash^{\Psi} (\rho : (a :^{\Phi}_{i} \kappa) \to \sigma)\, (\{a = \uprho'\}, \updelta) \rightsquigarrow \rho'' : \tau \dashv \Theta_2}$$

$$\frac{\Theta_0, \alpha\,[\cdot\,] :^{\Phi} \kappa \vdash^{\Psi} (\rho\,\alpha : [\alpha/a]\,\sigma)\, \updelta \rightsquigarrow \rho' : \tau \dashv \Theta_1}{\Theta_0 \vdash^{\Psi} (\rho : (a :^{\Phi}_{i} \kappa) \to \sigma)\, \updelta \rightsquigarrow \rho' : \tau \dashv \Theta_1}$$

$$\frac{\begin{array}{l}\Theta_0, \alpha\,[\cdot\,] :^{\forall} *, \beta\,[\cdot\,] :^{\forall} * \vdash \upsilon \sim (\alpha \to \beta) \rightsquigarrow \gamma \dashv \Theta_1 \\ \Theta_1 \vdash^{\Psi} (\rho \rhd \gamma : \alpha \to \beta)\, \updelta \rightsquigarrow \rho' : \tau \dashv \Theta_2\end{array}}{\Theta_0 \vdash^{\Psi} (\rho : \upsilon)\, \updelta \rightsquigarrow \rho' : \tau \dashv \Theta_2}$$

$\boxed{\Theta_0 \vdash \updelta : \mathbf{\Delta} \rightsquigarrow \delta \dashv \Theta_1}$ *(vector $\updelta$ in telescope $\mathbf{\Delta}$ elaborates to δ)*

$$\frac{}{\Theta \vdash \cdot : \cdot \rightsquigarrow \cdot \dashv \Theta} \qquad \frac{\begin{array}{c}\Theta_0 \vdash^{\Phi} \uprho : \sigma \rightsquigarrow \rho \dashv \Theta_1 \\ \Theta_1 \vdash \updelta : [\rho/a]\,\mathbf{\Delta} \rightsquigarrow \delta \dashv \Theta_2\end{array}}{\Theta_0 \vdash \uprho, \updelta : a :^{\Phi}_{e} \sigma, \mathbf{\Delta} \rightsquigarrow \rho, \delta \dashv \Theta_2}$$

$$\frac{\begin{array}{c}\Theta_0 \vdash^{\Phi} \uprho : \kappa \rightsquigarrow \rho \dashv \Theta_1 \\ \Theta_1 \vdash \updelta : [\rho/a]\,\mathbf{\Delta} \rightsquigarrow \delta \dashv \Theta_2\end{array}}{\Theta_0 \vdash \{a = \uprho\}, \updelta : a :^{\Phi}_{i} \kappa, \mathbf{\Delta} \rightsquigarrow \rho, \delta \dashv \Theta_2} \qquad \frac{\Theta_0, \alpha\,[\cdot\,] :^{\Phi} \kappa \vdash \updelta : [\alpha/a]\,\mathbf{\Delta} \rightsquigarrow \delta \dashv \Theta_1}{\Theta_0 \vdash \updelta : a :^{\Phi}_{i} \kappa, \mathbf{\Delta} \rightsquigarrow \alpha, \delta \dashv \Theta_1}$$

Figure 7.12: Elaboration of spines and vectors

$\boxed{\Theta_0 \vdash e : \sigma \prec \sigma' \leadsto e' \dashv \Theta_1}$ *(σ is subsumed by σ', converting e to e')*

$$\frac{\Theta \vdash e :^{\curlywedge} \lfloor\sigma\rfloor}{\Theta \vdash e : \sigma \prec \sigma \leadsto e \dashv \Theta_1} \qquad \frac{\Theta_0 \vdash \tau \sim \upsilon \leadsto \gamma \dashv \Theta_1}{\Theta_0 \vdash e : \tau \prec \upsilon \leadsto e \rhd \gamma \dashv \Theta_1}$$

$$\frac{\begin{array}{c}\Theta_0, y :^{\curlywedge}_{e} \sigma'_0 \vdash y : \sigma'_0 \prec \sigma_0 \leadsto e' \dashv \Theta_1 \\ \Theta_1 \vdash e\, e' : \sigma_1 \prec \sigma'_1 \leadsto e'' \dashv \Theta_2, y :^{\curlywedge}_{e} \sigma'_0, \Xi\end{array}}{\Theta_0 \vdash e : (x :^{\curlywedge}_{e} \sigma_0) \to \sigma_1 \prec (y :^{\curlywedge}_{e} \sigma'_0) \to \sigma'_1 \leadsto \lambda y{:}\lfloor\sigma'_0\rfloor \,.\, e'' \dashv \Theta_2, \Xi}$$

$$\frac{\begin{array}{l}\Theta_0 \vdash \upsilon \sim \tau \leadsto \gamma \dashv \Theta_1 \\ \Theta_1, b :^{\Upsilon}_{e} \upsilon \vdash e\,(b \rhd \gamma) : [b \rhd \gamma/a]\,\sigma \prec \sigma' \leadsto e' \dashv \Theta_2, b :^{\Upsilon}_{e} \upsilon, \Xi\end{array}}{\Theta_0 \vdash e : (a :^{\Upsilon}_{e} \tau) \to \sigma \prec (b :^{\Upsilon}_{e} \upsilon) \to \sigma' \leadsto \Lambda b{:}^{\Upsilon}\upsilon \,.\, e' \dashv \Theta_2, (b :^{\Upsilon} \upsilon) \nearrow \Xi}$$

$$\frac{\Theta_0, a :^{\Phi}_{i} \kappa \vdash e : \sigma \prec \sigma' \leadsto e' \dashv \Theta_1, a :^{\Phi}_{i} \kappa, \Xi}{\Theta_0 \vdash e : \sigma \prec (a :^{\Phi}_{i} \kappa) \to \sigma' \leadsto \Lambda a{:}^{\Phi}\kappa \,.\, e' \dashv \Theta_1, (a :^{\Phi} \kappa) \nearrow \Xi}$$

$$\frac{\Theta_0, \alpha\,[\cdot\,] :^{\Phi} \kappa \vdash e\,\alpha : [\alpha/a]\,\sigma \prec \sigma' \leadsto e' \dashv \Theta_1}{\Theta_0 \vdash e : (a :^{\Phi}_{i} \kappa) \to \sigma \prec \sigma' \leadsto e' \dashv \Theta_1}$$

Figure 7.13: Subsumption

7.5.1 Unification

The unification judgment $\Theta_0 \vdash \tau \sim \upsilon \leadsto \gamma \dashv \Theta_1$ means that unifying τ with υ in metacontext Θ_0 produces the proof γ in metacontext Θ_1. Conceptually, it is defined using the single rule

$$\frac{\Theta_0, \zeta\,[\cdot\,] :^{\Box} \tau \sim \upsilon \twoheadrightarrow^{*} \Theta_1}{\Theta_0 \vdash \tau \sim \upsilon \leadsto \zeta \dashv \Theta_1}$$

where a new proof obligation ζ (a metavariable at phase $\Box$, also known as a goal) is added to the metacontext and a backward chaining proof search procedure is invoked to take as many steps $\Theta \twoheadrightarrow \Theta'$ as possible, solving or simplifying goals.

I will not define the proof search algorithm (the $\twoheadrightarrow$ relation) fully, as my focus is on elaboration rather than constraint-solving, but a few comments on the steps it would take are in order.

The basic inference rules for backward chaining are the coercion constructors. For example, if the metacontext includes a goal of type $\tau_1{}^{\Upsilon}\upsilon_1 \sim \tau_2{}^{\Upsilon}\upsilon_2$, then backward chaining on congruence of application would turn this into subgoals with types $\tau_1 \sim \tau_2$ and $\upsilon_1 \sim \upsilon_2$. The coercion constructor allows a witness to the original goal to be built from the subgoal metavariables. In this example, the metacontext

$$\Theta, \zeta\,[\Delta] :^{\Box} \tau_1\, \upsilon_1 \sim \tau_2\, \upsilon_2$$

can be replaced with

$$\Theta, \zeta_0\,[\Delta] :^{\Box} \tau_1 \sim \tau_2, \zeta_1\,[\Delta] :^{\Box} \upsilon_1 \sim \upsilon_2, \zeta\,[\Delta] :=^{\Box} \mathbf{conga}^{\Upsilon}\, \zeta_0\, \zeta_1.$$

Similarly, other congruence rules can be used to decompose rigid-rigid constraints, the **step** ρ constructor can be used to reduce (compute) expressions and coherence can be used to remove coercions from equational goals.

Flex-flex or flex-rigid constraints (between two metavariables or a metavariable and a rigid term) can be solved by inversion and intersection, along the lines of the higher-order unification algorithm discussed in Section 4.2 (page 67).

The local parameter telescope of a goal contains the hypotheses available for proving that goal, which may allow it to be solved or simplified via backward chaining. For example, the goal $\zeta\,[c :^{\Box} b \sim a] :^{\Box} a \sim b$ can be solved by $\zeta\,[c :^{\Box} b \sim a] :=^{\Box} \mathbf{sym}\, c$. Introducing a hypothesis uses λ-abstraction for coercions.

Since the integers form an abelian group, constraint solving for linear integer constraints can follow the approach taken in Chapter 3.

Assuming that the proof search algorithm is sound (i.e. all steps are identity metasubstitutions), its embedding into elaboration is sound:

Lemma 7.3 (Soundness of unification)**.** *Suppose that for all* Θ *and* Θ'*,* $\Theta \twoheadrightarrow \Theta'$ *implies* $\Theta \sqsubseteq \Theta'$*. If* $\Theta_0 \vdash \tau \sim \upsilon \rightsquigarrow \gamma \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \gamma :^{\Box} \tau \sim \upsilon$*.*

Proof. By transitivity of metasubstitution and the typing rule for metavariables. □

7.5.2 Soundness of elaboration

The elaboration algorithm is related back to the non-deterministic specification by the following theorem, which states that the algorithm produces one possible elaboration of the input term.

Theorem 7.4 (Soundness of elaboration)**.** *Suppose* $\Psi \in \{\forall, \Pi, \curlywedge\}$*.*

(a) If $\Theta_0 \vdash^{\Psi} \mathsf{p} \rightsquigarrow^{\mathrm{sch}} \rho : \sigma \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \mathsf{p} \rightsquigarrow \rho :^{\Psi} \sigma$*.*

(b) If $\Theta_0 \vdash^{\Psi} \mathsf{p} \rightsquigarrow \rho : \tau \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \mathsf{p} \rightsquigarrow \rho :^{\Psi} \tau$*.*

(c) If $\Theta_0 \vdash^{\Psi} \mathsf{p} : \sigma \rightsquigarrow \rho \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \mathsf{p} \rightsquigarrow \rho :^{\Psi} \sigma$*.*

(d) *If* $\Theta_0 \vdash \mathsf{\sigma} \rightsquigarrow \sigma \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \mathsf{\sigma} \rightsquigarrow \sigma$.

(e) *If* $\Theta_0 \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \sigma$ *and* $\Theta_0 \vdash^{\Psi} (\rho : \sigma)\, \mathsf{\delta} \rightsquigarrow \rho' : \tau \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \mathsf{\rho}\, \mathsf{\delta} \rightsquigarrow \rho' :^{\Psi} \tau$.

(f) *If* $\Theta_0 \vdash \mathsf{\delta} : \mathbf{\Delta} \rightsquigarrow \delta \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash \mathsf{\delta} \rightsquigarrow \delta : \mathbf{\Delta}$.

(g) *If* $\Theta_0 \vdash e : \sigma \prec \sigma' \rightsquigarrow e' \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *and* $\Theta_1 \vdash e : \sigma \prec e' : \sigma'$.

Proof. By induction on derivations, using Lemma 7.3 for unification. □

7.6 Elaboration for case analysis

The system I have presented so far lacks case analysis, which is rather important in practice. Therefore, I will now present the elaboration rules for case expressions, extending the previous non-deterministic and deterministic systems.

I will consider only flat (non-nested) pattern matches; nested pattern matching is a well-studied topic (Augustsson, 1985) that can be presented via elaboration, but would complicate the presentation further.

Moreover, I will continue to assume that case expressions are covering. It is easy to amend the elaboration rules for case expressions to insert missing branches that generate an appropriate runtime error. True coverage checking is less straightforward, because for each omitted data constructor the constraint solver must establish that the constraints it introduces are unsolvable. Goguen et al. (2006) suggest extending the language of patterns with 'refutations', which allow the programmer to indicate arguments that are uninhabited.

The grammar of expressions $\mathsf{\rho}$ (and correspondingly terms t and types $\mathsf{\tau}$) is extended by **case** and **dcase** expressions:

$$\begin{array}{lcl} \mathsf{\rho} & ::= & (\mathbf{d})\mathbf{case}\,\mathsf{\rho}\,\mathbf{of}\,\overline{\mathrm{br}_i}^{\,i} \mid \cdots \\ \mathrm{br} & ::= & \mathsf{K}\,\mathrm{vs} \to \mathsf{\rho} \\ \mathrm{vs} & ::= & \cdot \mid x, \mathrm{vs} \mid \{a = b\}, \mathrm{vs} \end{array}$$

Each branch br in the *inch* source language matches a single data constructor K and binds variables vs, some of which may be implicit. The syntax $\{a = b\}$ means that the implicitly bound variable a in the telescope of the data constructor should be brought into scope with name b, that is, the right-hand side of the equation is the binding occurrence.

$\boxed{\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \sigma}$ *(*$\mathsf{\rho}$ *can elaborate to* ρ *with scheme* σ *at phase* Ψ*)*

$$\frac{\begin{array}{l}\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \qquad \mathbf{\Gamma} \vdash \tau :^{\forall} * \\ \mathbf{\Gamma} \vdash \mathrm{br}_0 \rightsquigarrow br_0 :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau \quad \ldots \quad \mathbf{\Gamma} \vdash \mathrm{br}_n \rightsquigarrow br_n :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau\end{array}}{\mathbf{\Gamma} \vdash \mathbf{case}\,\mathsf{\rho}\,\mathbf{of}\,\mathrm{br}_0 \ldots \mathrm{br}_n \rightsquigarrow \mathbf{case}\,\rho\,\mathbf{of}\,br_0 \ldots br_n :^{\Psi} \tau}$$

$$\frac{\begin{array}{l}\mathbf{\Gamma} \vdash \mathsf{\rho} \rightsquigarrow \varepsilon :^{\Pi /\!\!/ \Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \qquad \mathbf{\Gamma} \vdash \tau :^{\forall} * \\ \mathbf{\Gamma} \vdash \mathrm{br}_0 \rightsquigarrow br_0 :^{\Psi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau \quad \ldots \quad \mathbf{\Gamma} \vdash \mathrm{br}_n \rightsquigarrow br_n :^{\Psi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau\end{array}}{\mathbf{\Gamma} \vdash \mathbf{dcase}\,\mathsf{\rho}\,\mathbf{of}\,\mathrm{br}_0 \ldots \mathrm{br}_n \rightsquigarrow \mathbf{dcase}\,\varepsilon\,\mathbf{of}\,br_0 \ldots br_n :^{\Psi} \tau}$$

$\boxed{\mathbf{\Gamma} \vdash \mathrm{br} \rightsquigarrow br :^{\Psi} \mathsf{D}\,\psi \blacktriangleright \tau}$ *(*br *can elaborate to* br *at phase* Ψ*)*

$$\frac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall}_{i} \kappa_i}^{\,i}, \mathbf{\Delta}) \to \mathsf{D}\,\overline{a_i}^{\,i} \qquad \Phi \hookrightarrow \Psi \\ \mathrm{vs} \;:\; [\overline{\upsilon_i/a_i}^{\,i}]\,\mathbf{\Delta} \;\twoheadrightarrow\; \mathbf{\Delta}' \\ \mathbf{\Gamma}, \mathbf{\Delta}' \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \tau\end{array}}{\mathbf{\Gamma} \vdash (\mathsf{K}\,\mathrm{vs} \to \mathsf{\rho}) \rightsquigarrow (\mathsf{K}\,\lfloor \mathbf{\Delta}' \rfloor \to \rho) :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau}$$

$\boxed{\mathbf{\Gamma} \vdash \mathrm{br} \rightsquigarrow br :^{\Psi} (\varepsilon : \mathsf{D}\,\psi) \blacktriangleright \tau}$ *(*br *can elaborate to* br *at phase* Ψ*)*

$$\frac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall}_{i} \kappa_i}^{\,i}, \mathbf{\Delta}) \to \mathsf{D}\,\overline{a_i}^{\,i} \qquad \Phi \hookrightarrow \Pi /\!\!/ \Psi \\ \mathrm{vs} \;:\; [\overline{\upsilon_i/a_i}^{\,i}]\,\mathbf{\Delta} /\!\!/ \Pi \;\twoheadrightarrow\; \mathbf{\Delta}' \\ \mathbf{\Delta}'' = \mathbf{\Delta}', c :^{\square}_{i} \varepsilon \sim (\mathsf{K}\,\overline{\upsilon_i}^{\,i} \mathbf{\Delta}') \\ \mathbf{\Gamma}, \mathbf{\Delta}'' \vdash \mathsf{\rho} \rightsquigarrow \rho :^{\Psi} \tau\end{array}}{\mathbf{\Gamma} \vdash (\mathsf{K}\,\mathrm{vs} \to \mathsf{\rho}) \rightsquigarrow (\mathsf{K}\,\lfloor \mathbf{\Delta}'' \rfloor \to \rho) :^{\Psi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau}$$

$\boxed{\mathrm{vs} \;:\; \mathbf{\Delta} \;\twoheadrightarrow\; \mathbf{\Delta}'}$

$$\frac{}{\cdot \;:\; \cdot \;\twoheadrightarrow\; \cdot} \qquad\qquad \frac{\mathrm{vs} \;:\; [x/y]\,\mathbf{\Delta} \;\twoheadrightarrow\; \mathbf{\Delta}'}{x, \mathrm{vs} \;:\; y :^{\Phi}_{e} \sigma, \mathbf{\Delta} \;\twoheadrightarrow\; x :^{\Phi}_{e} \sigma, \mathbf{\Delta}'}$$

$$\frac{\mathrm{vs} \;:\; \mathbf{\Delta} \;\twoheadrightarrow\; \mathbf{\Delta}'}{\mathrm{vs} \;:\; a :^{\Phi}_{i} \kappa, \mathbf{\Delta} \;\twoheadrightarrow\; a :^{\Phi}_{i} \kappa, \mathbf{\Delta}'} \qquad\qquad \frac{\mathrm{vs} \;:\; [b/a]\,\mathbf{\Delta} \;\twoheadrightarrow\; \mathbf{\Delta}' \qquad \Phi \neq \square}{\{a\!=\!b\}, \mathrm{vs} \;:\; a :^{\Phi}_{i} \kappa, \mathbf{\Delta} \;\twoheadrightarrow\; b :^{\Phi}_{e} \kappa, \mathbf{\Delta}'}$$

Figure 7.14: Non-deterministic elaboration of case expressions

7.6.1 Extending the non-deterministic system

Figure 7.14 gives the new non-deterministic elaboration rules (extending those in Figure 7.4) for non-dependent and dependent case expressions. In each rule, the scrutinee is elaborated to give an expression of type $\mathsf{D}\,\overline{\upsilon_i}^{\,i}$, then each of the branches is elaborated and must deliver a common type τ, the type of the whole expression, which must not depend on variables in any of the branches. For the dependent case, the scrutinee is elaborated at phase $\Pi \mathbin{/\!\!/} \Psi$ (rather than Ψ) to ensure that it can appear in types and at runtime (if necessary).

The judgment $\Gamma \vdash \mathrm{br} \rightsquigarrow \mathit{br} :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau$ means that the case branch br can be elaborated to br, where the scrutinee has type $\mathsf{D}\,\overline{\upsilon_i}^{\,i}$, and the result has type τ. Branches must be of the form $\mathsf{K}\,\mathrm{vs} \to \mathsf{\rho}$ where K is a constructor of D and is accessible at the current phase. The implicit and explicit variable bindings vs are elaborated in the constructor's telescope $\boldsymbol{\Delta}$ to produce another telescope $\boldsymbol{\Delta}'$ that is in scope when the result of the branch is elaborated. For GADT matches, this telescope will include the equational constraints encoded by the GADT.

Similarly, the judgment $\Gamma \vdash \mathrm{br} \rightsquigarrow \mathit{br} :^{\Psi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau$ means that the dependent case branch br can be elaborated to br, under the assumption that the scrutinee is equal to ε.

The judgment $\mathrm{vs} : \boldsymbol{\Delta} \twoheadrightarrow \boldsymbol{\Delta}'$ means that matching the source language bindings vs against the annotated telescope $\boldsymbol{\Delta}$ of a data constructor results in the annotated telescope $\boldsymbol{\Delta}'$. This gives the telescope needed to elaborate the result of a case branch, using the binding names from vs but obtaining their types from $\boldsymbol{\Delta}$. Implicit bindings are introduced silently, or the user can explicitly bind a name that would usually be implicitly bound. This resembles the judgment for elaborating vectors in Figure 7.5, but for patterns rather than general expressions.

As an example, consider the following definition of append via case analysis:

```
append zs ys = case zs of
  Nil                 → Nil
  Cons {m = m′} x xs → Cons x (append xs ys)
```

To check the Cons branch, recall that the type scheme for Cons (after the GADT translation), is $(\boldsymbol{\Delta}) \to \mathsf{Vec}\,a\,b$ where

$$\boldsymbol{\Delta} = a :^{\forall}_{i} *, b :^{\forall}_{i} \mathbb{N}, m :^{\forall}_{i} \mathbb{N}, x :^{\curlywedge}_{e} a, xs :^{\curlywedge}_{e} \mathsf{Vec}\,a\,m, c :^{\square}_{i} (b \sim \mathsf{Suc}\,m).$$

The bindings $\{m = m'\}, x, xs$ are successfully matched against the telescope $\boldsymbol{\Delta}$, renaming m to m' and introducing a, b and c implicitly. The branch result $\mathsf{Cons}\ x\ (\mathsf{append}\ xs\ ys)$ is then elaborated under the renamed telescope of bindings.

Soundness of non-deterministic elaboration (Theorem 7.1) must be extended with the following additional cases:

Lemma 7.5 (Soundness of non-deterministic elaboration for case analysis)**.**

(a) If $\Gamma \vdash \mathrm{br} \leadsto br :^{\Phi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau$ *then* $\Gamma \vdash br\ :^{\Phi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau$.

(b) If $\Gamma \vdash \mathrm{br} \leadsto br :^{\Phi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau$ *then* $\Gamma \vdash br\ :^{\Phi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau$.

Proof. By structural induction, mutually with Theorem 7.1. □

7.6.2 Extending the deterministic system

Extension of the deterministic elaboration system is mostly routine, following the non-deterministic system. Figure 7.15 gives the additional elaboration rules for case expressions (extending the rules in Figures 7.10 and 7.9). As in the non-deterministic system, auxiliary judgments $\Theta_0 \vdash^{\Psi} \mathrm{br} : \upsilon \blacktriangleright \tau \leadsto br \dashv \Theta_1$ and $\Theta_0 \vdash^{\Psi} \mathrm{br} : (\varepsilon : \upsilon) \blacktriangleright \tau \leadsto br \dashv \Theta_1$ describe elaboration for individual branches. I write a list of semicolon-separated elaboration judgments to mean that the metacontexts are threaded through from one to another.

In the deterministic system, case expressions are checked, rather than having a type inferred. Inference is dealt with by generating a fresh metavariable β and checking that the expression has type β. It is not immediately apparent how the datatype D is to be determined: it might be obvious from the type υ of the scrutinee, but it might not (if a constraint must be solved to show that υ is an algebraic datatype). Alternatively, the types of the data constructors from the branches can be consulted, provided the case expression is non-empty.

Soundness of elaboration (Theorem 7.4) is extended with the following:

Lemma 7.6 (Soundness of elaboration for case expressions)**.**

(a) If $\Theta_0 \vdash^{\Psi} \mathrm{br} : \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau \leadsto br \dashv \Theta_1$ *then* $\Theta_1 \vdash \mathrm{br} \leadsto br :^{\Psi} \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau$ *and* $\Theta_0 \sqsubseteq \Theta_1$.

(b) If $\Theta_0 \vdash^{\Psi} \mathrm{br} : (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau \leadsto br \dashv \Theta_1$ *then* $\Theta_1 \vdash \mathrm{br} \leadsto br :^{\Psi} (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau$ *and* $\Theta_0 \sqsubseteq \Theta_1$.

Proof. By structural induction on derivations, mutually with Theorem 7.4. □

$\boxed{\Theta_0 \vdash^{\Psi} \mathsf{\rho} \rightsquigarrow \rho : \tau \dashv \Theta_1}$ *($\mathsf{\rho}$ elaborates to ρ with inferred type τ)*

$$\frac{\Theta_0, \beta\,[\cdot\,] :^{\forall} * \vdash^{\Psi} (\mathbf{d})\mathbf{case}\,\mathsf{\rho}\,\mathbf{of}\ \mathrm{br}_0 \dots \mathrm{br}_n : \beta \rightsquigarrow \rho \dashv \Theta_1}{\Theta_0 \vdash^{\Psi} (\mathbf{d})\mathbf{case}\,\mathsf{\rho}\,\mathbf{of}\ \mathrm{br}_0 \dots \mathrm{br}_n \rightsquigarrow \rho : \beta \dashv \Theta_1}$$

$\boxed{\Theta_0 \vdash^{\Psi} \mathsf{\rho} : \sigma \rightsquigarrow \rho \dashv \Theta_1}$ *(checking $\mathsf{\rho}$ at scheme σ and phase Ψ delivers ρ)*

$$\frac{\begin{array}{l}\Theta_0 \vdash^{\Psi} \mathsf{\rho} \rightsquigarrow \rho : \upsilon \dashv \Theta_1 \qquad \Theta_1, \overline{\alpha_i\,[\cdot\,] :^{\forall} \kappa_i}^{\,i} \vdash \upsilon \sim \mathsf{D}\,\overline{\alpha_i}^{\,i} \rightsquigarrow \gamma \dashv \Theta_2 \\ \Theta_2 \vdash^{\Psi}\ \mathrm{br}_0 : \mathsf{D}\,\overline{\alpha_i}^{\,i} \blacktriangleright \tau \rightsquigarrow br_0\ ;\ \dots;\ \mathrm{br}_n : \mathsf{D}\,\overline{\alpha_i}^{\,i} \blacktriangleright \tau \rightsquigarrow br_n\ \dashv \Theta_3\end{array}}{\Theta_0 \vdash^{\Psi} \mathbf{case}\,\mathsf{\rho}\,\mathbf{of}\ \mathrm{br}_0 \dots \mathrm{br}_n : \tau \rightsquigarrow \mathbf{case}\,\rho \rhd \gamma\,\mathbf{of}\ br_0 \dots br_n \dashv \Theta_3}$$

$$\frac{\begin{array}{l}\Theta_0 \vdash^{\Pi /\!\!/ \Psi} \mathsf{\rho} \rightsquigarrow \varepsilon : \upsilon \dashv \Theta_1 \qquad \Theta_1, \overline{\alpha_i\,[\cdot\,] :^{\forall} \kappa_i}^{\,i} \vdash \upsilon \sim \mathsf{D}\,\overline{\alpha_i}^{\,i} \rightsquigarrow \gamma \dashv \Theta_2 \\ \Theta_2 \vdash^{\Psi}\ \mathrm{br}_0 : (\varepsilon \rhd \gamma : \mathsf{D}\,\overline{\alpha_i}^{\,i}) \blacktriangleright \tau \rightsquigarrow br_0\ ;\ \dots;\ \mathrm{br}_n : (\varepsilon \rhd \gamma : \mathsf{D}\,\overline{\alpha_i}^{\,i}) \blacktriangleright \tau \rightsquigarrow br_n\ \dashv \Theta_3\end{array}}{\Theta_0 \vdash^{\Psi} \mathbf{dcase}\,\mathsf{\rho}\,\mathbf{of}\ \mathrm{br}_0 \dots \mathrm{br}_n : \tau \rightsquigarrow \mathbf{dcase}\,\varepsilon \rhd \gamma\,\mathbf{of}\ br_0 \dots br_n \dashv \Theta_3}$$

$\boxed{\Theta_0 \vdash^{\Psi} \mathrm{br} : \upsilon \blacktriangleright \tau \rightsquigarrow br \dashv \Theta_1}$ *(case branch* br *elaborates to br)*

$$\frac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall}_{i} \kappa_i}^{\,i}, \mathbf{\Delta}) \to \mathsf{D}\,\overline{a_i}^{\,i} \qquad \Phi \hookrightarrow \Psi \\ \mathrm{vs}\ :\ [\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\mathbf{\Delta}\ \twoheadrightarrow\ \mathbf{\Delta}' \\ \Theta_0, \mathbf{\Delta}' \vdash^{\Psi} \mathsf{\rho} : \tau \rightsquigarrow \rho \dashv \Theta_1, \mathbf{\Delta}', \Xi\end{array}}{\Theta_0 \vdash^{\Psi} \mathsf{K}\,\mathrm{vs} \to \mathsf{\rho} : \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau \rightsquigarrow \mathsf{K}\,\Delta \to \rho \dashv \Theta_1, \Delta \nearrow \Xi}$$

$\boxed{\Theta_0 \vdash^{\Psi} \mathrm{br} : (\varepsilon : \upsilon) \blacktriangleright \tau \rightsquigarrow br \dashv \Theta_1}$ *(dependent case branch* br *elaborates to br)*

$$\frac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall}_{i} \kappa_i}^{\,i}, \mathbf{\Delta}) \to \mathsf{D}\,\overline{a_i}^{\,i} \qquad \Phi \hookrightarrow \Pi /\!\!/ \Psi \\ \mathrm{vs}\ :\ [\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\mathbf{\Delta} /\!\!/ \Pi\ \twoheadrightarrow\ \mathbf{\Delta}' \\ \mathbf{\Delta}'' = \mathbf{\Delta}', c :^{\square}_{i} \varepsilon \sim \mathsf{K}\,\overline{\upsilon_i}^{\,i}\mathbf{\Delta}' \\ \Theta_0, \mathbf{\Delta}'' \vdash^{\Psi} \mathsf{\rho} : \tau \rightsquigarrow \rho \dashv \Theta_1, \mathbf{\Delta}'', \Xi\end{array}}{\Theta_0 \vdash^{\Psi} \mathsf{K}\,\mathrm{vs} \to \mathsf{\rho} : (\varepsilon : \mathsf{D}\,\overline{\upsilon_i}^{\,i}) \blacktriangleright \tau \rightsquigarrow \mathsf{K}\,\Delta' \to \rho \dashv \Theta_1, \Delta' \nearrow \Xi}$$

Figure 7.15: Elaboration of case expressions

7.6.3 Example of elaborating a function definition

Recall the $\mathsf{replicate}$ example from previous chapters:

$$\begin{array}{l}
\mathsf{replicate} :: \forall\, a :: * \,.\, \Pi\; n :: \mathbb{N} \to a \to \mathsf{Vec}\; a\; n \\
\mathsf{replicate}\; n\; x = \textbf{dcase}\; n\; \textbf{of} \\
\quad \mathsf{Zero} \quad\; \to \mathsf{Nil} \\
\quad \mathsf{Suc}\; m \to \mathsf{Cons}\; x\; (\mathsf{replicate}\; m\; x)
\end{array}$$

How will this be elaborated, as a shared function? Elaborating the type scheme produces $(\boldsymbol{\Delta}) \to \mathsf{Vec}\; a\; n$ where $\boldsymbol{\Delta} = a :^{\forall}_{i} *, n :^{\Pi}_{e} \mathbb{N}, x :^{⅄}_{e} a$, so the body should be elaborated at phase Π in the context $\mathsf{replicate}\; [\boldsymbol{\Delta}] :^{\Pi} \mathsf{Vec}\; a\; n, \boldsymbol{\Delta}$. Thus recursive calls to $\mathsf{replicate}$ can be made at phase Π, and its arguments are in scope.

To elaborate the body, the **dcase** expression must be checked at type $\mathsf{Vec}\; a\; n$. The scrutinee n is inferred to have type $\mathbb{N}$. It must then be checked that each of the branches accepts this scrutinee and produces a result of type $\mathsf{Vec}\; a\; n$.

In the Zero branch, the constructor telescope is empty so the only variable brought into scope is an implicit proof $c :^{\Box}_{i} n \sim \mathsf{Zero}$. The result Nil must then be elaborated at type $\mathsf{Vec}\; a\; n$ under this hypothesis. Since its type scheme is

$$(a :^{\forall}_{i} *, n :^{\forall}_{i} \mathbb{N}, c :^{\Box}_{i} n \sim \mathsf{Zero}) \to \mathsf{Vec}\; a\; n$$

the rule for elaborating a term applied to a spine of arguments (empty, in this case) generates metavariables α, β, ζ for the implicit arguments, so Nil elaborates to $\mathsf{Nil}\; \alpha\; \beta\; \zeta$ of inferred type $\mathsf{Vec}\; \alpha\; \beta$ with the proof obligation $\zeta :^{\Box} \beta \sim \mathsf{Zero}$ in the context. Subsumption allows this term to be checked at type $\mathsf{Vec}\; a\; n$, adding another proof obligation $\zeta' :^{\Box} \mathsf{Vec}\; \alpha\; \beta \sim \mathsf{Vec}\; a\; n$ and resulting in the term $\mathsf{Nil}\; \alpha\; \beta\; \zeta \rhd \zeta'$. It should not be difficult for the unifier to solve $\alpha := a$ and $\beta := n$, so ζ' is reflexive. Then $\zeta :^{\Box} n \sim \mathsf{Zero}$ can be solved by c. The final branch is:

$$\mathsf{Zero}\; (c :^{\Box} n \sim \mathsf{Zero}) \to \mathsf{Nil}\; a\; n\; c \rhd \langle \mathsf{Vec}\; a\; n \rangle$$

The coercion by reflexivity can be removed. Other solutions to the constraints are possible, but they affect only the coercions, which are operationally irrelevant.

In the Suc branch, the constructor has telescope $y :^{⅄}_{e} \mathbb{N}$, and matching the source-level bindings against it gives $m : (y :^{⅄}_{e} \mathbb{N}) \mathbin{/\!\!/} \Pi \twoheadrightarrow m :^{\Pi}_{e} \mathbb{N}$ so the match brings into scope m and a proof $c :^{\Box}_{i} n \sim \mathsf{Suc}\; m$. Insertion of implicit arguments proceeds similarly to the Nil case. The scheme $\boldsymbol{\Delta}$ for the recursive call to $\mathsf{replicate}$ is supplied by the context, and is used to check its vector of arguments a, m, x. The final result of elaborating the branch (omitting coercions) is:

$$\mathsf{Suc}\; (m :^{\Pi} \mathbb{N}, c :^{\Box} n \sim \mathsf{Suc}\; m) \to \mathsf{Cons}\; a\; n\; m\; x\; (\mathsf{replicate}\; (a, m, x))\; c$$

7.7 Discussion

In this chapter, I have presented an algorithm for elaborating the *inch* source language of Chapter 5 to the *evidence* language of Chapter 6. While it does not cover every feature available in Haskell, it does demonstrate the way in which an elaborator can be built up to cover a large source language, retaining confidence in the system through translation of source programs into an intermediate representation. The elaborator supports dependent Π-types with type-refining case analysis, higher-rank types and GADTs, though the exact capabilities will depend on the underlying unification algorithm. I have also presented an approach to implicit argument synthesis that generalises the current Haskell policy of 'invisible types, visible terms' to allow for explicit type application and implicit Π-types.

7.7.1 Generalisation

Chapter 2 demonstrated that generalisation of polymorphic let-definitions can be performed through 'skimming off' metavariables from the context after inferring the type of the definiens. Chapter 3 extended this to deal with abelian group unification. However, in the more complicated situation of *inch* elaboration, generalisation becomes yet more problematic. The presence of local constraints and parameterised metavariables means there is no reasonable way to decide which metavariables to generalise: attempting to generalise over parameterised metavariables leads to non-principal solutions.

For example, suppose the expression being generalised has type $\alpha \to \alpha$, and the context suffix is $\alpha\,[\cdot\,] :^{\forall} *, \zeta\,[c :^{\Box} \beta \sim \mathsf{Bool}] :^{\Box} \alpha \sim \mathsf{Bool}$. In this case, the type of ζ does not depend on its parameters, so we could discard the hypothesis c and generalise to produce a result of type $(a :^{\forall}_{i} *) \to (z :^{\Box}_{i} a \sim \mathsf{Bool}) \to a \to a$, i.e. $\mathsf{Bool} \to \mathsf{Bool}$ up to isomorphism. However, if we refrain from generalising and later discover that $\alpha \sim \beta$ then the result has type $\alpha \to \alpha$ for α an unconstrained metavariable. The order of constraint solving is now crucial, different solutions may be found as a result of slight variations in the program, and in general elaboration becomes fragile.

