Complete and Easy Type Inference for First-Class Polymorphism

Frank Emrich

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2023

Abstract

The Hindley-Milner (HM) typing discipline is remarkable in that it allows statically typing programs without requiring the programmer to annotate programs with types themselves. This is due to the HM system offering *complete* type inference, meaning that if a program is well typed, the inference algorithm is able to determine all the necessary typing information. Let bindings implicitly perform generalisation, allowing a let-bound variable to receive the most general possible type, which in turn may be instantiated appropriately at each of the variable's use sites. As a result, the HM type system has since become the foundation for type inference in programming languages such as Haskell as well as the ML family of languages and has been extended in a multitude of ways.

The original HM system only supports *prenex polymorphism*, where type variables are universally quantified only at the outermost level. This precludes many useful programs, such as passing a data structure to a function in the form of a fold function, which would need to be polymorphic in the type of the accumulator. However, this would require a nested quantifier in the type of the overall function. As a result, one direction of extending the HM system is to add support for *first-class polymorphism*, allowing arbitrarily nested quantifiers and instantiating type variables with polymorphic types. In such systems, restrictions are necessary to retain decidability of type inference.

This work presents FreezeML, a novel approach for integrating first-class polymorphism into the HM system, focused on simplicity. It eschews sophisticated yet hard to grasp heuristics in the type systems or extending the language of types, while still requiring only modest amounts of annotations. In particular, FreezeML leverages the mechanisms for generalisation and instantiation that are already at the heart of ML. Generalisation and instantiation are performed by let bindings and variables, respectively, but extended to types beyond prenex polymorphism. The defining feature of FreezeML is the ability to *freeze* variables, which prevents the usual instantiation of their types, allowing them instead to keep their original, fully polymorphic types.

We demonstrate that FreezeML is as expressive as System F by providing a translation from the latter to the former; the reverse direction is also shown. Further, we prove that FreezeML is indeed a conservative extension of ML: When considering only ML programs, FreezeML accepts exactly the same programs as ML itself.

We show that type inference for FreezeML can easily be integrated into HM-like type systems by presenting a sound and complete inference algorithm for FreezeML that extends Algorithm W, the original inference algorithm for the HM system.

Since the inception of Algorithm W in the 1970s, type inference for the HM system and its descendants has been modernised by approaches that involve *constraint solving*, which proved to be more modular and extensible. In such systems, a term is translated to a logical constraint, whose solutions correspond to the types of the original term. A solver for such constraints may then be defined independently. To this end, we demonstrate such a constraint-based inference approach for FreezeML.

We also discuss the effects of integrating the *value restriction* into FreezeML and provide detailed comparisons with other approaches towards first-class polymorphism in ML alongside a collection of examples found in the literature.

Lay Summary

Many programming languages employ a *static type system* to increase the reliability of programs, by checking if programs are *well typed* before executing them. A type system may then guarantee that some formal properties hold for all well typed programs. For example, a type system may guarantee that well typed programs will not crash due to a function being called with an insufficient number of arguments.

Establishing whether a program is well typed works by assigning *types* to expressions written in the language, describing their possible values. For example, the expression 123 may receive the type Int of integer numbers. The type system then defines when a certain expression may be given a certain type. For example, the type system may dictate that in order for a function call to be well typed, the type of the function (encoding the number of expected parameters and their individual types) must be compatible with the number and types of the arguments. This in turn allows giving the overall function call under consideration a type, derived from the function's type.

As a more concrete example, the function inc may be defined to increase integers by one. It may thus be given the *function type* $\mathsf{Int} \rightarrow \mathsf{Int}$, meaning that it is a function that takes exactly one integer argument and returns an integer. The expression $\mathsf{inc}\ 1$, where inc is applied to 1, is then well typed with type Int, while inc "hello" is not.

Some type systems offer *polymorphism*, supporting types that can indicate that a certain expressions can be used as if it had multiple types. For example, a user may define a function $\mathsf{reverse}$ that takes an argument of type $\mathsf{List\ Int}$ (i.e., a list containing integer values), and returns a list containing the same elements, but in reversed order. A type system could thus give $\mathsf{reverse}$ the type $\mathsf{List\ Int} \rightarrow \mathsf{List\ Int}$. However, a straightforward definition of $\mathsf{reverse}$ would usually not depend on the fact that the input list contains integers, but only depend on the fact that it is a list. Thus, the same implementation of the function may be used to reverse any kind of list.

A type system supporting polymorphism can represent this by allowing $\mathsf{reverse}$ to receive the *polymorphic* type $\forall a.\mathsf{List\ a} \rightarrow \mathsf{List\ a}$. Here, a is a type variable; the universal quantification expresses that $\mathsf{reverse}$'s type may be *instantiated* by picking what a should be. Thus, one type that $\forall a.\mathsf{List\ a} \rightarrow \mathsf{List\ a}$ may be instantiated to is the earlier type $\mathsf{List\ Int} \rightarrow \mathsf{List\ Int}$. Giving $\mathsf{reverse}$ the former polymorphic type then allows re-using the same definition of the function to act on many types of lists, instead of creating duplicate versions for each list type to act on.

Another way to characterise a type system is whether it supports some degree of

type inference. Some type systems require programmers to annotate all functions and variables with their types. This can be laborious, which is why some languages, such as recent versions of Java and C++, allow omitting types on certain local variables, meaning that the language's type system will *infer* the variable's type by reasoning about the value used to initialise it. Some type systems go further, meaning that the programmers do not need to write any type annotations at all. The theoretical foundation for this kind of type inference was established by the Hindley-Milner (HM) type system, first incorporated into the ML language. A key formal property is that there exist type inference algorithms for the HM system that are *complete*, meaning that if there exists a way of assigning types to expressions in a program to make it well typed, the algorithm will find it. This required a careful design of the HM type system: It supports polymorphism, but imposes several restrictions regarding the nesting of quantifiers within types and what types a quantified type variable may be instantiated with. Easing these restrictions can easily result in the completeness property of type inference to be lost.

Several type systems have been proposed that lift the restrictions imposed on polymorphism by the HM system, supporting what is called *first-class polymorphism* instead. In order to have any hope of providing a complete type inference algorithm, such systems require type annotations in some situations. This requires trade-offs, balancing the amount of type annotations required against how complicated the inference algorithm and rules of the type system are.

This work explores *FreezeML*, a new type system in this design space. It is designed to follow ML and the HM system particularly closely: The decision-making about how polymorphism is introduced when, say, defining a function, and how it is eliminated by instantiating the types of expressions is directly inspired by the corresponding rules of ML. This results in somewhat eager instantiation, where polymorphism may be instantiated in cases where the user would like it to be preserved. FreezeML thus provides a *freezing* operator, which allows avoiding instantiation. We present a type inference algorithm for FreezeML, which is closely based on Algorithm W, the original type inference algorithm for the HM system. We also investigate a more modern approach towards type inference for FreezeML, which acts in two steps: Programs are first translated to a *constraint*, which is then separately processed by a *constraint solver*. This involves the design of an appropriate constraint language and solver. We also explore potential extensions and variations of FreezeML, and compare it with existing systems in the design space.

Acknowledgements

First, and foremost, I would like to thank my supervisor, James Cheney, for his continued guidance throughout my PhD studies. James' course *Elements of Programming Languages*, which I attended as an exchange student visiting the University of Edinburgh, was one of the driving forces behind my decision to further my interest in programming languages, and eventually deciding to pursue a PhD in that area. I am very grateful for the great effort and patience with which he has supported me.

Next, I would like to thank my secondary supervisor, Sam Lindley. His idea to fix the handling of first-class polymorphism in Links took me on a research journey that turned out to be much larger than we may have originally excepted, ultimately leading to FreezeML and this thesis. Sam's optimism towards research has been inspirational, encouraging me to tackle problems head on rather than shy away from them.

A big thank you to my current and former fellow PhD students at the University of Edinburgh: Bogdan Manghiuc, Cat Wedderburn, Daniel Hillerström, Nuiok Dicaire, Stefan Fehrenbach, Simon Fowler, Rudi Horn and Wenzhi Fu. You have made my time as a PhD student ever so much more enjoyable. Daniel in particular has been a great source of guidance and support in times of doubt.

I am also very thankful to Andrew Kennedy and Igor Sheludko for hosting me during my internships at Meta and Google, respectively. I thoroughly enjoyed applying my PL knowledge in practice, and you have been a big part of that.

My friends and family have been an immeasurable source of support, thank you for being there for me during this journey.

Finally, I would like to thank Daan Leijen and Don Sannella for agreeing to be my examiners. I would likewise want to acknowledge the European Research Council for the funding provided by the ERC Consolidator Grant "Skye", as well as the Laboratory for Foundations of Computer Science for providing additional funding towards the end of my PhD.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

This work uses material from the following, previously published work. The concrete relationships between individual parts of this thesis and my previous work is outlined in the introduction.