What hope, then, for generalisation? In *inch*, I follow the advice of Vytiniotis et al. (2010) that local 'let should not be generalised'.[1] This strategy has the advantage of simplicity, but other choices (some with a more heuristic character) are available. One might choose to generalise whenever a let-expression did not give

[1]Top level let-bindings can be generalised, because parameterised metavariables can either be solved or reported as errors.

rise to parameterised metavariables at all, perhaps because no local constraints were introduced by case analysis of GADTs or subexpressions with higher-rank types, or because all the constraints introduced were solved by unification on the fly. This has the advantage of allowing generalisation in common cases, but it may be difficult for programmers to predict whether generalisation will take place without knowing the details of the inference algorithm.

7.7.2 Related and future work

The non-deterministic elaboration system is reminiscent of the approach taken in the Definition of Standard ML (Milner et al., 1997), which specifies elaboration via a syntax-directed inductive relation, but leaves matters such as the use of metavariables in type inference to implementations. Such a declarative specification can be turned into a logic program via mode assignment (Berghofer and Nipkow, 2002), with the underlying constraints solved by first-order unification. In the setting of this chapter, however, constraints are more complex and the non-deterministic system is not so easily operationalised.

Brady (2013) describes elaboration for Idris in terms of imperative tactics, taking inspiration from the work of McBride (1999) on the Oleg system.

A full specification of unification in such a rich setting is complex, and I have only outlined the way it fits into the elaboration framework. The careful management of variable scope means that unification could be specified similarly to the Miller pattern unification algorithm of Chapter 4. The algorithm used by GHC, described by Vytiniotis et al. (2011), is very powerful but not straightforward to understand or implement. Further work in this area is desirable.

I have outlined the treatment of higher-rank types, but have not discussed the role of bidirectional type inference in detail. Dunfield and Krishnaswami (2013) give an excellent account of a sound and complete typechecking algorithm for higher-rank polymorphism, in a similar spirit.

Soundness of the elaboration algorithm with respect to the non-deterministic specification is easy to show, and termination[2] follows from its structurally recursive definition, but it would be valuable to prove further properties. In particular, Luther (2003) discusses the *coherence* property, which requires that all possible (non-deterministic) elaborations of a term should be behaviourally equivalent. This formalises the intuition that elaboration should fill in details for which there is only one sensible choice.

[2]Termination in the sense of reduction to constraint solving, that is; termination of the constraint solver is another matter.

Chapter 8

Applications

In this chapter, the hard work of the previous chapters finally pays off. Having introduced the *inch* language and explained how to elaborate it into the *evidence* language, I now give examples of using it to write programs. I start with some familiar operations on vectors (8.1), before implementing merge sort (8.2) and left-leaning red-black tree insertion and deletion (8.3). I demonstrate an approach to checking the time complexity of function definitions (8.4). Finally, I show how to implement units of measure based on numeric constraints (8.5), in contrast to the built-in support for abelian groups described in Chapter 3.

The *inch* preprocessor

The examples in this chapter have been checked with a prototype implementation of *inch*[1]. The prototype consists of a preprocessor that typechecks a source file and converts it into type-correct GHC Haskell, erasing type dependencies. This means that certain features cannot be supported. For example, large eliminations (where types depend on shared terms) are impossible to implement.

The prototype implementation differs from the language laid out in the previous chapters in a number of respects. In particular, it retains a strong distinction between the term, type and kind levels, which limits its flexibility compared to the final design. The kind system consists only of $*$, $\mathbb{Z}$ and higher kinds; other promoted datatypes and kind polymorphism are not implemented.

The language of shared expressions, that may occur in terms and in types, is heavily restricted: only integers and arithmetic operations are available. Similarly, type equality constraints may involve only integers, and GADTs may use only integer indices. The kind $\mathbb{N}$ is represented by $\mathbb{Z}$ with an inequality constraint.

[1]`http://hackage.haskell.org/package/inch`, `https://github.com/adamgundry/inch`

The flexible approach to implicit and explicit arguments based on type schemes, described in Section 7.1 (page 145), is not implemented. Rather, $\forall$-quantifiers are always implicit and Π-quantifiers are always explicit, even though they are written with a dot (so $\Pi\ (m :: \mathbb{N}) \,.\, \tau$ means $\Pi\ (m :: \mathbb{N}) \to \tau$).

Terms that lie in the shared fragment must be marked with braces. This includes applications of Π-quantified functions and the patterns that define them. For example, if $f :: \Pi\ (n :: \mathbb{N}) \,.\, \mathsf{Vec}\ a\ n$ then $f\ \{x + 2\} :: \mathsf{Vec}\ a\ (x + 2)$. Otherwise, the syntax is broadly that of Haskell extended with kind signatures, scoped type variables, GADTs and higher-rank types. One minor extension is that multiple variables may share a kind signature: for example, $\forall (m\ n :: \mathbb{N}) \,.\, t$ is legal.

Type inference is implemented along the lines of elaboration as described in Chapter 7, although instead of generating evidence terms, dependency-erased Haskell programs are produced. Constraint solving is based on the abelian group unification algorithm in Chapter 3, extended to the ring $\mathbb{Z}$. Any remaining purely numeric constraints are checked using a decision procedure for Presburger arithmetic (Diatchki, 2011). This works well for linear constraints, but means that support for constraints involving multiplication is more limited.

Kind inference is not performed, so kinds must be annotated explicitly (otherwise they default to $*$). This means that variables will usually be explicitly quantified. In a more complete implementation, this would not be necessary.

Newtypes are not supported; where they are used in examples, they have been manually translated to the corresponding single-constructor data type behind the scenes. Support for typeclasses is extremely limited, and they will generally not be used in the examples.

Despite these restrictions, it is still possible to implement useful examples. Where relevant, I will point out opportunities to improve the examples given a full-scale implementation of the *inch* system.

8.1 Vectors

Recall the definition of vectors as an indexed family of types:[2]

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec} :: * \to \mathbb{N} \to *\ \textbf{where} \\
\quad \mathsf{Nil}\quad :: \mathsf{Vec}\ a\ 0 \\
\quad \mathsf{Cons} :: \forall\, a\ (n :: \mathbb{N}) \,.\, a \to \mathsf{Vec}\ a\ n \to \mathsf{Vec}\ a\ (n + 1)
\end{array}
$$

[2]Sensitive Haskell programmers may wonder why the kind of Vec is not $\mathbb{N} \to * \to *$, since then $\mathsf{Vec}\ n$ is a monad for any n with the diagonal join, as shown in Subsection 5.2.4 (page 99). Unfortunately, this would make it harder to regard Vec as a type indexed by $\mathbb{N}$, since Haskell treats type application as injective.

Here are some standard functions on vectors. The types of head and tail ensure they are never called on the empty vector, and lengths are tracked appropriately in the other cases. Most of the function definitions use polymorphic recursion and pattern-matching on GADTs, so their types must be specified. As discussed in Subsection 5.1.1 (page 90), the helper function for reverse implicitly requires a proof that $(m + 1) + n \sim m + (n + 1)$, so additional constraint solving beyond the inductive definition of $+$ is required. The lookup function demonstrates the use of Π-types: the index m must be supplied at runtime, but statically known to be below the length n.

$$
\begin{array}{l}
\mathsf{head} :: \forall (n :: \mathbb{N})\ a\,.\,\mathsf{Vec}\ a\ (n+1) \to a \\
\mathsf{head}\ (\mathsf{Cons}\ x\ \_) = x \\[4pt]
\mathsf{tail} :: \forall (n :: \mathbb{N})\ a\,.\,\mathsf{Vec}\ a\ (n+1) \to \mathsf{Vec}\ a\ n \\
\mathsf{tail}\ (\mathsf{Cons}\ \_\ xs) = xs \\[4pt]
\mathsf{append} :: \forall a\ (m\ n :: \mathbb{N})\,.\,\mathsf{Vec}\ a\ m \to \mathsf{Vec}\ a\ n \to \mathsf{Vec}\ a\ (m+n) \\
\mathsf{append}\ \mathsf{Nil}\qquad\qquad\ \ ys = ys \\
\mathsf{append}\ (\mathsf{Cons}\ x\ xs)\ ys = \mathsf{Cons}\ x\ (\mathsf{append}\ xs\ ys) \\[4pt]
\mathsf{reverse} :: \forall (n :: \mathbb{N})\ a\,.\,\mathsf{Vec}\ a\ n \to \mathsf{Vec}\ a\ n \\
\mathsf{reverse}\ xs = \mathsf{help}\ xs\ \mathsf{Nil} \\
\quad \mathbf{where} \\
\qquad \mathsf{help} :: \forall (m\ n :: \mathbb{N})\ a\,.\,\mathsf{Vec}\ a\ m \to \mathsf{Vec}\ a\ n \to \mathsf{Vec}\ a\ (m+n) \\
\qquad \mathsf{help}\ \mathsf{Nil}\qquad\qquad\ \ ys = ys \\
\qquad \mathsf{help}\ (\mathsf{Cons}\ x\ xs)\ ys = \mathsf{help}\ xs\ (\mathsf{Cons}\ x\ ys) \\[4pt]
\mathsf{lookup} :: \forall (n :: \mathbb{N})\ a\,.\,\Pi\ (m :: \mathbb{N})\,.\,m < n \Rightarrow \mathsf{Vec}\ a\ n \to a \\
\mathsf{lookup}\ \{0\}\qquad\ (\mathsf{Cons}\ x\ \_)\ \ = x \\
\mathsf{lookup}\ \{k+1\}\ (\mathsf{Cons}\ \_\ xs) = \mathsf{lookup}\ \{k\}\ xs
\end{array}
$$

The fold for vectors has a rank-2 type, because for the Cons constructor it needs to abstract over the length m of the tail. Apart from the more informative type signature, it is essentially the same as the traditional foldr for lists. Indeed, it will erase to such a function at runtime.

$$
\begin{array}{l}
\mathsf{foldVec} :: \forall (f :: \mathbb{N} \to *)\ a\ (n :: \mathbb{N})\,. \\
\qquad\qquad f\ 0 \to (\forall (m :: \mathbb{N})\,.\,a \to f\ m \to f\ (m+1)) \to \mathsf{Vec}\ a\ n \to f\ n \\
\mathsf{foldVec}\ n\ c\ \mathsf{Nil}\qquad\qquad\ \ = n \\
\mathsf{foldVec}\ n\ c\ (\mathsf{Cons}\ x\ xs) = c\ x\ (\mathsf{foldVec}\ n\ c\ xs)
\end{array}
$$

As one would expect, foldVec Nil Cons is well-typed and equal to the identity function on vectors. Unfortunately the usual definition of append via a fold,

```
append′ xs ys = foldVec ys Cons xs
```

does not typecheck, because of the lack of type-level λ-abstraction. It is possible to work around this, at the cost of some syntactic overhead, using a newtype:

```
newtype Plus a m n = Plus {unPlus :: a (m + n)}

append″ :: ∀ a (m n :: ℕ) . Vec a m → Vec a n → Vec a (m + n)
append″ xs ys = unPlus (foldVec (Plus ys)
                                  (\ z zs → Plus (Cons z (unPlus zs))) xs)
```

8.2 Merge sort

I now implement merge sort, based on a similar example by Altenkirch et al. (2005) in the dependently typed programming language Epigram (McBride and McKinna, 2004). The type of the sorting function guarantees that it preserves the length of the vector and returns a sorted result, if anything. No proof manipulation is necessary, and the program erases to a natural implementation of merge sort for lists of integers. I do not show that the result is a permutation of the input, as this would require a more expressive type system; Xi (2008) does so for quicksort in ATS. On similar lines, Xi (1998) gives an example of merge sort in Dependent ML that verifies the length of the input is preserved.

Of course, Haskell's type system does not check the totality of our programs, so this is only a partial correctness result. Higher-rank types allow me to express the fold-based recursion structure of the key functions, making the termination reasoning more obvious to the reader, if not the compiler.

In principle, it is possible to express something similar using GADTs and type families, but the complexity of the implementation and the manual proofs involved would be much greater. Mu (2007) provides an impressive example that verifies length-preservation, but not ordering, in this manner.

The point of this example is not that it is a verified implementation of merge sort, as there are many such programs already. Rather, it shows the utility of type-level numbers in Haskell and the ease with which they integrate with Haskell programming idioms (such as folds) and features (higher-rank types and polymorphic recursion).

A Tree is a leaf-labelled binary tree indexed by the number of leaves. Its construction ensures that it is balanced, as the subtrees of each node differ in size by at most one.

```
data Tree :: * → ℕ → * where
  Empty :: Tree a 0
  Leaf  :: a → Tree a 1
  Even  :: ∀ a (n :: ℕ) . 1 ⩽ n ⇒ Tree a n → Tree a n →
             Tree a (2 * n)
  Odd   :: ∀ a (n :: ℕ) . 1 ⩽ n ⇒ Tree a (n + 1) → Tree a n →
             Tree a (2 * n + 1)
```

Just like for vectors, the fold for trees uses higher-rank types. This version is slightly simplified, as it hides the distinction between even and odd nodes.

```
foldTree :: ∀ (f :: ℕ → *) a (n :: ℕ) .
              f 0 → (a → f 1) → (∀ (m n :: ℕ) . f m → f n → f (m + n)) →
              Tree a n → f n
foldTree e l n Empty      = e
foldTree e l n (Leaf a)   = l a
foldTree e l n (Even x y) = n (foldTree e l n x) (foldTree e l n y)
foldTree e l n (Odd x y)  = n (foldTree e l n x) (foldTree e l n y)
```

A tree can be built by folding over a vector, replacing Nil with Empty and inserting elements using the balance-preserving insert function:

```
mkTree :: ∀ a (n :: ℕ) . Vec a n → Tree a n
mkTree = foldVec Empty insert
  where
    insert :: ∀ a (n :: ℕ) . a → Tree a n → Tree a (n + 1)
    insert i Empty      = Leaf i
    insert i (Leaf j)   = Even (Leaf i) (Leaf j)
    insert i (Even l r) = Odd (insert i l) r
    insert i (Odd l r)  = Even l (insert i r)
```

A simple definition such as mkTree, which does not pattern-match on GADTs or use polymorphic recursion, does not need a top-level type signature. The bidirectional type inference algorithm is quite capable of inferring this type. However, I include the signature for consistency and clarity.

Ordered vectors are indexed by lower and upper bounds, plus length. They are restricted to containing integers (by the Π-quantifier). Ideally one should extend $\mathbb{Z}$ with top and bottom elements, to allow unbounded data. These restrictions derive from the limitations of the preprocessor; the theory given in Chapter 6 can support the general case.

```
data OVec :: ℤ → ℤ → ℕ → ∗ where
  ONil   :: ∀(l u :: ℤ) . l ⩽ u ⇒ OVec l u 0
  OCons :: ∀(l u :: ℤ) (n :: ℕ) . Π (x :: ℤ) . l ⩽ x ⇒
                 OVec x u n → OVec l u (n + 1)
```

Given two ordered vectors, the merge function combines them to produce a single ordered vector. It uses the syntax for guards that introduce local constraints described in Subsection 5.2.7. The second guard is redundant, but to see this the implementation would need to negate the results of previous tests when checking patterns, which is not currently supported.

```
merge ::   ∀(l u :: ℤ) (m n :: ℕ) .
                OVec l u m → OVec l u n → OVec l u (m + n)
merge ONil                ys    = ys
merge xs                  ONil = xs
merge (OCons {x} xs) (OCons {y} ys)
   | {x ⩽ y} = OCons {x} (merge xs (OCons {y} ys))
   | {x > y}  = OCons {y} (merge (OCons {x} xs) ys)
```

The type In l u represents integers in the interval $[l, u]$:

```
data In :: ℤ → ℤ → ∗ where
  In :: ∀(l u :: ℤ) . Π (x :: ℤ) . (l ⩽ x, x ⩽ u) ⇒ In l u
```

The flatten function converts a binary tree of numbers in an interval to an ordered vector on that same interval, by invoking the higher-rank fold over the tree, calling merge at each node and converting each leaf value into a vector of length 1.

```
flatten :: ∀(l u :: ℤ) (m :: ℕ) . l ⩽ u ⇒ Tree (In l u) m → OVec l u m
flatten = foldTree ONil invec merge
   where invec :: ∀(l u :: ℤ) . In l u → OVec l u 1
            invec (In {i}) = OCons {i} ONil
```

To merge sort a vector of numbers in an interval to produce an ordered vector, it is enough to construct and flatten a tree:

```
sort :: ∀(l u :: ℤ) (m :: ℕ) . l ⩽ u ⇒ Vec (In l u) m → OVec l u m
sort = flatten ∘ mkTree
```

Now evaluating sort (Cons (In {3}) (Cons (In {1}) (Cons (In {2}) Nil))) produces the sorted list OCons {1} (OCons {2} (OCons {3} ONil)) as expected.

8.3 Left-leaning red-black trees

I now move on to a more advanced example data structure, red-black trees. A *left-leaning red-black tree* is a self-balancing binary search tree, designed to give good performance for insertion, deletion and membership test operations.[3] Every node is coloured either red or black, subject to the following invariants:

1. All leaves, and both children of a red node, are black.

2. The right child of a black node is black.

3. Both children of an internal node have the same black height (the number of black nodes on any path to a leaf).

There has been much research on implementing red-black trees in functional languages, building on foundations laid by Okasaki (1998), who dealt with insertion but not deletion. Might (n.d.) showed how to extend Okasaki's implementation to deletion by adding two extra colours for tracking temporary invariant violations. Yamamoto (2011) applied Okasaki's work to left-leaning trees.

Another strand of research focused on provably correct functional implementations. Kahrs (2001) demonstrated an ingenious technique for enforcing the balance invariant of red-black trees using the Haskell type system. Ek et al. (2011) used Agda to verify the binary search tree and colour invariants of left-leaning red-black tree insertion, and Oster (2011) extended this to deletion.

Most implementations of red-black trees (both functional and imperative) work by constructing unbalanced trees and then applying a separate rebalancing operation. This does not work well when enforcing the invariants through the type system, because of the need to represent slightly malformed trees. Xi (2007) implemented red-black trees in ATS, following Okasaki's approach, indexing trees by the number of red-red colour violations they contain, and requiring that well-formed trees contain no colour violations. In this implementation, I will use McBride and McKinna's idea[4] of representing the path to the point where there would be an invariant violation using a Huet-style zipper. This avoids the need to represent trees that do not obey the invariants.

The choice of left-leaning red-black trees here is not crucial. The technique of avoiding malformed trees using a zipper works well for other self-balancing binary search trees such as normal red-black trees or AVL trees.

[3]Left-leaning red-black trees were introduced by Sedgewick (2008), as a simplification of the original red-black trees of Bayer (1972), obtained by omitting invariant 2. Regarding red nodes as part of their parent nodes, an LLRBT is a 2-3 tree; a normal red-black tree is a 2-3-4 tree.

[4]Red-black tree insertion was implemented as an example with the Epigram 1 distribution.

8.3.1 Enforcing red-black tree invariants via types

To keep track of colours in the type system, I define the following singleton GADT. This is a limitation of the preprocessor: in a full implementation, one could simply define an algebraic data type for colours (or use the booleans) and use its constructors promoted to the type level, which would be slightly neater.

```
type Black = 0
type Red   = 1

data Colour :: ℤ → ∗ where
  Black :: Colour Black
  Red   :: Colour Red
```

The type RBTree represents well-formed red-black trees. Trees are indexed by lower and upper bounds, their colour and black height, and the type checker guarantees that all the invariants hold. Each leaf E stores a proof that its lower bound is strictly smaller than its upper bound, ensuring that the keys are stored in ascending order and there are no duplicated keys. There are separate constructors for red and black internal nodes (TR and TB respectively). The indexing ensures that the colour invariants are observed. A Π-type is used to store the key x on an internal node, so that the ordering invariant can be maintained.

```
data RBTree :: ℤ → ℤ → ℤ → ℕ → ∗ where
  E  :: ∀(i j :: ℤ) . i < j ⇒ RBTree i j Black 0
  TR :: ∀(i j :: ℤ) (n :: ℕ) . Π (x :: ℤ) .
            RBTree i x Black n → RBTree x j Black n → RBTree i j Red n
  TB :: ∀(i j c :: ℤ) (n :: ℕ) . Π (x :: ℤ) .
            RBTree i x c n → RBTree x j Black n → RBTree i j Black (n + 1)
```

The interface that would exposed to the user of the red-black tree library hides the colour (always black) and the black height using the existential type RBT. However, the lower and upper bounds are visible. This distinguishes between invariants used only for the implementation of the library, which will change as nodes are inserted and deleted, from those relevant for the user. Alternative choices, such as concealing the bounds as well, are also possible.

```
data RBT :: ℤ → ℤ → ∗ where
  RBT :: ∀(i j :: ℤ) (n :: ℕ) . RBTree i j Black n → RBT i j
```

Given the type RBTree, the corresponding type of one-hole contexts can be derived mechanically (McBride, 2001). These can be used to navigate a tree via a zipper (Huet, 1997). The type of one-hole contexts is indexed by two copies of the RBTree indices: those provided at the root, and those required at the hole. Since the root is always black, however, I can do away with one of the indices.

```
data TreeZip :: ℤ → ℤ → ℕ →              -- root indices
                ℤ → ℤ → ℤ → ℕ →          -- hole indices
                * where
  Root :: ∀(i j :: ℤ) (n :: ℕ). TreeZip i j n i j Black n
  ZRL  :: ∀(i j i′ j′ :: ℤ) (n n′ :: ℕ). Π (x :: ℤ).
            TreeZip i′ j′ n′ i j Red n → RBTree x j Black n →
              TreeZip i′ j′ n′ i x Black n
  ZRR  :: ∀(i′ j′ i j :: ℤ) (n′ n :: ℕ). Π (x :: ℤ).
            RBTree i x Black n → TreeZip i′ j′ n′ i j Red n →
              TreeZip i′ j′ n′ x j Black n
  ZBL  :: ∀(i′ j′ i j c :: ℤ) (n′ n :: ℕ). Π (x :: ℤ).
            TreeZip i′ j′ n′ i j Black (n + 1) → RBTree x j Black n →
              TreeZip i′ j′ n′ i x c n
  ZBR  :: ∀(i′ j′ i j c :: ℤ) (n′ n :: ℕ). Π (x :: ℤ).
            RBTree i x c n → TreeZip i′ j′ n′ i j Black (n + 1) →
              TreeZip i′ j′ n′ x j Black n
```

Given a context and a tree that fits in the hole, the whole tree can be rebuilt by plug. This function is well-typed because the indexing discipline of TreeZip exactly matches the demands of RBTree. This also could be obtained for free using generic programming techniques (Löh and Magalhães, 2011).

```
plug :: ∀(i′ j′ i j c :: ℤ) (n n′ :: ℕ).
          RBTree i j c n → TreeZip i′ j′ n′ i j c n → RBTree i′ j′ Black n′
plug t Root             = t
plug t (ZRL {x} z r) = plug (TR {x} t r) z
plug t (ZRR {x} l z) = plug (TR {x} l t) z
plug t (ZBL {x} z r) = plug (TB {x} t r) z
plug t (ZBR {x} l z) = plug (TB {x} l t) z
```

8.3.2 Search

When searching for a key x in a red-black tree, it can either be Found z t, where z is the context in which it was found and t is the subtree with x at the root, or Missing z, where z is the context that should have contained x. This detailed search result information will later be used to implement insertion and deletion.

```
data SearchResult :: ℤ → ℤ → ℤ → ℕ → ∗ where
  Found   :: ∀(x i′ j′ i j c :: ℤ) (n′ n :: ℕ) .
               TreeZip i′ j′ n′ i j c n → RBTree i j c n →
                 SearchResult x i′ j′ n′
  Missing :: ∀(x i′ j′ i j :: ℤ) (n′ :: ℕ) . (i < x, x < j) ⇒
               TreeZip i′ j′ n′ i j Black 0 →
                 SearchResult x i′ j′ n′
```

To search a tree, a context is built up by comparing the key x to the value y stored at each node, and descending into the appropriate subtree, until the key is found or a leaf is reached. The invariants make it hard to get wrong: if a conditional test is omitted, or an invalid result returned, the typechecker will object.

```
search :: ∀(i′ j′ :: ℤ) (n′ :: ℕ) . Π (x :: ℤ) . (i′ < x, x < j′) ⇒
            RBTree i′ j′ Black n′ → SearchResult x i′ j′ n′
search {x} = help Root
  where
    help :: ∀(i j c :: ℤ) (n :: ℕ) . (i < x, x < j) ⇒
              TreeZip i′ j′ n′ i j c n → RBTree i j c n →
                SearchResult x i′ j′ n′
    help z E                          = Missing z
    help z (TR {y} l r) | {x < y}     = help (ZRL {y} z r) l
    help z (TR {y} l r) | {x ∼ y}     = Found z (TR {y} l r)
    help z (TR {y} l r) | {x > y}     = help (ZRR {y} l z) r
    help z (TB {y} l r) | {x < y}     = help (ZBL {y} z r) l
    help z (TB {y} l r) | {x ∼ y}     = Found z (TB {y} l r)
    help z (TB {y} l r) | {x > y}     = help (ZBR {y} l z) r
```

The user of the library can be presented with a simple membership test:

```
member :: ∀(i j :: ℤ) . Π (x :: ℤ) . (i < x, x < j) ⇒ RBT i j → Bool
member {x} (RBT t) = case search {x} t of
  Missing _   → False
  Found _ _   → True
```

8.3.3 Insertion

To insert an element into a red-black tree, use search to find the appropriate location, then add a new node and proceed back up the tree, rebalancing on the way. The InsProb datatype represents the kind of problems that may be encountered when rebalancing the tree: either it is on the level (inserting a tree into a hole of the correct black height, though not necessarily the same colour) or in a panic (because a red child would have a red parent).

```
data InsProb :: $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{N} \to *$ where
  Level   :: $\forall (i\ j\ c\ c' :: \mathbb{Z})\ (n :: \mathbb{N})$ .
                 Colour $c'$ $\to$ RBTree $i\ j\ c'\ n$ $\to$
                   InsProb $i\ j\ c\ n$
  PanicRB :: $\forall (i\ j :: \mathbb{Z})\ (n :: \mathbb{N})\ .\ \Pi\ (x :: \mathbb{Z})$ .
                 RBTree $i\ x$ Red $n$ $\to$ RBTree $x\ j$ Black $n$ $\to$
                   InsProb $i\ j$ Red $n$
  PanicBR :: $\forall (i\ j :: \mathbb{Z})\ (n :: \mathbb{N})\ .\ \Pi\ (x :: \mathbb{Z})$ .
                 RBTree $i\ x$ Black $n$ $\to$ RBTree $x\ j$ Red $n$ $\to$
                   InsProb $i\ j$ Red $n$
```

The insertRBT function searches for the element x, and if it is not present, calls the ins function defined below with the appropriate insertion problem.

```
insertRBT :: $\forall (i\ j :: \mathbb{Z})\ .\ \Pi\ (x :: \mathbb{Z})\ .\ (i < x, x < j) \Rightarrow$
               RBT $i\ j$ $\to$ RBT $i\ j$
insertRBT $\{x\}$ (RBT $t$) = solveIns (search $\{x\}$ $t$)
  where
    solveIns :: $\forall (n :: \mathbb{N})$ . SearchResult $x\ i\ j\ n$ $\to$ RBT $i\ j$
    solveIns (Missing $z$)  = ins (Level Red (TR $\{x\}$ E E)) $z$
    solveIns (Found _ _) = RBT $t$
```

To solve an insertion problem, move out through the context, updating the problem appropriately at each step. While this definition looks intimidating (!), the types make it difficult to get wrong: the typechecker will object if a tree is ever constructed that breaks the invariants (either ordering or colouring). It is much easier to construct interactively than in batch mode. In fact, I first implemented it in Agda using the support for interactive construction, then transcribed it for *inch*. The Agsy proof search tool (Lindblad and Benke, 2006) is able to fill in many cases automatically, further reducing the effort involved.

```
ins :: ∀(i′ j′ i j c :: ℤ) (n′ n :: ℕ) .
        InsProb i j c n → TreeZip i′ j′ n′ i j c n → RBT i′ j′
ins (Level Red (TR {x} t₀ t₁)) Root = RBT (TB {x} t₀ t₁)
ins (Level Black t)            Root = RBT t

ins (Level Red   t) (ZRL {x} z t′)  = ins (PanicRB {x} t t′) z
ins (Level Red   t) (ZRR {x} t′ z)  = ins (PanicBR {x} t′ t) z
ins (Level Black t) (ZRL {x} z t′)  = ins (Level Red (TR {x} t t′)) z
ins (Level Black t) (ZRR {x} t′ z)  = ins (Level Red (TR {x} t′ t)) z
ins (Level c     t) (ZBL {x} z t′)  = ins (Level Black (TB {x} t t′)) z
ins (Level Black t) (ZBR {x} t′ z)  = ins (Level Black (TB {x} t′ t)) z
ins (Level Red (TR {y} t₁ t₂)) (ZBR {x} E z) =
  RBT (plug (TB {y} (TR {x} E t₁) t₂) z)
ins (Level Red (TR {y} t₁ t₂)) (ZBR {x} (TB {w} t t′) z) =
  RBT (plug (TB {y} (TR {x} (TB {w} t t′) t₁) t₂) z)
ins (Level Red (TR {y} t₁ t₂)) (ZBR {x} (TR {w} t t′) z) =
  ins (Level Red (TR {x} (TB {w} t t′) (TB {y} t₁ t₂))) z

ins (PanicRB {y} (TR {w} t₀ t₁) t₂) (ZBL {x} z t) =
  ins (Level Red (TR {y} (TB {w} t₀ t₁) (TB {x} t₂ t))) z
ins (PanicBR {y} t₀ (TR {w} t₁ t₂)) (ZBL {x} z t) =
  ins (Level Red (TR {w} (TB {y} t₀ t₁) (TB {x} t₂ t))) z

ins (PanicRB {y} (TR {w} t₀ t₁) t₂) (ZBR {x} t z) =
  ins (Level Red (TR {w} (TB {x} t t₀) (TB {y} t₁ t₂))) z
ins (PanicBR {y} t₀ (TR {w} t₁ t₂)) (ZBR {x} t z) =
  ins (Level Red (TR {y} (TB {x} t t₀) (TB {w} t₁ t₂))) z
```

8.3.4 Deletion

Deleting a key from a red-black tree is slightly more complicated than insertion. The search function positions the focus on the node to be deleted, then calls delFocus, assuming the key exists.

```
delete :: ∀(i j :: ℤ) . Π (x :: ℤ) . (i < x, x < j) ⇒ RBT i j → RBT i j
delete {x} (RBT t) = solveDel (search {x} t)
  where
    solveDel :: ∀(n :: ℕ) . SearchResult x i j n → RBT i j
    solveDel (Missing _) = RBT t
    solveDel (Found z t) = delFocus t z
```

To delete the node at the focus, provided the right subtree has black height 1 or more, the deleted key can be replaced with the minimum of its right subtree (using findMin defined below). The base cases (where the right subtree has black height zero) are handled individually.

```
delFocus :: ∀(i′ j′ i j c :: ℤ) (n′ n :: ℕ) .
              RBTree i j c n → TreeZip i′ j′ n′ i j c n → RBT i′ j′
delFocus E                          z = RBT (plug E z)
delFocus (TR {x} E E)               z = RBT (plug E (wantBlack z))
delFocus (TB {x} E E)               z = del E z
delFocus (TB {x} (TR {y} E E) E)    z = RBT (plug (TB {y} E E) z)
delFocus (TR {x} t_0 (TB {y} t_1 t_2)) z =
  findMin (TB {y} t_1 t_2) (\ {k} → ZRR {k} (wkTree t_0) z)
delFocus (TB {x} t_0 (TB {y} t_1 t_2)) z =
  findMin (TB {y} t_1 t_2) (\ {k} → ZBR {k} (wkTree t_0) z)
```

The only context in which a red node can occur is the left child of a black node, which also accepts black nodes. Thus the wantBlack function can change the type from the former to the latter.

```
wantBlack :: ∀(i′ j′ i j :: ℤ) (n′ n :: ℕ) .
               TreeZip i′ j′ n′ i j Red n → TreeZip i′ j′ n′ i j Black n
wantBlack (ZBL {x} z r) = ZBL {x} z r
```

Deletion may require the upper bound of a subtree to be weakened, which needs a traversal of its right spine to satisfy the type checker. This could be replaced with unsafeCoerce, since the inequality proofs being manipulated are not retained at runtime, so it is operationally the identity function.

```
wkTree :: ∀(i j j′ c n :: ℤ) . j < j′ ⇒ RBTree i j c n → RBTree i j′ c n
wkTree E                = E
wkTree (TR {x} t_0 t_1) = TR {x} t_0 (wkTree t_1)
wkTree (TB {x} t_0 t_1) = TB {x} t_0 (wkTree t_1)
```

The findMin function works inside the right subtree of the node whose key is being deleted, looking for the minimum key, which will be used to replace the deleted one. The zipper context abstracts over the (as yet unknown) minimum key. If the minimum is found on a red node, it can simply be removed and the tree be reconstructed. However, if the minimum is on a black node or leaf, then the del function is called to decrease the black height.

```
findMin :: ∀(i' j' i j c :: ℤ) (n' n :: ℕ) . RBTree i j c (n + 1) →
             (Π (k :: ℤ) . i < k ⇒ TreeZip i' j' n' k j c (n + 1)) →
               RBT i' j'
findMin (TB {x} E E)              f = del E (f {x})
findMin (TB {x} (TR {y} E E) t) f = RBT (plug E (ZBL {x} (f {y}) t))
findMin (TR {x} (TB {y} E E) t) f = del E (ZRL {x} (f {y}) t)
findMin (TR {x} (TB {y} (TR {k} E E) t₀) t₁)  f =
  RBT (plug E (ZBL {y} (ZRL {x} (f {k}) t₁) t₀))
findMin (TR {x} (TB {y} (TB {w} t₀ t₁) t₂) t₃) f =
  findMin (TB {w} t₀ t₁) (\ {k} → ZBL {y} (ZRL {x} (f {k}) t₃) t₂)
findMin (TB {x} (TB {y} t₀ t₁) t₂)              f =
  findMin (TB {y} t₀ t₁) (\ {k} → ZBL {x} (f {k}) t₂)
```

When deleting a black leaf (either directly or because it is the minimum in the right subtree of a deleted internal node), the black height must be decremented. Generally, the problem is to fit a tree of black height n into a hole that expects a tree of height $n + 1$. The del function works its way upwards, rebalancing after deletion, in a similar way to ins. Again, this definition is much easier to write than to read, thanks to the automation tool Agsy (Lindblad and Benke, 2006).

```
del :: ∀(i' j' i j :: ℤ) (n' n :: ℕ) . RBTree i j Black n →
         TreeZip i' j' n' i j Black (n + 1) → RBT i' j'
del t Root = RBT t
del t (ZRL {x} z (TB {y} t₀ t₁)) = colourOf t₀
  (RBT (plug (TB {y} (TR {x} t t₀) t₁) (wantBlack z)))
  (\ {w} t₀' t₀'' → RBT (plug (TR {w} (TB {x} t t₀') (TB {y} t₀'' t₁)) z))
del t (ZRR {x} (TB {y} t₀ t₁) z) = colourOf t₀
  (RBT (plug (TB {x} (TR {y} t₀ t₁) t) (wantBlack z)))
  (\ {w} t₀' t₀'' → RBT (plug (TR {y} (TB {w} t₀' t₀'') (TB {x} t₁ t)) z))
del t (ZBL {x} z (TB {y} t₀ t₁)) = colourOf t₀
  (del (TB {y} (TR {x} t t₀) t₁) z)
  (\ {w} t₀' t₀'' → RBT (plug (TB {w} (TB {x} t t₀') (TB {y} t₀'' t₁)) z))
del t (ZBR {x} (TR {y} t₀ (TB {w} t₁ t₂)) z) = colourOf t₁
  (RBT (plug (TB {y} t₀ (TB {x} (TR {w} t₁ t₂) t)) z))
  (\ {v} t₁' t₁'' → RBT (plug (TB {w} (TR {y} t₀ (TB {v} t₁' t₁''))
                                        (TB {x} t₂ t)) z))
del t (ZBR {x} (TB {y} t₀ t₁) z) = colourOf t₀
  (del (TB {x} (TR {y} t₀ t₁) t) z)
  (\ {w} t₀' t₀'' → RBT (plug (TB {y} (TB {w} t₀' t₀'') (TB {x} t₁ t)) z))
```

The colourOf eliminator determines if a tree is red or black, and calls the corresponding argument. For red trees, it provides the children of the node to the callback. This reduces the number of cases in del, because each case depends on the colour of a subtree, but not whether it is a leaf or an internal node.

```
colourOf :: ∀ a (i j c n :: ℤ) .
   RBTree i j c n →
   ((c ∼ Black) ⇒ a) →
   ((c ∼ Red) ⇒ Π (x :: ℤ) . RBTree i x Black n →
                              RBTree x j Black n → a) → a
colourOf E                b g = b
colourOf (TB {x} _ _)     b g = b
colourOf (TR {x} t0 t1)   b g = g {x} t0 t1
```

8.4 Tracking time complexity

Danielsson (2008) introduced the Thunk library for verifying the time complexity of purely functional data structures in the dependently typed programming language Agda. He indexes a monad by the number of computation steps required to deliver a value in weak head normal form. Function definitions must be annotated with calls to an operation that increments this number.

```
newtype Cost (n :: ℕ) a = Hide {force :: a}
```

The implementation of Cost and the primitive functions on it are hidden, because Cost is really a newtype with phantom type parameter n. This avoids runtime overhead, but if it was exposed to the user then the library invariants would be easily violated. Agda provides a language construct **abstract** to support this, and a similar abstraction barrier can be created in Haskell using modules.

The return and bind functions witness the fact that Cost is a monad indexed by the monoid $(\mathbb{N}, +)$. That is, any value can be computed in no steps, and if some a can be computed in m steps, and used to compute some b in n steps, then the overall computation takes $m + n$ steps.

```
return :: a → Cost 0 a
return = Hide

bind :: ∀(m n :: ℕ) a b . Cost m a → (a → Cost n b) → Cost (m + n) b
bind (Hide x) f = wait (f x)
```

If a value can be computed in m steps, then it can be computed in n steps for any n larger than m. Unlike Danielsson's version, which requires the caller to specify a number of steps to wait, this exploits the inequality constraints of *inch* to provide a more flexible interface.

$$\begin{aligned}
&\mathsf{wait} :: \forall (m\ n :: \mathbb{N})\ a \,.\, m \leqslant n \Rightarrow \mathsf{Cost}\ m\ a \rightarrow \mathsf{Cost}\ n\ a \\
&\mathsf{wait}\ (\mathsf{Hide}\ a) = \mathsf{Hide}\ a
\end{aligned}$$

A crucial part of the methodology is to annotate every line of every function definition being counted with a call to tick, which increments the counter.

$$\begin{aligned}
&\mathsf{tick} :: \forall (n :: \mathbb{N})\ a \,.\, \mathsf{Cost}\ n\ a \rightarrow \mathsf{Cost}\ (n+1)\ a \\
&\mathsf{tick} = \mathsf{wait}
\end{aligned}$$

A useful helper function, returnW, allows a value to be injected into the monad with an arbitrary weakening of the upper bound.

$$\begin{aligned}
&\mathsf{returnW} :: \forall (n :: \mathbb{N})\ a \,.\, a \rightarrow \mathsf{Cost}\ n\ a \\
&\mathsf{returnW}\ x = \mathsf{wait}\ (\mathsf{return}\ x)
\end{aligned}$$

Danielsson's approach works well for verifying the time complexity of the merge sort and red-black tree operations defined in the previous sections. The *inch* constraint solver is able to deal with the proof obligations automatically, rather than requiring the user to supply proofs of trivial arithmetic properties. There are some obligations on the user of the library not captured by the types: every user function must be annotated with calls to tick, the force function must not occur inside code being timed, and library functions must not be partially applied.

To show how the approach works, I will reimplement red-black tree search with complexity annotations, proving that the time for the membership test is linear in the height of the tree.[5]

[5]That is, it is logarithmic in the number of elements.

First, the data type declaration for the zipper must have an extra index, to count its depth. This is needed to express some of the complexity invariants that the helper functions satisfy.

```
data TreeZip′ :: ℤ → ℤ → ℕ →              -- root indices
                 ℤ → ℤ → ℤ → ℕ →          -- hole indices
                 ℕ →                      -- depth
                  * where
  Root′ :: ∀(i j :: ℤ) (n :: ℕ) . TreeZip′ i j n i j Black n 0
  ZRL′  :: ∀(i j i′ j′ :: ℤ) (n n′ d :: ℕ) . Π (x :: ℤ) .
             TreeZip′ i′ j′ n′ i j Red n d → RBTree x j Black n →
               TreeZip′ i′ j′ n′ i x Black n (d + 1)
  ZRR′  :: ∀(i′ j′ i j :: ℤ) (n′ n d :: ℕ) . Π (x :: ℤ) .
             RBTree i x Black n → TreeZip′ i′ j′ n′ i j Red n d →
               TreeZip′ i′ j′ n′ x j Black n (d + 1)
  ZBL′  :: ∀(i′ j′ i j c :: ℤ) (n′ n d :: ℕ) . Π (x :: ℤ) .
             TreeZip′ i′ j′ n′ i j Black (n + 1) d → RBTree x j Black n →
               TreeZip′ i′ j′ n′ i x c n (d + 1)
  ZBR′  :: ∀(i′ j′ i j c :: ℤ) (n′ n d :: ℕ) . Π (x :: ℤ) .
             RBTree i x c n → TreeZip′ i′ j′ n′ i j Black (n + 1) d →
               TreeZip′ i′ j′ n′ x j Black n (d + 1)
```

The SearchResult type packs up the extra index, but is otherwise unchanged.

```
data SearchResult′ :: ℤ → ℤ → ℤ → ℕ → * where
  Found′   :: ∀(x i′ j′ i j c :: ℤ) (n′ n d :: ℕ) .
                TreeZip′ i′ j′ n′ i j c n d → RBTree i j c n →
                  SearchResult′ x i′ j′ n′
  Missing′ :: ∀(x i′ j′ i j :: ℤ) (n′ d :: ℕ) . (i < x, x < j) ⇒
                TreeZip′ i′ j′ n′ i j Black 0 d →
                  SearchResult′ x i′ j′ n′
```

The searchCost function returns a result in the Cost monad, showing that it takes $2n' + 2$ steps where n' is the black height of the tree. Some work is needed to choose the appropriate invariant to be maintained in the helper function. In this case, the invariant depends on the colour of the tree, so a separate helper function is needed when the subtree is black. The lines of the helper functions are annotated with calls to tick. Pure values are inserted into the Cost monad with returnW. The wait function is used to weaken a bound where the result is computed more quickly than the type requires.

```
searchCost :: $\forall (i'\ j' :: \mathbb{Z})\ (n' :: \mathbb{N})\,.\,\Pi\ (x :: \mathbb{Z})\,.\,(i' < x, x < j') \Rightarrow$
                  RBTree $i'\ j'$ Black $n' \rightarrow$
                    Cost $(2 * n' + 2)$ (SearchResult' $x\ i'\ j'\ n'$)
searchCost $\{x\}\ t$ = tick (helpB Root' $t$)
  where


help :: $\forall (i\ j\ c :: \mathbb{Z})\ (n\ d :: \mathbb{N})\,.$
        $((1 + (2 * n) + d) \leqslant (2 * n'), i < x, x < j) \Rightarrow$
        TreeZip' $i'\ j'\ n'\ i\ j\ c\ n\ d \rightarrow$ RBTree $i\ j\ c\ n \rightarrow$
          Cost $(2 + 2 * n)$ (SearchResult' $x\ i'\ j'\ n'$)
help $z$ E                              = tick (returnW (Missing' $z$))
help $z$ (TR $\{y\}\ l\ r$) | $\{x < y\}$   = tick (helpB (ZRL' $\{y\}\ z\ r$) $l$)
help $z$ (TR $\{y\}\ l\ r$) | $\{x \sim y\}$ = tick (returnW (Found' $z$ (TR $\{y\}\ l\ r$)))
help $z$ (TR $\{y\}\ l\ r$) | $\{x > y\}$   = tick (helpB (ZRR' $\{y\}\ l\ z$) $r$)
help $z$ (TB $\{y\}\ l\ r$) | $\{x < y\}$   = tick (wait (help (ZBL' $\{y\}\ z\ r$) $l$))
help $z$ (TB $\{y\}\ l\ r$) | $\{x \sim y\}$ = tick (returnW (Found' $z$ (TB $\{y\}\ l\ r$)))
help $z$ (TB $\{y\}\ l\ r$) | $\{x > y\}$   = tick (wait (help (ZBR' $\{y\}\ l\ z$) $r$))
helpB :: $\forall (i\ j :: \mathbb{Z})\ (n\ d :: \mathbb{N})\,.$
        $(((2 * n) + d) \leqslant (2 * n'), i < x, x < j) \Rightarrow$
          TreeZip' $i'\ j'\ n'\ i\ j$ Black $n\ d \rightarrow$ RBTree $i\ j$ Black $n \rightarrow$
            Cost $(2 * n + 1)$ (SearchResult' $x\ i'\ j'\ n'$)
helpB $z$ E                              = tick (returnW (Missing' $z$))
helpB $z$ (TB $\{y\}\ l\ r$) | $\{x < y\}$ = tick (help (ZBL' $\{y\}\ z\ r$) $l$)
helpB $z$ (TB $\{y\}\ l\ r$) | $\{x \sim y\}$ = tick (returnW (Found' $z$ (TB $\{y\}\ l\ r$)))
helpB $z$ (TB $\{y\}\ l\ r$) | $\{x > y\}$ = tick (help (ZBR' $\{y\}\ l\ z$) $r$)
```

The membership test can be implemented as before, inserting the necessary monadic plumbing and calls to tick. Thus it returns a result in $2n + 4$ steps.

```
memberCost :: $\forall (i\ j :: \mathbb{Z})\ (n :: \mathbb{N})\,.\,\Pi\ (x :: \mathbb{Z})\,.\,(i < x, x < j) \Rightarrow$
                 RBTree $i\ j$ Black $n \rightarrow$ Cost $(2 * n + 4)$ Bool
memberCost $\{x\}\ t$ = tick (bind (searchCost $\{x\}\ t$) $f$)
  where
    $f$ :: SearchResult' $x\ i\ j\ n \rightarrow$ Cost 1 Bool
    $f$ (Missing' _)  = tick (return False)
    $f$ (Found' _ _) = tick (return True)
```

The force function can be used to escape the Cost monad and acquire a value.

$$\mathsf{member'} :: \forall (i\ j :: \mathbb{Z})\,.\,\Pi\ (x :: \mathbb{Z})\,.\,(i < x, x < j) \Rightarrow \mathsf{RBT}\ i\ j \to \mathsf{Bool}$$
$$\mathsf{member'}\ \{x\}\ (\mathsf{RBT}\ t) = \mathsf{force}\ (\mathsf{memberCost}\ \{x\}\ t)$$

This approach can be used to show that both insertion and deletion are linear in the black height of the tree. The types given to the main functions are:

$$\begin{aligned}&\mathsf{insert} :: \forall (i\ j :: \mathbb{Z})\ (n :: \mathbb{N})\,.\,\Pi\ (x :: \mathbb{Z})\,.\,(i < x, x < j) \Rightarrow\\ &\qquad\qquad \mathsf{Tree}\ i\ j\ \mathsf{Black}\ n \to \mathsf{Cost}\ (4 * n + 6)\ (\mathsf{RBT}\ i\ j)\\ &\mathsf{delete} :: \forall (i\ j :: \mathbb{Z})\ (n :: \mathbb{N})\,.\,\Pi\ (x :: \mathbb{Z})\,.\,(i < x, x < j) \Rightarrow\\ &\quad \mathsf{Tree}\ i\ j\ \mathsf{Black}\ n \to \mathsf{Cost}\ (5 * n + 6)\ (\mathsf{RBT}\ i\ j)\end{aligned}$$

As in the member example, the main difficulty is in choosing appropriate invariants; the annotation is routine. Interactive program construction makes this easier, as it enables exploratory programming.

8.5 Units of measure

This section demonstrates a use for type-level integers, rather than natural numbers: a library for representing units of measure. Unlike the approach taken in Chapter 3, which requires a language extension but can support an arbitrary set of base units, this library can be implemented using existing features of *inch*, but the base units must be fixed ahead of time. Moreover, type errors will reveal the underlying representation of units, rather than being expressed in an easy-to-understand format. The `dimensional` package of Buckwalter (n.d.) is a much more comprehensive implementation of units of measure using this approach, but with type-level integers implemented via existing features of GHC Haskell.

The Unit constructor has arguments for the powers of three base units (metres, seconds and kilograms). A real units of measure implementation would supply more base units, but the number would still be fixed. The Quantity newtype wraps a numeric value, and has a phantom type parameter that will be instantiated with some application of Unit. This separation makes it easy to write functions that are completely polymorphic in the units.

$$\textbf{data}\ \mathsf{Unit} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to *$$
$$\textbf{newtype}\ \mathsf{Quantity}\ u\ a = \mathsf{Q}\ \{\mathsf{value} :: a\}$$

The Q constructor should not be exported from the module in which it is defined, in order to prevent clients of the library from changing units arbitrarily. Instead, all access must be through the functions defined below.

In the full *inch* system, with support for promoted datatypes, Unit would be a data constructor rather than a type constructor. Type synonyms can be defined for common units. If type families or type-level functions were available, one could define operations to combine units (such as multiplication of units).

```
type Dimensionless   = Unit 0 0 0
type Metres          = Unit 1 0 0
type Seconds         = Unit 0 1 0
type Kilograms       = Unit 0 0 1
type MetresPerSecond = Unit 1 (−1) 0
type Newtons         = Unit 1 (−2) 1
```

Users of the library will have access to smart constructors, which wrap the underlying newtype constructor Q, but specify the units.

```
dimensionless :: a → Quantity Dimensionless a
metres        :: a → Quantity Metres a
seconds       :: a → Quantity Seconds a
kilograms     :: a → Quantity Kilograms a
(dimensionless, metres, seconds, kilograms) = (Q, Q, Q, Q)
```

The usual arithmetic operations can be defined on quantities, with the types ensuring that the units are respected. However, Quantity u a cannot be made an instance of the Num typeclass, because multiplication does not preserve units.

```
plus :: Num a ⇒ Quantity u a → Quantity u a → Quantity u a
plus (Q x) (Q y) = Q (x + y)
minus :: Num a ⇒ Quantity u a → Quantity u a → Quantity u a
minus (Q x) (Q y) = Q (x − y)
```

The type signatures of the following operations would be significantly simpler if type-level functions could be defined.

```
times :: ∀(m s g m′ s′ g′ :: ℤ) a . Num a ⇒
  Quantity (Unit m s g) a → Quantity (Unit m′ s′ g′) a →
    Quantity (Unit (m + m′) (s + s′) (g + g′)) a
times (Q x) (Q y) = Q (x ∗ y)
inv :: ∀(m s g :: ℤ) a . Fractional a ⇒
  Quantity (Unit m s g) a → Quantity (Unit (−m) (−s) (−g)) a
inv (Q x) = Q (recip x)
```

```
over :: ∀(m s g m′ s′ g′ :: ℤ) a . (Num a, Fractional a) ⇒
  Quantity (Unit m s g) a → Quantity (Unit m′ s′ g′) a →
    Quantity (Unit (m − m′) (s − s′) (g − g′)) a
over x y = times x (inv y)

pow :: ∀(m s g :: ℤ) a . Fractional a ⇒
  Π (k :: ℕ) . Quantity (Unit m s g) a →
    Quantity (Unit (k ∗ m) (k ∗ s) (k ∗ g)) a
pow {k} (Q x) = Q (x^^k)
```

Scaling a quantity by a dimensionless constant is useful:

```
scale :: Num a ⇒ a → Quantity u a → Quantity u a
scale x (Q y) = Q (x ∗ y)

minutes = scale 60 ∘ seconds
hours   = scale 60 ∘ minutes
```

More generally, unit prefixes can be written as transformers of the constructors that scale by an appropriate constant:

```
type Prefix u a = (a → Quantity u a) → a → Quantity u a

prefix :: Num a ⇒ a → Prefix u a
prefix n f = scale n ∘ f

kilo  = prefix 1000
centi = prefix (recip 100)
milli = prefix (recip 1000)
```

This allows prefixed units to be expressed neatly:

```
km = kilo metres
cm = centi metres
mm = milli metres
```

Finally, a special case of flipped application allows expressions such as units 3 cm and units 15 km ‘over‘ units 3 hours.

```
units :: a → (a → Quantity u b) → Quantity u b
units x f = f x
```

As an example of using the library, here is a variant of the function from the introduction to Chapter 3 that calculates the distance travelled over time by an object with a fixed initial velocity and constant acceleration. The top-level type annotation is entirely optional.

```
distanceTravelled :: (Num a, Fractional a) ⇒
                        Quantity Seconds a → Quantity Metres a
distanceTravelled t = plus (times vel t) (times accel (pow {2} t))
  where
    vel   = over (units 20 metres) (units 1 seconds)
    accel = over (units 36 metres) (pow {2} (units 1 seconds))
```

Kennedy (2010, §3.10) gave an example of a function whose type cannot be inferred by the units-of-measure type system in F#, because of difficulties with generalisation, as explained in Subsection 3.0.1.

```
trouble = \ x → let d = over x
                in (d mass, d time)
  where
    mass = units 5 kilograms
    time = units 3 seconds
```

The *inch* system has no trouble inferring the most general type for this function

```
trouble :: ∀ a (m :: ℤ) (s :: ℤ) (g :: ℤ) .
  (Num a, Fractional a) ⇒
    Quantity (Unit m s g) a →
      (Quantity (Unit m s (g − 1)) a, Quantity (Unit m (s − 1) g) a)
```

although the fixed basis of units means it is more limited than Kennedy's solution or the algorithm in Chapter 3.

Chapter 9

Conclusion

The *inch* language described in this thesis is an experiment in re-imagining GHC Haskell. It shows how insights from work on dependent type theory can contribute to the development of Haskell's type system, intermediate language and elaboration process. It is not intended as a finished product or a rival system; rather, I have investigated some of the ways in which Haskell might develop.

Haskell as implemented in GHC is a moving target, with new language extensions being introduced frequently. The recent enrichment of the kind system with polymorphism and datatype promotion paves the way for the identification of kinds with types, a key aspect of the design of *inch*, and work to implement this is ongoing. Weirich et al. (2013) and the *evidence* language of Chapter 6 show that this gives a reasonable core language; the discussion of elaboration in Chapter 7 gives some idea of how type inference will continue to work.

The addition of Π-types to the language offers the possibility of significantly simplifying Haskell programming with dependent types. In particular, it avoids the need for singleton constructions that result in many incompatible names for essentially the same object. If Haskell's type system is to become more dependent, the key requirement is for the operational semantics of the term and type levels to be aligned, breaking the strict distinction between functions and type families. The shared functions of this thesis offer a possible way forward. While not requiring the identification of kinds and types, Π-types are much more useful if the identification is made, since then indexed datatypes can be quantified over.

Another key aspect of Chapter 7 is the increased flexibility it offers for which arguments are expected to be inferred by the machine. Milner's compromise, particularly the insistence that type-level expressions be invisible in terms, is no longer tenable in a world of advanced type-level features. By providing the machine with a small amount of help, we can gain significant expressive power.

The case for permitting explicit type application and quantification grows ever stronger. Π-types benefit from case-by-case decisions on whether they should be explicit or implicit, and extending the same mechanism to ∀-quantifiers seems natural. In any case, wherever an argument is supposed to be inferred by the machine, it should be possible for the user to supply it.

Type inference and unification with nontrivial equational theories has been a key theme of the first part of this thesis, including the theory of abelian groups for units of measure in Chapter 3 and the theory of $\beta\eta$-conversion in Chapter 4. A desirable feature for a system of type-level numbers is automatically solving the constraints that arise, and the abelian group structure of the integers provides a starting point for this, though the presence of local hypotheses complicates matters and more research is needed. As I have outlined, the careful management of variable scope (using dependency-ordered contexts) can help make it clear how to solve constraints in a most general fashion.

Elaboration of full-spectrum dependently-typed languages is another topic in need of further work, as practical implementations are not always theoretically well-understood. I hope that the higher-order unification algorithm in Chapter 4 may provide a useful base for describing elaboration more precisely.

Appendix A

Reference implementation of Hindley-Milner type inference

In this appendix and the two that follow I will present reference implementations for the unification and type inference algorithms described in Part I of this thesis. The implementations are presented in literate Haskell, and I will take slight liberties with the Haskell syntax. In particular, I will use italicised capital letters (e.g. A) for Haskell variables, while sans-serif capital letters (e.g. A) will continue to stand for data constructors. This allows me to retain more of the syntactic conventions of the earlier chapters, such as using Θ for a metacontext and A for an object language type. I will omit boilerplate code such as module import lists and straightforward typeclass instances, and routine support code for pretty-printing and testing.

The code has been tested using version 7.6.3 of the Glasgow Haskell Compiler, with version 2013.2.0.0 of the Haskell Platform and version 0.6 of the Strathclyde Haskell Enhancement (McBride, 2010b). It is available online[1] and with the electronic version of this thesis. In addition to the standard libraries, the Binders Unbound library of Weirich et al. (2011b) is used to represent syntax with names and bindings, deriving α-equivalence and substitution functions automatically.

In this appendix, I implement syntactic unification and Hindley-Milner type inference, as described in Chapter 2. Section A.1 gives datatypes representing types, terms and contexts in the object language; Section A.2 gives the implementation of unification, and this is used in Section A.3 to implement type inference. Finally, Section A.4 contains an implementation of elaboration from Hindley-Milner terms into System F, based on a zipper.

[1] `https://github.com/adamgundry/type-inference/`

A.1 Representation of types and terms

This section implements type, contexts and terms, as in Section 2.1 (page 11).

The datatype Type represents types of the object language, which may contain metavariables M and variables V as well as functions and a base type. The Name constructor is provided by the Binders Unbound library.

```
data Type = M (Name Type) | V (Name Type) | Type ⇾ Type
```

The fmv function computes the free metavariables of a type.

```
fmv :: Type → Set (Name Type)
fmv (M α)    = {α}
fmv (V a)    = ∅
fmv (τ ⇾ υ) = fmv τ ∪ fmv υ
```

The datatype Scheme represents type schemes. Binding variables uses a locally nameless representation where bound variables have de Bruijn indices and free variables (those bound in the context) have names (McBride and McKinna, 2004).

```
data Scheme = T Type | All (Bind (Name Type) Scheme)
```

Bwd is the type of backwards lists with • for the empty list and :< for snoc. Lists are traversable functors, and monoids under concatenation (⊙), in the usual way. Datatype declarations are cheap, so rather than reusing the forwards list type [ ], I prefer to make the code closer to the specification.

```
data Bwd a = • | Bwd a :< a
```

Contexts are backwards lists of entries, which are either metavariables E (possibly carrying a definition), term variables Z or generalisation markers ⁏. A context suffix contains only metavariable entries, and can be appended to a context with the 'fish' operator (<><).

```
type Context = Bwd Entry
type Suffix  = [(Name Type, Decl Type)]
data Decl υ  = HOLE | DEFN υ
data Entry   = E (Name Type) (Decl Type) | Z (Name Tm) Scheme | ⁏
```

```
(<><) :: Context → Suffix → Context
Θ <>< ((α, d) : es) = (Θ :< E α d) <>< es
Θ <>< [ ]            = Θ
```

The Contextual monad represents computations that can mutate the context, generate fresh names and throw exceptions. It thus encapsulates the effects needed to implement unification and type inference. I will use the throwError operation in the monad to abort due to 'expected' errors, such as impossible unification problems, and the Haskell built-in error for violations of invariants that would indicate bugs in the implementation itself.

```
newtype Contextual a = Contextual
  (StateT Context (FreshMT (ErrorT String Identity)) a)
```

The popL function removes and returns an entry from the metacontext.

```
popL :: Contextual Entry
popL = do  Θ :< e ← get
           put Θ
           return e
```

The freshMeta function generates a fresh metavariable name and appends a HOLE to the context.

```
freshMeta :: String → Contextual (Name Type)
freshMeta a = do  α ← fresh (s2n a)
                  modify (:<E α HOLE)
                  return α
```

The datatype Tm represents terms in the object language. As with type schemes, it uses a locally nameless representation.

```
data Tm = X (Name Tm)                  | App Tm Tm
        | Lam (Bind (Name Tm) Tm) | Let Tm (Bind (Name Tm) Tm)
```

The Contextual monad supports the find function, which looks up a term variable in the context and returns its scheme.

```
find :: Name Tm → Contextual Scheme
find x = get ≫= help
  where
    help :: Context → Contextual Scheme
    help •                    = throwError $ "Out of scope: " ++ show x
    help (Θ :< Z y σ) | x ≡ y = return σ
    help (Θ :< _)             = help Θ
```

The inScope operator runs a Contextual computation with an additional term variable in scope, then removes the variable afterwards.

```
inScope :: Name Tm → Scheme → Contextual $a$ → Contextual $a$
inScope $x$ $\sigma$ $m$ = do  modify (:<Z $x$ $\sigma$)
                               $a \leftarrow m$
                               modify dropVar
                               return $a$
  where
    dropVar •                          = error "Invariant violation"
    dropVar ($\Theta$ :< Z $y$ _) | $x \equiv y$ = $\Theta$
    dropVar ($\Theta$ :< $e$)                = dropVar $\Theta$ :< $e$
```

A.2 Unification

Having set up the necessary data structures, I will now implement the unification algorithm of Section 2.2 (page 19).

The onTop operator delivers the typical access pattern for contexts, locally bringing the top variable declaration into focus and working over the remainder. The local operation f, passed as an argument, may restore the previous entry, or it may return a context extension (containing at least as much information as the entry that has been removed) with which to replace it.

```
data Extension = Restore | Replace Suffix


onTop :: (Name Type → Decl Type → Contextual Extension)
             → Contextual ()
onTop $f$ = popL >>= \ $e$ → case $e$ of
  E $\alpha$ $d$ → $f$ $\alpha$ $d$ >>= \ $m$ → case $m$ of
            Replace $\Xi$ → modify (<>< $\Xi$)
            Restore    → modify (:<$e$)
  _ → onTop $f$ >> modify (:<$e$)


restore :: Contextual Extension
restore = return Restore

replace :: Suffix → Contextual Extension
replace = return ∘ Replace
```

The unify function actually implements unification. This proceeds structurally over types. If it reaches a pair of metavariables, it examines the context, using onTop to pick out a declaration to consider. Depending on the metavariables, it then either succeeds, restoring the old entry or replacing it with a new one, or continues with an updated constraint.

```
unify :: Type → Type → Contextual ()
unify ($\tau_0$ ⇾ $\tau_1$) ($\upsilon_0$ ⇾ $\upsilon_1$) = unify $\tau_0$ $\upsilon_0$ ≫ unify $\tau_1$ $\upsilon_1$
unify (M $\alpha$) (M $\beta$) = onTop $ \ $\gamma$ $d$ → case
   ($\gamma \equiv \alpha, \gamma \equiv \beta, d$         ) of
   (True,   True,   _        ) → restore
   (True,   False,  HOLE     ) → replace [($\alpha$, DEFN (M $\beta$))]
   (False,  True,   HOLE     ) → replace [($\beta$, DEFN (M $\alpha$))]
   (True,   False,  DEFN $\tau$) → unify (M $\beta$) $\tau$       ≫ restore
   (False,  True,   DEFN $\tau$) → unify (M $\alpha$) $\tau$       ≫ restore
   (False,  False,  _        ) → unify (M $\alpha$) (M $\beta$) ≫ restore
unify (M $\alpha$) $\tau$                  = solve $\alpha$ [] $\tau$
unify $\tau$       (M $\alpha$)            = solve $\alpha$ [] $\tau$
unify _        _                   = throwError "Rigid-rigid mismatch"
```

The solve function is called to unify a metavariable with a rigid type (one that is not a metavariable). It works similarly to the way unify works on pairs of metavariables, but must also accumulate a list of the type's dependencies and push them left through the context. It performs the occurs check and throws an exception if an illegal occurrence (leading to an infinite type) is detected.

```
solve :: Name Type → Suffix → Type → Contextual ()
solve $\alpha$ $\Xi$ $\tau$ = onTop $
   \ $\gamma$ $d$ → case
     ($\gamma \equiv \alpha, \gamma \in$ fmv $\tau, d$          ) of
     (_,       _,            DEFN $\upsilon$) → modify (<>< $\Xi$)
                                    ≫ unify (subst $\gamma$ $\upsilon$ (M $\alpha$)) (subst $\gamma$ $\upsilon$ $\tau$)
                                    ≫ restore
     (True,   True,        HOLE   ) → throwError "Occurrence detected!"
     (True,   False,       HOLE   ) → replace ($\Xi \odot$ [($\alpha$, DEFN $\tau$)])
     (False,  True,        HOLE   ) → solve $\alpha$ (($\gamma$, HOLE) : $\Xi$) $\tau$
                                    ≫ replace []
     (False,  False,       HOLE   ) → solve $\alpha$ $\Xi$ $\tau$
                                    ≫ restore
```

A.3 Type inference

Building on the implementation of unification in the previous section, I now implement the type inference algorithm described in Section 2.3 (page 23).

The metaBind and metaUnbind functions extend the bind and unbind functions provided by the Binders Unbound library, so that binding a metavariable converts it into a variable, and vice versa.

```
metaBind :: (Alpha t, Subst Type t) ⇒
                Name Type → t → Bind (Name Type) t
metaBind α = bind α ∘ subst α (V α)
metaUnbind :: (Alpha t, Subst Type t, Fresh m) ⇒
                  Bind (Name Type) t → m (Name Type, t)
metaUnbind b = do  (a, t) ← unbind b
                   return (a, subst a (M a) t)
```

Specialisation of type schemes is implemented by the specialise function, which unpacks a scheme with fresh metavariables for the bound variables.

```
specialise :: Scheme → Contextual Type
specialise (T τ)   = return τ
specialise (All b) = do  (β, σ) ← metaUnbind b
                         modify (:<E β HOLE)
                         specialise σ
```

Generalisation turns a type into a scheme by 'skimming' entries off the top of the metacontext. The generaliseOver control operator runs a Contextual computation in a new locality (extending the context by ⁏), then generalises the resulting type until it finds the ⁏ again. This depends on the ⇑ function which generalises a suffix of metavariables over a type to produce a scheme.

```
generaliseOver :: Contextual Type → Contextual Scheme
generaliseOver $x$ = do  modify (:<$⁏$)
                         $\tau$ ← $x$
                         $\Xi$ ← skimContext [ ]
                         return ($\Xi$ $\Uparrow$ $\tau$)
  where
    skimContext :: Suffix → Contextual Suffix
    skimContext $\Xi$ = popL ≫= \ $e$ → case $e$ of
      E $\alpha$ $d$      → skimContext (($\alpha$, $d$) : $\Xi$)
      $⁏$          → return $\Xi$
($\Uparrow$) :: Suffix → Type → Scheme
[ ]                     $\Uparrow$ $\tau$ = T $\tau$
(($\alpha$, HOLE)    : $\Xi$) $\Uparrow$ $\tau$ = All (metaBind $\alpha$ ($\Xi$ $\Uparrow$ $\tau$))
(($\alpha$, DEFN $\upsilon$) : $\Xi$) $\Uparrow$ $\tau$ = subst $\alpha$ $\upsilon$ ($\Xi$ $\Uparrow$ $\tau$)
```

Finally, the infer function implements the type inference algorithm. It proceeds structurally through the term, following the rules in Figure 2.9 (page 26) and using the monadic operations defined earlier.

```
infer :: Tm → Contextual Type
infer (X $x$)       = find $x$ ≫= specialise
infer (Lam $b$)     = do  ($x$, $t$) ← unbind $b$
                          $\alpha$      ← M ⟨$⟩ freshMeta "alpha"
                          $\upsilon$      ← inScope $x$ (T $\alpha$) $ infer $t$
                          return ($\alpha$ ⊸ $\upsilon$)
infer (App $f$ $s$) = do  $\chi$      ← infer $f$
                          $\upsilon$      ← infer $s$
                          $\beta$      ← M ⟨$⟩ freshMeta "beta"
                          unify $\chi$ ($\upsilon$ ⊸ $\beta$)
                          return $\beta$
infer (Let $s$ $b$) = do  $\sigma$      ← generaliseOver (infer $s$)
                          ($x$, $t$) ← unbind $b$
                          inScope $x$ $\sigma$ $ infer $t$
```

A.4 Elaboration, zipper style

In this section, I implement the zipper-based elaboration algorithm described in Section 2.4 (page 27). This transforms source language terms Tm (defined in Section A.1) into System F terms FTm, represented thus:

```
data FTm = VarF (Name FTm) | AppTm FTm FTm | AppTy FTm Type
         | LamTm Scheme (Bind (Name FTm) FTm)
         | LamTy (Bind (Name Type) FTm)
```

As described in the text, context entries now consist of metavariables and layers:

```
data TermLayer = AppLeft () Tm
               | AppRight (FTm, Type) ()
               | LamBody (Name Tm, Type) ()
               | LetBinding () (Bind (Name Tm) Tm)
               | LetBody (Name Tm) (FTm, Scheme) ()
data Entry     = E (Name Type) (Decl Type) | L TermLayer
```

Most functions from the previous sections, including the unification algorithm, remain unchanged. The find function, which looks up a term variable in the context and returns its type scheme, is easily adapted to the new structure:

```
find :: Name Tm → Contextual Scheme
find $x$ = get ≫= help
  where
    help :: Context → Contextual Scheme
    help • = throwError $ "Out of scope: " ++ show $x$
    help ($\Theta$ :< L (LamBody ($y, \tau$) ()))   | $x \equiv y$ = return (T $\tau$)
    help ($\Theta$ :< L (LetBody $y$ (_, $\sigma$) ())) | $x \equiv y$ = return $\sigma$
    help ($\Theta$ :< _)                              = help $\Theta$
```

The specialise function takes an elaborated term and its scheme, and applies the term to fresh metavariables to produce a witness of the specialised type.

```
specialise :: FTm → Scheme → Contextual (FTm, Type)
specialise $t$ (T $\tau$)  = return ($t, \tau$)
specialise $t$ (All $b$) = do ($\beta, \sigma$) ← metaUnbind $b$
                          modify (:<E $\beta$ HOLE)
                          specialise ($t$ `AppTy` M $\beta$) $\sigma$
```

Now elaboration can be implemented as a tail-recursive function elab. To elaborate a variable, it looks up the type scheme and instantiates it with fresh metavariables, then calls the next function to navigate the zipper structure and find the next elaboration problem. For λ-abstractions, applications and let-bindings, it extends the zipper and elaborates the appropriate subterm.

```
elab :: Tm → Contextual (FTm, Type)
elab (X $x$)         = do $\sigma$      ← find $x$
                          next [] =≪ specialise (VarF $x$) $\sigma$
elab (Lam $b$)       = do $(x, t)$ ← unbind $b$
                          $\alpha$      ← freshMeta "alpha"
                          modify (:<L (LamBody $(x$, M $\alpha)$ ())) ≫ elab $t$
elab ($f$ `App` $a$) = modify (:<L (AppLeft () $a$)) ≫ elab $f$
elab (Let $s$ $b$)   = modify (:<L (LetBinding () $b$)) ≫ elab $s$
```

The next function is called with the term at the current location and its type. It navigates through the zipper structure to find the next elaboration problem, updating the current term and type as it does so. The accumulator Ξ collects metavariables that encountered along the way. These are reinserted into the context once the new problem is found, or if a LetBinding layer is encountered, Ξ contains exactly the metavariables to generalise over.

```
next :: Suffix → (FTm, Type) → Contextual (FTm, Type)
next $\Xi$ $(t, \tau)$ = optional popL ≫= \ $e$ → case $e$ of
  Just (L (AppLeft () $a$))          → do modify (<>< $\Xi$)
                                          modify (:<L (AppRight $(t, \tau)$ ()))
                                          elab $a$
  Just (L (AppRight $(f, \sigma)$ ())) → do modify (<>< $\Xi$)
                                          $\beta$ ← M ⟨$⟩ freshMeta "beta"
                                          unify $\sigma$ $(\tau \rightarrowtriangle \beta)$
                                          next [] ($f$ `AppTm` $t, \beta$)
  Just (L (LamBody $(x, \upsilon)$ ())) → next $\Xi$ $(\lambda x{:}\upsilon.\, t, \upsilon \rightarrowtriangle \tau)$
  Just (L (LetBinding () $b$))       → do $(x, w)$ ← unbind $b$
                                          let $(t', \sigma) = (\Lambda\Xi.\, t, \Xi \Uparrow \tau)$
                                          modify (:<L (LetBody $x$ $(t', \sigma)$ ()))
                                          elab $w$
  Just (L (LetBody $x$ $(s, \sigma)$ ())) → next $\Xi$ ($\lambda x{:}\sigma.\, t$ `AppTm` $s, \tau$)
  Just (E $\alpha$ $d$)                → next $((\alpha, d) : \Xi)$ $(t, \tau)$
  Nothing                          → modify (<>< $\Xi$) ≫ return $(t, \tau)$
```

Appendix B

Reference implementation of units of measure

This appendix extends the implementation of unification in Appendix A to support the units of measure of Chapter 3. Section B.1 introduces the data types representing units of measure in normal form, using the `signed-multiset` library of Holdermans (2013). Section B.2 extends the representation of types and contexts from Section A.1 to support the syntax of units. The implementation of unification for units of measure is given in Section B.3, and this is used to implement type unification in Section B.4. There is no change to the implementation of type inference from Section A.3, other than using the new unification algorithm.