- Emrich, F., Lindley, S., Stolarek, J., Cheney, J., and Coates, J. (2020b). FreezeML: Complete and easy type inference for first-class polymorphism. In Donaldson, A. F. and Torlak, E., editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 423–437. ACM

- Emrich, F., Stolarek, J., Cheney, J., and Lindley, S. (2022). Constraint-based type inference for FreezeML. *Proceedings of the ACM on Programming Languages*, 6(ICFP):570–595

- Emrich, F., Lindley, S., and Stolarek, J. (2020a). The virtues of semi-explicit polymorphism. Extended abstract presented at the ML workshop @ ICFP 2020

(*Frank Emrich*)

Contents

Chapter 1

Introduction

Programming languages featuring a *sound* static type systems provide programmers with a strong safety guarantee: If a program is well typed, it cannot exhibit type errors at runtime, therefore ruling out whole classes of errors. Type inference can then ease the burden of having to specify types in the program on the programmer, possibly avoiding the need for type annotations altogether.

The Hindley-Milner (HM) typing discipline [40, 77, 13] is remarkable in that it guarantees the existence of a *complete* type inference algorithm, meaning that if a program is well typed, the algorithm is able to determine all necessary typing information. It requires no type annotations at all. As a result, the HM typing discipline has provided the basis for the type systems of several industrial-strength programming languages, such as Haskell and members of the ML family of languages, including Standard ML and OCaml.

The HM system allows variables bound by let terms to receive polymorphic types, reflecting that a binding can act on data of different shapes. For example, the following program, written in an imaginary dialect of ML, defines the function every_other such that calling every_other l returns a list containing only every second element from the input list l.

```
let rec every_other = λl.
   case l of
      | x :: y :: xs  =>  y :: every_other xs
      | _ => []
in
(every_other [1;2;3], every_other ["hello";"world"])
```

Here, the HM system can correctly determine that the argument to every_other must be a list, but the type of the list's elements is irrelevant to the function. The system can therefore give the function the following, polymorphic type.

$$\forall a.\mathsf{List}\ a \to \mathsf{List}\ a$$

This type then allows the two subsequent calls of every_other using arguments of different list types to be accepted.

The HM system has been extended in a large variety of ways in the decades following its inception. Examples include features such as extensible records [121, 98, 82], type classes [120, 87, 47, 46], qualified types [44], recursive types [97], and object-oriented programming [102], to name a few.

1.1 Beyond HM-style polymorphism

The HM type system imposes restrictions with respect to where polymorphic types may occur. Consider a situation where we want to remove elements from two lists using a data-agnostic removal function cull, such as every_other, before applying a computationally expensive operation combine that turns the *i*th elements from the culled lists into an element of the result list. We may hope to define a helper function doing this as follows:

```
let cull_and_combine = $\lambda$cull combine $l_1$ $l_2$ .
  let $l_1$ = cull $l_1$ in
  let $l_2$ = cull $l_2$ in
  let rec map2_prefix = $\lambda l\ l'$.
    case $(l, l')$ of
      |$(x :: xs,\ y :: ys)$ => combine $x\ y$ :: map2_prefix $xs\ ys$
      | _ => []
  in
  map2_prefix $l_1$ $l_2$
```

In order for the function cull to work on both lists, we intend for it to have the polymorphic type $\forall d.\mathsf{List}\ d \to \mathsf{List}\ d$ inside cull_and_combine, meaning that the overall type of cull_and_combine should be the following:

$$\forall a\,b\,c.\underbrace{(\forall d.\mathsf{List}\ d \to \mathsf{List}\ d)}_{\mathsf{cull}} \to \underbrace{(a \to b \to c)}_{\mathsf{combine}} \to \underbrace{\mathsf{List}\ a}_{l_1} \to \underbrace{\mathsf{List}\ b}_{l_2} \to \mathsf{List}\ c \qquad (1.1)$$

However, this type is not supported in the HM system because it only supports *prenex polymorphism*, where quantifiers may only appear at the outermost level of any type. As an immediate consequence, lambda-bound variables can never be given polymorphic types. Milner himself described this as the "main limitation" of the HM system [77]. As a result, the definition of cull_and_combine on the previous page would not receive the type shown in (1.1), but instead force l_1 and l_2 to have the same list types, which cull acts on. Following the definition of a type's *rank* [61], systems whose type language allows arbitrarily nested quantifiers are said to support *higher-rank types*.

Note that the issue regarding the typing of cull_and_combine is independent of the helper function map2_prefix[1] it defines; hoisting map2_prefix out of cull_and_combine has no effect on the overall issue.

As another simple example that requires higher-rank types, consider a function f that can act on various data structures containing elements of some type a. Thus, rather than fixing the data structure to be received by f, we may want to keep it unspecified and pass the data structure to f represented by a fold function. This would require f to have a type of the following shape, where b is the type of the chosen accumulator while folding.

$$\forall a.\, \underbrace{(\forall b.(a \to b \to b) \to b \to b)}_{\mathsf{fold}} \to \;\ldots$$

Again, this type is not supported in the HM system.

As a further restriction, HM style type systems only permit *predicative* polymorphism, where type variables may only be instantiated with non-polymorphic types. As an example, consider the case where we redefine cull_and_combine to take a *list* of culling functions instead, to be applied in sequence. We call the resulting function list_cull_and_combine, with the following type.

$$\forall ab.\, \underbrace{(\mathsf{List}\ (\forall c.\mathsf{List}\ c \to \mathsf{List}\ c))}_{\mathsf{list\ of\ culls}} \to \underbrace{(a \to b \to c)}_{\mathsf{combine}} \to \underbrace{\mathsf{List}\ a}_{l_1} \to \underbrace{\mathsf{List}\ b}_{l_2} \to \mathsf{List}\ c,$$

Even after hypothetically extending the HM system with higher-rank types, we cannot call it using list_cull_and_combine (singleton every_other), where singleton is defined in the environment with the usual type $\forall a.a \to \mathsf{List}\ a$ to create a singleton list from its argument. Doing so would require instantiating the quantified type variable a in singleton's type with $\forall c.\mathsf{List}\ c \to \mathsf{List}\ c$, which predicative polymorphism does not

[1]This function is known to Haskell programmers as zipWith. The OCaml standard library provides map2, but unlike our function map2_prefix it fails if the two lists do not have the same length.

permit. As another example, this limitation means that in the presence of higher-rank polymorphism, the infix application operator \$ with type $\forall ab.(a \to b) \to a \to b$ cannot be used on functions with polymorphic arguments, unless an ad-hoc change is made to the type-checker. Serrano et al. go as far as calling the lack of polymorphic instantiation in the HM system an "embarrassing shortcoming" [109]. Offering *first-class polymorphism* refers to lifting both restrictions, thus supporting higher-rank types *and* polymorphic instantiation of type variables.

Examples for programming techniques that require forms of polymorphism beyond what the HM system offers include state threads [54], *short-cut deforestation* [28], dynamic typing in the style of Baars and Swierstra [2], and the *scrap your boilerplate* approach towards generic programming [51]. Shan [111] provides a more detailed overview of these and more use-cases.

The limitations of the HM system to prenex polymorphism and predicative instantiation are not arbitrary, but a direct result of the balancing act that is required to obtain a complete type inference algorithm. Easing the restrictions can easily lead to type inference becoming undecidable [123, 94, 48]. For example, a term such as $\lambda f.f\ f$ may become typeable when polymorphism is unrestricted, but it lacks a type that is most general in the usual sense of the HM system, therefore subsuming all other possible choices. While a lack of principal types does not automatically preclude the existence of a complete type inference algorithm, the example illustrates the difficulties faced in the presence of richer forms of polymorphism.

As a result, a large field of research into type systems that integrate higher-rank types or full first-class polymorphism into HM-style systems has emerged. The resulting landscape includes a variety of different approaches, making different trade-offs between aspects such as the complexity of the system's specification and implementation as well as the amount of type annotations required by programmers. Some systems use types that directly contain constraints on how a quantified type variable may be instantiated [56, 59]. Other systems utilise sophisticated heuristics built into the typing rules, guiding for example where instantiation and generalisation may be performed [58, 110, 109]. These systems are aimed at catching the programmer's intent in most situations, but may be difficult for users to follow in those cases where they do not.

Yet another group of systems requires users to be more explicit when using first-class polymorphism, often asking them to choose between different sorts of polymorphic types [106, 89, 26].

1.2 The Links programming language

The author's work on a type system supporting first-class polymorphism originated in his work on the Links programming language [11, 69]. Links is a strict, functional, multi-tier programming language, allowing parts of an application written entirely in Links to be executed on a web-server, an associated database server, and end users' browsers. This involves translating Links code to JavaScript and SQL where needed. Over the years since its inception, Links' database-related features have been extended to offer strong guarantees about the efficiency of the generated SQL queries [67, 7, 8] and support relational lenses [42] as well as temporal data [23]. An experimental version of Links supports data provenance tracking [22]. Further, support for effect handlers [37, 39, 38] and session types has been added to Links [68, 24].

Links has also supported first-class polymorphism since its early days. However, the existing treatment of polymorphism was a complex, ad-hoc combination of existing approaches that had never been formalised or reached a satisfying level of reliability. During the author's work on a type-checker for the explicitly typed intermediate representation (IR) of Links and subsequent investigations into the causes of ill typed IR code being generated, it became clear that the existing support for first-class polymorphism needed to be overhauled. Ultimately, the decision was made to replace the existing system with a more principled design.

1.3 The road to FreezeML

While investigating the approach towards first-class polymorphism with which to replace the existing one in Links, a series of required properties were crystallised. The overarching theme was the desire to find a solution that values simplicity – in terms of its formal specification, implementation, and the rules needed to be understood by end users (e.g., where to place type annotations) – rather than aggressively trying to minimise the number of type annotations.

Predictable behaviour Many existing approaches towards first-class polymorphism rely on an elaborate type system with intricate heuristics to reduce the number of type annotations required by the user. Vytiniotis et al. say about their own Boxy-Types system that it is "too complicated for Joe Programmer to understand", and expect programmers to use the type-checker itself as the specification, experimenting with the placement of annotations if needed [117]. We follow a different philosophy, wanting to implement a system that is simple enough for programmers to understand, without relying on potentially surprising heuristics.

System F types Our ideal solution would use the type language of System F and ML, already familiar to programmers. Some systems, such as MLF [56], go beyond the expressive power of System F types by allowing constraints in types on how variables may be instantiated. Other systems, such as IFX [89], Poly-ML [26] and QML [106], distinguish two sorts of quantified types, namely ML-style types schemes and System F style polymorphic types, where only the latter stays polymorphic when passed around. We agree with Russo and Vytiniotis' assessment that this distinction is potentially confusing for programmers [106]. Further, staying within the limits of just System F types is compatible with the existing implementation for first-class polymorphism in Links.

Close to ML type inference It should be possible to implement the solution as a simple extension of Hindley Milner type inference, allowing it to be integrated into existing type inference systems, rather than making strong assumptions about the underlying inference algorithm. Systems like MLF [56] and FPH [118] rely on much more sophisticated inference and/or unification algorithms. Other systems, such as QuickLook [109], rely on a bidirectional inference system.

The current implementation of type inference in Links – despite its multitude of typing features – resembles an efficient implementation of Algorithm W. It traverses the abstract syntax tree while maintaining a union-find structure of type variables.

Thus, type inference for our approach should be flexible enough to be integrated into the existing inference algorithm. The approach's implementation should not get in the way if longer term plans to overhaul type inference in Links are realised, such as adding a degree of bidirectionality or adopting a constraint-based inference approach.

Low syntactic overhead Of course, the desired properties stated so far could easily be achieved when the verbosity of the resulting programs is of no concern: The usual, Church-style explicitly typed representation of System F satisfies all properties above. The challenge lies in the tension between the simplicity of the system while still keeping the necessary amount of type annotations in check. Thus, the chosen approach should still have a reasonably low syntactic overhead.

Embracing the value restriction The *value restriction* [126] is a simple syntactic mechanism to ensure the soundness of polymorphic type systems in the presence of computations with side effects. It restricts what terms may be given polymorphic types and therefore influences the design of a system providing richer polymorphism. It was adopted in Standard ML and OCaml, using a relaxed version of it for the latter, as well as Links. Our goal was therefore to use a system that works well in the presence of the value restriction.

While these properties were in part motivated by the need to implement the new approach for Links, we consider these as – somewhat subjective – desirable properties of an approach towards first-class polymorphism on their own. As a result, we study FreezeML as a core calculus in this work, independently from Links, but return to the implementation of FreezeML in Links in one of the later chapters.

Ultimately, the Links authors found that none of the existing approaches towards first-class polymorphism had these properties, meaning that they became design goals of a new system to be developed. The new system would be focused on the simplicity of the underlying specification. In particular, the resulting system directly leverages the existing mechanisms for instantiation (i.e., using polymorphic values) and generalisation (i.e., creating polymorphic values): Clément et al. [10] showed that in ML, generalisation and instantiation are inherently tied to let bindings and variables, respectively. Their presentation of ML makes it clear that generalisation is only meaningful when performed on let-bound terms prior to determining the type of the bound variable, while instantiation happens exactly at variables, instantiating all toplevel quantifiers. We revisit these findings more formally in the next chapter.

As a result, our approach directly uses let bindings and variables as the only mechanisms for generalisation and instantiation, respectively, and observe that some helpful operators may be obtained from them as syntactic sugar. The remaining puzzle piece then becomes how to *keep* values polymorphic, if polymorphism is introduced at let bindings, but immediately lost at each use site of the variable. Our answer is the ability to *freeze* variables, denoted $\lceil x \rceil$, which preserves the type of x unchanged rather than performing instantiation. It is the defining feature of FreezeML, our type system satisfying the requirements listed earlier.

Freezing immediately resolves the potential ambiguity when typing examples such as $\mathsf{singleton}\ \mathsf{id}$. Here, id is the identity function with type $\forall b.b \to b$, the type of $\mathsf{singleton}$ is $\forall a.a \to \mathsf{List}\ a$, as discussed in Section 1.1. This term is often used in existing literature to discuss the design decisions facing systems supporting first-class polymorphism.

We observe that $\mathsf{singleton}$ *must* be instantiated in this term to obtain a function type. The question remains whether or not to instantiate id, which in turn dictates the type of the overall term. If the quantifier b in id's type is instantiated with some type A, then the overall application results in a list of type $\mathsf{List}\ (A \to A)$. Otherwise, if id is kept polymorphic, we obtain a singleton list of polymorphic identity functions of type $\mathsf{List}\ (\forall b.b \to b)$.

Instead of relying on complex mechanisms to determine the most likely intention of the programmer, such as taking the potential surrounding context into account, the freezing operator allows programmers to make their intentions explicit. While $\mathsf{singleton}\ \mathsf{id}$ will always instantiate id, writing $\mathsf{singleton}\ \lceil \mathsf{id} \rceil$ instead keeps it polymorphic. This means that while FreezeML preserves the instantiating character of variables, a defining feature of ML, frozen variables provide users with variables that behave closer to what they do in a logic system: They simply reflect the types associated with them in the environment, as they do in System F.

1.4 Structure and contributions of this thesis

This thesis is structured as follows. We point out the contributions C1 to C12 made by this work as they appear in the document.

- Chapter 2 provides necessary background. We introduce ML and System F as core calculi and discuss existing approaches for integrating first-class polymorphism into the HM system.

- In Chapter 3, we introduce FreezeML itself, a novel design for a type system that integrates first-class polymorphism into ML, satisfying the design goals listed in Section 1.3. The chapter begins with an outline of the design of FreezeML (C1), followed by its formal definition (C2). Crucially, the type system ensures that all occurrences of polymorphism in the types of variables are fully determined. This means that for each variable there exists a unique type such that all other possible types of the variable may be obtained by substituting type variables in the former type with *monomorphic* types. This is achieved by limiting where polymorphic types can occur in typing rules and requiring the use of principal types for some terms.

  We discuss translations from System F to FreezeML and backwards in Section 3.3 (C3). Finally, we show that the FreezeML typing relation is well defined, despite containing typing rules that universally reason about all possible typings of certain terms (C4). While other authors have used typing rules with similar conditions in their systems, this issue has not been treated formally in existing work.

- A type inference algorithm for FreezeML based on Algorithm W is presented in Chapter 4 (C5). First, we introduce a type substitution property for FreezeML, before presenting the actual inference algorithm. Our inference algorithm augments the original Algorithm W by globally tracking which type variables may be substituted with polymorphic types.

- Since the inception of Algorithm W in the 1970s, type inference has been modernised by techniques that rely on *constraint solving*. Chapter 5 discusses an alternative inference algorithm for FreezeML based on this concept. We first present a suitable constraint language and how to translate FreezeML terms to it (C6), followed by a solver for the constraint language (C7).

  To the best of the author's knowledge, the constraint language presented here is the first whose semantics impose the usage of principal solutions for certain constraints.

  However, an important part of the solver's metatheory remains as a conjecture, which is required for the overall correctness of the constraint-based inference approach. We provide a detailed discussion of the challenges faced when attempting to finalise the missing proof.

- Chapter 6 provides additional discussion. We show how some real-world use cases of first-class polymorphism can be implemented using FreezeML (C8) as well as sketching potential variations and extensions of FreezeML (C9). Two implementations of FreezeML are discussed, in Links and as a standalone application (C10). We outline which properties of ML and other approaches for first-class polymorphism in the HM system hold for FreezeML, or how they need to be adapted to hold (C11). Finally, we compare FreezeML in detail with select systems discussed in Chapter 2, in particular based on a collection of example programs used in previous literature (C12).

- We conclude and discuss future work in Chapter 7.

Those proofs omitted from the main text are provided in Appendices A to C.