B.1 Representation of units of measure

I begin by introducting the semantic representation of units of measure, along with operations on them, as described in Section 3.1 (page 35). A unit of measure is represented as a Unit value with signed multisets of metavariables and constants. For simplicity, the type of base units is fixed.

```
data Unit = Unit (SignedMultiset (Name Type)) (SignedMultiset BaseUnit)
data BaseUnit = METRE | SEC | KG
```

The mkUnit function creates a unit from lists of powers of metavariables and base units. As a special case, metaUnit creates a unit from a single metavariable.

```
mkUnit :: [(Name Type, Int)] → [(BaseUnit, Int)] → Unit
mkUnit vs bs = Unit (fromList vs) (fromList bs)

metaUnit :: Name Type → Unit
metaUnit a = mkUnit [(a, 1)] []
```

Utility functions determine if a unit is the identity or constant, the number of variables it contains, and the power of a metavariable in it.

```
isIdentity :: Unit → Bool
isIdentity (Unit vs bs)  = null vs ∧ null bs
isConstant :: Unit → Bool
isConstant (Unit vs bs) = null vs
numVariables :: Unit → Int
numVariables (Unit vs _) = size vs
powerIn :: Name Type → Unit → Int
α `powerIn` Unit vs _ = multiplicity α vs
```

The dividesPowers function determines if an integer divides all the powers of metavariables and base units.

```
dividesPowers :: Int → Unit → Bool
n `dividesPowers` (Unit vs bs) = dividesAll vs ∧ dividesAll bs
  where
    dividesAll :: SignedMultiset a → Bool
    dividesAll = all ((0 ≡) . (‘mod‘n) . snd) . toList
```

The notMax function determines if the power of a variable is less than the power of at least one other variable.

```
notMax :: (Name Type, Int) → Unit → Bool
notMax (α, n) (Unit vs _) = any bigger (toList vs)
  where bigger (β, m) = α ≢ β ∧ abs n ⩽ abs m
```

The (⊛), (⊘) and (⦸) operators respectively multiply and divide units, and raise a unit to a constant power.

```
(⊛) :: Unit → Unit → Unit
Unit vs bs ⊛ Unit vs′ bs′ = Unit (additiveUnion vs vs′) (additiveUnion bs bs′)
(⊘) :: Unit → Unit → Unit
d ⊘ e = d ⊛ invert e
(⦸) :: Unit → Int → Unit
Unit vs bs ⦸ k = Unit (multiply k vs) (multiply k bs)
invert :: Unit → Unit
invert (Unit vs bs) = Unit (shadow vs) (shadow bs)
```

The pivot function removes the given metavariable from the unit, inverts it and takes the quotient of its powers by the power of the removed variable.

```
pivot :: Name Type → Unit → Unit
pivot α e = invert $ quotient $ e ⊘ (metaUnit α ⊙ n)
  where
    n = α `powerIn` e
    quotient (Unit vs bs) = mkUnit (map (second ( `quot` n)) (toList vs))
                                   (map (second ( `quot` n)) (toList bs))
```

The substUnit function substitutes a unit for a metavariable in another unit.

```
substUnit :: Name Type → Unit → Unit → Unit
substUnit α d e = ((d ⊘ metaUnit α) ⊙ (α `powerIn` e)) ⊛ e
```

B.2 Representation of types

Now I extend the representation of types and contexts from Section A.1 to include units of measure, as described in Subsection 3.0.2 (page 33). The datatype of types retains metavariables, variables and functions, and gains syntax for units (types of kind $\mathcal{U}$): the identity, multiplication, constant expontentiation and base units. The Float constructor is an example of a type parameterised by a unit.

```
data Kind = ⋆ | 𝒰
data Type = M (Name Type) | V (Name Type) | Type ⇾ Type
          | Float Type | One | Type :∗ Type | Type :∧ Int | Base BaseUnit
```

The set of free metavariables is computed in the obvious way.

```
fmv :: Type → Set (Name Type)
fmv (M α)     = {α}
fmv (V a)     = ∅
fmv (τ ⇾ υ)   = fmv τ ∪ fmv υ
fmv (Float ν) = fmv ν
fmv One       = ∅
fmv (ν :∗ ν′)  = fmv ν ∪ fmv ν′
fmv (ν :∧ _)  = fmv ν
fmv (Base _)  = ∅
```

It is easy to convert a semantic Unit to a syntactic expression Type, while the other direction may fail if the type is not well-kinded.

```
unitToType :: Unit → Type
unitToType (Unit xs ys) = foldr (\ α k τ → (M α :∧ k) :∗ τ) One xs
                          :∗ foldr (\ u k τ → (Base u :∧ k) :∗ τ) One ys

typeToUnit :: Type → Unit
typeToUnit (M α)     = metaUnit α
typeToUnit One       = mkUnit [] []
typeToUnit (ν :∗ ν′) = typeToUnit ν ⊛ typeToUnit ν′
typeToUnit (ν :∧ k)  = typeToUnit ν ∧⃝ k
typeToUnit (Base b)  = mkUnit [] [(b, 1)]
typeToUnit _         = error "typeToUnit: kind error"
```

Type schemes are defined as in Appendix A, except that each ∀ quantifier carries a kind.

```
data Scheme = T Type | All Kind (Bind (Name Type) Scheme)
```

Similarly, contexts are generalised to record the kinds of metavariables:

```
type Context = Bwd Entry
type Suffix  = [(Name Type, Kind, Decl Type)]
data Entry   = E (Name Type) Kind (Decl Type)
             | Z (Name Tm) Scheme
             | ⨾
data Decl v  = HOLE | DEFN v
```

The type Tm of terms is unchanged from Appendix A. Likewise, the Contextual monad and popL, find and inScope operations use the new definition of Context but are otherwise identical. The freshMeta operation is parameterised over the kind of the metavariable to create:

```
freshMeta :: String → Kind → Contextual (Name Type)
freshMeta a κ = do α ← fresh (s2n a)
                   modify (:<E α κ HOLE)
                   return α
```

The unification algorithm must searching the context for metavariable declarations (perhaps of a particular kind), make some changes and either choose to restore the existing declaration or replace it with a new one. As before, the onTop function captures this pattern, and it is used to implement onTop* and onTop$^\mathcal{U}$ that look for a metavariable of the corresponding kind.

```
data Extension = Restore | Replace Suffix

restore :: Contextual Extension
restore = return Restore

replace :: Suffix → Contextual Extension
replace = return ∘ Replace

onTop :: (Name Type → Kind → Decl Type → Contextual Extension) →
             Contextual ()
onTop f = popL ≫= \ e → case e of
   E α κ d → f α κ d ≫= \ m → case m of
                  Replace Ξ → modify (<>< Ξ)
                  Restore    → modify (:<e)
   _          → onTop f ≫ modify (:<e)

onTop⋆ :: (Name Type → Decl Type → Contextual Extension) →
              Contextual ()
onTop⋆ f = onTop $ \ α κ d → case κ of
   ⋆ → f α d
   𝒰 → onTop⋆ f ≫ restore

onTop𝒰 :: (Name Type → Decl Type → Contextual Extension) →
              Contextual ()
onTop𝒰 f = onTop $ \ α κ d → case κ of
   𝒰 → f α d
   ⋆ → onTop𝒰 f ≫ restore
```

B.3 Unification of unit expressions

I now implement the abelian group unification algorithm given in Section 3.1 (page 35). This is based around an algorithm for unifying single expressions with the group identity. A pair of expressions can then be unified thus:

```
unifyUnit :: Type → Type → Contextual ()
unifyUnit d e = unifyId Nothing $ typeToUnit d ⊘ typeToUnit e
```

To unify a unit expression e with the identity, first check if it is already the identity (and win) or is another constant (and lose). Otherwise, search the context for group variables that occur in e. When one is found, either substitute it into the expression (if it has a definition) or examine the coefficients to determine how to proceed. If its coefficient n divides all the others, it can be defined to solve the equation. Otherwise, either reduce the coefficients modulo n or just collect the variable and move it back in the context.

```
unifyId :: Maybe (Name Type) → Unit → Contextual ()
unifyId Ψ e
    | isIdentity e   = return ()
    | isConstant e = throwError "Unit mismatch!"
    | otherwise      = onTop^U $ \ α d →
      let n = α `powerIn` e in
      case d of
        _ | n ≡ 0                → do unifyId Ψ e
                                     restore
        DEFN x                   → do modify (ins Ψ)
                                     let e′ = substUnit α (typeToUnit x) e
                                     unifyId Nothing e′
                                     restore
        HOLE
          | n `dividesPowers` e  → do modify (ins Ψ)
                                     let p = pivot α e
                                     replace [(α, U, DEFN (unitToType p))]
          | (α, n) `notMax` e    → do modify (ins Ψ)
                                     β ← fresh (s2n "beta")
                                     let p = pivot α e ⊛ metaUnit β
                                     unifyId (Just β) $ substUnit α p e
                                     replace [(α, U, DEFN (unitToType p))]
          | numVariables e > 1 → do unifyId (Just α) e
                                     replace []
          | otherwise            → throwError "No way!"

ins :: Maybe (Name Type) → Context → Context
ins Nothing  Θ = Θ
ins (Just α) Θ = Θ :< E α U HOLE
```

B.4 Unification of types

Here I implement the type unification algorithm given in Section 3.2 (page 39). The implementation of unify for types with units of measure is very similar to the version in Section A.2, except that it calls unifyUnit to unify the unit annotations of Float types, and uses startSolve in place of solve as discussed below.

```
unify :: Type → Type → Contextual ()
unify (τ₀ ⇾ τ₁) (υ₀ ⇾ υ₁) = unify τ₀ υ₀ ≫ unify τ₁ υ₁
unify (Float d)  (Float e)   = unifyUnit d e
unify (M α)      (M β)       = onTop⋆ $ \ γ d → case
    (γ ≡ α, γ ≡ β, d) of
    (True,   True,   _      ) → restore
    (True,   False,  HOLE   ) → replace [(α, ⋆, DEFN (M β))]
    (False,  True,   HOLE   ) → replace [(β, ⋆, DEFN (M α))]
    (True,   False,  DEFN τ) → unify (M β) τ        ≫ restore
    (False,  True,   DEFN τ) → unify (M α) τ        ≫ restore
    (False,  False,  _      ) → unify (M α) (M β) ≫ restore
unify (M α) τ                 = startSolve α τ
unify τ       (M α)           = startSolve α τ
unify _       _               = throwError "Rigid-rigid mismatch"
```

When starting to solve a flex-rigid constraint, one has to be careful not to accidentally lose polymorphism, as explained in Subsection 3.2.1 (page 40). The syntactic occurs check performed by solve is not quite right, because the richer equational theory of abelian groups may exhibit apparent dependency when there is in fact none. Thus startSolve replaces units in the rigid type with fresh variables, solves the flex-rigid constraint first, then unifies the units.

```
startSolve :: Name Type → Type → Contextual ()
startSolve α τ = do  (ρ, xs) ← rigidHull τ
                     solve α (constraintsToSuffix xs) ρ
                     solveConstraints xs
```

The rigidHull operation computes the 'hull' of a type of kind ⋆, replacing unit subexpressions with fresh variables. Along with the hull, it returns the constraints between the fresh variables and the units they replaced.

```
rigidHull :: Type → Contextual (Type, [(Name Type, Type)])
rigidHull (M $a$)      = return (M $a$, [])
rigidHull (V $a$)      = return (V $a$, [])
rigidHull ($\tau$ ⇾ $\upsilon$)  = do  ($\tau'$, $xs$) ← rigidHull $\tau$
                          ($\upsilon'$, $ys$) ← rigidHull $\upsilon$
                          return ($\tau'$ ⇾ $\upsilon'$, $xs \odot ys$)
rigidHull (Float $d$) = do  $\beta$ ← fresh (s2n "beta")
                          return (Float (M $\beta$), [($\beta$, $d$)])
```

A list of constraints can be turned into the appropriate context suffix by discarding the types and adding unit declarations for the metavariables:

```
constraintsToSuffix :: [(Name Type, Type)] → Suffix
constraintsToSuffix = map (\ ($\alpha$, _) → ($\alpha$, $\mathcal{U}$, HOLE))
```

Or they can be solved by repeatedly invoking unifyUnit:

```
solveConstraints :: [(Name Type, Type)] → Contextual ()
solveConstraints = mapM_ (uncurry $ unifyUnit ∘ M)
```

The implementation of solve is almost identical to the version in Appendix A.

```
solve :: Name Type → Suffix → Type → Contextual ()
solve $\alpha$ $\Xi$ $\tau$ = onTop$^\star$ $
  \ $\gamma$ $d$ → case
    ($\gamma \equiv \alpha$, $\gamma \in$ fmv $\tau$, $d$       ) of
    (_,      _,           DEFN $\upsilon$) → modify (<>< $\Xi$)
                                  ≫ unify (subst $\gamma$ $\upsilon$ (M $\alpha$)) (subst $\gamma$ $\upsilon$ $\tau$)
                                  ≫ restore
    (True,   True,        HOLE   ) → throwError "Occurrence detected!"
    (True,   False,       HOLE   ) → replace $ $\Xi \odot$ [($\alpha$, $\star$, DEFN $\tau$)]
    (False,  True,        HOLE   ) → solve $\alpha$ (($\gamma$, $\star$, HOLE) : $\Xi$) $\tau$
                                  ≫ replace []
    (False,  False,       HOLE   ) → solve $\alpha$ $\Xi$ $\tau$
                                  ≫ restore
```

Appendix C

Reference implementation of Miller pattern unification

Having specified the pattern unification algorithm in Chapter 4, I now implement it in Haskell. The code is organised along similar lines to the previous two appendices, although the details differ substantially. First I describe the representation of object language terms (Section C.1) and the domain-specific language in which I will implement the algorithm (Section C.2). I then give implementations of type and equality checking (Section C.3), and unification (Section C.4).

C.1 Representation of terms

First I define terms and machinery for working with them (including evaluation and occurrence checking), based on the description in Subsection 4.1.1 (page 53).

Object language terms are represented using the data type Tm. The Binders Unbound library of Weirich et al. (2011b) defines the Bind type constructor and gives a cheap locally nameless representation with operations including α-equivalence and substitution for first-order datatypes containing terms. I use a single constructor for all the canonical forms (that do not involve binding) so as to factor out common patterns in the typechecker.

```
data Tm where
   λ    :: Bind Nom Tm → Tm
   ·    :: Head → Bwd Elim → Tm
   C    :: Can Tm → Tm
   Π, Σ :: Type → Bind Nom Type → Tm

type Nom  = Name Tm
```

```
data Can t = Set | Type | Pair t t | Bool | Tt | Ff | ℕ | Ze | Su t
data Head  = V Nom Twin | M Nom
data Twin  = Only | TwinL | TwinR
data Elim  = A Tm | Hd | Tl | If (Bind Nom Type) Tm Tm
type Type  = Tm
```

The non-binding canonical forms Can induce a Foldable functor (which can be derived automatically by GHC). Annoyingly, Elim cannot be made a functor in the same way, because Bind Nom is not a functor on $*$ but only on the subcategory induced by Alpha. However, the action on morphisms can be defined thus:

```
mapElim :: (Tm → Tm) → Elim → Elim
mapElim f (A a)       = A (f a)
mapElim _ Hd          = Hd
mapElim _ Tl          = Tl
mapElim f (If T s t) = If (bind x (f T′)) (f s) (f t)
  where (x, T′) = unsafeUnbind T

foldMapElim :: Monoid m ⇒ (Tm → m) → Elim → m
foldMapElim f (A a)       = f a
foldMapElim _ Hd          = mempty
foldMapElim _ Tl          = mempty
foldMapElim f (If T s t) = f T′ ⊙ f s ⊙ f t
  where (_, T′) = unsafeUnbind T
```

Despite the single-constructor representation of canonical forms, it is often neater to write code as if Tm had a data constructor for each canonical constructor of the object language. This is possible thanks to pattern synonyms (Aitken and Reppy, 1992) as implemented by the Strathclyde Haskell Enhancement (McBride, 2010b). Pattern synonyms are abbreviations that can be used 'on the left' (in patterns) as well as 'on the right' (in expressions).

```
pattern Type     = C Type
pattern Set      = C Set
pattern pair s t = C (Pair s t)
pattern 𝔹        = C Bool
pattern tt       = C Tt
pattern ff       = C Ff
pattern ℕ        = C ℕ
pattern ze       = C Ze
pattern su n     = C (Su n)
```

Free variables

Rather than definining functions to determine the free metavariables and variables of terms directly, I use a typeclass to make them available on the whole syntax.

```
data Flavour = Vars | RigVars | Metas
class Occurs $t$ where
  free :: Flavour $\to t \to$ Set Nom

fv, fv$^{\mathsf{rig}}$, fmv :: Occurs $t \Rightarrow t \to$ Set Nom
fv    = free Vars
fv$^{\mathsf{rig}}$ = free RigVars
fmv = free Metas

instance Occurs Tm where
  free $l$ ($\lambda$ $b$)       = free $l$ $b$
  free $l$ (C $c$)       = free $l$ $c$
  free $l$ ($\Pi$ $S$ $T$) = free $l$ $S \cup$ free $l$ $T$
  free $l$ ($\Sigma$ $S$ $T$) = free $l$ $S \cup$ free $l$ $T$
  free RigVars (V $x$ _ $\cdot$ $e$) = $\{x\} \cup$ free RigVars $e$
  free RigVars (M _ $\cdot$ _)   = $\emptyset$
  free $l$           ($h \cdot e$)       = free $l$ $h \cup$ free $l$ $e$
instance Occurs $t \Rightarrow$ Occurs (Can $t$) where
  free $l$ (Pair $s$ $t$) = free $l$ $s \cup$ free $l$ $t$
  free $l$ (Su $n$)       = free $l$ $n$
  free $l$ _             = $\emptyset$
instance Occurs Head where
  free Vars      (M _)   = $\emptyset$
  free RigVars (M _)   = $\emptyset$
  free Metas   (M $\alpha$)   = $\{\alpha\}$
  free Vars      (V $x$ _) = $\{x\}$
  free RigVars (V $x$ _) = $\{x\}$
  free Metas   (V _ _) = $\emptyset$
instance Occurs Elim where
  free $l$ (A $a$)        = free $l$ $a$
  free $l$ Hd           = $\emptyset$
  free $l$ Tl            = $\emptyset$
  free $l$ (If $T$ $s$ $t$) = free $l$ $T \cup$ free $l$ $s \cup$ free $l$ $t$
```

Evaluation by hereditary substitution

Substitutions are implemented as finite maps from names to terms; as a technical convenience there is no distinction between substitution and metasubstitution.

```
type Subs = [(Nom, Tm)]


(∘) :: Subs → Subs → Subs
δ′ ∘ δ = unionBy ((≡) `on` fst) δ′ (substs δ′ δ)
```

The evaluator is an implementation of hereditary substitution defined in Figure 4.2 (page 54): it proceeds structurally through terms, replacing variables with their values and eliminating redexes using the (%%) operator defined below.

```
eval :: Subs → Tm → Tm
eval g (λ b)      = λ (evalUnder g b)
eval g (h · e)    = foldl ( %% ) (evalHead g h) (fmap (mapElim (eval g)) e)
eval g (C c)      = C (fmap (eval g) c)
eval g (Π S T)    = Π (eval g S) (evalUnder g T)
eval g (Σ S T)    = Σ (eval g S) (evalUnder g T)

evalHead :: Subs → Head → Tm
evalHead g (V x _) | Just t ← lookup x g = t
evalHead g (M α)   | Just t ← lookup α g = t
evalHead g h                             = h · •

evalUnder :: Subs → Bind Nom Tm → Bind Nom Tm
evalUnder g b = bind x (eval g t)
  where (x, t) = unsafeUnbind b
```

The (%%) operator reduces a redex (a term with an eliminator) to normal form: this re-invokes hereditary substitution when a λ-abstraction meets an application.

```
( %% ) :: Tm → Elim → Tm
λ b        %% (A a)   = eval [(x, a)] t  where (x, t) = unsafeUnbind b
pair x _   %% Hd      = x
pair _ y   %% Tl      = y
tt         %% If _ t _ = t
ff         %% If _ _ f = f
h · e      %% z       = h · (e :< z)
t          %% a       = error "bad elimination"
```

I define some convenient abbreviations: ($$) for applying a function to an argument, ($*$) for applying a function to a telescope of arguments, $\cdot\{\cdot\}$ for substituting out a single binding and hd and tl for the projections from Σ-types.

```
( $$ ) :: Tm → Tm → Tm
f $$ a = f %% A a
( $*$ ) :: Tm → Bwd (Nom, Type) → Tm
f $*$ Γ = foldl ( $$ ) f (fmap (var . fst) Γ)
·{·} :: Bind Nom Tm → Tm → Tm
f{s} = λ f $$ s
hd, tl :: Tm → Tm
hd = ( %% Hd)
tl  = ( %% Tl)
```

C.2 Problems and contexts

I will now define unification problems, metacontexts and operations for working on them in the Contextual monad. The notions of metacontext and context in use were given in Subsection 4.1.2 (page 55), and the monadic approach develops that of the previous appendices. Metacontext entries now consist of metavariables, as before, or problems, which carry a status bit used to record whether they have been solve as far as possible given their current type (see Subsection C.4.6). Problems are equations under universally quantified parameters, and parameters may include twins.

```
data Decl v    = HOLE | DEFN v
data Entry     = E (Name Tm) (Type, Decl Tm) | Q Status Problem
data Status    = Blocked | Active
data Param     = P Type | Type‡Type
type Params    = Bwd (Nom, Param)
data Equation = (Tm : Type) ≈ (Tm : Type)
data Problem  = Unify Equation | All Param (Bind Nom Problem)
```

The sym function swaps the two sides of an equation:

```
sym :: Equation → Equation
sym ((s : S) ≈ (t : T)) = (t : T) ≈ (s : S)
```

The metacontext is represented as a list zipper: a pair of lists representing the entries before and after the cursor. Entries after the cursor may include substitutions, being propagated lazily.

```
type ContextL  = Bwd Entry
type ContextR  = [Either Subs Entry]
type Context   = (ContextL, ContextR)
```

The Contextual monad stores the current context and parameters, generates fresh names when required for going under binders, and handles exceptions.

```
newtype Contextual a = Contextual
  (ReaderT Params (StateT Context (FreshMT (ErrorT String Identity))) a)
```

Reading and modifying state

I define versions of the usual state-manipulating get, modify and put operations that act on the left or right part of the context (before or after the cursor).

```
getL :: Contextual ContextL
getL = gets fst

getR :: Contextual ContextR
getR = gets snd

modifyL :: (ContextL → ContextL) → Contextual ()
modifyL = modify ∘ first

modifyR :: (ContextR → ContextR) → Contextual ()
modifyR = modify ∘ second

putL :: ContextL → Contextual ()
putL = modifyL ∘ const

putR :: ContextR → Contextual ()
putR = modifyR ∘ const
```

Here are operations to push to, or pop from, either side of the cursor, or move the cursor one entry to the left:

```
pushL :: Entry → Contextual ()
pushL e = modifyL (:<e)

pushR :: Either Subs Entry → Contextual ()
pushR e = modifyR (e:)
```

```
pushLs :: Traversable f ⇒ f Entry → Contextual ()
pushLs es = traverse pushL es ≫ return ()

popL :: Contextual Entry
popL = do  Θ ← getL
           case Θ of (Θ' :< e) → putL Θ' ≫ return e
                     •         → throwError "popL: out of context"

popR :: Contextual (Either Subs Entry)
popR = do  Θ ← getR
           case Θ of (x : Θ')  → putR Θ' ≫ return x
                     []        → throwError "popR: out of context"

goLeft :: Contextual ()
goLeft = popL ≫= pushR ∘ Right
```

Variable and metavariable lookup

The context of local parameters is tracked using the ReaderT monad transformer, so the local operation can be used to bring a parameter into scope, and the ask operation can be used to look up a variable.

```
inScope :: Nom → Param → Contextual a → Contextual a
inScope x p = local (:<(x, p))

lookupVar :: Nom → Twin → Contextual Type
lookupVar x w = help w =≪ ask
  where
    help Only  (Γ :< (y, P T))  | x ≡ y = return T
    help TwinL (Γ :< (y, S‡T))  | x ≡ y = return S
    help TwinR (Γ :< (y, S‡T))  | x ≡ y = return T
    help w     (Γ :< _)                 = help w Γ
    help _     • = throwError $ "lookupVar: missing " ++ show x
```

The type of a metavariable can be determined from its name by searching the metacontext. Only metavariables left of the cursor are in scope.

```
lookupMeta :: Nom → Contextual Type
lookupMeta x = look =≪ getL
  where
    look (Θ :< E y (T, _)) | x ≡ y = return T
    look (Θ :< _)                  = look Θ
    look • = error $ "lookupMeta: missing " ++ show x
```

C.3 Type and equality checking

Here I give a typechecker and definitional equality test for the type theory defined in Subsection 4.1.3 (page 56). With the Contextual monad operations, I define a bidirectional typechecker, based on a typed definitional equality test between $\beta\delta$-normal forms that produces an η-long standard form. The equalise $T\ s\ t$ function implements the judgment $\Theta \mid \Gamma \vdash T \ni s \equiv\!\!\lbrack\ u\ \rbrack\!\!\equiv t$, defined in Figure 4.4 (page 59), where u is the result.

```
equalise :: Type → Tm → Tm → Contextual Tm
equalise Type Set Set = return Set
equalise Type $S$    $T$    = equalise Set $S\ T$
equalise Set  $\mathbb{B}$    $\mathbb{B}$    = return $\mathbb{B}$
equalise $\mathbb{B}$    tt   tt   = return tt
equalise $\mathbb{B}$    ff   ff   = return ff
equalise Set  $(\Pi\ A\ B)\ (\Pi\ S\ T)$ = do
  $U$ ← equalise Set $A\ S$
  $\Pi\ U$ ⟨$⟩ bindsInScope $U\ B\ T$
             $(\backslash\ x\ B'\ T' \rightarrow$ equalise Set $B'\ T')$
equalise $(\Pi\ U\ V)\ f\ g$ =
  $\lambda$ ⟨$⟩ bindInScope $U\ V$
                $(\backslash\ x\ V' \rightarrow$ equalise $V'\ (f$ $$ var $x)\ (g$ $$ var $x))$
equalise Set $(\Sigma\ A\ B)\ (\Sigma\ S\ T)$ = do
  $U$ ← equalise Set $A\ S$
  $\Sigma\ U$ ⟨$⟩ bindsInScope $U\ B\ T$
             $(\backslash\ x\ B'\ T' \rightarrow$ equalise Set $B'\ T')$
equalise $(\Sigma\ U\ V)\ s\ t$ = do
  $u_0$ ← equalise $U$ (hd $s$) (hd $t$)
  $u_1$ ← equalise $(V\{u_0\})$ (tl $s$) (tl $t$)
  return (pair $u_0\ u_1$)
equalise $U\ (h \cdot e)\ (h' \cdot e')$ = do
  $(h'', e'', V)$ ← equaliseN $h\ e\ h'\ e'$
  equalise Type $U\ V$
  return $(h'' \cdot e'')$
```

Similarly, the equaliseN $h\ e\ h'\ e'$ function implements the equality judgment $\Theta \mid \Gamma \vdash h \cdot e \equiv\!\!\lbrack\ h'' \cdot e''\ \rbrack\!\!\equiv h' \cdot e' \in T$, defined in Figure 4.5 (page 60), where h'', e'' and T are the results.

```
equaliseN :: Head → Bwd Elim → Head → Bwd Elim →
                  Contextual (Head, Bwd Elim, Type)
equaliseN $h$ • $h'$ • | $h \equiv h'$          = $(h, \bullet, )$ ⟨\$⟩ infer $h$
equaliseN $h$ $(e$ :< A $s)$ $h'$ $(e'$ :< A $t)$ = do
  $(h'', e'', \Pi\ U\ V)$ ← equaliseN $h$ $e$ $h'$ $e'$
  $u$                 ← equalise $U$ $s$ $t$
  return $(h'', e''$ :< A $u$, $V\{u\})$
equaliseN $h$ $(e$ :< Hd$)$ $h'$ $(e'$ :< Hd$)$ = do
  $(h'', e'', \Sigma\ U\ V)$ ← equaliseN $h$ $e$ $h'$ $e'$
  return $(h'', e''$ :< Hd, $U)$
equaliseN $h$ $(e$ :< Tl$)$  $h'$ $(e'$ :< Tl$)$  = do
  $(h'', e'', \Sigma\ U\ V)$ ← equaliseN $h$ $e$ $h'$ $e'$
  return $(h'', e''$ :< Tl, $V\{h'' \cdot (e''$ :< Hd$)\})$
equaliseN $h$ $(e$ :< If $T$ $u$ $v)$ $h'$ $(e'$ :< If $T'$ $u'$ $v')$ = do
  $(h'', e'', \mathbb{B})$ ← equaliseN $h$ $e$ $h'$ $e'$
  $U''$        ← bindsInScope $\mathbb{B}$ $T$ $T'$ (\ $x$ $U$ $U'$ → equalise Type $U$ $U'$)
  $u''$        ← equalise $(U''\{\mathbf{tt}\})$ $u$ $u'$
  $v''$        ← equalise $(U''\{\mathbf{ff}\})$ $v$ $v'$
  return $(h'', e''$ :< If $U''$ $u''$ $v''$, $U''\{h'' \cdot e''\})$
```

The infer function looks up the type of a head, using lookupVar or lookupMeta from the previous section as appropriate.

```
infer :: Head → Contextual Type
infer (V $x$ $w$) = lookupVar $x$ $w$
infer (M $x$)   = lookupMeta $x$
```

The bindInScope and bindsInScope helper operations introduce a binding or two and call the continuation with a variable of the given type in scope.

```
bindInScope :: Type → Bind Nom Tm →
                 (Nom → Tm → Contextual Tm) →
                 Contextual (Bind Nom Tm)
bindInScope $T$ $b$ $f$ = do  $(x, b')$ ← unbind $b$
                          bind $x$ ⟨\$⟩ inScope $x$ (P $T$) $(f\ x\ b')$

bindsInScope :: Type → Bind Nom Tm → Bind Nom Tm →
                  (Nom → Tm → Tm → Contextual Tm) →
                  Contextual (Bind Nom Tm)
bindsInScope $T$ $a$ $b$ $f$ = do  Just $(x, a', \_, b')$ ← unbind2 $a$ $b$
                              bind $x$ ⟨\$⟩ inScope $x$ (P $T$) $(f\ x\ a'\ b')$
```

Equality checking can return a Boolean instead of throwing an error when the terms are not equal. Since typing is the diagonal of equality, it is easy to define a typechecking function as well.

```
equal :: Type → Tm → Tm → Contextual Bool
equal T s t = (equalise T s t ≫ return True) ⊕ (return False)

typecheck :: Type → Tm → Contextual Bool
typecheck T t = equal T t t
```

Finally, a convenience function that tests if a heterogeneous equation is reflexive, by checking that the types are equal and the terms are equal.

```
isReflexive :: Equation → Contextual Bool
isReflexive ((s : S) ≈ (t : T)) = optional (equalise Type S T) ≫=
                                      maybe (return False) (\ U → equal U s t)
```

C.4 Unification

With the preliminaries out of the way, I can now present the pattern unification algorithm as specified in Section 4.2 (page 67). I begin with utilities for working with metavariables and problems, then give the implementations of inversion, intersection, pruning, metavariable simplification and problem simplification. Finally, I show how the order of constraint solving is managed.

Making and filling holes

A telescope is a list of binding names and their types. Any type can be viewed as consisting of a Π-bound telescope followed by a non-Π-type.

```
type Telescope = Bwd (Nom, Type)

telescope :: Type → Contextual (Telescope, Type)
telescope (Π S T) = do  (x, T′) ← unbind T
                        (Δ, U) ← telescope T′
                        return ((• :< (x, S)) ⊙ Δ, U)
telescope T       = return (•, T)
```

The hole control operator creates a metavariable of the given type (under a telescope of parameters), and calls the continuation with the metavariable in scope. Finally, it moves the cursor back to the left of the metavariable, so it will be examined again in case further progress can be made on it. The continuation must not move the cursor.

```
hole :: Telescope → Type → (Tm → Contextual $a$) → Contextual $a$
hole $\Gamma$ $T$ $f$ = do  $\alpha$ ← fresh (s2n "alpha")
                 pushL \$ E $\alpha$ ($\Pi\Gamma.\ T$, HOLE)
                 $r$ ← $f$ (meta $\alpha$ \$*\$ $\Gamma$)
                 goLeft
                 return $r$
```

Once a solution for a metavariable is found, the define function adds a definition to the context. (The declaration of the metavariable should already have been removed.) This also propagates a substitution that replaces the metavariable with its value.

```
define :: Telescope → Nom → Type → Tm → Contextual ()
define $\Gamma$ $\alpha$ $S$ $v$ = do  pushR \$ Left [($\alpha$, $t$)]
                    pushR \$ Right \$ E $\alpha$ ($T$, DEFN $t$)
  where $T$ = $\Pi\Gamma.\ S$
        $t$ = $\lambda\Gamma.\ v$
```

Postponing problems

When a problem cannot be solved immediately, it can be postponed by adding it to the metacontext. The postpone functions wraps a problem in the current context (as returned by ask) and stores it in the metacontext with the given status. The active function postpones a problem on which progress can be made, while the block function postpones a problem that cannot make progress until its type becomes more informative, as discussed in Subsection C.4.6.

```
postpone :: Status → Problem → Contextual ()
postpone $s$ $p$ = pushR ∘ Right ∘ Q $s$ ∘ wrapProb $p$ =≪ ask
  where
    wrapProb :: Problem → Params → Problem
    wrapProb = foldr (\ ($x$, $e$) $p$ → All $e$ (bind $x$ $p$))
active, block :: Problem → Contextual ()
active = postpone Active
block  = postpone Blocked
```

A useful combinator

The following combinator executes its first argument, and if this returns False then it also executes its second argument.

```
(⊚) :: Monad m ⇒ m Bool → m () → m ()
a ⊚ b = do  x ← a
            unless x b
```

C.4.1 Inversion

A flexible unification problem is one where one side is an applied metavariable and the other is an arbitrary term. The algorithm moves left in the context, accumulating a list of metavariables Ξ that the term depends on, to construct the necessary dependency-respecting permutation. Once the target metavariable is reached, it can attempt to find a solution by inversion. This implements step (4.16) in Figure 4.15 (page 80), as described in Subsection 4.2.1 (page 67).

```
flexTerm :: [Entry] → Equation → Contextual ()
flexTerm Ξ q@(M α · _ ≈ _) = do
  Γ ← fmap snd ⟨$⟩ ask
  popL ≫= \ e → case e of
    E β (T, HOLE)
      | α ≡ β ∧ α ∈ fmv Ξ → do  pushLs (e : _Xi)
                                block (Unify q)
      | α ≡ β             → do  pushLs Ξ
                                tryInvert q T
                                  ⊚ (block (Unify q) ≫ pushL e)
      | β ∈ fmv (Γ, Ξ, q) → flexTerm (e : Ξ) q
    _                     → do  pushR (Right e)
                                flexTerm Ξ q
```

A flex-flex unification problem is one where both sides are applied metavariables. As in the general case above, the algorithm proceeds leftwards through the context, looking for one of the metavariables so it can try to solve one with the other. If it reaches one of the metavariables and cannot solve for the metavariable by inversion, it continues (using flexTerm), which ensures it will terminate after trying to solve for both. For example, consider the case $\alpha\,\overline{t_i}^{\,i} \approx \beta\,\overline{x_j}^{\,j}$ where only $\overline{x_j}^{\,j}$ is a list of variables. If it reaches α first then it might get stuck even if it could potentially solve for β. This would be correct if order were important in the metacontext, for example when implementing let-generalisation as discussed in Chapter 2. Here it is not, so the algorithm can simply pick up α and carry on.

```
flexFlex :: [Entry] → Equation → Contextual ()
flexFlex $\Xi$ $q$@(M $\alpha \cdot ds \approx$ M $\beta \cdot es$) = do
  $\Gamma$ ← fmap snd ⟨$⟩ ask
  popL ≫= \ $e$ → case $e$ of
    E $\gamma$ ($T$, HOLE)
      | $\gamma \in [\alpha, \beta] \wedge \gamma \in$ fmv ($\Xi$) → do pushLs ($e : \Xi$)
                                              block (Unify $q$)
      | $\gamma \equiv \alpha$              → do pushLs $\Xi$
                                   tryInvert $q$ $T$ ⊘ flexTerm [$e$] (sym $q$)
      | $\gamma \equiv \beta$              → do pushLs $\Xi$
                                   tryInvert (sym $q$) $T$ ⊘ flexTerm [$e$] $q$
      | $\gamma \in$ fmv ($\Gamma, \Xi, q$) → flexFlex ($e : \Xi$) $q$
    _                          → do pushR (Right $e$)
                                   flexFlex $\Xi$ $q$
```

Given a flexible equation whose head metavariable has just been found in the context, the tryInvert control operator calls invert to seek a solution to the equation. If it finds one, it defines the metavariable.

```
tryInvert :: Equation → Type → Contextual Bool
tryInvert $q$@(M $\alpha \cdot e \approx s$) $T$ = invert $\alpha$ $T$ $e$ $s$ ≫= \ $m$ → case $m$ of
  Nothing → return False
  Just $v$   → do active (Unify $q$)
                 define • $\alpha$ $T$ $v$
                 return True
```

Given a metavariable α of type T, spine e and term t, invert attempts to find a value for α that solves the equation $\alpha \cdot e \approx t$. It will throw an error if the problem is unsolvable due to an impossible occurrence.

```
invert :: Nom → Type → Bwd Elim → Tm → Contextual (Maybe Tm)
invert $\alpha$ $T$ $e$ $t$ | occurCheck True $\alpha$ $t$ = throwError "occur check"
              | $\alpha \notin$ fmv $t$, Just $xs$ ← toVars $e$, linearOn $t$ $xs$ = do
                $b$ ← local (const •) (typecheck $T$ ($\lambda xs.\ t$))
                return $ if $b$ then Just ($\lambda xs.\ t$) else Nothing
              | otherwise = return Nothing
```

Note that the solution $\lambda xs.\, t$ is typechecked under no parameters, so typechecking will fail if an out-of-scope variable is used.