1.4.1 Relationship with previous work

The material in Chapters 3 and 4 is based on the following paper.

> Emrich, F., Lindley, S., Stolarek, J., Cheney, J., and Coates, J. (2020b). FreezeML: Complete and easy type inference for first-class polymorphism. In Donaldson, A. F. and Torlak, E., editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 423–437. ACM

Part of this work also appears in Sections 6.3 and 6.5.

The material in Chapter 5 is based on the following paper.

> Emrich, F., Stolarek, J., Cheney, J., and Lindley, S. (2022). Constraint-based type inference for FreezeML. *Proceedings of the ACM on Programming Languages*, 6(ICFP):570–595

The material in Section 7.2.2 is based on the following work.

> Emrich, F., Lindley, S., and Stolarek, J. (2020a). The virtues of semi-explicit polymorphism. Extended abstract presented at the ML workshop @ ICFP 2020

Chapter 2

Background

This chapter provides background material that aids understanding of the remainder of this thesis. We introduce notations and present ML as well as System F formally. We then provide an overview of existing approaches towards combining the Hindley-Milner typing discipline with richer forms of polymorphism, such as arbitrary-rank types or fully-fledged first-class polymorphism.

2.1 Notations

Sequences and sets We write $\overline{X}$ to denote a sequence of some elements X. Sequences are concatenated using commas, meaning that $\overline{X}, \overline{Y}$ denotes the sequence consisting of all elements from $\overline{X}$, followed by those of $\overline{Y}$. We also use the comma operator on singleton elements (e.g., we may write $\overline{X}, x$ to append x to the sequence $\overline{X}$).

The relation # indicates disjointness between sequences as well as sets, including singleton ones. We use $\approx$ to indicate that two sequences are permutations of each other.

Given a finite set or sequence X, we denote its cardinality using $|X|$.

Given a set X, we write 2^X for its powerset.

Functions We write $\mathsf{dom}(f)$ and $\mathsf{codom}(f)$ for the domain and co-domain of a function f, respectively. Given a set Z with $Z \subseteq \mathsf{dom}(f)$, we write $f\restriction_Z$ to denote the restriction of f to Z.

2.2 Core calculi

We now introduce ML and System F in a style that makes bound type variables explicit.

2.2.1 ML and the Hindley-Milner type system

What is now known as the Hindley-Milner (HM) type system was independently developed by Hindley [40] and Milner [77]. Hindley's work was in the context of combinatory logic. Milner's work was in the context of the ML programming language, originally developed as part of the LCF theorem proving system [32]. Crucially, Milner established the ability of let bindings to implicitly introduce polymorphism. The now typical presentation of the HM type system is due to Damas and Milner [13]. This led to the system also being referred to as Damas-Milner or Damas–Hindley–Milner.

Several extensions to the original ML language, such as the addition of datatypes and a powerful module system [70], ultimately led to the definition of Standard ML [79]. Further, it marked the inception of the ML *family* of languages, with OCaml and F# being prominent members. The underlying HM type system also laid the foundation for type inference in other languages, such as Haskell, that do not share the characteristic features of ML-like languages (e.g., mutability and call-by-value semantics). A comprehensive account of the history of ML has recently been published by MacQueen et al. [71].

In this work, unless stated otherwise, we use the term "ML" in the sense of the core calculus studied by Damas and Milner [13], eschewing features such as datatypes, recursion, records, and mutable references.

The syntax of this ML core calculus is shown in Figure 2.1 on the next page. We assume that the type constructors D contain at least functions and the unit type, where each constructor has a fixed arity, which we refer to as $\mathsf{arity}(D)$. Ordinary types, called *monotypes* in our presentation, then consist of type variables and type constructor applications. As usual, type schemes E allow universal quantification of type variables, but only at the outermost level. This enshrines ML's limitation to *prenex polymorphism* in the syntax of types. Throughout this work, we identify type contexts Δ and sequences of variables in quantified types. Thus, given $\Delta = \overline{a}$, we use $\forall\Delta.E$ and $\forall\overline{a}.E$ interchangeably. Further, we identify $\forall\overline{a}.E$ and E if $\overline{a}$ is empty.

Instantiations δ map type variables to monotypes.

As mentioned before, we limit ourselves to a minimal number of syntactic forms, meaning that terms M consist of variables, abstractions, applications, and let bindings.

Type variables	a, b, c
Type constructors	$D ::= \to \mid \mathsf{Unit} \mid \ldots$
Monotypes	$S, T ::= a \mid D\overline{S}$
Type schemes	$E ::= S \mid \forall a.E$
Type instantiations	$\delta ::= \emptyset \mid \delta[a \mapsto S]$
Term variables	x, y, z
Terms	$M, N ::= x \mid \lambda x.M \mid M\,N \mid \textbf{let}\ x = M\ \textbf{in}\ N$
Values	$V, W ::= x \mid \lambda x.M \mid \textbf{let}\ x = V\ \textbf{in}\ W$
Type contexts	$\Delta ::= \cdot \mid \Delta, a$
Term contexts	$\Gamma ::= \cdot \mid \Gamma, x : E$

Figure 2.1: ML syntax

We return to the role of values V in Section 2.2.1.2.

Traditional presentations of ML often do not explicitly track what type variables are in scope – Gundry et al. [35] say that they let type variables "float in space". In contrast, our formalism uses *type contexts* Δ that contain all type variables in scope. As usual, *term contexts* Γ assign type schemes to the term variables in scope. Throughout this work, we use the convention that the elements of each type context or term context appearing in a judgement are implicitly required to be pairwise disjoint. For example, premises mentioning Δ, a or $\Gamma, x : E$ in typing rules implicitly imposes $\Delta \# a$ and $\Gamma \# x$. This allows us to avoid dealing with shadowing by assuming appropriate alpha-conversion has taken place.

The typing rules of ML are shown in Figure 2.2 on the following page. They are presented in a declarative style, which is indicated by the superscript MLD. Our explicit treatment of type variable scoping is reflected by the additional requirement that in order for a judgement $\Delta; \Gamma \vdash^{\text{MLD}} M : E$ to hold, any type scheme E appearing in Γ must be well formed under Δ, denoted $\Delta \vdash E\ \textbf{ok}$. This simply means that all free type variables of E must occur in Δ. Similar requirements are imposed on the subsequently introduced variations of ML.

Type schemes for variables are retrieved from the term context. Abstractions and applications are typed as usual, but limited to monotypes. Let bindings allow obtaining a type scheme for the bound term M as the type for x in N. The rules INST and GEN allow performing instantiation and generalisation on arbitrary terms. Crucially, only variables that do not appear in the surrounding context may be generalised.

$$\boxed{\Delta;\Gamma \vdash^{\text{MLD}} M : E}$$

$$\textsc{Var}\ \frac{x : E \in \Gamma}{\Delta;\Gamma \vdash^{\text{MLD}} x : E} \qquad \textsc{Lam}\ \frac{\Delta;(\Gamma, x : S) \vdash^{\text{MLD}} M : T}{\Delta;\Gamma \vdash^{\text{MLD}} \lambda x.M : S \to T} \qquad \textsc{App}\ \frac{\Delta;\Gamma \vdash^{\text{MLD}} M : S \to T \quad \Delta;\Gamma \vdash^{\text{MLD}} N : S}{\Delta;\Gamma \vdash^{\text{MLD}} M\,N : T}$$

$$\textsc{Let}\ \frac{\Delta;\Gamma \vdash^{\text{MLD}} M : E \qquad \Delta;(\Gamma, x : E) \vdash^{\text{MLD}} N : T}{\Delta;\Gamma \vdash^{\text{MLD}} \textbf{let}\ x = M\ \textbf{in}\ N : T}$$

$$\textsc{Inst}\ \frac{\Delta;\Gamma \vdash^{\text{MLD}} M : E \qquad \Delta \vdash E \leq E'}{\Delta;\Gamma \vdash^{\text{MLD}} M : E'} \qquad \textsc{Gen}\ \frac{(\Delta, a);\Gamma \vdash^{\text{MLD}} M : E}{\Delta;\Gamma \vdash^{\text{MLD}} M : \forall a.E}$$

Figure 2.2: ML typing rules (declarative)

The Inst rule uses the *generic instance* relation, shown in Figure 2.3. Here,

$$\boxed{\Delta \vdash E \leq E'}$$

$$\frac{}{\Delta \vdash E \leq E} \qquad \frac{\Delta, a \vdash E \leq E'}{\Delta \vdash E \leq \forall a.E'} \qquad \frac{\Delta \vdash S \qquad \Delta \vdash E[S/a] \leq E'}{\Delta \vdash \forall a.E \leq E'}$$

Figure 2.3: Generic instance relation

$E[S/a]$ denotes capture-avoiding substitution of a with S in E. We have $\Delta \vdash E \leq E'$ if E is more general than E'.[1] Note that we again make the type variables in scope explicit by using a type context Δ; we require E and E' to be well formed under it. If $\Delta \vdash E \leq E'$ holds, then E' may be obtained from E by instantiating quantifiers in E, and quantifying over variables introduced during this instantiation step.