The occur check, used to tell if an equation is definitely unsolvable, looks for occurrences of a metavariable inside a term. In a strong rigid context (where the first argument is True), any occurrence is fatal. In a weak rigid context (where it is False), the evaluation context of the metavariable must be a list of variables.

```
occurCheck :: Bool → Nom → Tm → Bool
occurCheck w α (λ b)       = occurCheck w α t
                                 where (_, t) = unsafeUnbind b
occurCheck w α (V _ _ · e) = getAny $ foldMap
                                (foldMapElim (Any ∘ occurCheck False α)) e
occurCheck w α (M β · e)   = α ≡ β ∧ (w ∨ isJust (toVars e))
occurCheck w α (C c)       = getAny $ foldMap (Any ∘ occurCheck w α) c
occurCheck w α (Π S T)     = occurCheck w α S ∨ occurCheck w α T′
                                 where (_, T′) = unsafeUnbind T
occurCheck w α (Σ S T) = occurCheck w α S ∨ occurCheck w α T′
                                 where (_, T′) = unsafeUnbind T
```

Here toVars tries to convert a spine to a list of variables, and linearOn determines if a list of variables is linear on the free variables of a term. Since it is enough for a term in a spine to be η-convertible to a variable, the etaContract function implements η-contraction for terms.

```
linearOn :: Tm → Bwd Nom → Bool
linearOn _ •           = True
linearOn t (as :< a) = ¬ (a ∈ fv t ∧ a ∈ as) ∧ linearOn t as
```

```
etaContract :: Tm → Tm
etaContract (λ b) = case etaContract t of
  x · (e :< A (V y′ _ · •)) | y ≡ y′, ¬ (y ∈ fv e) → x · e
  t′                                            → λy. t′
   where (y, t) = unsafeUnbind b
etaContract (x · as)     = x · (fmap (mapElim etaContract) as)
etaContract (pair s t) = case (etaContract s, etaContract t) of
  (x · (as :< Hd), y · (bs :< Tl)) | x ≡ y, as ≡ bs → x · as
  (s′, t′)                                           → pair s′ t′
etaContract (C c)        = C (fmap etaContract c)
```

```
toVar :: Tm → Maybe Nom
toVar $v$ = case etaContract $v$ of V $x$ _ · • → Just $x$
                                   _           → Nothing

toVars :: Traversable $f$ ⇒ $f$ Elim → Maybe ($f$ Nom)
toVars = traverse (unA >=> toVar)
  where unA (A $t$) = Just $t$
        unA _       = Nothing
```

C.4.2 Intersection

When a flex-flex equation has the same metavariable on both sides, i.e. it has the form $\alpha\,\overline{x_i}^{\,i} \approx \alpha\,\overline{y_i}^{\,i}$ where $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$ are both lists of variables, the equation can be solved by restricting α to the arguments on which $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$ agree (i.e. creating a new metavariable β and using it to solve α). This implements step (4.18) in Figure 4.15 (page 80), as described in Subsection 4.2.2 (page 70).

The flexFlexSame function extracts the type of α as a telescope and calls intersect to generate a restricted telescope. If this succeeds, it calls instantiate to create a new metavariable and solve the old one. Otherwise, it leaves the equation in the context. Twin annotations can be ignored here here because any twins will have definitionally equal types anyway.

```
flexFlexSame :: Equation → Contextual ()
flexFlexSame $q$@(M $\alpha$ · $e$ ≈ M $\alpha$ · $e'$) = do
  ($\Delta$, $T$) ← telescope =<< lookupMeta $\alpha$
  case intersect $\Delta$ $e$ $e'$ of
    Just $\Delta'$ | fv $T \subset$ vars $\Delta'$ → instantiate ($\alpha, \Pi\Delta'.\, T$, \ $\beta \to \lambda\Delta.\, \beta$ $*$ $\Delta$)
    _                          → block (Unify $q$)
```

Given a telescope and the two evaluation contexts, intersect checks the evaluation contexts are lists of variables and produces the telescope on which they agree.

```
intersect :: Telescope → Bwd Elim → Bwd Elim → Maybe Telescope
intersect •              •            •             = return •
intersect ($\Delta$ :< ($z, S$)) ($e$ :< A $s$) ($e'$ :< A $t$) = do
  $\Delta'$ ← intersect $\Delta$ $e$ $e'$
  $x$  ← toVar $s$
  $y$  ← toVar $t$
  if $x \equiv y$ then return ($\Delta'$ :< ($z, S$)) else return $\Delta'$
intersect _ _ _ = Nothing
```

C.4.3 Pruning

Given a flex-rigid or flex-flex equation, it might be possible to make some progress by pruning the metavariables contained within it, as described in Subsection 4.2.3 (page 71). The tryPrune function calls pruneTm: if it learns anything from pruning, it leaves the current problem active and instantiates the pruned metavariable.

```
tryPrune :: Equation → Contextual Bool
tryPrune $q$@(M $\alpha \cdot e \approx t$) = pruneTm (fv $e$) $t$ ≫= \ $u$ → case $u$ of
   $d$ : _      → active (Unify $q$) ≫ instantiate $d$ ≫ return True
   [ ]          → return False
```

Pruning a term requires traversing it looking for occurrences of forbidden variables. If any occur rigidly, the corresponding constraint is impossible. If a metavariable is encountered, it cannot depend on any arguments that contain rigid occurrences of forbidden variables, so it can be replaced by a fresh metavariable of restricted type. The pruneTm function generates a list of triples (β, U, f) where β is a metavariable, U is a type for a new metavariable γ and $f\ \gamma$ is a solution for β. It maintains the invariant that U and $f\ \gamma$ depend only on metavariables defined prior to β in the context.

```
pruneTm :: Set Nom → Tm → Contextual [Instantiation]
pruneTm $\mathcal{V}$ ($\Pi\ S\ T$)   = (++) ⟨$⟩ pruneTm $\mathcal{V}$ $S$ ⟨*⟩ pruneUnder $\mathcal{V}$ $T$
pruneTm $\mathcal{V}$ ($\Sigma\ S\ T$)   = (++) ⟨$⟩ pruneTm $\mathcal{V}$ $S$ ⟨*⟩ pruneUnder $\mathcal{V}$ $T$
pruneTm $\mathcal{V}$ (pair $s$ $t$) = (++) ⟨$⟩ pruneTm $\mathcal{V}$ $s$ ⟨*⟩ pruneTm $\mathcal{V}$ $t$
pruneTm $\mathcal{V}$ ($\lambda\ b$)       = pruneUnder $\mathcal{V}$ $b$
pruneTm $\mathcal{V}$ (M $\beta \cdot e$)   = pruneMeta $\mathcal{V}$ $\beta$ $e$
pruneTm $\mathcal{V}$ (C _)       = return [ ]
pruneTm $\mathcal{V}$ (V $z$ _ $\cdot$ $e$) | $z \in \mathcal{V}$      = pruneElims $\mathcal{V}$ $e$
                        | otherwise = throwError "pruning error"

pruneUnder :: Set Nom → Bind Nom Tm → Contextual [Instantiation]
pruneUnder $\mathcal{V}$ $b$ = do  $(x, t)$ ← unbind $b$
                        pruneTm ($\mathcal{V} \cup \{x\}$) $t$

pruneElims :: Set Nom → Bwd Elim → Contextual [Instantiation]
pruneElims $\mathcal{V}$ $e$ = fold ⟨$⟩ traverse pruneElim $e$
  where
    pruneElim (A $a$)       = pruneTm $\mathcal{V}$ $a$
    pruneElim (If $T$ $s$ $t$) = (++) ⟨$⟩ ((++) ⟨$⟩ pruneTm $\mathcal{V}$ $s$ ⟨*⟩ pruneTm $\mathcal{V}$ $t$)
                                        ⟨*⟩ pruneUnder $\mathcal{V}$ $T$
    pruneElim _            = return [ ]
```

Once a metavariable has been found, pruneMeta unfolds its type as a telescope $\Pi\mathit{\Delta}.\ T$, and calls prune with the telescope and list of arguments. If the telescope is successfully pruned ($\mathit{\Delta}'$ is not the same as $\mathit{\Delta}$) and the free variables of T remain in the telescope, then an instantiation of the metavariable is generated.

```
pruneMeta :: Set Nom → Nom → Bwd Elim → Contextual [Instantiation]
pruneMeta $\mathcal{V}$ $\beta$ $e$ = do
  ($\mathit{\Delta}$, $T$) ← telescope =≪ lookupMeta $\beta$
  case prune $\mathcal{V}$ $\mathit{\Delta}$ $e$ of
    Just $\mathit{\Delta}'$ | $\mathit{\Delta}' \not\equiv \mathit{\Delta}$, fv $T \subset$ vars $\mathit{\Delta}'$
                → return [($\beta$, $\Pi\mathit{\Delta}'.\ T$, \ beta' → $\lambda\mathit{\Delta}$. beta' $*$ $\mathit{\Delta}'$)]
    _           → return []
```

The prune function generates a restricted telescope, removing arguments that contain a rigid occurrence of a forbidden variable. This may fail if it is not clear which arguments must be removed.

```
prune :: Set Nom → Telescope → Bwd Elim → Maybe Telescope
prune $\mathcal{V}$ •                •          = Just •
prune $\mathcal{V}$ ($\mathit{\Delta}$ :< ($x$, $S$)) ($e$ :< A $s$) = do
  $\mathit{\Delta}'$ ← prune $\mathcal{V}$ $\mathit{\Delta}$ $e$
  case toVar $s$ of
    Just $y$ | $y \in \mathcal{V}$, fv $S \subset$ vars $\mathit{\Delta}'$ → Just ($\mathit{\Delta}'$ :< ($x$, $S$))
    _      | fv$^{\mathsf{rig}}$ $s \not\subset \mathcal{V}$             → Just $\mathit{\Delta}'$
           | otherwise                   → Nothing
prune _ _ _ = Nothing
```

A metavariable α can be instantiated to a more specific type by moving left through the context until it is found, creating a new metavariable and solving for α. The type must not depend on any metavariables defined after α.

```
type Instantiation = (Nom, Type, Tm → Tm)


instantiate :: Instantiation → Contextual ()
instantiate $d$@($\alpha$, $T$, $f$) = popL ≫= \ $e$ → case $e$ of
  E $\beta$ ($U$, HOLE) | $\alpha \equiv \beta$ → hole • $T$ (\ $t$ → define • $\beta$ $U$ ($f$ $t$))
  _                        → do pushR (Right $e$)
                                instantiate $d$
```

C.4.4 Metavariable simplification

Given the name and type of a metavariable, lower attempts to simplify it by removing Σ-types, according to the metavariable simplification steps (4.21) and (4.22) in Figure 4.15 (page 80), as described in Subsection 4.2.4 (page 74).

```
lower :: Telescope → Nom → Type → Contextual Bool
lower $\Phi$ $\alpha$ ($\Sigma$ $S$ $T$) = hole $\Phi$ $S$ \$ \ $s$ →
                          hole $\Phi$ ($T\{s\}$) \$ \ $t$ →
                          define $\Phi$ $\alpha$ ($\Sigma$ $S$ $T$) (pair $s$ $t$) ≫
                          return True
lower $\Phi$ $\alpha$ ($\Pi$ $S$ $T$) = do  $x$ ← fresh (s2n "x")
                             splitSig • $x$ $S$ ≫= maybe
                               (lower ($\Phi$ :< $(x, S)$) $\alpha$ ($T\{$var $x\}$))
                               (\ $(y, A, z, B, s, (u, v))$ →
                                 hole $\Phi$ ($\Pi y{:}A.\,\Pi z{:}B.\ T\{s\}$) \$ \ $w$ →
                                 define $\Phi$ $\alpha$ ($\Pi$ $S$ $T$) ($\lambda x.\ w$ \$\$ $u$ \$\$ $v$) ≫
                                 return True)
lower $\Phi$ $\alpha$ $T$ = return False
```

Lowering a metavariable needs to split Σ-types (possibly underneath a bunch of parameters) into their components. For example, $y : \Pi x : X.\, \Sigma z : S.\ T$ splits into $y_0 : \Pi x : X.\ S$ and $y_1 : \Pi x : X.\ T\{y_0\, x\}$. Given the name of a variable and its type, splitSig attempts to split it, returning fresh variables for the two components of the Σ-type, an inhabitant of the original type in terms of the new variables and inhabitants of the new types by projecting the original variable.

```
splitSig :: Telescope → Nom → Type →
              Contextual (Maybe (Nom, Type, Nom, Type, Tm, (Tm, Tm)))
splitSig $\Phi$ $x$ ($\Sigma$ $S$ $T$) = do  $y$ ← fresh (s2n "y")
                               $z$ ← fresh (s2n "z")
                               return \$ Just ($y, \Pi\Phi.\ S, z, \Pi\Phi.$ ($T\{$var $y$ \$*\$ $\Phi\}$),
                                        $\lambda\Phi.$ pair (var $y$ \$*\$ $\Phi$) (var $z$ \$*\$ $\Phi$),
                                        ($\lambda\Phi.$ var $x$ \$*\$ $\Phi$ %% Hd,
                                          $\lambda\Phi.$ var $x$ \$*\$ $\Phi$ %% Tl))
splitSig $\Phi$ $x$ ($\Pi$ $A$ $B$) = do  $a$ ← fresh (s2n "a")
                               splitSig ($\Phi$ :< $(a, A)$) $x$ ($B\{$var $a\}$)
splitSig _ _ _ = return Nothing
```

C.4.5 Problem simplification and unification

Given a problem, the solver simplifies it according to the rules in Figure 4.14 (page 79), introduces parameters and calls unify defined below if it is not already reflexive. In particular, problem simplification removes Σ-types from parameters, potentially eliminating projections, and replaces twins whose types are definitionally equal with a single parameter. This implements the steps described in Subsection 4.2.5 (page 75).

```
solver :: Problem → Contextual ()
solver (Unify $q$) = do  $b$ ← isReflexive $q$
                         unless $b$ (unify $q$)
solver (All $p$ $b$) = do
  $(x, q)$ ← unbind $b$
  case $p$ of
    _ | $x \notin$ fv $q$ → active $q$
    P $S$  → splitSig $\bullet$ $x$ $S$ $\ggg$= \ $m$ → case $m$ of
      Just $(y, A, z, B, s, \_)$ → solver ($\forall y\!:\!A.\,\forall z\!:\!B.$ subst $x$ $s$ $q$)
      Nothing                    → inScope $x$ (P $S$) \$ solver $q$
    $S \ddagger T$ → equal Set $S$ $T$ $\ggg$= \ $c$ →
      if $c$ then solver ($\forall x\!:\!S.$ subst $x$ (var $x$) $q$)
             else  inScope $x$ $(S \ddagger T)$ \$ solver $q$
```

The unify function performs a single unification step: η-expanding elements of Π or Σ types via the problem simplification steps (4.2) and (4.3) in Figure 4.14 (page 79), or invoking an auxiliary function in order to make progress.

```
unify :: Equation → Contextual ()
unify $((f : \Pi\ A\ B) \approx (g : \Pi\ S\ T))$ = do
  $x$ ← fresh (s2n "x")
  active \$ $\forall \hat{x}\!:\!A \ddagger S.\ (f \,\$\$\, \acute{x} : B\{\acute{x}\}) \approx (g \,\$\$\, \grave{x} : T\{\grave{x}\})$
unify $((t : \Sigma\ A\ B) \approx (w : \Sigma\ C\ D))$ = do
  active \$ (hd $t$ : $A$) $\approx$ (hd $w$ : $C$)
  active \$ (tl $t$ : $B\{$hd $t\}$) $\approx$ (tl $w$ : $D\{$hd $w\}$)
unify $q$@(M $\alpha \cdot e \approx$ M $\beta \cdot e'$)
  | $\alpha \equiv \beta$ = tryPrune $q$ $\varovee$ tryPrune (sym $q$) $\varovee$ flexFlexSame $q$
unify $q$@(M $\alpha \cdot e \approx$ M $\beta \cdot e'$) = tryPrune $q$ $\varovee$ tryPrune (sym $q$) $\varovee$ flexFlex [] $q$
unify $q$@(M $\alpha \cdot e \approx t$)           = tryPrune $q$ $\varovee$ flexTerm [] $q$
unify $q$@($t \approx$ M $\alpha \cdot e$)           = tryPrune (sym $q$) $\varovee$ flexTerm [] (sym $q$)
unify $q$                           = rigidRigid $q$
```

A rigid-rigid equation (between two non-metavariable terms) can either be decomposed into simpler equations or it is impossible to solve. For example, $\Pi x : A.\, B \approx \Pi x : S.\, T$ splits into $A \approx S, B \approx T$, but $\Pi x : A.\, B \approx \Sigma x : S.\, T$ cannot be solved. The rigidRigid function implements steps (4.4)–(4.7) from Figure 4.14 (page 79), as described in Subsection 4.2.5 (page 75). Both unify and rigidRigid will be called only when the equation is not already reflexive.

```
rigidRigid :: Equation → Contextual ()
rigidRigid ((Π $A$ $B$ : Set) ≈ (Π $S$ $T$ : Set)) = do
  $x$ ← fresh (s2n "x")
  active $ ($A$ : Set) ≈ ($S$ : Set)
  active $ $\forall \hat{x} : A \ddagger S.\, (B\{\acute{x}\} : \mathbf{Set}) \approx (T\{\grave{x}\} : \mathbf{Set})$
rigidRigid ((Σ $A$ $B$ : Set) ≈ (Σ $S$ $T$ : Set)) = do
  $x$ ← fresh (s2n "x")
  active $ ($A$ : Set) ≈ ($S$ : Set)
  active $ $\forall \hat{x} : A \ddagger S.\, (B\{\acute{x}\} : \mathbf{Set}) \approx (T\{\grave{x}\} : \mathbf{Set})$
rigidRigid (V $x$ $w$ · $e$ ≈ V $x'$ $w'$ · $e'$) =
  matchSpine $x$ $w$ $e$ $x'$ $w'$ $e'$ ≫ return ()
rigidRigid $q$ | orthogonal $q$ = throwError "Rigid-rigid mismatch"
              | otherwise     = block $ Unify $q$
```

A constraint has no solutions if it equates two orthogonal terms, with different constructors or variables, as defined in Figure 4.13 (page 76).

```
orthogonal :: Equation → Bool
orthogonal ((Π _ _ : Set) ≈ (Σ _ _ : Set))  = True
orthogonal ((Π _ _ : Set) ≈ (𝔹 : Set))      = True
orthogonal ((Σ _ _ : Set) ≈ (Π _ _ : Set))  = True
orthogonal ((Σ _ _ : Set) ≈ (𝔹 : Set))      = True
orthogonal ((𝔹 : Set) ≈ (Π _ _ : Set))      = True
orthogonal ((𝔹 : Set) ≈ (Σ _ _ : Set))      = True
orthogonal ((tt : 𝔹) ≈ (ff : 𝔹))            = True
orthogonal ((ff : 𝔹) ≈ (tt : 𝔹))            = True
orthogonal ((Π _ _ : Set) ≈ (V _ _ · _ : _)) = True
orthogonal ((Σ _ _ : Set) ≈ (V _ _ · _ : _)) = True
orthogonal ((𝔹 : Set) ≈ (V _ _ · _ : _))     = True
orthogonal ((tt : 𝔹) ≈ (V _ _ · _ : _))      = True
orthogonal ((ff : 𝔹) ≈ (V _ _ · _ : _))      = True
orthogonal ((V _ _ · _ : _) ≈ (Π _ _ : Set)) = True
orthogonal ((V _ _ · _ : _) ≈ (Σ _ _ : Set)) = True
orthogonal ((V _ _ · _ : _) ≈ (𝔹 : Set))     = True
orthogonal ((V _ _ · _ : _) ≈ (tt : 𝔹))      = True
orthogonal ((V _ _ · _ : _) ≈ (ff : 𝔹))      = True
orthogonal _                                 = False
```

When there are rigid variables at the heads on both sides, proceed through the evaluation contexts, demanding that projections are identical and unifying terms in applications. Note that matchSpine returns the types of the two sides, used when unifying applications to determine the types of the arguments. For example, if $y : \Pi x : S.\ T\{x\} \to U$ then the constraint $y\,s\,t \approx y\,u\,v$ will decompose into $(s : S) \approx (u : S) \wedge (t : T\{s\}) \approx (v : T\{u\})$.

```
matchSpine :: Nom → Twin → Bwd Elim →
                Nom → Twin → Bwd Elim →
                  Contextual (Type, Type)
matchSpine x w • x' w' •
  | x ≡ x'    = (,) ⟨$⟩ lookupVar x w ⟨*⟩ lookupVar x' w'
  | otherwise = throwError "rigid head mismatch"
matchSpine x w (e :< A a) x' w' (e' :< A s) = do
  (Π A B, Π S T) ← matchSpine x w e x' w' e'
  active $ (a : A) ≈ (s : S)
  return (B{a}, T{s})
matchSpine x w (e :< Hd) x' w' (e' :< Hd) = do
  (Σ A B, Σ S T) ← matchSpine x w e x' w' e'
  return (A, S)
matchSpine x w (e :< Tl) x' w' (e' :< Tl) = do
  (Σ A B, Σ S T) ← matchSpine x w e x' w' e'
  return (B{V x w · (e :< Hd)}, T{V x' w' · (e' :< Hd)})
matchSpine x w (e :< If T s t) x' w' (e' :< If T' s' t') = do
  (𝔹, 𝔹) ← matchSpine x w e x' w' e'
  y ← fresh (s2n "y")
  active $ ∀y:𝔹. (T{var y} : Type) ≈ (T'{var y} : Type)
  active $ (s : T{tt}) ≈ (s' : T'{tt})
  active $ (t : T{ff}) ≈ (t' : T'{ff})
  return (T{V x w · e}, T'{V x' w' · e'})
matchSpine _ _ _ _ _ _ = throwError "spine mismatch"
```

C.4.6 Solvitur ambulando

Constraint solving is started by the ambulando function, which lazily propagates a substitution rightwards through the metacontext, making progress on problems where possible. It maintains the invariant that the entries to the left of the cursor include no active problems. This is not the only possible strategy: indeed, it is crucial for guaranteeing most general solutions that solving the constraints in any order would produce the same result. However, it is simple to implement and often works well with the heterogeneity invariant, because the problems making a constraint homogeneous will usually be solved before the constraint itself.

```
ambulando :: Subs → Contextual ()
ambulando $\theta$ = optional popR ≫= \ $x$ → case $x$ of
  Nothing          → return ()
  Just (Left $\theta'$)   → ambulando ($\theta \circ \theta'$)
  Just (Right $e$)  → case update $\theta$ $e$ of
    $e'$@(E $\alpha$ ($T$, HOLE)) → do lower • $\alpha$ $T$ ⊗ pushL $e'$
                                  ambulando $\theta$
    Q Active $p$               → do pushR (Left $\theta$)
                                  solver $p$
                                  ambulando []
    $e'$                       → do pushL $e'$
                                  ambulando $\theta$
```

Each problem records its status, which is either Active and ready to be worked on or Blocked and unable to make progress. The update function applies a substitution to an entry, updating the status of a problem if its type changes.

```
update :: Subs → Entry → Entry
update $\theta$ (Q $s$ $p$) = Q $s'$ $p'$
   where $p'$ = substs $\theta$ $p$
         $s'$ | $p \equiv p'$     = $s$
            | otherwise = Active
update $\theta$ $e$ = substs $\theta$ $e$
```

For simplicity, Blocked problems do not store any information about when they may be resumed. An optimisation would be to track the conditions under which they should become active, typically when particular metavariables are solved or types become definitionally equal.

Appendix D

Selected proofs

This appendix contains details of selected proofs from Chapters 2–6.

D.1 Correctness of unification and type inference

Lemma 2.6 (Soundness and generality of unification).

(a) If $\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1$ *then* $\Theta_0 \sqsubseteq \Theta_1$ *is a minimal solution of* $\tau \ \equiv\ \upsilon$.

(b) If $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$ *then* $\Theta_0, \Xi \sqsubseteq \Theta_1$ *is a minimal solution of* $\alpha \ \equiv\ \tau$.

Proof. By induction on the structure of derivations. For each 'unify' rule, one must verify that it gives a solution (i.e. $\Theta_0 \sqsubseteq \Theta_1$ and $\Theta_1 \vdash \tau \equiv \upsilon : *$), and that this solution is minimal (i.e. given any other solution $\theta : \Theta_0 \sqsubseteq \Theta'$ such that $\Theta' \vdash \theta\,\tau \equiv \theta\,\upsilon : *$, there is a cofactor $\zeta : \Theta_1 \sqsubseteq \Theta'$ with $\theta \ \equiv\ \zeta \cdot \iota$).

For each 'instantiate' rule one must verify $\Theta_0, \Xi \sqsubseteq \Theta_1$, $\Theta_1 \vdash \alpha \equiv \tau : *$ and that given any other solution $\theta : \Theta_0, \Xi \sqsubseteq \Theta'$ such that $\Theta' \vdash \theta\,\alpha \equiv \theta\,\tau : *$ there is a cofactor $\zeta : \Theta_1 \sqsubseteq \Theta'$ with $\theta \ \equiv\ \zeta \cdot \iota$.

The key idea is that the type variables of Θ_0 and Θ_1 are the same, and the definitions made in Θ_1 must hold as equations in Θ' for the problem to be solved, so the solution θ can be rearranged to produce the necessary cofactor. I consider some of the more interesting cases.

For the DECOMPOSE rule, solutions to $\tau_0 \rightarrow \tau_1 \equiv \upsilon_1 \rightarrow \upsilon_1$ are exactly those that solve $\tau_0 \equiv \upsilon_0 \ \wedge\ \tau_1 \equiv \upsilon_1$, so it gives a minimal solution by the Optimist's lemma (Lemma 2.4).

For the SKIP-SEMI rule, suppose that $\theta : \Theta_0 ; \ \sqsubseteq\ \Theta' \,;\, \Xi$ solves $\alpha \ \equiv\ \beta$, so $\Theta' \,;\, \Xi \vdash \theta\,\alpha \equiv \theta\,\beta : *$. Now $\theta|_{\Theta_0} : \Theta_0 \sqsubseteq \Theta'$ by definition of the $\sqsubseteq$ relation, so by induction there exists $\zeta : \Theta_1 \sqsubseteq \Theta'$ with $\theta \equiv \zeta \cdot \iota$. Then $\zeta : \Theta_1 \fatsemi \sqsubseteq \Theta' \fatsemi \Xi$ is the required cofactor.

For the INST-SKIP-SEMI rule, suppose that $\theta : \Theta_0 \fatsemi \Xi \sqsubseteq \Theta' \fatsemi \Xi'$ solves $\alpha \equiv \tau$, so $\Theta' \fatsemi \Xi' \vdash \theta\,\alpha \equiv \theta\,\tau : *$. Now Θ_0 declares α by the input conditions (Definition 2.1), so $\theta\,\alpha$ is a Θ'-type and $\theta\,\tau$ is equal to it. Hence $\theta\,\tau$ does not depend on any metavariables in Ξ'. Now all the metavariables declared in Ξ occur in τ, giving $\theta : \Theta_0 \fatsemi \Xi \sqsubseteq \Theta' \fatsemi$ and hence $\theta : \Theta_0, \Xi \sqsubseteq \Theta'$. By induction there exists $\zeta : \Theta_1 \sqsubseteq \Theta'$ such that $\theta \equiv \zeta \cdot \iota$. □

Lemma 2.11 (Soundness and generality of type inference)**.** *If $\Theta_0 \vdash t : \tau \dashv \Theta_1$, then $\Theta_0 \sqsubseteq \Theta_1$ is a minimal solution to the type inference problem for t with output τ. Similarly, if $\Theta_0 \vdash t : \sigma \dashv \Theta_1$ then $\Theta_0 \sqsubseteq \Theta_1$ is a minimal solution to the type scheme inference problem for t with output σ.*

Proof. Proceed by induction on derivations. It is straightforward to show that $\Theta_0 \sqsubseteq \Theta_1$ and $\Theta_1 \vdash t : \tau$ or $\Theta_1 \vdash t : \sigma$. The more interesting part is establishing that the solution is minimal, for which suppose $\theta : \Theta_0 \sqsubseteq \Theta'$ is a solution, and exhibit a cofactor $\zeta : \Theta_1 \sqsubseteq \Theta'$.

The Generalist's lemma proves the property required for the INFER-GEN rule.

For the INFER-VAR rule, suppose $x : (\forall \Xi . \upsilon) \in \Theta_0$, $\theta : \Theta_0 \sqsubseteq \Theta'$ and $\Theta' \vdash x : \upsilon'$. By inversion, the proof must consist of the VAR rule, so $\Theta' \vdash \theta\,(\forall \Xi . \upsilon) \succ \upsilon'$. Thus there is some substitution $\zeta : \Theta', \theta\Xi \sqsubseteq \Theta'$ such that $\Theta' \vdash \zeta\,(\theta\,\upsilon) \equiv \upsilon' : *$ and ζ is the identity on Θ'. Weakening θ gives $\theta' : \Theta_0, \Xi \sqsubseteq \Theta', \theta\Xi$ and hence $\zeta \cdot \theta' : \Theta_0, \Xi \sqsubseteq \Theta'$ is the required cofactor.

For the INFER-LAM rule, suppose $\theta : \Theta_0 \sqsubseteq \Theta'$ and $\Theta' \vdash \lambda x . t : \upsilon \to \tau'$, then $\Theta', x : \upsilon \vdash t : \tau'$ by inversion. Now $(\theta, \upsilon/\alpha) : \Theta_0, \alpha : *, x : \alpha \sqsubseteq \Theta', x : \upsilon$ so induction on the first premise gives $\zeta : \Theta_1, x : \alpha, \Xi \sqsubseteq \Theta', x : \upsilon$ such that $(\theta, \upsilon/\alpha) \equiv \zeta \cdot \iota$ and $\Theta' \vdash \tau' \equiv \zeta\,\tau : *$. Thus $\zeta : \Theta_1, \Xi \sqsubseteq \Theta'$ is the required cofactor.

For the INFER-APP rule, the Optimist's lemma does not directly apply because it does not apply to problems with outputs, but the same reasoning applies.[1] Suppose $\theta : \Theta_0 \sqsubseteq \Theta'$ and $\Theta' \vdash s\,t : \tau$. By inversion, $\Theta' \vdash s : \tau' \to \tau$ and $\Theta' \vdash t : \tau'$ for some τ'. Thus induction on the first premise gives $\zeta : \Theta_1 \sqsubseteq \Theta'$ such that $\theta \equiv \zeta \cdot \iota$ and $\Theta' \vdash \zeta\,\upsilon \equiv \tau' \to \tau : *$. Now induction on the second premise gives $\zeta' : \Theta_2 \sqsubseteq \Theta'$ such that $\zeta \equiv \zeta' \cdot \iota$ and $\Theta' \vdash \zeta'\,\upsilon' \equiv \tau' : *$. Since there is a solution $(\zeta', \tau/\alpha) : \Theta_2, \alpha : * \sqsubseteq \Theta'$ such that $\Theta' \vdash (\zeta', \tau/\alpha)\,\upsilon \equiv (\zeta', \tau/\alpha)\,(\upsilon' \to \alpha) : *$, Lemma 2.6 applied to the third premise gives $\zeta'' : \Theta_3 \sqsubseteq \Theta'$ with $(\zeta', \tau/\alpha) \equiv \zeta'' \cdot \iota$. Now $\theta \equiv \zeta'' \cdot \iota$ so ζ'' is the required cofactor.

[1]The lemma can be generalised to apply to this rule (Gundry et al., 2010), but I omit the more general formulation here for simplicity of presentation.

For the INFER-LET rule, suppose $\theta : \Theta_0 \sqsubseteq \Theta'$ gives $\Theta' \vdash \mathbf{let}\, x = s\, \mathbf{in}\, t : \tau'$, then by inversion, $\Theta' ⨾ \Xi' \vdash s : \upsilon$ and $\Theta', x : (\forall \Xi'.\upsilon) \vdash t : \tau'$ for some υ. Now $\Theta' \vdash s : (\forall \Xi'.\upsilon)$ so by induction on the first premise there must be some $\zeta : \Theta_1 \sqsubseteq \Theta'$ such that $\theta \equiv \zeta \cdot \iota$ and $\Theta' \vdash \zeta\, \sigma \succ (\forall \Xi'.\upsilon)$. Now $\zeta : \Theta_1, x : \sigma \sqsubseteq \Theta', x : \zeta\, \sigma$ so by induction on the second premise there must be some $\zeta' : \Theta_2, x : \sigma, \Xi \sqsubseteq \Theta', x : \zeta\, \sigma$ such that $\zeta \equiv \zeta' \cdot \iota$ and $\Theta', x : \zeta\, \sigma \vdash \zeta'\, \tau \equiv \tau' : *$. Thus $\zeta' : \Theta_2, \Xi \sqsubseteq \Theta'$ is the required cofactor since $\theta \equiv \zeta' \cdot \iota$ and $\Theta' \vdash \zeta'\, \tau \equiv \tau' : *$. □

Lemma 2.12 (Completeness of type inference)**.**

(a) If (Θ_0, t) *is a type inference problem with solution* $(\theta : \Theta_0 \sqsubseteq \Theta', \upsilon)$*, then* $\Theta_0 \vdash t : \tau \dashv \Theta_1$ *for some* Θ_1 *and* τ*.*

(b) If (Θ_0, t) *is a scheme inference problem with solution* $(\theta : \Theta_0 \sqsubseteq \Theta', \sigma')$*, then* $\Theta_0 \vdash t : \sigma \dashv \Theta_1$ *for some* Θ_1 *and* σ*.*

Proof. Proceed by induction on the derivation of $\Theta' \vdash t : \upsilon$ or $\Theta' \vdash t : \sigma'$ in the transformed declarative system (Figure 2.8, page 25).

For the VAR case, $\Theta' \ni x : \sigma$ so $\Theta_0 \ni x : \sigma_0$ for some σ_0 by definition of information increase, and hence the INFER-VAR rule applies.

For the LAM case, $(\theta, \tau/\alpha) : \Theta_0, \alpha : *, x : \alpha \sqsubseteq \Theta', x : \tau$ with υ is a solution to the type inference problem for t, so by induction, $\Theta_0, \alpha : *, x : \alpha \vdash t : \tau' \dashv \Theta_1'$ for some Θ' and τ'. Moreover, $\Theta_1' = \Theta_1, x : \alpha, \Xi$ by soundness of type inference and they definition of information increase, so the INFER-LAM rule applies.

For the APP case, inversion gives $\Theta' \vdash s : \tau' \to \tau$ and $\Theta' \vdash t : \tau'$. Two appeals to the inductive hypothesis show that inference succeeds for s and t, with types υ and υ'. Now generality of type inference gives $\zeta : \Theta_2 \sqsubseteq \Theta'$ such that $\theta \equiv \zeta \cdot \iota$ and $\Theta' \vdash \zeta\, \upsilon \equiv \tau' \to \tau \;\wedge\; \zeta\, \upsilon' \equiv \tau'$. Then $(\zeta, \tau/\alpha) : \Theta_2, \alpha : * \sqsubseteq \Theta'$ and $\Theta' \vdash \zeta\, \upsilon \equiv \zeta\, \upsilon' \to \tau : *$ so Lemma 2.8 shows that the INFER-APP rule applies.

For the LET case, observe that $\Theta' \vdash s : \forall \Xi.\upsilon$ so by induction using part (b), $\Theta_0 \vdash s : \sigma \dashv \Theta_1$ for some Θ_1 and σ. By generality of type inference, there exists $\zeta : \Theta_1 \sqsubseteq \Theta'$ such that $\Theta' \vdash \zeta\, \sigma \succ \forall \Xi.\upsilon$. Note that $\zeta : \Theta_1, x : \sigma \sqsubseteq \Theta', x : \zeta\, \sigma$. Now $\Theta', x : \forall \Xi.\upsilon \vdash t : \tau$ and hence $\Theta', x : \zeta\, \sigma \vdash t : \tau$, so the INFER-LET rule applies by induction.

In part (b), suppose $\theta : \Theta_0 \sqsubseteq \Theta'$ is a solution to the scheme inference problem for t, with output $\forall \Xi'.\upsilon$. Then $\Theta' ⨾ \Xi' \vdash t : \upsilon$. Now $\theta : \Theta_0 ⨾ \sqsubseteq \Theta' ⨾ \Xi'$ so induction using part (a) gives $\Theta_0 ⨾ \vdash t : \tau \dashv \Theta_1 ⨾ \Xi$ and hence the INFER-GEN rule applies. □

D.2 Correctness of abelian group unification

Lemma 3.2 (Soundness and generality of abelian group unification). *If the group unification algorithm succeeds with $\Theta_0 \,\|\, \Upsilon \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1$, then $\Theta_0, \Upsilon \sqsubseteq \Theta_1$ is a minimal solution of $\nu \equiv 1 : \mathcal{U}$.*

Proof. Proceed by induction on derivations. For soundness, it is easy to verify that $\theta : \Theta_0, \Upsilon \sqsubseteq \Theta_1$ and $\Theta_1 \vdash \theta\,\nu \equiv 1 : \mathcal{U}$. Now consider generality for each rule in Figure 3.5 (page 37). In each case, suppose $\theta : \Theta_0, \Upsilon \sqsubseteq \Theta'$ is such that $\Theta' \vdash \nu \equiv 1 : \mathcal{U}$, and exhibit a cofactor $\zeta : \Theta_1 \sqsubseteq \Theta'$.