For example, we have the following.

$$\begin{array}{rcl} \vdash \forall b.b \to b & \leq & \mathsf{Int} \to \mathsf{Int} \\ a \vdash \forall b.b \to b & \leq & a \to a \\ \cdot \vdash \forall ab.a \to b \to (a \times b) & \leq & \forall c.c \to c \to (c \times c) \end{array}$$

[1]Historically, both $\leq$ and $\geq$ (and similar symbols) have been used for the generic instance relation.

2.2.1.1 Syntax-directed presentation

The typing rules in Figure 2.2 are declarative in the sense that each rule is concerned with just a single aspect of the type system. As a result, more than one typing rule may be applicable to a term M.

An important observation about the ML type system is that instantiation may be limited to variables while generalisation may be limited to let bindings, while preserving typeability. This has led to syntax-directed presentations of the ML type system, where exactly one rule applies to each term form [10].

The typing rules for ML in this style are shown in Figure 2.4. The standalone

$$\boxed{\Delta;\Gamma \vdash^{\text{MLS}} M : S}$$

$$\textsc{Var}\ \frac{x : \forall\Delta'.S \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow \cdot}{\Delta;\Gamma \vdash^{\text{MLS}} x : \delta(S)} \qquad \textsc{Lam}\ \frac{\Delta;(\Gamma, x : S) \vdash^{\text{MLS}} M : T}{\Delta;\Gamma \vdash^{\text{MLS}} \lambda x.M : S \to T}$$

$$\textsc{App}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash^{\text{MLS}} M : S \to T \\ \Delta;\Gamma \vdash^{\text{MLS}} N : S\end{array}}{\Delta;\Gamma \vdash^{\text{MLS}} M\,N : T} \qquad \textsc{Let}\ \frac{\begin{array}{c}(\Delta,\Delta');\Gamma \vdash^{\text{MLS}} M : S \\ \Delta;(\Gamma, x : \forall\Delta'.S) \vdash^{\text{MLS}} N : T\end{array}}{\Delta;\Gamma \vdash^{\text{MLS}} \mathbf{let}\, x = M\, \mathbf{in}\, N : T}$$

Figure 2.4: ML typing rules (syntax-directed)

instantiation and generalisation rules are removed and fused into the VAR and LET rules, respectively.

To denote the eager instantiation of all quantifiers performed in the VAR rule in our notation, we use instantiations δ, as defined in Figure 2.1. Judgements of the form $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$ then impose that δ maps variables from Δ' to types using variables from Δ and Δ''. Further, δ maps all variables from Δ, which denotes some kind of ambient context, to themselves. The formal rules of the judgement are shown in Figure 2.5 on the following page. Note that the rules impose $\Delta \# \Delta'$ and $\Delta \# \Delta''$ as prerequisites of $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$.

The declarative and syntax-directed presentations are equivalent, besides the ability to generalise the overall type at the toplevel [10, Theorem 2.1]. Using our notations, this result can be stated as follows.

Lemma 2.1 (Relationship between $\vdash^{\text{MLD}}$ and $\vdash^{\text{MLS}}$)**.**

$$\boxed{\Delta \vdash \delta : \Delta' \Rightarrow \Delta''}$$

$$\frac{\Delta \# \Delta''}{\Delta \vdash \emptyset : \cdot \Rightarrow \Delta''} \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow \Delta'' \qquad (\Delta, \Delta'') \vdash S \qquad \Delta \# (\Delta', a)}{\Delta \vdash \delta[a \mapsto S] : (\Delta', a) \Rightarrow \Delta''}$$

Figure 2.5: ML instantiation rules

For all Δ, Γ, M *and* E*, the following conditions hold.*

1. *If* $\Delta; \Gamma \vdash^{MLS} M : E$ *then* $\Delta; \Gamma \vdash^{MLD} M : E$ *holds.*
2. *If* $\Delta; \Gamma \vdash^{MLD} M : E$ *then there exist* S *and* Δ' *such that* $(\Delta, \Delta'); \Gamma \vdash^{MLS} M : S$ *and* $\forall \Delta'. S \leq E$.

2.2.1.2 The value restriction

Since the early days of ML, it was known that the combination of polymorphism and mutability requires care to avoid unsoundness of the type system [77]. Consider the following example.

$$\begin{aligned}&\mathbf{let}\ \mathsf{pl} = \mathsf{ref}\ []\ \mathbf{in}\\ &\mathsf{pl} := [3];\\ &(\mathsf{head}\ (\mathsf{head}\ !\mathsf{pl})) + 3\end{aligned}$$

Naively, we may consider this program to be well typed: A possible type scheme for the empty list [] is $\forall a.\mathsf{List}\ a$. We may therefore assign pl the type scheme $\forall a.\mathsf{Ref}\ \mathsf{List}\ a$, where Ref is the type constructor for mutable references. We may then instantiate pl to have type Ref List Int, which makes the reference assignment with a value of type List Int well typed. Finally, we may instantiate pl again to yield Ref List List Int, which makes the last line of the example well typed, where we dereference pl, obtain the first element of the resulting list (using the function head), and then apply head again to the result.

However, this program should clearly be rejected: During execution of the program, once the last line is reached, the reference pl points to a list containing a single integer. Therefore, the result of (head !pl) is an integer, and applying head to it would cause havoc at runtime.

A multitude of solutions for this problem has been devised and refined over the years [12, 115, 64, 125, 63, 41, 113, 126, 33, 25]. One early observation was that a simple syntactic criterion may be used in tandem with more complex solutions: If a term is *non-expansive*, it is always safe to generalise its type [114]. Non-expansive terms, also called (syntactic) values, include variables and lambda terms, and possibly compound expressions if all of their subterms are non-expansive. The underlying idea is that non-expansive terms must not extend the *store* of reference cells during execution. Most importantly, applications are not values, ruling out the creation of references using the ref operator, such as ref [] in our example.

Wright [126] then provided empirical evidence that relying *exclusively* on this mechanism works well in practice. As a result, the *value restriction*, imposing that values are the only terms that may be generalised, was adopted in the revised definition of Standard ML [78]. A slightly relaxed version, taking variance of type constructor parameters into account, was adopted in OCaml [25].

We reflect this in Figure 2.1 on page 13 by introducing (syntactic) values V as a separate syntactic category.

In a syntax-directed presentation, the value restriction can be implemented by making the generalisation performed by let bindings depend on the syntactic form of the bound term M. This is shown in Figure 2.6, where the LET rule uses the auxiliary

$\boxed{\Delta;\Gamma \vdash^{\text{ML}} M : A}$

$$\textsc{Var}\quad \frac{x : \forall\Delta'.S \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow \cdot}{\Delta;\Gamma \vdash^{\text{ML}} x : \delta(S)} \qquad\qquad \textsc{Lam}\quad \frac{\Delta;(\Gamma, x : S) \vdash^{\text{ML}} M : T}{\Delta;\Gamma \vdash^{\text{ML}} \lambda x.M : S \to T}$$

$$\textsc{App}\quad \frac{\begin{array}{c}\Delta;\Gamma \vdash^{\text{ML}} M : S \to T \\ \Delta;\Gamma \vdash^{\text{ML}} N : S\end{array}}{\Delta;\Gamma \vdash^{\text{ML}} M\,N : T} \qquad \textsc{Let}\quad \frac{\begin{array}{c}\Delta' = \mathsf{gen}(\Delta, S, M) \quad (\Delta,\Delta');\Gamma \vdash^{\text{ML}} M : S \\ \Delta;(\Gamma, x : \forall\Delta'.S) \vdash^{\text{ML}} N : T\end{array}}{\Delta;\Gamma \vdash^{\text{ML}} \textbf{let}\, x = M \,\textbf{in}\, N : T}$$

Figure 2.6: ML typing rules (syntax-directed, using value restriction)

function gen, which is defined as follows. Here, $\mathsf{ftv}(S)$ refers to the free type variables of S.

$$\mathsf{gen}(\Delta, S, M) = \begin{cases} \mathsf{ftv}(S) - \Delta & \text{if } M \text{ is a value} \\ \cdot & \text{otherwise} \end{cases}$$

Unless stated otherwise, we generally refer to the system with the value restriction, shown in Figure 2.6, when referring to ML in this work.

2.2.1.3 Properties of ML

We now revisit some well known properties of ML to facilitate comparisons with FreezeML in subsequent chapters.

Typing in ML is stable under generalisation of variables in the environment. More specifically, types of terms are preserved when replacing any type scheme E in the term context with E', such that E is a generic instance of E' [13, Lemma 1].

Lemma 2.2.
For all $\Delta, \Gamma, M, E, E, E''$ *such that* $\Delta \vdash E' \leq E''$ *we have that* $\Delta; (\Gamma, x : E'') \vdash^{MLD} M : E$ *implies* $\Delta; (\Gamma, x : E') \vdash^{MLD} M : E$.

Next, typing in ML is stable under substitution of type variables [13, Proposition 2].

Lemma 2.3.
If $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$, *then* $(\Delta, \Delta'); \Gamma \vdash^{MLD} M : E$ *implies* $(\Delta, \Delta''); \delta(\Gamma) \vdash^{MLD} M : \delta(E)$.

The two previous lemmas equally hold for the typing relations $\vdash^{\mathrm{MLS}}$ and $\vdash^{\mathrm{ML}}$.

Principality We can now define *principal type schemes* [13] in our notation. We say that E is a principal type scheme of M under Δ and Γ if the following conditions hold.

1. We have $\Delta; \Gamma \vdash^{\mathrm{MLD}} M : E$.

2. For all E' such that $\Delta; \Gamma \vdash^{\mathrm{MLD}} M : E'$ we have $E \leq E'$.

Milner and Damas' seminal result was then that every well typed ML term has a principal type scheme and can be found using Algorithm W [13, 12].

Lemma 2.4 (Existence of principal type schemes)**.**
If $\Delta; \Gamma \vdash^{MLD} M : E$, *then there exists a principal type scheme of* M *under* Δ *and* Γ.

We observe that the lemma above and the idea of principal type schemes relies on the ability to generalise terms at the toplevel, only present in the declarative specification (and hence expressed using the relation $\vdash^{\mathrm{MLD}}$). In syntax-directed presentations, we may implicitly generalise after the fact to obtain a type scheme (as in Lemma 2.1 on page 15). Alternatively, we may state a property about all types directly derivable in the syntax-directed presentations simply in terms of free type variables rather than type schemes. To this end, we may also define a notion of *principal types*. We say that S is a principal type of M under Δ and Γ if there exists Δ' such that the following conditions hold.

1. We have $(\Delta, \Delta'); \Gamma \vdash^{\mathrm{ML}} M : S$.

2. For all Δ'', S'' such that $(\Delta, \Delta''); \Gamma \vdash^{\mathrm{ML}} M : S''$ there exists δ such that $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$ and $\delta(S) = S''$.

We observe that given Δ, Γ and a principal type S, we may reconstruct Δ' as any type context containing a superset of $\mathsf{ftv}(S) - \Delta$.

We can use this to state the following property.

Lemma 2.5 (Existence of principal types).
If $\Delta; \Gamma \vdash^{ML} M : S$*, then there exist* Δ', S'*, such that* S' *is a principal type of* M *under* Δ *and* Γ.

The definition of principal types and the corresponding lemma are stated in terms of the typing relation $\vdash^{\mathrm{ML}}$, but equally hold for $\vdash^{\mathrm{MLD}}$ and $\vdash^{\mathrm{MLS}}$.

The definition of principal types directly implies that they are unique up to renaming of generic variables (i.e., those not in the ambient context).

Lemma 2.6 (Uniqueness of principal types).
Let S *and* S' *both be principal types of* M *under* Δ *and* Γ*. Then there exists an instantiation* δ *that is a bijection between the variables in* $\mathsf{ftv}(S) - \Delta$ *and* $\mathsf{ftv}(S') - \Delta$ *such that* $\delta(S) = S'$.

Note that Lemma 2.5 immediately reveals the fact that principal types (and similarly, principal type schemes) capture *exactly* the possible types of M: The definition of principal types tells us that all other types of M can be obtained by substituting free variables in S, while Lemma 2.3 states that all such substitutions are valid types of M.

2.2.2 System F

System F, also called the polymorphic lambda calculus, was independently described by Girard [29, 30] and Reynolds [103]. It extends the simply typed lambda calculus with polymorphic types. We focus on a presentation of System F in the style of Church, where all binders are annotated with their types and polymorphism needs to be introduced and eliminated with dedicated syntactic forms, namely type abstractions and type applications. The syntax of System F in this style is given in Figure 2.7 on the next page. As opposed to ML, System F types A permit quantifiers anywhere. It is well known that it is possible to encode arbitrary inductive datatypes in System F [62, 6, 4, 119], using a generalisation of Church's encoding of data such as numerals and pairs [9, 3], for instance.

Type variables	a, b, c
Type constructors	$D ::= \to \mid \mathsf{Unit} \mid \ldots$
Types	$A, B ::= a \mid D\overline{A} \mid \forall a.A$
Term variables	x, y, z
Terms	$M, N ::= x \mid \lambda x^A.M \mid M\,N \mid \Lambda a.V \mid M\,A \mid \mathbf{let}\ x^A = M\ \mathbf{in}\ N$
Values	$V, W ::= x \mid \lambda x^A.M \mid \Lambda a.V \mid V\,A \mid \mathbf{let}\ x^A = V\ \mathbf{in}\ W$
Type contexts	$\Delta ::= \cdot \mid \Delta, a$
Term contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$

Figure 2.7: System F syntax

However, to facilitate simpler comparisons between ML and System F, we again assume that as in our definition of ML in Figure 2.1, there exists a collection of pre-defined data constructors D.

As mention before, term abstractions must be annotated with the type of the parameter. Type abstractions $\Lambda a.V$ and applications $M\,A$ provide first-class polymorphism, as mentioned. As we are mostly concerned with variants of ML that implement the value restriction, we present a version of System F that obeys the value restriction, too. This is why only values V may be abstracted over types. Further, we introduce let bindings of the form $\mathbf{let}\ x^A = M\ \mathbf{in}\ N$. In terms of their static and dynamic semantics, these behave as if they were syntactic sugar for $(\lambda x^A.N)\ M$. However, we keep them as primitives so that we may categorise bindings $\mathbf{let}\ x^A = V_1\ \mathbf{in}\ V_2$ as values. We assume a standard, call-by-value dynamic semantics of System F. Type and term contexts remain unchanged as compared to ML, but the latter now use types A.

The typing rules for System F are shown in Figure 2.8 on the next page. Similarly to ML, judgements $\Delta \vdash A\ \mathbf{ok}$ state that A is well formed in context Δ. We again require that in order for $\Delta; \Gamma \vdash^{\mathrm{F}} M : A$ to hold, all types in Γ must be well formed under Δ.

The existence of type annotations in terms means that a substitution property for System F has to apply the substitution to terms, too. This is in contrast to the corresponding property of ML (namely, Lemma 2.3), where this was not necessary.

Lemma 2.7.
If $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$*, then* $(\Delta, \Delta'); \Gamma \vdash^{\mathrm{F}} M : A$ *implies* $(\Delta, \Delta''); \delta(\Gamma) \vdash^{\mathrm{F}} \delta(M) : \delta(A)$.

Proof. By structural induction on M. □

$\boxed{\Delta;\Gamma \vdash^{\mathrm{F}} M : A}$

$$\frac{x : A \in \Gamma}{\Delta;\Gamma \vdash^{\mathrm{F}} x : A} \qquad \frac{\Delta;\Gamma \vdash^{\mathrm{F}} M : A \to B \quad \Delta;\Gamma \vdash^{\mathrm{F}} N : A}{\Delta;\Gamma \vdash^{\mathrm{F}} M N : B} \qquad \frac{(\Delta, a);\Gamma \vdash^{\mathrm{F}} V : A}{\Delta;\Gamma \vdash^{\mathrm{F}} \Lambda a.V : \forall a.A}$$

$$\frac{\Delta;(\Gamma, x : A) \vdash^{\mathrm{F}} M : B}{\Delta;\Gamma \vdash^{\mathrm{F}} \lambda x^{A}.M : A \to B} \qquad \frac{\Delta;\Gamma \vdash^{\mathrm{F}} M : \forall a.B \qquad \Delta \vdash A\ \mathbf{ok}}{\Delta;\Gamma \vdash^{\mathrm{F}} M A : B[A/a]}$$

$$\frac{\Delta;\Gamma \vdash^{\mathrm{F}} M : A \qquad \Delta;(\Gamma, x : A) \vdash^{\mathrm{F}} N : B}{\Delta;\Gamma \vdash^{\mathrm{F}} \mathbf{let}\, x^{A} = M\, \mathbf{in}\, N : B}$$

Figure 2.8: System F typing rules

Type inference for System F Type inference for System F is known to be undecidable. More specifically, the following two problems have been shown to be undecidable.

1. Wells [122] considered type inference for Curry-style System F, where terms contain no typing information at all.
2. Boehm [5] and Pfenning [93, 94] considered the version of the inference problem for Church-style System F, meaning that the term language is the one shown in Figure 2.7. However, it is extended with term abstractions and type applications not including types. Therefore, the latter new term form allows indicating the fact that a type application occurs, but without stating the argument.

Perhaps surprisingly, no relationship between the two problems is known in the sense that the negative result for each problem has not led to an answer to the other.

2.3 Existing approaches

This section gives on overview of the design space of type systems that extend the Hindley-Milner type system with first-class polymorphism. We do not discuss systems in detail that only provide higher-rank polymorphism but not polymorphic instantiation [92, 16].[2].