For U-TRIVIAL, the result is obvious.

For U-SKIP-SEMI, if Υ is empty then the result is straightforward. Otherwise, Υ contains a single unknown variable $\beta : \mathcal{U}$; let $\nu \equiv \beta^k * \nu'$. Moreover, suppose $\theta : \Theta_0 \fatsemi \beta : \mathcal{U} \sqsubseteq \Theta' \fatsemi \Xi$ is such that $\Theta' \fatsemi \Xi \vdash \theta\,(\beta^k * \nu') \equiv 1$. Rearranging gives $\Theta' \fatsemi \Xi \vdash (\theta\,\beta)^k \equiv (\theta\,\nu')^{-1}$ but $\theta\,\nu'$ is defined over Θ' so $\theta\,\beta$ must be defined over Θ'. Thus $\theta : \Theta_0, \beta : \mathcal{U} \sqsubseteq \Theta'$ and the result follows by the inductive hypothesis.

For U-SKIP-TY, U-SKIP-TMand U-SUBS, it is straightforward to check that the inductive hypothesis gives the required cofactor.

For U-DEFINE, suppose $\theta : \Theta_0, \alpha : \mathcal{U}, \Upsilon \sqsubseteq \Theta'$ is such that $\Theta' \vdash \theta\,(\alpha^k * \nu^k) \equiv 1$. Then $\Theta' \vdash (\theta\,(\alpha * \nu))^k \equiv 1$ and hence $\Theta' \vdash \theta\,(\alpha * \nu) \equiv 1$ for the *free* abelian group. Thus $\Theta' \vdash \theta\,\alpha \equiv \theta\,(\nu^{-1})$ and so $\theta \equiv \theta \cdot [\nu^{-1}/\alpha] : \Theta_0, \Upsilon \sqsubseteq \Theta'$.

For U-REDUCE, apply the isomorphism lemma (Lemma 2.5, page 18). The inductive hypothesis gives that $\Theta_0, \Upsilon, \beta : \mathcal{U} \sqsubseteq \Theta_1$ is a minimal solution of $\beta^k * \mathsf{R}_k(\nu) \equiv 1$. Moreover $[\alpha * \mathsf{Q}_k(\nu)^{-1}/\beta] : \Theta_0, \Upsilon, \beta : \mathcal{U} \sqsubseteq \Theta_0, \alpha : \mathcal{U}, \Upsilon$ is an isomorphism with inverse $[\beta * \mathsf{Q}_k(\nu)/\alpha] : \Theta_0, \alpha : \mathcal{U}, \Upsilon \sqsubseteq \Theta_0, \Upsilon, \beta : \mathcal{U}$, so the isomorphism lemma gives that $\Theta_0, \alpha : \mathcal{U}, \Upsilon \sqsubseteq \Theta_1, \alpha := \beta * \mathsf{Q}_k(\nu) : \mathcal{U}$ is a minimal solution of $\alpha^k * \nu \equiv 1$.

For U-COLLECT, appeal directly to the inductive hypothesis. □

Lemma 3.3 (Completeness of abelian group unification). *If ν is a well-formed unit of measure in Θ_0, and there is some $\theta : \Theta_0 \sqsubseteq \Theta'$ such that $\Theta' \vdash \theta\,\nu \equiv 1 : \mathcal{U}$, then the algorithm produces Θ_1 such that $\Theta_0 \,\|\, \cdot \vdash \nu \equiv 1 : \mathcal{U} \dashv \Theta_1$.*

Proof. First, establish termination of the rules when viewed as an algorithm, where hypotheses correspond to recursive calls. Termination is by the lexicographic order on the total length of the context (including Υ), the maximum power of a variable in the expression being unified, and the length of the first part of the context (excluding Υ). Only the U-REDUCE and U-COLLECT rules do not decrease the total length on recursive calls; moreover, U-REDUCE decreases the maximum power of a variable and U-COLLECT decreases the length of the first part of the context. Note that the final result may be longer than the original context, due to U-REDUCE.

The algorithm terminates, so proceeding by induction on the call graph allows reasoning about completeness. By inspection of the rules, observe that only two possible cases are not covered: either ν is a constant that is not equal to 1, or ν contains exactly one variable α, and the power of α does not divide the powers of the constants. In either case, there are no possible solutions of the unification problem $\nu \equiv 1 {:} \mathcal{U}$.

Finally, note that each rule preserves solutions: that is, if the initial problem (conclusion of the rule) has a solution then the rewritten problem (hypothesis of the rule) must also have a solution. Hence failure of the algorithm indicates that the original problem had no solutions. □

Lemma 3.4 (Soundness and generality of type unification)**.**

(a) If $\Theta_0 \vdash \tau \equiv \upsilon : * \dashv \Theta_1$*, then* $\Theta_0 \sqsubseteq \Theta_1$ *is a minimal solution of* $\tau \equiv \upsilon {:} *$.

(b) If $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$*, then* $\Theta_0, \Xi \sqsubseteq \Theta_1$ *is a minimal solution of* $\alpha \equiv \tau {:} *$.

Proof. Proceed by induction on the structure of derivations, as in Lemma 2.6 (page 22). The majority of the cases are similar to the previous proof, but the UNIT rule is new, the INST rule has been modified. The INST-SKIP-SEMI rule requires a more subtle generality proof, in order to verify that instantiation moves only genuine dependencies. The input conditions ensure that units always occur in the form $\mathbb{F}\langle\alpha\rangle$, so it is obvious that α is a dependency.

For the UNIT rule, the result follows from the soundness and generality of abelian group unification (Lemma 3.2).

For the INST rule, use the Optimist's lemma (Lemma 2.4, page 18), which states that the minimal solution to a conjunction of problems is found by 'optimistically' solving the first problem in the original context, then solving the second problem in the resulting context. This rule fits the pattern as solutions to $\alpha \equiv \tau\{\, \overline{\nu_i}^{\,i} \,\} {:} *$ are the same as solutions to $(\alpha \equiv \tau\{\, \overline{\beta_i}^{\,i} \,\} {:} *) \;\wedge\; \overline{\beta_i \equiv \nu_i {:} \mathcal{U}}^{\,i}$ up to the equational theory.

Recall the INST-SKIP-SEMI rule

$$\frac{\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1}{\Theta_0 \fatsemi \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1 \fatsemi}\,,$$

and suppose $\theta : \Theta_0 \fatsemi \Xi \sqsubseteq \Theta' \fatsemi \Xi'$ is such that $\Theta' \fatsemi \Xi' \vdash \theta\,\alpha \equiv \theta\,\tau : *$. Now $\alpha : * \in \Theta_0$ by the conditions for the algorithmic judgment, so $\theta\,\alpha$ is a Θ'-type and $\theta\,\tau$ is equal to it. In the previous proof, I argued that $\theta\,\tau$ could not depend on Ξ', but this does not hold for the equational theory of abelian groups, because equivalent expressions can have different sets of free variables. However, if $\beta : \mathcal{U} \in \Xi$ then $\mathbb{F}\langle\beta\rangle$ is a subterm of τ, so $\mathbb{F}\langle\theta\,\beta\rangle$ is a subterm of $\theta\,\tau$ and hence there is some of Θ'-unit ν with $\theta\,\beta \equiv \nu$. Similarly, if $\gamma : * \in \Xi$ then $\gamma \in \mathsf{fmv}(\tau)$ so $\theta\,\gamma$ is defined over Θ'. Hence there is some $\theta' : \Theta_0 \fatsemi \Xi \sqsubseteq \Theta' \fatsemi$ with $\theta \equiv \theta'$, so $\theta' : \Theta_0, \Xi \sqsubseteq \Theta'$ and by induction there exists $\zeta : \Theta_1 \sqsubseteq \Theta'$ as required. □

Lemma 3.5 (Completeness of type unification)**.**

(a) If the types υ and τ are well-formed in Θ_0 and there is some $\theta : \Theta_0 \sqsubseteq \Theta'$ with $\Theta' \vdash \theta\,\upsilon \equiv \theta\,\tau : *$, then unification produces Θ_1 such that $\Theta_0 \vdash \upsilon \equiv \tau : * \dashv \Theta_1$.

(b) Moreover, if $\theta : \Theta_0, \Xi \sqsubseteq \Theta'$ is such that $\Theta' \vdash \theta\,\alpha \equiv \theta\,\tau : *$ and the input conditions (Definition 3.1) are satisfied, then there is some context Θ_1 such that $\Theta_0 \,|\, \Xi \vdash \alpha \equiv \tau : * \dashv \Theta_1$.

Proof. First establish that the system terminates, if viewed as an algorithm with inputs Θ_0 (and Ξ), υ (or α) and τ, giving outputs Θ_1 and θ. The ‘unify’ judgments terminate because each recursive call removes a type metavariable from the context, decomposes the types or removes a unit metavariable. The ‘instantiate’ judgments either shorten the whole context or the part of the context before the bar. Note that the INST rule may add unit metavariables, but a type variable will be removed from the context by instantiation. Only the DECOMPOSE rule makes more than one recursive call to type unification, and it decomposes types so it does not matter that the intermediate context may have more unit metavariables.

Now proceed by structural induction on the call graph, observing that each rule in turn preserves solutions, and that all (potentially solvable) cases are covered. The only cases not covered are rigid-rigid mismatches (e.g. unifying $\upsilon \to \tau$ with $\mathbb{F}\langle\nu\rangle$) and the flex-rigid problem $\alpha \equiv \tau$ in context $\Theta_0, \alpha : *, \Xi$ where $\alpha \in \mathsf{fmv}(\tau)$. The latter has no solutions because the occurs check fails (if α is in Ξ then the conditions of the lemma ensure τ depends on it), as in Lemma 2.8. The algorithm may also fail in abelian group unification, for which completeness is by Lemma 3.3. □

D.3 Correctness of Miller pattern unification

D.3.1 Consistency of the unification logic

To prove consistency of the unification logic, as described in Section 4.3 (page 81), it is enough to show that every derivation has a normal form.

Lemma 4.9. *If* Θ *is solved,* $\Theta \mid \Gamma \vdash P$ *and* δ *is a substitution from* Γ *to* Δ *that identifies twins, then* $\Theta \mid \Delta \vdash \delta\, P$ **is**.

Proof. By induction on the derivation of $\Theta \mid \Gamma \vdash P$.

Case $\boxed{\dfrac{\Theta \mid \Gamma \vdash \textbf{ctx}}{\Theta \mid \Gamma \vdash \top}}$. Trivial.

Case $\boxed{\dfrac{\Theta \mid \Gamma \vdash \bot \qquad \Theta \mid \Gamma \vdash P\ \textbf{wf}}{\Theta \mid \Gamma \vdash P}}$. By induction, $\Theta \mid \Delta \vdash \bot$ **is**, which is impossible.

Case $\boxed{\dfrac{\Theta \mid \Gamma, x \mathbin{:} S \vdash P}{\Theta \mid \Gamma \vdash \forall x \mathbin{:} S.\, P}}$. $\Theta \mid \Delta, x \mathbin{:} \delta\, S \vdash \delta\, P$ **is** follows from the inductive hypothesis, and hence $\Theta \mid \Delta \vdash \forall x \mathbin{:} \delta\, S.\, \delta\, P$ **is**.

Case $\boxed{\dfrac{\begin{array}{l}\Theta \mid \Gamma \vdash (S \mathbin{:} \textbf{Type}) \approx (T \mathbin{:} \textbf{Type})\\ \Theta \mid \Gamma, \hat{x} \mathbin{:} S \ddagger T \vdash P\end{array}}{\Theta \mid \Gamma \vdash \forall \hat{x} \mathbin{:} S \ddagger T.\, P}}$. Similarly to the previous case, induction gives $\Theta \mid \Delta \vdash \textbf{Type} \ni \delta\, S \mathrel{\equiv\!\![}\ U \mathrel{]\!\!\equiv} \delta\, T$ and $\Theta \mid \Delta, x : U \vdash \delta\, P\{x, x\}$ **is**, and hence $\Theta \mid \Delta \vdash \forall \hat{x} \mathbin{:} \delta\, S \ddagger \delta\ T.\, \delta\, P$ **is**.

Case $\boxed{\dfrac{\begin{array}{l}\Theta \mid \Gamma \vdash \textbf{Type} \ni S \mathrel{\equiv\!\![}\ U \mathrel{]\!\!\equiv} T\\ \Theta \mid \Gamma, x \mathbin{:} U \vdash P\{x, x\}\end{array}}{\Theta \mid \Gamma \vdash \forall \hat{x} \mathbin{:} S \ddagger T.\, P}}$. Similar to the previous case.

Case $\boxed{\dfrac{\begin{array}{l}\Theta \mid \Gamma \vdash \forall x \mathbin{:} S.\, P\\ \Theta \mid \Gamma \vdash S \ni s\end{array}}{\Theta \mid \Gamma \vdash P\{s\}}}$. By induction, $\Theta \mid \Delta \vdash \forall x \mathbin{:} \delta\, S.\, \delta\, P$ **is**, so inversion gives $\Theta \mid \Delta, x \mathbin{:} \delta\, S \vdash \delta\, P$ **is**. Then $\Theta \mid \Delta \vdash \delta\, P\{s\}$ **is** by the substitution lemma.

Case $\boxed{\begin{array}{l} \Theta \mid \Gamma \vdash \forall \hat{x}{:}S \ddagger T.\, P \\ \Theta \mid \Gamma \vdash \mathbf{Type} \ni S \equiv\!\!\!\![\; U \;]\!\!\!\!\equiv T \\ \Theta \mid \Gamma \vdash U \ni u \\ \hline \multicolumn{1}{c}{\Theta \mid \Gamma \vdash P\{u, u\}} \end{array}}$. By induction, $\Theta \mid \Delta \vdash \forall \hat{x} : \delta\, S \ddagger \delta\, T.\, \delta\, P$ **is**, so $\Theta \mid \Delta \vdash \mathbf{Type} \ni \delta\, S \equiv\!\!\!\![\; U \;]\!\!\!\!\equiv \delta\, T$ and $\Theta \mid \Delta, x : U \vdash \delta\, P\{x, x\}$ **is** by inversion. Then $\Theta \mid \Delta \vdash U \ni \delta\, u$ so substitution gives $\Theta \mid \Delta \vdash \delta\, P\{\delta\, u, \delta\, u\}$ **is**.

Case $\boxed{\dfrac{\Theta \ni ?\, P \qquad \Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash P}}$. P is solved, so use Lemma 4.7 (page 81).

Conjunction introduction and elimination. Straightforward appeal to the inductive hypotheses.

Case $\boxed{\dfrac{\Theta \mid \Gamma \vdash \Pi x{:}A.\, B \approx \Pi x{:}S.\, T}{\Theta \mid \Gamma \vdash A \approx S \wedge \forall \hat{x}{:}A \ddagger S.\, B\{\acute{x}\} \approx T\{\grave{x}\}}}$. The inductive hypothesis gives $\Theta \mid \Delta \vdash \Pi x{:}\delta\, A.\, \delta\, B \approx \Pi x{:}\delta\, S.\, \delta\, T$ **is** so inversion using the definition of **is** gives $\Theta \mid \Delta \vdash \mathbf{Set} \ni \Pi x : \delta\, A.\, \delta\, B \equiv \Pi x : \delta\, S.\, \delta\, T$. Then inversion on the definitional equality gives $\Theta \mid \Delta \vdash \mathbf{Set} \ni \delta\, A \equiv\!\!\!\![\; U \;]\!\!\!\!\equiv \delta\, B$ and $\Theta \mid \Delta, x : U \vdash \mathbf{Set} \ni \delta\, B \equiv \delta\, T$. Thus $\Theta \mid \Delta \vdash \delta\, A \approx \delta\, B \wedge \forall \hat{x}{:}\delta\, S \ddagger \delta\, T.\, \delta\, B\{\acute{x}\} \approx \delta\, T\{\grave{x}\}$ **is**.

Case $\boxed{\dfrac{\Theta \mid \Gamma \vdash \Sigma x{:}A.\, B \approx \Sigma x{:}S.\, T}{\Theta \mid \Gamma \vdash A \approx S \wedge \forall \hat{x}{:}A \ddagger S.\, B\{\acute{x}\} \approx T\{\grave{x}\}}}$. Similar to the previous case.

Reflexivity, symmetry and transitivity. By Lemma 4.1 (page 65) and the definition of $\Theta \mid \Delta \vdash (s{:}S) \approx (t{:}\, T)$ **is**.

Case $\boxed{\dfrac{\Gamma \ni \hat{x}{:}S \ddagger T \qquad \Theta \mid \Gamma \vdash \mathbf{ctx}}{\Theta \mid \Gamma \vdash (\acute{x}{:}S) \approx (\grave{x}{:}\, T)}}$. Here δ identifies twins, so it must be the case that $\Theta \mid \Delta \vdash (\delta\, \acute{x}{:}\delta\, S) \approx (\delta\, \grave{x}{:}\delta\, T)$ **is**.

Congruence rules (Figure 4.7, page 62). Each congruence rule corresponds to a rule of the definitional equality, except for the presence of twins. The heterogeneity invariant means that the types of twins are provably equal, so induction means they are definitionally equal and can be replaced with a single variable. □

D.3.2 Soundness

If twins have definitionally equal types, they can be replaced with a single variable:

Lemma D.1. *Suppose* $\Theta \,|\, \Gamma \vdash \mathbf{Type} \ni A \mathrel{\equiv\!\![} U \mathrel{]\!\!\equiv} S$. *Then* $\Theta \,|\, \Gamma \vdash \forall \hat{x} : A \ddagger S.\, P$ *if and only if* $\Theta \,|\, \Gamma \vdash \forall x : U.\, P\{x, x\}$.

Proof. For the forward direction, observe that $\Theta \,|\, \Gamma, y : U \vdash \forall \hat{x} : A \ddagger S.\, P$ and instantiate this with $(y, y)/\hat{x}$ to get $\Theta \,|\, \Gamma, y : U \vdash P\{y, y\}$, so $\Theta \,|\, \Gamma \vdash \forall y : U.\, P\{y, y\}$. For the reverse direction, if $\Theta \,|\, \Gamma \vdash \forall x : U.\, P\{x, x\}$ then $\Theta \,|\, \Gamma, y : U \vdash \forall x : U.\, P\{x, x\}$ so $\Theta \,|\, \Gamma, y : U \vdash P\{y, y\}$ and hence $\Theta \,|\, \Gamma \vdash \forall \hat{x} : S \ddagger T.\, P\{s, t\}$ by an inference rule. $\square$

The following lemma justifies decomposition of rigid-rigid equations between eliminated variables, which is part of the soundness of problem decomposition.

Lemma D.2. *Suppose* $x \cdot e \bowtie x' \cdot e' \mapsto P$, $\Theta \,|\, \Gamma \vdash x \cdot e \in S$ *and* $\Theta \,|\, \Gamma \vdash x' \cdot e' \in S'$. *Then* $\Theta \,|\, \Gamma \vdash P\ \mathbf{wf}$, *and if* $\Theta \,|\, \Gamma \vdash P$ *then* $\Theta \,|\, \Gamma \vdash x \cdot e \approx x' \cdot e'$.

Proof. Prove both parts simultaneously by induction on e. $\square$

All the judgments are insensitive to η-contraction:

Lemma D.3.

(a) If $\Theta \,|\, \Gamma \vdash T \ni t\{\lambda x.n\, x\}$ *then* $\Theta \,|\, \Gamma \vdash T \ni t\{\lambda x.n\, x\} \equiv t\{n\}$.

(b) If $\Theta \,|\, \Gamma \vdash T \ni t\{(n\ \textsc{hd}, n\ \textsc{tl})\}$ *then* $\Theta \,|\, \Gamma \vdash T \ni t\{(n\ \textsc{hd}, n\ \textsc{tl})\} \equiv t\{n\}$.

(c) If $\Theta \,|\, \Gamma\{\lambda x.n\, x\} \vdash P\{\lambda x.n\, x\}$ *then* $\Theta \,|\, \Gamma\{n\} \vdash P\{n\}$.

(d) If $\Theta \,|\, \Gamma\{(n\ \textsc{hd}, n\ \textsc{tl})\} \vdash P\{(n\ \textsc{hd}, n\ \textsc{tl})\}$ *then* $\Theta \,|\, \Gamma\{n\} \vdash P\{n\}$.

(e) If $\Theta \,|\, \Gamma\{\lambda x.n\, x\} \vdash P\{\lambda x.n\, x\}\ \mathbf{is}$ *then* $\Theta \,|\, \Gamma\{n\} \vdash P\{n\}\ \mathbf{is}$.

(f) If $\Theta \,|\, \Gamma\{(n\ \textsc{hd}, n\ \textsc{tl})\} \vdash P\{(n\ \textsc{hd}, n\ \textsc{tl})\}\ \mathbf{is}$ *then* $\Theta \,|\, \Gamma\{n\} \vdash P\{n\}\ \mathbf{is}$.

(g) If $\Theta \,|\, \Gamma\{\lambda x.n\, x\} \vdash P\{\lambda x.n\, x\}\ \mathbf{wf}$ *and* $\Theta \,|\, \Gamma\{n\} \vdash P\{n\}$ *then* $\Theta \,|\, \Gamma\{\lambda x.n\, x\} \vdash P\{\lambda x.n\, x\}$.

(h) If $\Theta \,|\, \Gamma\{(n\ \textsc{hd}, n\ \textsc{tl})\} \vdash P\{(n\ \textsc{hd}, n\ \textsc{tl})\}\ \mathbf{wf}$ *and* $\Theta \,|\, \Gamma\{n\} \vdash P\{n\}$ *then* $\Theta \,|\, \Gamma\{(n\ \textsc{hd}, n\ \textsc{tl})\} \vdash P\{(n\ \textsc{hd}, n\ \textsc{tl})\}$.

Proof. Parts (a) and (b) are by structural induction on derivations. The remaining parts follow from them by induction on derivations, using context conversion (Lemma 4.5) and conversion (Lemma 4.6). $\square$

The problem decomposition operation, summarised in Figure 4.14 (page 79), is sound in that it preserves well-formedness and provability of problems:

Lemma 4.12. *If* $\Theta \,|\, \Gamma \vdash P\, \mathbf{wf}$ *and* $P \mapsto Q$ *then*

(a) $\Theta \,|\, \Gamma \vdash Q\, \mathbf{wf}$*, and*

(b) $\Theta \,|\, \Gamma \vdash Q$ *implies* $\Theta \,|\, \Gamma \vdash P$*.*

Proof of part (a). For reflexivity (4.1), Q is trivial and hence well-formed.

For η-expanion of functions (4.2), $\Theta \,|\, \Gamma \vdash \Pi x : A.\, B \approx \Pi x : S.\, T$ from the definition of problem well-formedness, so $\Theta \,|\, \Gamma \vdash A \approx S \wedge \forall \hat{x} \!:\! A \ddagger S.\, B\{\acute{x}\} \approx T\{\grave{x}\}$ by injectivity. The case of η-expansion of pairs (4.3) is similar.

For rigid-rigid decomposition of equations between Π-types (4.4) or Σ-types (4.5), the second component of the conjunction is well-formed because the first component may be assumed as a hypothesis.

For rigid-rigid decomposition of variable applications (4.6), use Lemma D.2.

For rigid-rigid mismatch (4.7), Q is false and hence well-formed.

For η-contraction of subterms (4.8), (4.9), use Lemma D.3.

The cases that drop unused parameters or twins (4.10)–(4.13) correspond to proving admissibility for appropriate forms of strengthening.

Simplification of identical twins (4.14) and Σ-splitting of parameters (4.15) give well-formed results by the substitution lemma (Lemma 4.1, page 65). $\square$

Proof of part (b). For reflexivity (4.1), P holds definitionally.

For the steps that perform η-expansion and rigid-rigid decomposition of Π or Σ-types (4.2)–(4.5), in each case, P follows from Q by a single application of the appropriate congruence rule from Figure 4.7.

For rigid-rigid decomposition of variable applications (4.6), use Lemma D.2.

For rigid-rigid mismatch (4.7), the proof of $Q = \bot$ can be eliminated to produce a proof of P.

For η-contraction steps (4.8) and (4.9), use Lemma D.3.

The cases that drop unused parameters or twins (4.10)–(4.13) correspond to proving admissibility for appropriate forms of weakening.

Lemma D.1 proves the required property for simplification of twins (4.14).

For Σ-splitting of parameters (4.15), instantiating Q with $\lambda\Delta.x\,\Delta\,\textsc{hd}$ for y and $\lambda\Delta.x\,\Delta\,\textsc{tl}$ for z gives the P (up to uses of surjective pairing, using Lemma D.3). $\square$

Lemma D.4 (Soundness of pruning). *Suppose* $\Theta = \Theta_0, \beta : \Pi\Delta.\ T, \Theta_1$.

(a) If $\Theta \vdash \mathbf{mctx}$ *and* $\mathsf{prune}\,\mathcal{V}\,\Delta\,\overline{t_i}^{\,i} \mapsto \Delta'$ *then* $\Theta_0 \,|\, \Delta' \vdash \mathbf{ctx}$*,* $\mathsf{vars}(\Delta') \subset \mathsf{vars}(\Delta)$*.*

(b) If $\Theta \vdash \mathbf{mctx}$ *and* $\mathsf{pruneTm}\,\mathcal{V}\,t \mapsto (\beta, \Delta')$ *then* $\Theta_0 \,|\, \cdot \vdash \mathbf{Type} \ni \Pi\Delta'.\ T$ *and* $\Theta_0, \gamma : \Pi\Delta'.\ T \,|\, \cdot \vdash \Pi\Delta.\ T \ni \lambda\Delta. \gamma\,\Delta'$*.*

Proof. Part (a) is by induction on the definition of prune, observing that the bindings in Δ' are a subset of those in Δ, and that prune retains a binding $x : S$ only if the free variables of S have been retained in Δ'. Part (b) then follows from part (a), and the fact that $\mathsf{pruneTm}$ checks that $\mathsf{fv}(T) \subset \mathsf{vars}(\Delta')$, so the type $\Pi\Delta'.\ T$ is well-formed. □

Lemma 4.13. *If* $\Theta \vdash \mathbf{mctx}$ *and* $\Theta \mapsto \Theta'$ *then* $\iota : \Theta \sqsubseteq \Theta'$*.*

Proof. By induction on the step taken.

For inversion (4.16), $\iota : \Theta, \alpha : T, \Theta \sqsubseteq \Theta, \Theta_0, \alpha := \lambda\,\overline{x_i}^{\,i}.t : T, \Theta_1$ since Θ_0, Θ_1 is a dependency-respecting permutation of Θ and the solution for α is well-typed. Moreover $\forall\Gamma.\, \alpha\,\overline{x_i}^{\,i} \approx t$ holds since $\alpha\,\overline{x_i}^{\,i} \equiv (\lambda\,\overline{x_i}^{\,i}.t)\,\overline{x_i}^{\,i} \equiv t$.

For occurs check failure (4.17), the result is trivial since any problem is true in a failed metacontext.

For equation solving by intersection (4.18), observe that $\forall\Gamma.\, \alpha\,\overline{x_i}^{\,i} \approx \alpha\,\overline{y_i}^{\,i}$ holds since $\alpha\,\overline{x_i}^{\,i} \equiv (\lambda\Delta.\beta\,\Delta')\,\overline{x_i}^{\,i} \equiv \beta\,\Delta'$ by the definition of intersection, and similarly $\alpha\,\overline{y_i}^{\,i} \equiv (\lambda\Delta.\beta\,\Delta')\,\overline{y_i}^{\,i} \equiv \beta\,\Delta'$.

For pruning (4.19), use Lemma D.4.

For pruning failure (4.20), the result is trivial since Θ' is failed.

For Σ-splitting (4.21), it suffices to check that if $\Theta \,|\, \cdot \vdash \mathbf{Type} \ni \Pi\Delta.\, \Sigma x : S.\ T$ then $\Theta \,|\, \cdot \vdash \mathbf{Type} \ni \Pi\Delta.\, S$; $\Theta, \alpha_0 : \Pi\Delta.\, S \,|\, \cdot \vdash \mathbf{Type} \ni \Pi\Delta.\ T\{\alpha_0\,\Delta\}$ and $\Theta, \alpha_0 : \Pi\Delta.\, S, \alpha_1 : \Pi\Delta.\ T\{\alpha_0\,\Delta\} \,|\, \cdot \vdash \Pi\Delta.\, \Sigma x : S.\ T \ni \lambda\Delta.(\alpha_0\,\Delta, \alpha_1\,\Delta)$.

For uncurrying (4.22), a similar check is needed.

For problem decomposition (4.23), Lemma 4.12 gives that $\Theta, ?\,\forall\Gamma.\, P \vdash \mathbf{mctx}$ and $P \Mapsto Q$ implies $\Theta, ?\,\forall\Gamma.\, Q \,|\, \cdot \vdash \forall\Gamma.\, P$, since $\Theta, ?\,\forall\Gamma.\, Q \,|\, \Gamma \vdash Q$.

For conjunction splitting (4.24) and removing trivial problems (4.25), the result is trivial.

For the symmetry step (4.26), the result follows by induction and symmetry of the definitional equality (Lemma 4.4).

For the suffix step (4.27), observe that if $\theta : \Theta \sqsubseteq \Theta'$ and $\Theta, \Theta_0 \vdash \mathbf{mctx}$ then $\Theta', \theta\Theta_0 \vdash \mathbf{mctx}$ and weakening means that $(\theta, \iota) : \Theta, \Theta_0 \sqsubseteq \Theta', \theta\Theta_0$. □

D.3.3 Generality

I will need standard no confusion and no cycle results for the definitional equality, in order to prove that the steps that reject impossible equations are most general.

Lemma D.5 (No confusion)**.** *If $s \perp\!\!\!\perp t$ then there are no Θ and Γ such that $\Theta \,|\, \Gamma \vdash T \ni s \equiv t$. Moreover, if $s \perp\!\!\!\perp t$ then $\theta\, s \perp\!\!\!\perp \theta\, t$ for any metasubstitution θ.*

Proof. By induction on the derivation of $s \perp\!\!\!\perp t$, and inversion on the definitional equality relation for the first part. □

Lemma D.6 (No cycle)**.** *Suppose t contains a strong rigid occurrence of $\underline{\alpha\, \overline{t_i}^{\,i}}$, or a rigid occurrence of $\underline{\alpha\, \overline{y_i}^{\,i}}$. Then there are no Θ, Γ, θ and T such that $\Theta \,|\, \Gamma \vdash T \ni \theta\,(\alpha\, \overline{x_i}^{\,i}) \equiv \theta\, t$.*

Proof. Suppose otherwise, and without loss of generality assume that θ substitutes $\lambda\, \overline{x_i}^{\,i} \,.s$ for α, so $\theta\,(\alpha\, \overline{x_i}^{\,i}) = s$. If $\underline{\alpha\, \overline{t_i}^{\,i}}$ occurs strong rigidly (under a canonical constructor such as Π) in t, then $[\, \overline{t_i/x_i}^{\,i}\,]\, s = [\, \overline{t_i/x_i}^{\,i}\,]\,(\theta\, t)$ occurs strong rigidly in $\theta\, t$. But substitution cannot remove strong rigid occurrences of subterms, so repeating this observation shows that s contains an infinitely deep tree of canonical constructors, which is a contradiction.

If $\underline{\alpha\, \overline{y_i}^{\,i}}$ occurs rigidly (under a canonical constructor or variable) in t, then $[\, \overline{y_i/x_i}^{\,i}\,]\, s$ occurs rigidly in $\theta\, t$. Now renaming does not change the size of a term, so s is the same size as a subterm of itself, which is a contradiction. □

Lemma D.7. *If $\Theta \,|\, \Gamma \vdash T \ni s \equiv t$ then $\mathsf{fv}(s) = \mathsf{fv}(t)$.*

Proof. By induction on the derivation. □

Lemma 4.15 (Generality of problem decomposition)**.** *If $\Theta \,|\, \Gamma \vdash P\, \mathbf{wf}$, the metasubstitution $\theta : \Theta, ?\,\forall\Gamma.\, P \sqsubseteq \Theta'$ is a solution and $P \mapsto Q$, then $\theta : \Theta, ?\,\forall\Gamma.\, Q \sqsubseteq \Theta'$.*

Proof. Lemma 4.2 (page 65) implies $\Theta' \,|\, \cdot\ \vdash \theta\,(\forall\Gamma.\, P)$, so $\Theta' \,|\, \cdot\ \vdash \theta\,(\forall\Gamma.\, P)\ \mathbf{is}$ by Corollary 4.10 (page 82). Now proceed by case analysis on $P \mapsto Q$, supposing that $\theta\,(\forall\Gamma.\, P)$ is solved and showing that $\theta\,(\forall\Gamma.\, Q)$ is solved. Without loss of generality assume that Γ contains no twins,[2] so suppose $\Theta' \,|\, \theta\Gamma \vdash P\ \mathbf{is}$ and show that $\Theta' \,|\, \theta\Gamma \vdash Q\ \mathbf{is}$.

For reflexivity (4.1), Q is trivial.

For the η-expansion and rigid-rigid decomposition steps (4.2)–(4.6), each case follows from inversion on the definitional equality: for example, consider the rule

[2]By definition, a problem involving twins is solved if the types are equal and the corresponding problem without twins is solved.

for Π-types (4.4). If $\Theta' \mid \theta\Gamma \vdash \mathbf{Set} \ni \Pi x{:}\theta\, A.\, \theta\, B \Lleftarrow \Pi x{:}\, U.\, V \models \Pi x{:}\theta\, S.\, \theta\, T$ then $\Theta' \mid \theta\Gamma \vdash \mathbf{Set} \ni \theta\, A \Lleftarrow U \models \theta\, S$ and $\Theta' \mid \theta\Gamma, x : U \vdash \mathbf{Set} \ni \theta\, B \Lleftarrow V \models \theta\, T$ by inversion. Hence $\Theta' \mid \cdot \ \vdash \theta\,(\forall\Gamma.\, A \approx S \wedge \forall\hat{x}{:}\, A\ddagger S.\, B\{\acute{x}\} \approx T\{\grave{x}\})$ **is**.

For the rigid-rigid mismatch step (4.7), observe that metasubstitution cannot remove rigid differences, and rigidly different terms cannot be definitionally equal, by Lemma D.5. Thus there can be no solution θ.

For the η-contraction steps (4.8) and (4.9), use Lemma D.3.

The cases that drop unused parameters or twins (4.10)–(4.13) correspond to proving admissibility for appropriate forms of strengthening.

For simplification of twins (4.14), there is nothing to prove, as the definition of $\forall\hat{x}{:}\, S\ddagger T.\, P$ **is** means $\Theta \mid \Gamma \vdash \mathbf{Type} \ni S \Lleftarrow U \models T$ and $\forall x{:}\, U.\, P\{x, x\}$ **is**.

For Σ-splitting of parameters (4.15), use Lemma 4.7 (page 81). □

Lemma D.8 (Generality of pruning)**.** *If* $\mathsf{pruneTm}\,(\mathsf{fv}(e))\; t \;\mapsto\; (\beta, \Delta')$ *and there is some* $\theta{:}\Theta, \beta{:}\Pi\Delta.\, T, \Theta' \sqsubseteq \Theta_1$ *such that* $\Theta_1 \mid \theta\Gamma \vdash U \ni \theta\,(\alpha \cdot e) \equiv \theta\, t$*, then there exists* $\zeta{:}\Theta, \gamma{:}\Pi\Delta'.\, T, \beta := \lambda\Delta.\gamma\,\Delta'{:}\Pi\Delta.\, T, \Theta', ?\,\forall\Gamma.\, \alpha \cdot e \approx t \sqsubseteq \Theta_1$ *with* $\theta \equiv \zeta \cdot \iota$*.*

Proof. Let $\theta = (\theta_0, s/\beta, \theta_1)$ and observe that $s \equiv \lambda\Delta.u$ up to η-conversion. To see that $\mathsf{fv}(u) \subset \mathsf{vars}(\Delta')$, suppose otherwise, i.e. assume $x_j \in \mathsf{fv}(u) \backslash \mathsf{vars}(\Delta')$. By definition of pruning there is some subterm $\beta\, \overline{t_i}^{\,i}$ of t such that $\mathsf{prune}\, \mathcal{V}\, \Delta\, \overline{t_i}^{\,i} \mapsto \Delta'$. Thus $\theta\, t$ contains some $\theta\, t_j$ with $\mathsf{fv}^{\mathsf{rig}}(\theta\, t_j) \not\subset \mathcal{V}$. Hence $\mathsf{fv}(\theta\, t) \not\subset \mathsf{fv}(\theta\,(\alpha \cdot e))$, which contradicts Lemma D.7. Thus $\mathsf{fv}(u) \subset \mathsf{vars}(\Delta')$, so the cofactor ζ can be taken to be $(\theta_0, (\lambda\Delta'.u)/\gamma, (\lambda\Delta.u)/\beta, \theta_1)$. □

Theorem 4.16 (Generality)**.** *If* $\Theta_0 \vdash \mathbf{mctx}$*, the metasubstitution* $\theta{:}\Theta_0 \sqsubseteq \Theta'$ *is a solution and* $\Theta_0 \mapsto \Theta_1$ *then there exists a cofactor* $\zeta{:}\Theta_1 \sqsubseteq \Theta'$ *such that* $\theta \equiv \zeta \cdot \iota$*.*

Proof. By induction on the step taken. In each case, construct a suitable cofactor ζ. If the induced metasubstitution $\iota{:}\Theta_0 \sqsubseteq \Theta_1$ is an isomorphism, its inverse can be composed with θ to obtain the required cofactor (Lemma 2.5, page 18).