[2]While our work has little in common with that of Dunfield and Krishnaswami [16] from a technical point of view, there is a noteworthy similarity: The latter inspired the title of this work!

We also eschew discussing systems that are not conservative extensions of ML [95, 88], with some select exceptions. Thus, all systems discussed here *do* extend the HM typing discipline, unless explicitly stated otherwise.

2.3.1 Nominal types

An early approach for adding first-class (existentially or universally) quantified types to ML was to wrap them in dedicated nominal datatypes [100, 52, 85, 45]. Thus, such systems allow the arguments of data constructors to have quantified types. Instead of passing values of such quantified types directly, programmers must perform the necessary injection and projection themselves. This simplifies inference, as no unification with quantified types is necessary, only with nominal types.

As an example, consider the function $\mathsf{cull\_and\_combine}$, which we discussed in Section 1.1. In this style, we would introduce a datatype cull using

$$\mathbf{type}\ \mathsf{cull} = \mathsf{Cull}\ \mathbf{of}\ \ \forall b.\mathsf{List}\ b \to \mathsf{List}\ b$$

and redefine the function $\mathsf{cull\_and\_combine}$ to have type

$$\forall a\,b\,c.\,\mathsf{cull} \to (a \to b \to c) \to \mathsf{List}\ a \to \mathsf{List}\ b \to \mathsf{List}\ c.$$

We may then call it for example using

$$\mathsf{cull\_and\_combine}\ (\mathsf{Cull}\ \mathsf{every\_other})\ (\lambda x y. x + \mathsf{floor}\ y)\ [1; 2; 3; 4]\ [1.1; 2.2; 3.3; 4.4].$$

This illustrates the disadvantages of this approach for one-off usages, due to the necessity to introduce a dedicated datatype for every quantified type to be used in the system. Further, the explicit injection and projection increases verbosity.

2.3.2 Explicit systems

The following systems rely on explicit constructs to introduce and/or eliminate first-class polymorphism. We discuss these type systems in greater detail than others due to their closer relationship with FreezeML. We keep editing to a minimum and use the terminology introduced by the authors of each system, rather than trying to integrate their systems into the naming scheme used in Section 2.2.

2.3.2.1 IFX

The language IFX, proposed by O'Toole and Gifford [89], seems to be the first to conservatively extend ML while offering first-class polymorphism. Its syntax subsumes both ML and System F; its type language distinguishes two sorts of quantifiers: Types τ may contain arbitrary quantifiers $\forall$, denoting System F-style first-class polymorphic types $\forall\overline{a}.\tau'$. The system then carries ML-style type schemes $\widetilde{\forall}\overline{\alpha}.\tau$ in the term context. As seen above, $\widetilde{\forall}$ and $\forall$ quantify over variables α and a respectively. Only the latter are user-denotable in type annotations and type applications. Finally, monomorphic types μ arise from τ by disallowing quantifiers.

The syntax is shown in Figure 2.9. The ML and System F fragment of the language

Type identifiers	a
Generic type variables	α
User-provided types	$\upsilon ::= a \mid \upsilon_1 \to \upsilon_2 \mid \forall a.\upsilon$
Monomorphic types	$\mu ::= a \mid \alpha \mid \mu_1 \to \mu_2$
Polymorphic types	$\tau ::= a \mid \alpha \mid \tau_1 \to \tau_2 \mid \forall a.\tau$
Type schemes	$\eta ::= \tau \mid \widetilde{\forall}\alpha.\eta$
Type contexts	$A ::= \cdot \mid A, x : \eta$
Expressions	$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
	$\mid \Lambda a.e \mid \lambda(x:\upsilon).e \mid e\,\upsilon \mid \mathbf{open}\ e \mid \mathbf{close}\ e$

Figure 2.9: IFX syntax

are typed mostly independently: The type schemes of variables are implicitly instantiated with monotypes, let bindings introduce type schemes, monotypes are inferred for un-annotated lambda parameters. System F style type abstractions and applications introduce and eliminate $\forall$ quantified types. To mediate between both worlds, the term **open** e instantiates the toplevel $\forall$-quantifiers of e's type monomorphically. Conversely, **close** e converts the principal type scheme $\widetilde{\forall}\overline{\alpha}.\tau$ of e into $\forall\overline{a}.\tau[\overline{a}/\overline{\alpha}]$. Crucially, **close** e acts on the principal type scheme of e, rather than an arbitrary type. This is expressed in the following typing rule for the operator.

$$\frac{A \vdash e : \tau \qquad \overline{\alpha} = \mathsf{ftv}(\tau) - \mathsf{ftv}(A) \\ \text{for all } \tau' :\ A \vdash e : \tau' \text{ implies } A \vdash \tilde{\forall}\overline{\alpha}.\tau\ \leq\ \tilde{\forall}(\mathsf{ftv}(\tau) - \mathsf{ftv}(A)).\tau}{A \vdash \mathbf{close}\ e : \forall\overline{a}.\tau[\overline{a}/\overline{\alpha}]}$$

Here, $A \vdash \eta_1 \leq \eta_2$ is the generic instance relation, analogously to Figure 2.3, and ftv denotes free general type variables. Similar principality conditions were later adopted in other systems.

When typing function applications $e_1\ e_2$, the type system treats $\forall$ quantifiers in e_1's type specially, allowing them to be instantiated monomorphically to reveal a function type below.

The authors describe this as the two forms of polymorphism "coexisting" in IFX; it leaves mostly the programmer in charge to convert between the two forms. In total, the type system imposes principality conditions for three constructs, namely the **close** operator (as shown), type abstractions, and let bindings.

2.3.2.2 Poly-ML

The characteristic feature of Poly-ML[3] [26] is its usage of *labels* ε to track the origins of polymorphic types. Its syntax is shown in Figure 2.10.

Variables	$\xi ::= \alpha \mid \varepsilon$
Simple types	$\tau ::= \alpha \mid \tau \to \tau \mid [\sigma]^\varepsilon$
Polymorphic types	$\sigma ::= \tau \mid \forall \alpha . \sigma$
Type schemes	$\varsigma ::= \sigma \mid \forall \varepsilon . \varsigma$
Type contexts	$A ::= \cdot \mid A, x : \varsigma$
Expressions	$a ::= x \mid \lambda x . a \mid a\ a \mid \textbf{let}\ x = a\ \textbf{in}\ a \mid [a : \sigma] \mid \langle a \rangle \mid (a : \tau)$

Figure 2.10: Syntax of Poly-ML

Poly-ML distinguishes simple types τ from *polymorphic types* σ, which allow quantifying simple types over type variables α at the outermost level. Poly-ML uses a special type constructor $[\cdot]^\varepsilon$ that boxes a polymorphic type σ, turning it into a *polytype*. Note that each polytype is annotated with a label ε, we return to their role shortly. Polytypes are considered to be simple types, meaning that boxing a polymorphic type σ, turning it into $[\sigma]^\varepsilon$, allows embedding polymorphic types into the simple types.

Given a term a with a polytype $[\sigma]^\varepsilon$, its quantifiers are never instantiated implicitly. Instead, a dedicated opening operator $\langle \cdot \rangle$ allows unwrapping polytypes by instantiating

[3]The system was not given a name at the point of its original publication, but the name Poly-ML seems to have been given to it retroactively [56]. It should not be confused with Poly/ML [73, 96], an implementation of Standard ML.

the toplevel quantifiers of σ.

The label system is used to determine when terms may be opened. Note that labels ε appear as a separate form of variables in the type language. In addition to the aforementioned polymorphic types σ, Poly-ML uses *type schemes* ς, which allow quantifying simple types over type variables *as well as labels* at the outermost level.

Poly-ML's type system follows other declarative presentations of ML in that it allows arbitrary instantiation of type schemes at any point (but not of *polytypes* $[\sigma]^{\varepsilon}$, as mentioned above), as well as generalisation, subject to the usual restriction that the variable to generalise does not appear in the context A. However, in Poly-ML, these generalisation rules apply to type variables and labels alike. As a result, a term a of polytype $[\sigma]^{\varepsilon}$ can be generalised to $\forall\varepsilon.[\sigma]^{\varepsilon}$ if and only if ε does not appear in the context. The typing rule for the opening operator $\langle\cdot\rangle$ then imposes that it can only be applied to terms with type schemes of the form $\forall\varepsilon.[\sigma]^{\varepsilon}$, but not to plain polytypes $[\sigma]^{\varepsilon}$. The rationale for this rule is that Poly-ML treats the existence of a label ε in the context as a sign that the type $[\sigma]^{\varepsilon}$ was guessed, rather than provided by the user at some point.

Conversely, the only introductory form for polytypes is the wrapping operator, where $[a : \sigma]$ has type $[\sigma]^{\varepsilon}$ for some arbitrary ε, if a has type σ. Note that this allows choosing a fresh label ε, which may later be generalised. This is in line with the fact that the polytype of the term $[a : \sigma]$ was introduced using an explicit type annotation and not guessed.