For equation solving by inversion (4.16), let ζ be the appropriate permutation of θ. Observe that θ is a solution so $\Theta' \mid \cdot \ \vdash \theta\,(\forall\Gamma.\, \alpha\, \overline{x_i}^{\,i} \approx t)$ **is** and hence $\Theta' \mid \theta\Gamma \vdash T \ni (\theta\,\alpha)\, \overline{x_i}^{\,i} \equiv \theta\, t$. Then $\Theta' \mid \cdot \ \vdash \Pi\Delta.\, T \ni \theta\,\alpha \equiv \theta\,(\lambda\, \overline{x_i}^{\,i}.t)$ by congruence of λ, η-expansion and strengthening, so $\theta \equiv \zeta \cdot \iota$.

For occurs check failure (4.17), there can be no solution θ by Lemma D.6.

For equation solving by intersection (4.18), $\Theta' \mid \cdot \ \vdash \theta\,(\forall\Gamma.\, \alpha\, \overline{x_i}^{\,i} \approx \alpha\, \overline{y_i}^{\,i})$ **is** implies $\Theta' \mid \theta\Gamma \vdash T \ni (\theta\,\alpha)\, \overline{x_i}^{\,i} \equiv (\theta\,\alpha)\, \overline{y_i}^{\,i}$. Up to η, $\theta\,\alpha$ is of the form $\lambda\Delta.t$, and any variable bound in Δ corresponding to distinct variables in $\overline{x_i}^{\,i}$ and $\overline{y_i}^{\,i}$

must not occur in t, as the above definitional equality would fail. Hence ζ can substitute $\lambda\Delta'.t$ for β.

For pruning (4.19), use Lemma D.8.

For pruning failure (4.20), observe that metasubstitution cannot add free variables (i.e. $\mathsf{fv}(\theta\, s) \subset \mathsf{fv}(s)$) or remove rigid occurrences of free variables (i.e. $\mathsf{fv}^{\mathsf{rig}}(s) \subset \mathsf{fv}^{\mathsf{rig}}(\theta\, s)$), so the existence of a solution would contradict Lemma D.7.

For Σ-splitting (4.21), the induced metasubstitution is an isomorphism, with the inverse given by substituting $\lambda\Delta.\alpha\ \textsc{hd}$ for α_0 and $\lambda\Delta.\alpha\ \textsc{tl}$ for α_1.

Similarly, uncurrying (4.22) induces an isomorphism (with the inverse given by currying).

For problem decomposition (4.23), Lemma 4.15 shows that $\zeta = \theta$ suffices.

For conjunction splitting (4.24) and removal of trivial problems (4.25), the induced metasubstitution is an isomorphism.

For the symmetry step (4.26), the result follows from the inductive hypothesis and the fact that definitional equality is symmetric.

For the suffix step (4.27), the result follows by induction. □

D.3.4 Partial completeness

Lemma 4.17. *Suppose Θ is a well-formed metacontext in the pattern fragment that is not solved or failed. Then $\Theta \mapsto \Theta'$ for some Θ' in the pattern fragment.*

Proof. By case analysis on the first unsolved problem in Θ, using step (4.27) to skip later problems. If the first problem is a conjunction, step (4.24) applies. If not, it is of the form $\forall\Gamma.\,(s : S) \approx (t : T)$. Without loss of generality, assume that Γ contains no twins (otherwise they can be removed by step (4.14)). Now $\Theta \,|\, \Gamma \vdash (S : \mathbf{Type}) \approx (T : \mathbf{Type})$ by the heterogeneity invariant, and hence $\Theta \,|\, \Gamma \vdash \mathbf{Type} \ni S \equiv T$ by Corollary 4.10. In particular, $\mathsf{fv}(S) = \mathsf{fv}(T)$ by Lemma D.7.

If $\beta \cdot e'$ is a subterm of s or t, the pattern condition means that e' consists only of projections and applications to variables. But any projections may be eliminated by the lowering step (4.21), so assume it includes only variables.

Now consider the possible cases for s and t. If they are identical, then step (4.1) removes the reflexive equation. If one of them is a function or pair, then the appropriate η-expansion step (4.2) or (4.3) applies.

If they are both rigid, then either the heads match so one of the decomposition steps (4.4)–(4.6) applies, or they do not and the algorithm fails with (4.7).

Otherwise, one of them is flexible. Suppose without loss of generality, using the symmetry step (4.26) if necessary, that $s = \alpha\,\overline{x_i}^{\,i}$, and consider the possible cases for t.

If $t = \alpha\,\overline{y_i}^{\,i}$ then step (4.18) applies: intersection always succeeds, and the condition on the free variables must hold since S and T have the same free variables, so any variable removed by intersection cannot occur in the type of α.

If t has a flexible occurrence of a variable that is not one of the $\overline{x_i}^{\,i}$, then pruning will take a step (4.19); the pattern condition ensures it will not get stuck. If t has a rigid occurrence of a forbidden variable, then unification will fail with step (4.20).

If t contains a rigid occurrence of α, then the occur check step (4.17) applies, since the evaluation context of α consists only of variables. By the pattern condition, t contains no flexible occurrences of α.

Finally, to apply the solution step (4.16), an appropriate permutation of the metacontext must exist, so that all the dependencies of t can be moved before α. Observe that the type of t does not transitively depend on α, since it is equal to the type of $\alpha\,\overline{x_i}^{\,i}$. Now by induction on the typing derivation for t, using the pattern condition and the fact that t does not contain α, none of the subterms of t have types that depend on α. In particular, none of the metavariables that occur in t have types that depend on α, so an appropriate permutation exists. (This induction requires the result type of an if-expression to contain no metavariables.) □

D.4 Consistency of evidence language coercions

The overall structure of the consistency proof for coercions in the evidence language is described in Section 6.5 (page 131). Here I will detail the proofs that were previously omitted, and prove required additional results.

Note that the reduction relation is closed under substitution:

Lemma D.9. *If* $\rho \xrightarrow{\text{kpush}} \rho'$ *then* $[\delta/\Delta]\,\rho \xrightarrow{\text{kpush}} [\delta/\Delta]\,\rho'$.

Proof. By induction on the reduction step used. □

Lemma 6.14 (Transitivity)**.** *If* $\mathbf{A}_k(\tau \sim \upsilon)$ *and* $\mathbf{A}_k(\upsilon \sim \kappa)$ *then* $\mathbf{A}_k(\tau \sim \kappa)$.

Proof. Proceed by induction on k and inversion on $\mathbf{A}_k(\varphi)$.

Consider the case for quantifiers, where $\mathbf{A}_k((a_1 :^{\Upsilon} \kappa_1) \to \tau_1 \sim (a_2 :^{\Upsilon} \kappa_2) \to \tau_2)$ and $\mathbf{A}_k((a_2 :^{\Upsilon} \kappa_2) \to \tau_2 \sim (a_3 :^{\Upsilon} \kappa_3) \to \tau_3)$. By definition, $\mathbf{A}_k(\gamma_1 : \kappa_1 \sim \kappa_2)$

for some γ_1, and $\mathbf{A}_k(\kappa_2 \sim \kappa_3)$, so induction gives $\mathbf{A}_k(\kappa_1 \sim \kappa_3)$. In order to demonstrate that $\mathbf{A}_k((a_1 :^{\Upsilon} \kappa_1) \to \tau_1 \sim (a_3 :^{\Upsilon} \kappa_3) \to \tau_3)$, suppose υ_1 and υ_3 have $\mathbf{A}_l((\upsilon_1 : \kappa_1) \sim (\upsilon_3 : \kappa_3))$ for $l < k$, and seek to prove $\mathbf{A}_l([\upsilon_1/a_1]\,\tau_1 \sim [\upsilon_3/a_3]\,\tau_3)$. But $\mathbf{A}_l([\upsilon_1/a_1]\,\tau_1 \sim [\upsilon_1 \triangleright \gamma_1/a_2]\,\tau_2)$ and $\mathbf{A}_l([\upsilon_1 \triangleright \gamma_1/a_2]\,\tau_2 \sim [\upsilon_3/a_3]\,\tau_3)$, so the result follows by induction.

The other cases where all three types are structural are similar.

If all three types are computational, then they can each take a step by definition, and the reducts are related by induction.

If τ is computational but υ and κ are structural, then the definition gives τ' structural or coerced such that $\tau \longrightarrow^* \tau'$ and $\mathbf{A}_k(\tau' \sim \upsilon)$. Then induction gives $\mathbf{A}_k(\tau' \sim \kappa)$ and hence $\mathbf{A}_k(\tau \sim \kappa)$ by definition.

If τ and υ are computational but κ is structural, then the definition gives υ' structural or coerced such that $\upsilon \longrightarrow^* \upsilon'$ and $\mathbf{A}_k(\upsilon' \sim \kappa)$. Then there exists τ' such that $\tau \longrightarrow^* \tau'$ and $\mathbf{A}_k(\tau' \sim \upsilon')$, so induction gives $\mathbf{A}_k(\tau' \sim \kappa)$ and hence $\mathbf{A}_k(\tau \sim \kappa)$.

The other cases where some of the types are computational and some are structural are similar.

If any of τ, υ and κ are coerced, then the coercion(s) can be removed and the underlying types are compatible by induction. $\square$

I need a couple of auxiliary results to prove that compatibility is closed under reduction. The first is straightforward.

Lemma D.10. *Suppose* $\cdot \vdash \mathsf{H} :^{\forall} (\Delta) \to \tau$ *and* $\cdot \vdash^{\text{tc}} \omega : \Delta$*. Then for any* k*,* $\mathbf{A}_k(\mathsf{H}\,\overleftarrow{\omega} \sim \mathsf{H}\,\overrightarrow{\omega})$ *if and only if* $\mathbf{A}_k(\omega : \Delta)$*.*

Proof. By induction on the length of ω. $\square$

Showing that compatible expressions satisfy progress is more interesting. This does not imply progress in general, because only type expressions (at phase $\forall$) are covered and they must be in the diagonal of compatibility.

Lemma D.11 (Progress for compatible expressions)**.** *If* $\mathbf{A}_k(\tau \sim \tau)$ *for* $k > 0$ *then either* τ *is a coerced value type or* τ *can take a step.*

Proof. By induction on τ and inversion on $\mathbf{A}_k(\tau \sim \tau)$. If τ is computational then the definition states that it can take a step. If $\tau = \tau' \triangleright \gamma$ is coerced then $\mathbf{A}_k(\tau' \sim \tau')$ so by induction either τ' is a coercion, a coerced value or can take a step, which implies the result. Otherwise, τ is structural: either it is immediately a value type, or it is an application $\tau'\,\rho$ and $\mathbf{A}_k(\tau' \sim \tau')$ so induction on τ' implies the result. $\square$

To deal with one coercion being cast by another, I need to show that compatibility of two propositions ($\varphi_1 \sim \varphi_2$) means compatibility of φ_1 implies compatibility of φ_2. Observe that φ_1 and φ_2 are syntactically restricted to be quantified equations, not arbitrary types. Proving this lemma is the motivation for restricting quantification at phase $\square$ to syntactic propositions only.

Lemma D.12. *If $\mathbf{A}_k(\varphi_1)$ and $\mathbf{A}_k(\varphi_1 \sim \varphi_2)$ then $\mathbf{A}_k(\varphi_2)$.*

Proof. Proceed by induction on k and case analysis on φ_1 and φ_2. Since they are both equations or quantified propositions, the definition of $\mathbf{A}_k(\varphi_1 \sim \varphi_2)$ implies that they have the same form.

If $\varphi_1 = \tau_1 \sim \upsilon_1$ then $\varphi_2 = \tau_2 \sim \upsilon_2$ where $\mathbf{A}_k(\tau_1 \sim \tau_2)$ and $\mathbf{A}_k(\upsilon_1 \sim \upsilon_2)$. Moreover $\mathbf{A}_k(\tau_1 \sim \upsilon_1)$, so $\mathbf{A}_k(\tau_2 \sim \upsilon_2)$ by transitivity (Lemma 6.14).

If $\varphi_1 = (c_1 :^{\square} \varphi_1') \to \varphi_1''$ then $\varphi_2 = (c_2 :^{\square} \varphi_2') \to \varphi_2''$. For $\mathbf{A}_k((c_2 :^{\square} \varphi_2') \to \varphi_2'')$, suppose η is such that $\mathbf{A}_l(\eta : \varphi_2')$ for some $l < k$. Now $\mathbf{A}_l(\gamma : \varphi_2' \sim \varphi_1')$ for some γ by definition of $\mathbf{A}_k(\varphi_1 \sim \varphi_2)$ and downward closure (Lemma 6.15, page 135). By induction, $\mathbf{A}_l(\eta \rhd \gamma : \varphi_1')$. Then $\mathbf{A}_l([\eta \rhd \gamma / c_1]\, \varphi_1'' \sim [\eta/c_2]\, \varphi_2'')$ by definition of $\mathbf{A}_k(\varphi_1 \sim \varphi_2)$. Moreover, $\mathbf{A}_l([\eta \rhd \gamma / c_1]\, \varphi_1'')$ by definition of $\mathbf{A}_k(\varphi_1)$. Hence $\mathbf{A}_l([\eta/c_2]\, \varphi_2'')$ by induction, so $\mathbf{A}_k((c_2 :^{\square} \varphi_2') \to \varphi_2'')$ as required.

If $\varphi_1 = (x_1 :^{\curlywedge} \tau_1) \to \varphi_1'$ then $\varphi_2 = (x_2 :^{\curlywedge} \tau_2) \to \varphi_2'$. Now the assumptions imply $\mathbf{A}_k(\varphi_1')$ and $\mathbf{A}_k(\varphi_1' \sim \varphi_2')$, so $\mathbf{A}_k(\varphi_2')$ by induction, and hence $\mathbf{A}_k(\varphi_2)$.

Finally, if $\varphi_1 = (a_1 :^{\Upsilon} \kappa_1) \to \varphi_1'$ then $\varphi_2 = (a_2 :^{\Upsilon} \kappa_2) \to \varphi_2'$. To show $\mathbf{A}_k((a_2 :^{\Upsilon} \kappa_2) \to \varphi_2')$, suppose $\cdot \vdash \tau :^{\forall} \kappa_2$ and $\mathbf{A}_l(\tau \sim \tau)$ for some $l < k$. Now $\mathbf{A}_k(\varphi_1 \sim \varphi_2)$ implies $\mathbf{A}_k(\eta : \kappa_2 \sim \kappa_1)$ for some η. Moreover, $\mathbf{A}_l([\tau \rhd \eta / a_1]\, \varphi_1')$ and $\mathbf{A}_l([\tau \rhd \eta / a_1]\, \varphi_1' \sim [\tau/a_2]\, \varphi_2')$ follow from the assumptions, so $\mathbf{A}_l([\tau/a_2]\, \varphi_2')$ by induction. Hence $\mathbf{A}_k((a_2 :^{\Upsilon} \kappa_2) \to \varphi_2')$ as required. $\square$

The following result shows that the **step** coercion preserves compatibility.

Lemma 6.16 (Reduction preserves compatibility)**.** *If $\tau \xrightarrow{\text{kpush}} \upsilon$ and $\mathbf{A}_k(\tau \sim \tau)$ then $\mathbf{A}_{k-1}(\tau \sim \upsilon)$.*

Proof. By induction on k and the reduction step $\tau \longrightarrow \upsilon$.

Case $\boxed{\dfrac{\rho \longrightarrow \rho'}{\rho \rhd \eta \longrightarrow \rho' \rhd \eta}}$. If $\mathbf{A}_k(\rho \rhd \eta \sim \rho \rhd \eta)$ then $\mathbf{A}_k(\rho \sim \rho)$ and $\mathbf{A}_{k-1}(\eta : \varphi)$.

Hence $\mathbf{A}_{k-1}(\rho \sim \rho')$ by induction, so $\mathbf{A}_{k-1}(\rho \rhd \eta \sim \rho' \rhd \eta)$ as required.

Case $\boxed{\dfrac{\rho \longrightarrow \rho'}{\rho\,\rho'' \longrightarrow \rho'\,\rho''}}$. If $\mathbf{A}_k(\,\rho\,\rho'' \sim \rho\,\rho'')$ then $\mathbf{A}_k(\,\rho \sim \rho)$ so by induction $\mathbf{A}_{k-1}(\,\rho \sim \rho')$ and hence $\mathbf{A}_{k-1}(\,\rho\,\rho'' \sim \rho'\,\rho'')$.

Case $\boxed{\dfrac{\rho \xrightarrow{\text{kpush}} \rho'}{\mathbf{case}\,\rho\,\mathbf{of}\,\overline{br_j}^{\,j} \longrightarrow \mathbf{case}\,\rho'\,\mathbf{of}\,\overline{br_j}^{\,j}}}$. If $\mathbf{A}_k(\,\mathbf{case}\,\rho\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \mathbf{case}\,\rho\,\mathbf{of}\,\overline{br_i}^{\,i})$ then $\mathbf{A}_{k-1}(\,\mathbf{case}\,\rho'\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \mathbf{case}\,\rho'\,\mathbf{of}\,\overline{br_i}^{\,i})$ by definition. By Lemma D.11, there is τ with $\mathbf{case}\,\rho'\,\mathbf{of}\,\overline{br_i}^{\,i} \longrightarrow \tau$, and induction gives $\mathbf{A}_{k-2}(\,\mathbf{case}\,\rho'\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \tau)$. Hence by definition $\mathbf{A}_{k-1}(\,\mathbf{case}\,\rho\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \mathbf{case}\,\rho'\,\mathbf{of}\,\overline{br_i}^{\,i})$.

Case $\boxed{\dfrac{\varepsilon \xrightarrow{\text{kpush}} \varepsilon' \qquad br'_0 = br_0 \rhd \mathbf{step}\,\varepsilon \quad ... \quad br'_n = br_n \rhd \mathbf{step}\,\varepsilon}{\mathbf{dcase}\,\varepsilon\,\mathbf{of}\,br_0\,...\,br_n \longrightarrow \mathbf{dcase}\,\varepsilon'\,\mathbf{of}\,br'_0\,...\,br'_n}}$. Similar to previous case.

Case $\boxed{\dfrac{\mathsf{K}\,\Delta \to \rho \in \overline{br_i}^{\,i}}{\mathbf{case}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \longrightarrow [\delta/\Delta]\,\rho}}$.

If $\mathbf{A}_k(\mathbf{case}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \mathbf{case}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i})$ then $\mathbf{A}_{k-1}(\,[\delta/\Delta]\,\rho \sim [\delta/\Delta]\,\rho)$ by definition. If $[\delta/\Delta]\,\rho$ is computational, then proceed as in the previous two cases. If it is structural, then $\mathbf{A}_{k-1}(\,\mathbf{case}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \sim [\delta/\Delta]\,\rho)$ is immediate from the definition. If it is coerced, then unwrap coercions until a computational or structural type is reached, and the required property follows as before.

Case $\boxed{\dfrac{\mathsf{K}\,\Delta \to \rho \in \overline{br_i}^{\,i}}{\mathbf{dcase}\,\mathsf{K}\,\psi\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \longrightarrow [(\delta, \langle \mathsf{K}\,\psi\,\delta\rangle)/\Delta]\,\rho}}$. Similar to previous case.

Case $\boxed{\dfrac{\Sigma \ni f\,[\mathbf{\Delta}] = \rho :^{\Phi} \kappa}{f(\delta) \longrightarrow [\delta/\Delta]\,\rho}}$. Similar to previous case.

Case $\boxed{\dfrac{\Gamma \vdash \gamma :^{\square} ((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2) \qquad \gamma_0 = \mathbf{sym}\,(\mathbf{left}\,\gamma) \qquad \gamma_1 = \gamma@(\mathbf{coh}\,\langle\tau\rangle\,\gamma_0)}{(v \rhd \gamma)^{\Upsilon}\tau \longrightarrow v^{\Upsilon}(\tau \rhd \gamma_0) \rhd \gamma_1}}$.

If $\mathbf{A}_k(\,(v \rhd \gamma)\,\tau \sim (v \rhd \gamma)\,\tau)$ then the definition gives $\mathbf{A}_k(\,v \sim v)$, $\mathbf{A}_k(\tau \sim \tau)$ and $\mathbf{A}_{k-1}(\gamma : (a_1 :^{\Upsilon} \kappa_1) \to \tau_1 \sim (a_2 :^{\Upsilon} \kappa_2) \to \tau_2)$. Hence $\mathbf{A}_{k-1}(\,v \rhd \gamma \sim v)$. Now $\mathbf{A}_{k-1}(\gamma_0 : \kappa_2 \sim \kappa_1)$, so $\mathbf{A}_{k-1}(\tau \sim \tau \rhd \gamma_0)$ and $\mathbf{A}_{k-2}(\gamma_1 : [\tau \rhd \gamma_0/a_1]\,\tau_1 \sim [\tau/a_2]\,\tau_2)$. Thus $\mathbf{A}_{k-1}(\,(v \rhd \gamma)\,\tau \sim v\,(\tau \rhd \gamma_0) \rhd \gamma_1)$.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \gamma \ :^{\Box}\ ((c_1 :^{\Box} \varphi_1) \to \tau_1) \sim ((c_2 :^{\Box} \varphi_2) \to \tau_2) \\ \gamma_0 = \mathbf{sym}\,(\mathbf{left}\,\gamma) \qquad\qquad \gamma_1 = \gamma @ (\eta \triangleright \gamma_0, \eta)\end{array}}{(v \triangleright \gamma)^{\Box}\eta \longrightarrow v^{\Box}(\eta \triangleright \gamma_0) \triangleright \gamma_1}}$. If $\mathbf{A}_k(\,(v \triangleright \gamma)\,\eta \sim (v \triangleright \gamma)\,\eta)$ then $\mathbf{A}_k(\,v \sim v)$, $\mathbf{A}_{k-1}(\eta : \varphi_2)$ and $\mathbf{A}_{k-1}(\gamma : (c_1 :^{\Box} \varphi_1) \to \tau_1 \sim (c_2 :^{\Box} \varphi_2) \to \tau_2)$. Hence $\mathbf{A}_{k-1}(\,v \triangleright \gamma \sim v)$. Now $\mathbf{A}_{k-1}(\gamma_0 : \varphi_2 \sim \varphi_1)$, so $\mathbf{A}_{k-2}(\eta \triangleright \gamma_0 : \varphi_1)$ by Lemma D.12, and $\mathbf{A}_{k-2}(\gamma_1 : [\eta \triangleright \gamma_0 / c_1]\,\tau_1 \sim [\eta / c_2]\,\tau_2)$. From this it follows that $\mathbf{A}_{k-1}(\,(v \triangleright \gamma)\,\eta \sim v\,(\eta \triangleright \gamma_0) \triangleright \gamma_1)$.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \gamma \ :^{\Box}\ ((a_1 :^{\curlywedge} \kappa_1) \to \tau_1) \sim ((a_2 :^{\curlywedge} \kappa_2) \to \tau_2) \\ \gamma_0 = \mathbf{sym}\,(\mathbf{left}\,\gamma) \qquad\qquad \gamma_1 = \mathbf{right}\,\gamma\end{array}}{(v \triangleright \gamma)^{\curlywedge}\rho \longrightarrow v^{\curlywedge}(\rho \triangleright \gamma_0) \triangleright \gamma_1}}$. If $\mathbf{A}_k(\,(v \triangleright \gamma)\,\rho \sim (v \triangleright \gamma)\,\rho)$ then $\mathbf{A}_k(\,v \sim v)$, $\mathbf{A}_k(\,\rho \sim \rho)$ and $\mathbf{A}_{k-1}(\gamma : (a_1 :^{\curlywedge} \kappa_1) \to \tau_1 \sim (a_2 :^{\curlywedge} \kappa_2) \to \tau_2)$. Hence $\mathbf{A}_{k-1}(\,v \triangleright \gamma \sim v)$. Now $\mathbf{A}_{k-1}(\gamma_0 : \kappa_2 \sim \kappa_1)$ and $\mathbf{A}_{k-2}(\gamma_1 : \tau_1 \sim \tau_2)$. Hence $\mathbf{A}_{k-1}(\,\rho \sim \rho \triangleright \gamma_0)$ and so $\mathbf{A}_{k-1}(\,(v \triangleright \gamma)\,\rho \sim v\,(\rho \triangleright \gamma_0) \triangleright \gamma_1)$.

Case $\boxed{\dfrac{}{(v \triangleright \gamma) \triangleright \gamma' \longrightarrow v \triangleright (\gamma; \gamma')}}$. If $\mathbf{A}_k(\,(v \triangleright \gamma) \triangleright \gamma' \sim (v \triangleright \gamma) \triangleright \gamma')$ then $\mathbf{A}_k(\,v \sim v)$, $\mathbf{A}_{k-1}(\gamma : \tau_0 \sim \tau_1)$ and $\mathbf{A}_{k-1}(\gamma' : \tau_1 \sim \tau_2)$. Transitivity gives $\mathbf{A}_{k-1}(\tau_0 \sim \tau_2)$ and hence $\mathbf{A}_k(\,(v \triangleright \gamma) \triangleright \gamma' \sim v \triangleright (\gamma; \gamma'))$.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \gamma \ :^{\Box}\ \mathsf{D}\,\overline{\tau_i}^{\,i} \sim \mathsf{D}\,\overline{\upsilon_i}^{\,i} \\ \Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i} \\ \omega = \overline{(\tau_i, \upsilon_i, \mathbf{nth}^i\,\gamma)}^{\,i} : \overline{a_i :^{\forall} \kappa_i}^{\,i} \ \prec\ \delta : \Delta\end{array}}{(\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta) \triangleright \gamma \xrightarrow{\text{kpush}} \mathsf{K}\,\overline{\upsilon_i}^{\,i}\,\overrightarrow{\omega}}}$.

If $\mathbf{A}_k((\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta) \triangleright \gamma \sim (\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta) \triangleright \gamma)$ then the definition gives $\mathbf{A}_k(\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta \sim \mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta)$ and $\mathbf{A}_{k-1}(\gamma : \mathsf{D}\,\overline{\tau_i}^{\,i} \sim \mathsf{D}\,\overline{\upsilon_i}^{\,i})$. Lemma D.10 gives $\mathbf{A}_{k-1}(\overline{(\tau_i, \upsilon_i, \mathbf{nth}^i\,\gamma)}^{\,i} : \overline{a_i :^{\forall} \kappa_i}^{\,i})$ and $\mathbf{A}_{k-1}(\overline{(\tau_i, \upsilon_i, \mathbf{nth}^i\,\gamma)}^{\,i}, \omega : \overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta)$ follows from the definition on coerced types since telescoped coercion extension appends coerced copies of types. Hence $\mathbf{A}_{k-1}(\mathsf{K}\,\overline{\tau_i}^{\,i}\,\overleftarrow{\omega} \sim \mathsf{K}\,\overline{\upsilon_i}^{\,i}\,\overrightarrow{\omega})$ and so $\mathbf{A}_{k-1}((\mathsf{K}\,\overline{\tau_i}^{\,i}\,\delta) \triangleright \gamma \sim \mathsf{K}\,\overline{\upsilon_i}^{\,i}\,\overrightarrow{\omega})$ as required. □

To prove congruence for case analysis, I need that whenever an expression is equivalent to an applied constructor, the expression reduces to the same head constructor (possibly under a coercion). This follows from the definition of compatibility on structural expressions.

Lemma D.13. *If* $\mathbf{A}_k(\mathsf{H}\,\delta \sim \tau)$ *then either* $\tau \longrightarrow^* \mathsf{H}\,\delta'$ *or* $\tau \longrightarrow^* \mathsf{H}\,\delta' \triangleright \gamma$*, and* $\mathbf{A}_k(\mathsf{H}\,\delta \sim \mathsf{H}\,\delta')$.

Proof. By induction on the length of δ and the structure of τ. $\square$

Lemma 6.17 (Congruence for case analysis)**.** *If* $\mathbf{A}_k(\varepsilon \sim \varepsilon')$ *and* $\mathbf{A}_k(br_i \approx br'_i)$ *for all* i*, then* $\mathbf{A}_k((\mathbf{d})\mathbf{case}\,\varepsilon\,\mathbf{of}\,\overline{br_i}^{\,i} \sim (\mathbf{d})\mathbf{case}\,\varepsilon'\,\mathbf{of}\,\overline{br'_i}^{\,i})$.

Proof. By induction on k, ε and ε'.

If $\varepsilon \xrightarrow{\text{kpush}} \varepsilon_0$ and $\varepsilon' \xrightarrow{\text{kpush}} \varepsilon'_0$, then Lemma 6.16 (page 135) and transitivity give $\mathbf{A}_{k-1}(\varepsilon_0 \sim \varepsilon'_0)$, and $\mathbf{A}_{k-1}((\mathbf{d})\mathbf{case}\,\varepsilon_0\,\mathbf{of}\,\overline{br_i}^{\,i} \sim (\mathbf{d})\mathbf{case}\,\varepsilon'_0\,\mathbf{of}\,\overline{br'_i}^{\,i})$ by induction, so the result follows.

Suppose without loss of generality that ε cannot step, then by Lemma D.11 either $k = 0$ (and the result is trivial) or ε is a value. It cannot have an outermost coercion, since Lemma D.13 ensures the case scrutinee push step would be applicable. The canonical forms lemma (Lemma 6.12) means that $\varepsilon = \mathsf{K}\,\overline{\tau_j}^{\,i}\,\delta$. By Lemma D.13, $\varepsilon' \longrightarrow^* \mathsf{K}\,\overline{\tau'_j}^{\,i}\,\delta'$ and $\mathbf{A}_k(\mathsf{K}\,\overline{\tau_j}^{\,i}\,\delta \sim \mathsf{K}\,\overline{\tau'_j}^{\,i}\,\delta')$.

For **case** expressions, there are $\mathsf{K}\,\Delta_0 \to \tau_0 \in \overline{br_i}^{\,i}$ and $\mathsf{K}\,\Delta'_0 \to \tau'_0 \in \overline{br'_i}^{\,i}$, so

$$\mathbf{case}\,\mathsf{K}\,\overline{\tau_j}^{\,i}\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \longrightarrow [\delta/\Delta_0]\,\tau_0 \text{ and } \mathbf{case}\,\mathsf{K}\,\overline{\tau'_j}^{\,i}\,\delta'\,\mathbf{of}\,\overline{br'_i}^{\,i} \longrightarrow [\delta'/\Delta'_0]\,\tau'_0.$$

Moreover $\mathbf{A}_k(\mathsf{K}\,\Delta_0 \to \tau_0 \approx \mathsf{K}\,\Delta'_0 \to \tau'_0)$ gives $\mathbf{A}_k((\Delta_0 \mathrel{/\!\!\!\wedge\!\!\!\backslash} \Delta'_0) \to (\tau_0 \sim \tau'))$. Instantiating this with δ and δ' yields $\mathbf{A}_{k-1}([\delta/\Delta_0]\,\tau_0 \sim [\delta'/\Delta'_0]\,\tau'_0)$, so

$$\mathbf{A}_k(\mathbf{case}\,\mathsf{K}\,\overline{\tau_j}^{\,i}\,\delta\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \mathbf{case}\,\mathsf{K}\,\overline{\tau'_j}^{\,i}\,\delta'\,\mathbf{of}\,\overline{br'_i}^{\,i}).$$

Now $\mathbf{A}_k(\mathbf{case}\,\varepsilon'\,\mathbf{of}\,\overline{br'_i}^{\,i} \sim \mathbf{case}\,\mathsf{K}\,\overline{\tau'_j}^{\,i}\,\delta'\,\mathbf{of}\,\overline{br'_i}^{\,i})$ follows from Lemma 6.16 since the left side reduces to the right side, so $\mathbf{A}_k(\mathbf{case}\,\varepsilon\,\mathbf{of}\,\overline{br_i}^{\,i} \sim \mathbf{case}\,\varepsilon'\,\mathbf{of}\,\overline{br'_i}^{\,i})$.

The argument for **dcase** expressions is similar: δ and δ' are replaced with $\delta, \langle\mathsf{K}\,\overline{\tau_j}^{\,i}\,\delta\rangle$ and $\delta', \langle\mathsf{K}\,\overline{\tau'_j}^{\,i}\,\delta'\rangle$; proof irrelevance means nothing more is needed. $\square$

If δ is a vector then let $\langle\!\langle\delta\rangle\!\rangle$ be the telescoped coercion with $\overleftarrow{\langle\!\langle\delta\rangle\!\rangle} = \delta = \overrightarrow{\langle\!\langle\delta\rangle\!\rangle}$ and the coercion proofs given by reflexivity. Note that $\Gamma \vdash \delta : \Delta$ is equivalent to $\Gamma \vdash^{\text{tc}} \langle\!\langle\delta\rangle\!\rangle : \Delta$.

Finally, I can prove the key result, that well-typed coercions are compatible. This is a massive mutual structural induction on typing derivations, using the preceding results. Unlike most of the previous results, however, k is quantified inside the inductive hypothesis, because some cases need to increase it when making appeals to induction.

Lemma 6.19 (Basic Lemma)**.**

(a) If $\Gamma \vdash \tau :^{\forall} \kappa$ *then for all* k, $\mathbf{A}_k(\omega_0 : \Gamma)$ *implies* $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\overrightarrow{\omega_0}/\Gamma]\,\tau)$.

(b) If $\Gamma \vdash br \;:^{\forall} \upsilon \blacktriangleright \tau$ *or* $\Gamma \vdash br \;:^{\forall} (\varepsilon : \upsilon) \blacktriangleright \tau$ *then for all* k, $\mathbf{A}_k(\omega_0 : \Gamma)$ *implies* $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,br \approx [\overrightarrow{\omega_0}/\Gamma]\,br)$.

(c) If $\Gamma \vdash \gamma :^{\Box} \varphi$ *then for all* k, $\mathbf{A}_k(\omega_0 : \Gamma)$ *implies* $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\varphi)$ *and* $\mathbf{A}_k([\overrightarrow{\omega_0}/\Gamma]\,\varphi)$.

(d) If $\Gamma \vdash^{\mathrm{tc}} \omega : \Delta$ *then for all* k, $\mathbf{A}_k(\omega_0 : \Gamma)$ *implies* $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$.

Proof of part (a). Fix k and ω_0 such that $\mathbf{A}_k(\omega_0 : \Gamma)$. Proceed by induction on the derivation of $\Gamma \vdash \tau :^{\forall} \kappa$ to show $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\overrightarrow{\omega_0}/\Gamma]\,\tau)$.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \mathbf{ctx}\\ \Gamma \ni a :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash a :^{\Psi} \kappa}}$. Here $\mathbf{A}_k(\omega_0 : \Gamma)$ gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,a \sim [\overrightarrow{\omega_0}/\Gamma]\,a)$.

Cases $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \mathbf{ctx}\\ \Sigma \ni \mathsf{D} :^{\forall} \kappa\end{array}}{\Gamma \vdash \mathsf{D} :^{\forall} \kappa}}$ **and** $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \mathbf{ctx}\\ \Sigma \ni \mathsf{K} :^{\Phi} \kappa \qquad \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash \mathsf{K} :^{\Psi} \kappa}}$. Trivial.

Case $\boxed{\dfrac{\Sigma \ni f\,[\boldsymbol{\Delta}] :^{\Phi} \kappa \qquad \Gamma \vdash \delta : \Delta \mathbin{/\!\!/} \Psi \qquad \Phi \hookrightarrow \Psi}{\Gamma \vdash f(\delta) :^{\Psi} [\delta/\Delta]\,\kappa}}$. Let $\omega = [\omega_0/\Gamma]\langle\!\langle\delta\rangle\!\rangle$ so that $\overleftarrow{\omega} = [\overleftarrow{\omega_0}/\Gamma]\,\delta$ and $\overrightarrow{\omega} = [\overrightarrow{\omega_0}/\Gamma]\,\delta$. Then the goal $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,f(\delta) \sim [\overrightarrow{\omega_0}/\Gamma]\,f(\delta))$ is $\mathbf{A}_k(f(\overleftarrow{\omega}) \sim f(\overrightarrow{\omega}))$. Now $f(\overleftarrow{\omega}) \longrightarrow [\overleftarrow{\omega}/\Delta]\,\tau$ and $f(\overrightarrow{\omega}) \longrightarrow [\overrightarrow{\omega}/\Delta]\,\tau$ where $\Sigma \ni f\,[\boldsymbol{\Delta}] = \tau :^{\Phi} \kappa$. Moreover, induction using part (d) gives $\mathbf{A}_{k-1}(\omega : \Delta)$, and $\mathbf{A}_{k-1}([\overleftarrow{\omega}/\Delta]\,\tau \sim [\overrightarrow{\omega}/\Delta]\,\tau)$ follows since the function definition is good. Hence $\mathbf{A}_k(f(\overleftarrow{\omega}) \sim f(\overrightarrow{\omega}))$ as required.

Case $\boxed{\dfrac{\Gamma \vdash \rho :^{\Psi} (a{:}^{\Phi}\kappa_1) \to \kappa_2 \qquad \Gamma \vdash \rho' :^{\Phi \mathbin{/\!\!/} \Psi} \kappa_1}{\Gamma \vdash \rho^{\Phi}\rho' :^{\Psi} [\rho'/a]\,\kappa_2}}$.

By induction, $\mathbf{A}_k(\,[\overleftarrow{\omega_0}/\Gamma]\,\rho \sim [\overrightarrow{\omega_0}/\Gamma]\,\rho)$ and $\mathbf{A}_k(\,[\overleftarrow{\omega_0}/\Gamma]\,\rho' \sim [\overrightarrow{\omega_0}/\Gamma]\,\rho')$. Now $\mathbf{A}_{k-1}([\overleftarrow{\omega_0}/\Gamma]\,((a :^{\Phi} \kappa_1) \to \kappa_2) \sim [\overrightarrow{\omega_0}/\Gamma]\,((a :^{\Phi} \kappa_1) \to \kappa_2))$ by Lemma 6.18, so the definition on quantifiers gives $\mathbf{A}_{k-1}([\overleftarrow{\omega_0}/\Gamma]\,([\rho'/a]\,\kappa_2) \sim [\overrightarrow{\omega_0}/\Gamma]\,([\rho'/a]\,\kappa_2))$. Hence $\mathbf{A}_k(\,[\overleftarrow{\omega_0}/\Gamma]\,(\rho\,\rho') \sim [\overrightarrow{\omega_0}/\Gamma]\,(\rho\,\rho'))$ as required.

Case $\boxed{\dfrac{\Gamma \vdash \kappa \;:^{\forall}\; * \qquad \Gamma, a :^{\Phi} \kappa \vdash \tau \;:^{\forall}\; *}{\Gamma \vdash (a{:}^{\Phi}\kappa) \to \tau \;:^{\forall}\; *}}$.

To show $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,((a :^{\Phi} \kappa) \to \tau) \sim [\overrightarrow{\omega_0}/\Gamma]\,((a :^{\Phi} \kappa) \to \tau))$, first observe that $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\kappa \sim [\overrightarrow{\omega_0}/\Gamma]\,\kappa)$ follows by induction. Suppose $l < k$, υ and υ' with $\mathbf{A}_l(\upsilon \sim \upsilon')$, then $\mathbf{A}_l([\upsilon/a]\,[\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\upsilon'/a]\,[\overrightarrow{\omega_0}/\Gamma]\,\tau)$ also follows by induction.

Case $\boxed{\dfrac{\Gamma \vdash \rho \;:^{\Psi}\; \kappa \qquad \Gamma \vdash \gamma \;:^{\Box}\; \kappa \sim \kappa' \qquad \Psi \neq \Box}{\Gamma \vdash \rho \rhd \gamma \;:^{\Psi}\; \kappa'}}$.

Here $\mathbf{A}_k(\,[\overleftarrow{\omega_0}/\Gamma]\,\rho \sim [\overrightarrow{\omega_0}/\Gamma]\,\rho)$ by induction, and $\mathbf{A}_{k-1}([\overleftarrow{\omega_0}/\Gamma]\,(\kappa \sim \kappa'))$ and $\mathbf{A}_{k-1}([\overrightarrow{\omega_0}/\Gamma]\,(\kappa \sim \kappa'))$ by part (c). Hence $\mathbf{A}_k(\,[\overleftarrow{\omega_0}/\Gamma]\,(\rho \rhd \gamma) \sim [\overrightarrow{\omega_0}/\Gamma]\,(\rho \rhd \gamma))$.

Cases $\boxed{\dfrac{\Gamma \vdash \mathbf{ctx}}{\Gamma \vdash * \;:^{\forall}\; *}}$ **and** $\boxed{\dfrac{\Gamma \vdash \mathbf{ctx}}{\Gamma \vdash (\sim) \;:^{\forall}\; (a{:}^{\forall}*) \to (b{:}^{\forall}*) \to a \to b \to *}}$. Trivial.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \rho \;:^{\Psi}\; \upsilon \qquad \Psi \neq \Box \\ \Gamma \vdash br_0 \;:^{\Psi}\; \upsilon \blacktriangleright \tau \quad \ldots \quad \Gamma \vdash br_n \;:^{\Psi}\; \upsilon \blacktriangleright \tau\end{array}}{\Gamma \vdash \mathbf{case}\,\rho\,\mathbf{of}\,br_0 \ldots br_n \;:^{\Psi}\; \tau}}$. By induction, using part (b), and congruence for case analysis (Lemma 6.17).

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \varepsilon \;:^{\Pi /\!\!/ \Psi}\; \upsilon \qquad \Psi \neq \Box \\ \Gamma \vdash br_0 \;:^{\Psi}\; (\varepsilon : \upsilon) \blacktriangleright \tau \quad \ldots \quad \Gamma \vdash br_n \;:^{\Psi}\; (\varepsilon : \upsilon) \blacktriangleright \tau\end{array}}{\Gamma \vdash \mathbf{dcase}\,\varepsilon\,\mathbf{of}\,br_0 \ldots br_n \;:^{\Psi}\; \tau}}$. As previous case.

$\Box$

Proof of part (b). Fix k and ω_0 such that $\mathbf{A}_k(\omega_0 : \Gamma)$. Proceed by induction on the derivation to show $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,br \approx [\overrightarrow{\omega_0}/\Gamma]\,br)$.

Case $\boxed{\dfrac{\begin{array}{l}\Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\,\overline{a_i}^{\,i} \\ \Gamma, [\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\Delta \vdash \rho \;:^{\Psi}\; \tau \\ \Gamma \vdash \tau \;:^{\forall}\; * \qquad \Phi \hookrightarrow \Psi\end{array}}{\Gamma \vdash \mathsf{K}\,([\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\Delta) \to \rho \;:^{\Psi}\; \mathsf{D}\,\overline{\upsilon_i}^{\,i} \blacktriangleright \tau}}$. First let $\Delta' = [\,\overline{\upsilon_i/a_i}^{\,i}\,]\,\Delta$ and $\Delta'' = [\overleftarrow{\omega_0}/\Gamma]\,\Delta' \mathrel{\wedge\!\!\!\wedge} [\overrightarrow{\omega_0}/\Gamma]\,\Delta'$. The goal is $\mathbf{A}_k(\,(\Delta'') \to [\overleftarrow{\omega_0}/\Gamma]\,\rho \sim [\overrightarrow{\omega_0}/\Gamma]\,\rho)$. Equivalently, suppose ω is such that $\mathbf{A}_k(\omega_0, \omega : \Gamma, \Delta')$, then it suffices to show $\mathbf{A}_k(\,[\overleftarrow{(\omega_0, \omega)}/\Gamma, \Delta']\,\rho \sim [\overrightarrow{(\omega_0, \omega)}/\Gamma, \Delta']\,\rho)$, which follows from part (a).

Case $\boxed{\begin{array}{l} \Sigma \ni \mathsf{K} :^{\Phi} (\overline{a_i :^{\forall} \kappa_i}^{\,i}, \Delta) \to \mathsf{D}\, \overline{a_i}^{\,i} \\ \Delta' = [\,\overline{\upsilon_i/a_i}^{\,i}\,]\, \Delta \mathbin{/\!\!/} \Pi, c :^{\Box} \varepsilon \sim (\mathsf{K}\, \overline{\upsilon_i}^{\,i} \Delta) \\ \Gamma, \Delta' \vdash \rho \ :^{\Psi}\ \tau \qquad \Gamma \vdash \tau\ :^{\forall}\ * \qquad \Phi \hookrightarrow \Pi \mathbin{/\!\!/} \Psi \\ \hline \multicolumn{1}{c}{\Gamma \vdash \mathsf{K}\, \Delta' \to \rho\ :^{\Psi}\ (\varepsilon : \mathsf{D}\, \overline{\upsilon_i}^{\,i}) \blacktriangleright \tau} \end{array}}$. Similar. $\Box$

Proof of part (c). Fix k and ω_0 such that $\mathbf{A}_k(\omega_0 : \Gamma)$. Proceed by induction on the derivation of $\Gamma \vdash \gamma\ :^{\Box}\ \varphi$ to show $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, \varphi)$. In each case, it is straightforward to further show $\mathbf{A}_k([\overrightarrow{\omega_0}/\Gamma]\, \varphi)$.

Case $\boxed{\begin{array}{l} \Gamma \ni c :^{\Box} \varphi \\ \Gamma \vdash \mathbf{ctx} \\ \hline \Gamma \vdash c\ :^{\Box}\ \varphi \end{array}}$. Here the definition of $\mathbf{A}_k(\omega_0 : \Gamma)$ gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, \varphi)$.

Case $\boxed{\begin{array}{l} \Gamma \vdash \gamma\ :^{\Box}\ (a{:}^{\Phi} \kappa) \to \varphi \\ \Gamma \vdash \tau\ :^{\forall}\ \kappa \qquad\quad \Phi \neq \Box \\ \hline \multicolumn{1}{c}{\Gamma \vdash \gamma^{\Phi}\tau\ :^{\Box}\ [\tau/a]\, \varphi} \end{array}}$. Here $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, ((a :^{\Phi} \kappa) \to \varphi))$ by induction, and part (a) gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, \tau \sim [\overrightarrow{\omega_0}/\Gamma]\, \tau)$, so $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, [\tau/a]\, \varphi)$ follows immediately from the definition.

Case $\boxed{\begin{array}{l} \Gamma \vdash \gamma\ :^{\Box}\ (c{:}^{\Box} \varphi') \to \varphi \\ \Gamma \vdash \eta\ :^{\Box}\ \varphi' \\ \hline \multicolumn{1}{c}{\Gamma \vdash \gamma^{\Box}\eta\ :^{\Box}\ [\eta/c]\, \varphi} \end{array}}$. Induction gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (c{:}^{\Box} \varphi') \to \varphi)$ from the first hypothesis and $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, \varphi')$ from the second, so $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, [\eta/c]\, \varphi)$ follows from the definition.

Case $\boxed{\begin{array}{c} \Gamma, a :^{\Phi} \kappa \vdash \gamma\ :^{\Box}\ \tau \\ \hline \Gamma \vdash \Lambda a{:}^{\Phi}\kappa \,.\, \gamma\ :^{\Box}\ (a{:}^{\Phi} \kappa) \to \tau \end{array}}$. First suppose $\Phi \neq \Box$, and let τ be such that $\cdot \vdash \tau\ :^{\forall}\ \kappa$ and $\mathbf{A}_k(\tau \sim \tau)$. Induction gives $\mathbf{A}_k([(\overleftarrow{\omega_0}, \tau)/\Gamma, a :^{\Phi} \kappa]\, \varphi)$, so $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, ((a{:}^{\Phi} \kappa) \to \varphi))$ as required. The case $\Phi = \Box$ is similar.

Case $\boxed{\begin{array}{c} \Gamma \vdash \mathbf{ctx} \qquad\quad \Sigma \ni C :^{\Box} \varphi \\ \hline \Gamma \vdash C\ :^{\Box}\ \varphi \end{array}}$. Here the goodness of Σ gives $\mathbf{A}_k(\varphi)$ and hence $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, \varphi)$ since φ is closed.

Case $\boxed{\dfrac{\Gamma \vdash^{\mathrm{tc}} \omega \,:\, \Delta \qquad \Gamma, \Delta \vdash \tau \,:^{\forall}\, \kappa}{\Gamma \vdash \mathbf{resp}\, \omega\, \Delta\, \tau \,:^{\square}\, [\overleftarrow{\omega}/\Delta]\, \tau \sim [\overrightarrow{\omega}/\Delta]\, \tau}}$. Part (d) gives $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$, then part (a) gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, ([\overleftarrow{\omega}/\Delta]\, \tau \sim [\overrightarrow{\omega}/\Delta]\, \tau))$ as required.

Case $\boxed{\dfrac{\Gamma \vdash \gamma \,:^{\square}\, \tau\, \tau' \sim \upsilon\, \upsilon'}{\Gamma \vdash \mathbf{left}\, \gamma \,:^{\square}\, \tau \sim \upsilon}}$. By induction, $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (\tau\, \tau' \sim \upsilon\, \upsilon'))$, and hence $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (\tau \sim \upsilon))$ by definition.

Case $\boxed{\dfrac{\Gamma \vdash \gamma \,:^{\square}\, \tau\, \tau' \sim \upsilon\, \upsilon'}{\Gamma \vdash \mathbf{right}\, \gamma \,:^{\square}\, \tau' \sim \upsilon'}}$. Similar to the previous case.

Case $\boxed{\dfrac{\Gamma \vdash \gamma \,:^{\square}\, ((a_1 :^{\Phi} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Phi} \kappa_2) \to \tau_2)}{\Gamma \vdash \mathbf{left}\, \gamma \,:^{\square}\, \kappa_1 \sim \kappa_2}}$.

By induction, $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (((a_1 :^{\Phi} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Phi} \kappa_2) \to \tau_2)))$, and hence $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (\kappa_1 \sim \kappa_2))$ by definition.

Case $\boxed{\dfrac{\Gamma \vdash \gamma \,:^{\square}\, (\kappa_1 \to \tau_1) \sim (\kappa_2 \to \tau_2)}{\Gamma \vdash \mathbf{right}\, \gamma \,:^{\square}\, \tau_1 \sim \tau_2}}$. Here the inductive hypothesis gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, ((\kappa_1 \to \tau_1) \sim (\kappa_2 \to \tau_2)))$, so $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (\tau_1 \sim \tau_2))$ by definition.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \gamma \,:^{\square}\, (\tau_1 : (a_1 :^{\Upsilon} \kappa_1) \to \kappa_1') \sim (\tau_2 : (a_2 :^{\Upsilon} \kappa_2) \to \kappa_2') \\ \Gamma \vdash \eta \,:^{\square}\, (\upsilon_1 : \kappa_1) \sim (\upsilon_2 : \kappa_2)\end{array}}{\Gamma \vdash \mathbf{conga}^{\Upsilon}\, \gamma\, \eta \,:^{\square}\, (\tau_1\, \upsilon_1) \sim (\tau_2\, \upsilon_2)}}$.

By induction, $\mathbf{A}_{k+1}([\overleftarrow{\omega_0}/\Gamma]\, (\tau_1 \sim \tau_2))$ and $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (\upsilon_1 \sim \upsilon_2))$. Moreover Lemma 6.18 gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (((a_1 :^{\Phi} \kappa_1) \to \kappa_1') \sim ((a_2 :^{\Phi} \kappa_2) \to \kappa_2')))$ and hence $\mathbf{A}_{k-1}([\overleftarrow{\omega_0}/\Gamma]\, ([\upsilon_1/a_1]\, \kappa_1' \sim [\upsilon_2/a_2]\, \kappa_2'))$. Thus $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\, (\tau_1\, \upsilon_1 \sim \tau_2\, \upsilon_2))$ as required. Note that this case relies on the fact that k is universally quantified inside the inductive hypothesis.

Case $\boxed{\dfrac{\begin{array}{l}\Gamma \vdash \gamma \,:^{\square}\, (\tau_1 : (c_1 :^{\square} \varphi_1) \to \kappa_1) \sim (\tau_2 : (c_2 :^{\square} \varphi_2) \to \kappa_2) \\ \Gamma \vdash \eta_1 \,:^{\square}\, \varphi_1 \qquad \Gamma \vdash \eta_2 \,:^{\square}\, \varphi_2\end{array}}{\Gamma \vdash \mathbf{conga}^{\square}\, \gamma\, (\eta_1,\, \eta_2) \,:^{\square}\, (\tau_1\, \eta_1) \sim (\tau_2\, \eta_2)}}$.

Similar to previous case.

Case $\boxed{\begin{array}{c} \Gamma, a_1 :^{\Upsilon} \kappa_1 \vdash \tau_1 :^{\forall} * \qquad \Gamma, a_2 :^{\Upsilon} \kappa_2 \vdash \tau_2 :^{\forall} * \qquad \Gamma \vdash \eta :^{\Box} \kappa_1 \sim \kappa_2 \\ \Gamma \vdash \gamma :^{\Box} (a_1 :^{\Upsilon} \kappa_1, a_2 :^{\Upsilon} \kappa_2, c :^{\Box} a_1 \sim a_2) \to \tau_1 \sim \tau_2 \\ \hline \Gamma \vdash \mathbf{cong}\, \Upsilon\, \eta\, \gamma :^{\Box} ((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2) \end{array}}$.

$\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(\kappa_1 \sim \kappa_2))$ and $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,((a_1 :^{\Phi} \kappa_1, a_2 :^{\Phi} \kappa_2, c :^{\Box} a_1 \sim a_2) \to \tau_1 \sim \tau_2))$ by induction. Hence $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(((a_1 :^{\Phi} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Phi} \kappa_2) \to \tau_2)))$.

Case $\boxed{\begin{array}{c} \Gamma, c_1 :^{\Box} \varphi_1 \vdash \tau_1 :^{\forall} * \qquad \Gamma, c_2 :^{\Box} \varphi_2 \vdash \tau_2 :^{\forall} * \\ \Gamma \vdash \eta :^{\Box} \varphi_1 \sim \varphi_2 \qquad \Gamma \vdash \gamma :^{\Box} (c_1 :^{\Box} \varphi_1, c_2 :^{\Box} \varphi_2) \to \tau_1 \sim \tau_2 \\ \hline \Gamma \vdash \mathbf{cong}\, \Box\, \eta\, \gamma :^{\Box} ((c_1 :^{\Box} \varphi_1) \to \tau_1) \sim ((c_2 :^{\Box} \varphi_2) \to \tau_2) \end{array}}$.

Similar to previous case.

Case $\boxed{\begin{array}{c} \Gamma \vdash \gamma :^{\Box} \varepsilon \sim \varepsilon' \qquad \Gamma \vdash \eta_0 :^{\Box} br_0 \approx br_0' \;\ldots\; \Gamma \vdash \eta_n :^{\Box} br_n \approx br_n' \\ \hline \Gamma \vdash (\mathbf{cong}\,(\mathbf{d})\mathbf{case}\, \gamma\, \overline{\eta_i}^{\,i}) :^{\Box} ((\mathbf{d})\mathbf{case}\, \varepsilon\, \mathbf{of}\, \overline{br_i}^{\,i}) \sim ((\mathbf{d})\mathbf{case}\, \varepsilon'\, \mathbf{of}\, \overline{br_i'}^{\,i}) \end{array}}$.

By induction and Lemma 6.17.

Case $\boxed{\begin{array}{c} \Gamma \vdash \gamma :^{\Box} \varphi \\ \Gamma \vdash \eta :^{\Box} \varphi \sim \varphi' \\ \hline \Gamma \vdash \gamma \rhd \eta :^{\Box} \varphi' \end{array}}$. By induction, $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\varphi)$ and $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(\varphi \sim \varphi'))$.

Then Lemma D.12 gives the required result.

Case $\boxed{\begin{array}{c} \Gamma \vdash \gamma :^{\Box} ((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2) \\ \Gamma \vdash \eta :^{\Box} (\upsilon_1 : \kappa_1) \sim (\upsilon_2 : \kappa_2) \\ \hline \Gamma \vdash \gamma @ \eta :^{\Box} [\upsilon_1/a_1]\,\tau_1 \sim [\upsilon_2/a_2]\,\tau_2 \end{array}}$.

Here induction gives $\mathbf{A}_{k+1}([\overleftarrow{\omega_0}/\Gamma]\,(((a_1 :^{\Upsilon} \kappa_1) \to \tau_1) \sim ((a_2 :^{\Upsilon} \kappa_2) \to \tau_2)))$ and $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(\upsilon_1 \sim \upsilon_2))$, so by definition, $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,([\upsilon_1/a_1]\,\tau_1 \sim [\upsilon_2/a_2]\,\tau_2))$.

Case $\boxed{\begin{array}{c} \Gamma \vdash \gamma :^{\Box} ((c_1 :^{\Box} \varphi_1) \to \tau_1) \sim ((c_2 :^{\Box} \varphi_2) \to \tau_2) \\ \Gamma \vdash \eta_1 :^{\Box} \varphi_1 \qquad \Gamma \vdash \eta_2 :^{\Box} \varphi_2 \\ \hline \Gamma \vdash \gamma @ (\eta_1, \eta_2) :^{\Box} [\eta_1/c_1]\,\tau_1 \sim [\eta_2/c_2]\,\tau_2 \end{array}}$. Similar to previous case.

Case $\boxed{\begin{array}{c} \Gamma \vdash \gamma :^{\Box} (\tau_1 : \kappa_1) \sim (\tau_2 : \kappa_2) \\ \Gamma \vdash \eta :^{\Box} \kappa_1 \sim \upsilon \\ \hline \Gamma \vdash \mathbf{coh}\, \gamma\, \eta :^{\Box} \tau_1 \rhd \eta \sim \tau_2 \end{array}}$. By induction, $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(\tau_1 \sim \tau_2))$ and $\mathbf{A}_{k-1}([\overleftarrow{\omega_0}/\Gamma]\,(\kappa_1 \sim \kappa_2))$. Hence $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(\tau_1 \rhd \eta \sim \tau_2))$ as required.

Case $\boxed{\dfrac{\Gamma \vdash \tau \,:^{\forall}\, \kappa \qquad \Gamma \vdash \tau' \,:^{\forall}\, \kappa \qquad \tau \xrightarrow{\text{kpush}} \tau'}{\Gamma \vdash \mathbf{step}\,\tau \,:^{\Box}\, \tau \sim \tau'}}$.

Here induction using part (a) gives $\mathbf{A}_{k+1}([\overleftarrow{\omega_0}/\Gamma]\,(\tau \sim \tau))$, and Lemma D.9 gives $[\overleftarrow{\omega_0}/\Gamma]\,\tau \xrightarrow{\text{kpush}} [\overleftarrow{\omega_0}/\Gamma]\,\tau'$, so Lemma 6.16 gives $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,(\tau \sim \tau'))$.

Case $\boxed{\dfrac{\Gamma \vdash \gamma \,:^{\Box}\, (\tau_1 \!:\! \kappa_1) \sim (\tau_2 \!:\! \kappa_2)}{\Gamma \vdash \mathbf{kind}\,\gamma \,:^{\Box}\, \kappa_1 \sim \kappa_2}}$. By induction and Lemma 6.18.

□

Proof of part (d). Fix k and ω_0 such that $\mathbf{A}_k(\omega_0 : \Gamma)$. Proceed by induction on the derivation of $\Gamma \vdash^{\text{tc}} \omega : \Delta$ to show $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$.

Case $\boxed{\dfrac{\Gamma \vdash \ \mathbf{ctx}}{\Gamma \vdash^{\text{tc}} \cdot \,:\, \cdot}}$. Trivial.

Case $\boxed{\dfrac{\begin{array}{cc}\Gamma \vdash^{\text{tc}} \omega \,:\, \Delta & \Gamma \vdash \gamma \,:^{\Box}\, \tau \sim \upsilon \\ \Gamma \vdash \tau \,:^{\Upsilon}\, [\overleftarrow{\omega}/\Delta]\,\kappa & \Gamma \vdash \upsilon \,:^{\Upsilon}\, [\overrightarrow{\omega}/\Delta]\,\kappa\end{array}}{\Gamma \vdash^{\text{tc}} (\omega, (\tau, \upsilon, \gamma)) \,:\, (\Delta, a \,:^{\Upsilon}\, \kappa)}}$. Here $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$ by induction, and $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\overleftarrow{\omega_0}/\Gamma]\,\upsilon)$ and $\mathbf{A}_k([\overrightarrow{\omega_0}/\Gamma]\,\tau \sim [\overrightarrow{\omega_0}/\Gamma]\,\upsilon)$ from part (c). Moreover $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\overrightarrow{\omega_0}/\Gamma]\,\tau)$ from part (a). Hence symmetry and transitivity give $\mathbf{A}_k([\overleftarrow{\omega_0}/\Gamma]\,\tau \sim [\overrightarrow{\omega_0}/\Gamma]\,\upsilon)$ as required.

Case $\boxed{\dfrac{\begin{array}{c}\Gamma \vdash^{\text{tc}} \omega \,:\, \Delta \\ \Gamma \vdash \eta \,:^{\Box}\, [\overleftarrow{\omega}/\Delta]\,\varphi \\ \Gamma \vdash \eta' \,:^{\Box}\, [\overrightarrow{\omega}/\Delta]\,\varphi\end{array}}{\Gamma \vdash^{\text{tc}} (\omega, (\eta, \eta')) \,:\, (\Delta, c \,:^{\Box}\, \varphi)}}$. Let $\omega' = \omega_0, [\omega_0/\Gamma]\omega$. By induction, $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$, $\mathbf{A}_{k-1}([\overleftarrow{\omega'}/\Gamma, \Delta]\,\varphi)$ and $\mathbf{A}_{k-1}([\overrightarrow{\omega'}/\Gamma, \Delta]\,\varphi)$ as required.

Case $\boxed{\dfrac{\begin{array}{c}\Gamma \vdash^{\text{tc}} \omega \,:\, \Delta \\ \Gamma \vdash e \,:^{⅄}\, [\overleftarrow{\omega}/\Delta]\,\tau \\ \Gamma \vdash e' \,:^{⅄}\, [\overrightarrow{\omega}/\Delta]\,\tau\end{array}}{\Gamma \vdash^{\text{tc}} (\omega, (e, e')) \,:\, (\Delta, x \,:^{⅄}\, \tau)}}$. By induction, $\mathbf{A}_k([\omega_0/\Gamma]\omega : \Delta)$. □

Bibliography

Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Typed Lambda Calculi and Applications (TLCA '11)*, pages 10–26. Springer, 2011.

Alfonso Acosta. ForSyDe tutorial, 2008. URL `http://www.ict.kth.se/forsyde/files/tutorial/`.

W. E. Aitken and J. H. Reppy. Abstract value constructors. Technical Report 92-1290, Department of Computer Science, Cornell University, 1992.

Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Unpublished manuscript, 2005. URL `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`.

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming Languages meets Program Verification (PLPV '07)*, pages 57–68. ACM, 2007.

Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture (FPLCA '85)*, LNCS 201, pages 368–381. Springer, 1985.

Lennart Augustsson and Kent Petersson. Silly type families. Unpublished manuscript, 1994. URL `http://web.cecs.pdx.edu/~sheard/papers/silly.pdf`.

Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, LNCS 2277, pages 24–40. Springer, 2002.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.

Jason J. Brown. *Presentations of Unification in a Logical Framework*. PhD thesis, University of Oxford, 1996.

Björn Buckwalter. The `numtype` package, 2009. URL `http://hackage.haskell.org/package/numtype`. Haskell package.

Björn Buckwalter. Dimensional — statically checked physical dimensions for Haskell, n.d.. URL `http://dimensional.googlecode.com/`.

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*, pages 241–253. ACM, 2005.

James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ETT. In *Trends in Functional Programming (TFP '05)*, pages 79–94, 2005.

Chiyan Chen. *Type inference in applied type system*. PhD thesis, Boston University, 2006.

Feng Chen, Grigore Roşu, and Ram Prasad Venkatesan. Rule-based analysis of dimensional safety. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA '03)*, LNCS 2706, pages 197–207. Springer, 2003.

James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003.

Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM conference on LISP and Functional Programming (LFP '86)*, pages 13–27. ACM, 1986.

Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.4*, 2013. URL `http://coq.inria.fr/refman/`. Software manual.

Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996.

Luís Damas. Unpublished manuscript, 1984.

Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '82)*, pages 207–212. ACM, 1982.

Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '08)*, pages 133–144. ACM, 2008.

N.G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, 1991.

Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, pages 143–155, 2011.

Iavor Diatchki. The `presburger` package, 2011. URL `http://hackage.haskell.org/package/presburger`. Haskell package.

Iavor Diatchki. Type-level naturals, n.d.. URL `http://hackage.haskell.org/trac/ghc/wiki/TypeNats`.

Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming (JICSLP '96)*. MIT Press, 1996.

Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206(1–2):1–50, 1998.

Joshua Dunfield. Greedy bidirectional polymorphism. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML (ML '09)*, pages 15–26, 2009. URL `http://www.cs.cmu.edu/~joshuad/papers/poly/`.

Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*, pages 429–442. ACM, 2013.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.

Frederik Eaton. Statically typed linear algebra in Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell (Haskell '06)*, pages 120–121. ACM, 2006.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 symposium on Haskell (Haskell '12)*, pages 117–130. ACM, 2012.

Linus Ek, Ola Holmström, and Stevan Andjelkovic. Formalizing Arne Andersson trees and left-leaning red-black trees in Agda. Unpublished manuscript, 2011. URL `http://web.student.chalmers.se/groups/datx02-dtp/`.

Conal Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. URL `http://conal.net/papers/elliott90.pdf`.

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *21st International Symposium on Implementation and Application of Functional Languages (IFL '09)*, LNCS 6041. Springer, 2009.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge University Press, 1989. ISBN 0-521-37181-3.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation*, LNCS 4060, pages 521–540. Springer, 2006.

Benjamin Gregoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS 3603, pages 98–113. Springer, 2005.

Adam Gundry. Type inference for units of measure. In Ricardo Peña and Marko van Eekelen, editors, *Draft Proceedings of the 12th International Symposium on Trends in Functional Programming (TFP '11)*, pages

17–35, 2011. URL `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-7-11.pdf`.

Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically Structured Functional Programming (MSFP '10)*, pages 43–54. ACM, 2010.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

Stefan Holdermans. The `signed-multiset` package, 2013. URL `http://hackage.haskell.org/package/signed-multiset`. Haskell package.

Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Programming Languages and Systems*, LNCS 1782, pages 230–244. Springer, 2000.

Stefan Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11(4):425–432, July 2001.

Andrew Kennedy. Type inference and equational theories. Research Report LIX/RR/96/09, École Polytechnique, 1996a. URL `http://research.microsoft.com/en-us/um/people/akenn/other/EqnTypeInf.ps`.

Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996b. URL `http://research.microsoft.com/en-us/um/people/akenn/units/ProgrammingLanguagesAndDimensions.pdf`.

Andrew Kennedy. Types for units-of-measure: Theory and practice. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming (CEFP '09)*, LNCS 6299, pages 268–305. Springer, 2010.

Oleg Kiselyov. Number-parameterized types. *The Monad.Reader*, 5, 2005. URL `http://okmij.org/ftp/Haskell/number-parameterized-types.html`.

Oleg Kiselyov. How OCaml type checker works — or what polymorphism and garbage collection have in common, February 2013. URL `http://okmij.org/ftp/ML/generalization.html`.

George Kuan and David MacQueen. Efficient ML type inference using ranked type variables. In Claudio V. Russo and Derek Dreyer, editors, *Proceedings of the 2007 workshop on ML (ML '07)*, pages 3–14. ACM, 2007.

Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications (ML '92)*. ACM, 1992.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '00)*, pages 108–118. ACM, 2000.

Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In *Proceedings of the 2004 international conference on Types for Proofs and Programs (TYPES '04)*, pages 154–169. Springer, 2006.

Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed Haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*, pages 81–92. ACM, 2013.

Andres Löh and José Pedro Magalhães. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming (WGP '11)*, pages 1–12. ACM, 2011.

Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta Informaticæ*, 102(2): 177–207, 2010.

Marko Luther. *Elaboration and Erasure in Type Theory.* PhD thesis, Universität Ulm, Germany, 2003. URL `ftp://ftp.informatik.uni-ulm.de/pub/KI/papers/luther03-diss.pdf`.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic*

Colloquium, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984. ISBN 88-7088-105-9. Notes by Giovanni Sambin.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, May 1996.

Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory.* Oxford University Press, 1998. ISBN 9780198501275.

Bruce J. McAdam. On the unification of substitutions in type inference. In *Implementation of Functional Languages (IFL' 98)*, pages 139–154. Springer, 1998.

Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999. URL `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

Conor McBride. The derivative of a regular type is its type of one-hole contexts, 2001. URL `http://strictlypositive.org/diff.pdf`. Unpublished manuscript.

Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12:375–392, 6 2002.

Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6), 2003.

Conor McBride. Clowns to the left of me, jokers to the right. Unpublished manuscript, 2008. URL `https://personal.cis.strath.ac.uk/conor.mcbride/Dissect.pdf`.

Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP '10)*, pages 1–12. ACM, 2010a.

Conor McBride. Strathclyde Haskell Enhancement, 2010b. URL `http://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`. Computer software.

Conor McBride and James McKinna. Functional pearl: I am not a number–I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (Haskell '04)*, pages 1–9. ACM, 2004.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

Matt Might. The missing method: Deleting from Okasaki's red-black trees, n.d.. URL `http://matt.might.net/articles/red-black-delete/`.

Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN 9780262631815.

Stefan Monnier and David Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming Languages meets Program Verification (PLPV '10)*, pages 1–8. ACM, 2010.

Shin-Cheng Mu. Developing programs and proofs spontaneously using GADT, 2007. URL `http://www.iis.sinica.edu.tw/~scm/2007/developing-programs-and-proofs-spontaneously-using-gadt/`.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):23:1–23:49, 2008.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN 3-540-65410-0.

Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990. URL `http://www.cse.chalmers.se/research/group/logic/book/`.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1998. ISBN 9780521631242.

Julien Oster. An Agda implementation of deletion in left-leaning red-black trees. Unpublished manuscript, 2011. URL `http://www.reinference.net/llrb-delete-julien-oster.pdf`.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*, pages 50–61. ACM, 2006.

Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–182. Cambridge University Press, 1991a.

Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Logic in Computer Science (LICS '91)*, pages 74–85. IEEE, 1991b.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

Robert Pollack. Implicit syntax. In Gérard Huet and Gordon Plotkin, editors, *Informal Proceedings of First Workshop on Logical Frameworks*, 1990.

Możesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu matematyków krajów slowiańskich, Warszawa 1929 (Comptes-rendus du I Congrés des Mathématiciens des Pays Slaves, Varsovie 1929)*, pages 92–101, 395, 1930.

David Pym. A unification algorithm for the $\lambda\pi$-calculus. *International Journal of Foundations of Computer Science*, 3(3):333–378, 1992.

Jason Reed. Higher-order constraint simplification in dependent type theory. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09)*, pages 49–56. ACM, 2009a.

Jason Reed. *A Hybrid Logical Framework.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2009b. URL `http://www.cs.cmu.edu/~rwh/theses/reed.pdf`.

Didier Rémy. Extension of ML type system with a sorted equational theory on types. Research Report RR-1766, INRIA, 1992.

John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, LNCS 19, pages 408–425. Springer, 1974.

Mikael Rittri. Dimension inference under polymorphic recursion. In *Functional Programming and Computer Architecture (FPCA '95)*, pages 147–159. ACM, 1995.

John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

Colin Runciman. What about the natural numbers? *Computer Languages*, 14: 181–191, 1989.

Matthias C. Schabel and Steven Watanabe. Boost.Units 1.1.0, 2013. URL `http://www.boost.org/doc/libs/1_54_0/doc/html/boost_units.html`. Computer software.

Robert Sedgewick. Left-leaning red-black trees. Unpublished manuscript, 2008. URL `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, ENTCS 199, pages 49–65, 2008.

Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29, 2007.

Ryan Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR84–639, Computer Science Department, Cornell University, 1984.

Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical Report ACRC-99-009, University of South Australia, School of Computer and Information Science, July 1999.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in Language Design and Implementation (TLDI '07)*, pages 53–66. ACM, 2007.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. Unpublished manuscript, 2009. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/tldi22-sulzmann-with-appendix.pdf`.

Don Syme. *The F# 2.0 Language Specification.* Microsoft, 2010. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=79948`.

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalized. In *Types in Language Design and Implementation (TLDI '10)*, pages 39–50. ACM, 2010.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4–5):333–412, 2011.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*, pages 341–352. ACM, 2012.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '89)*, pages 60–76. ACM, 1989.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, School of Computer Science, Carnegie Mellon University, 2003.

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '11)*, pages 227–240. ACM, 2011a.

Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, pages 333–345. ACM, 2011b.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*, pages 275–286. ACM, 2013.

J. B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*, pages 913–925. Springer, 2002.

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1998. URL `http://www.cs.bu.edu/~hwxi/academic/papers/thesis.2.ps`.

Hongwei Xi. Applied Type System. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, LNCS 3085, pages 394–408. Springer, 2004.

Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

Hongwei Xi. A verified implementation of quicksort on lists, 2008. URL `http://www.ats-lang.org/EXAMPLE/MISC/quicksort_list_dats.html`.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation (PLDI '98)*, pages 249–257. ACM, 1998.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '03)*, pages 224–235. ACM, 2003.

Kazu Yamamoto. Purely functional left-leaning red-black trees. 2011. URL `http://www.mew.org/~kazu/proj/red-black-tree/`.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in Language Design and Implementation (TLDI '12)*, pages 53–66. ACM, 2012.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

This thesis was written in literate Haskell and converted to LaTeX by `lhs2TeX` version 1.17, developed by Ralf Hinze and Andres Löh. Formal grammars and typing rules (but sadly no machine-checked proofs) were produced using `Ott` version 0.21.1, developed by Francesco Zappa Nardelli, Peter Sewell, and Scott Owens. It was typeset in 12 point Latin Modern using `pdfTeX` version 1.40.13.