As an example, consider the term $\lambda f.\langle f\rangle$. Poly-ML only allows simple types τ for function parameters. This leaves us with $f : [\sigma]^{\varepsilon}$ as the only choice for f's type, for some σ and ε. However, this means that $f : [\sigma]^{\varepsilon}$ (and hence ε) appears in the context while typing the body $\langle f\rangle$ of the function. This prevents ε from being generalised prior to opening it, which makes the opening operation ill-typed.

As Garrigue and Rémy point out, labels never need to appear in code in their system: Polytypes appearing in annotations may always omit labels, meaning that fresh labels are implicitly used in their place. However, users need to be able to reason about labels appearing in error messages and during inference.

Type annotations can be used in Poly-ML to freshen labels, which means that we may write $\lambda f.\langle(f : [\sigma])\rangle$ to make the earlier example well typed. Here, we may give the subterm $(f : [\sigma])$ the type $[\sigma]^{\varepsilon_1}$ for some ε_1 not appearing in the context, and can therefore generalise its type to $\forall\varepsilon_1.[\sigma]^{\varepsilon_1}$. This subsequently allows the term to be opened.

In addition to tracking that a polymorphic type originates from a type annotation rather than being guessed, the system can also determine when a polymorphic type was deduced from already confirmed polymorphism. Consider the following example, where $\lambda x : \tau.\, a$ is syntactic sugar for $\lambda x.\,\mathbf{let}\ x = (x : \tau)\ \mathbf{in}\ a$.

$$\begin{aligned}&\mathbf{let} f = \lambda(g : [\forall\alpha.\alpha \to \alpha]^{\varepsilon}).3\ \mathbf{in}\\ &\lambda x.(f\,x, \langle x\rangle)\end{aligned}$$

As usual, let-bound variables may be assigned type schemes. Recall that in Poly-ML, this includes quantifying labels at the toplevel.

We may therefore give f the type $\forall\varepsilon.([\forall\alpha.\alpha \to \alpha]^{\varepsilon} \to \mathsf{Int})$ while type-checking the body of the let binding. When type-checking $f\,x$, we may then instantiate ε to an arbitrary ε' not appearing in the context and infer $[\forall\alpha.\alpha \to \alpha]^{\varepsilon'}$ as the type for x. This in turn allows us to open x. This demonstrates how Poly-ML is able to propagate information from the confirmed polytype in f's type to x, without relying on bidirectional typing.

However, the label system can be somewhat difficult to understand, as it is sensitive to where type annotations appear. The first of the following terms is well typed, while the second one is not.

$$\begin{aligned}&\lambda f.\langle(f : [\forall\alpha.\alpha \to \alpha])\rangle\ f\\ &\lambda f.\langle f\rangle \qquad\qquad\qquad (f : [\forall\alpha.\alpha \to \alpha])\end{aligned}$$

Further, annotations are not propagated inwards in the way they would in a bidirectional setting. The terms $\mathbf{let}\ (g : [\forall\alpha.\alpha \to \alpha] \to \mathsf{Int}) = \lambda f.\langle f\rangle\ 3\ \mathbf{in}\ \ldots$ and $(\lambda f.\langle f\rangle\ 3 : [\forall\alpha.\alpha \to \alpha] \to \mathsf{Int})$ are both ill-typed.[4]

2.3.2.3 QML

Russo and Vytiniotis' QML follows IFX and Poly-ML by distinguishing two sorts of polymorphic types. Like IFX, they allow $\forall$ quantifiers to appear arbitrarily nested in types τ, while the type environment carries type schemes $\Pi\overline{\alpha}.\tau$. As opposed to IFX and Poly-ML, QML also allows existentially quantified types in order to allow data abstraction. The syntax of QML is shown in Figure 2.11; it subsumes the syntax of ML.

As in Poly-ML, universally quantified types are introduced using a type annotation in terms, they have the form $\{\overline{\alpha}\,\forall\beta.\tau\}\ e$. Here, the type variables $\overline{\alpha}$ may appear freely in τ and are considered existentially quantified in the sense of flexible type variables to be determined by the type-checker. Hence, the overall type of $\{\overline{\alpha}\,\forall\beta.\tau\}\ e$ is $\forall\beta.\tau[\overline{\tau'}/\overline{\alpha}]$ for appropriate $\overline{\tau'}$. Note that the $\overline{\tau'}$ may be arbitrarily quantified types, but must not mention β.

[4]Here, we consider $\mathbf{let}\ (f : \sigma) = a_1\ \mathbf{in}\ a_2$ to be syntactic sugar for $\mathbf{let}\ f = \langle[a_1 : \sigma]\rangle\ \mathbf{in}\ a_2$

$$\begin{array}{ll}
\text{Types} & \tau, \mu ::= \alpha \mid \tau \to \tau \mid \forall \alpha . \tau \mid \exists \alpha . \tau \\
\text{Type schemes} & \varsigma ::= \Pi \overline{\alpha} . \tau \\
\text{Type contexts} & \Gamma ::= \cdot \mid \Gamma, x : \varsigma \\
\text{Expressions} & e, u ::= x \mid \lambda x . e \mid e\ e \mid \textbf{let } x = u \textbf{ in } e \\
 & \qquad \mid \{\overline{\alpha}\ \forall \beta . \tau\}\ e \mid e\ \{\overline{\alpha}\ \forall \beta . \tau\} \\
 & \qquad \mid \{\overline{\alpha}\ \exists \beta . \tau\}\ e \mid \textbf{open } \{\overline{\alpha}\ \exists \beta . \tau\}\ x = u \textbf{ in } e
\end{array}$$

Figure 2.11: Syntax of QML

All $\forall$ quantifiers are eliminated by a dual term form $\{\overline{\alpha}\ \forall \beta . \tau\}\ e$, where $\forall \beta . \tau$ is the type of e, again modulo the instantiation of the free variables $\overline{\alpha}$ of τ. In particular, note that the original type of e *before* the instantiation is given by the user. This is in contrast to IFX, which allows monomorphic instantiation without type information and requires System F style type applications for polymorphic instantiation, and in contrast to Poly-ML, where the elimination form $\langle a \rangle$ of first-class polymorphism allows instantiation with (boxed) polymorphic types. However, as a result, binders never need type annotations in QML.

Existential packages are introduced analogously to universally quantified ones, using terms of the form $\{\overline{\alpha}\ \exists \beta . \tau\}\ e$. They are eliminated using the usual **open** construct, resulting in terms of the form **open** $\{\overline{\alpha}\ \exists \beta . \tau\}\ x = u$ **in** e. Note that again the full type of the term u is given, modulo the substitution of the variables $\overline{\alpha}$.

Russo and Vytiniotis see their approach towards first-class polymorphism based on fully explicit introduction and elimination forms as a refinement of earlier work relying on explicitly declared datatypes (as discussed in Section 2.3.1), but keeping types anonymous.

2.3.3 Systems using heuristics

We now discuss systems that use some sort of heuristics – in the widest sense of the term – in their type systems.

2.3.3.1 HMF

HMF [58] uses just System F types; its term language extends ML with type annotations on lambda parameters as well as general type annotations on terms. Unlike most other systems, it permits a somewhat declarative account for instantiation and generalisation, by using standalone typing rules for both applicable to arbitrary terms, in the style of the ordinary ML rules GEN and INST (as shown in Figure 2.2 on page 14), but with the latter rule allowing polymorphic instantiation. To retain decidable inference, the system imposes several restrictions.

1. Function parameters have monomorphic types, unless they are annotated.

2. Let bindings must use the principal type of the let-bound term.

3. An application $e_1\ e_2$ can only be typed in such a way that minimises the number of introduced polymorphic types compared to all other typings of the expression.

4. Functions must have return types without toplevel quantifiers, unless their bodies are type-annotated.

The first two conditions are fairly standard. The third condition means that the system prefers monomorphic instantiation over polymorphic instantiation whenever possible. This means that the term $\mathsf{singleton}\ \mathsf{id}$ can be typed as $\mathsf{List}\ (\sigma \to \sigma)$ for monomorphic types σ but choosing a polymorphic type for σ or using the overall type $\mathsf{List}\ (\forall\alpha.\alpha \to \alpha)$ for the term requires a type annotation.

As a result, considering binary applications only would result in an increased need for annotations. For example, consider the term $\mathsf{cons}\ \mathsf{id}\ \mathsf{ids}$, where $\mathsf{ids} : \mathsf{List}\ (\forall\alpha.\alpha \to \alpha)$ and $\mathsf{cons} : \forall\alpha.\alpha \to \mathsf{List}\ \alpha$. When only considering the subterm $\mathsf{cons}\ \mathsf{id}$ first, the minimality condition would not allow instantiating α with $\forall\beta.\beta \to \beta$. The fact that this instantiation is the *only* one that makes the overall term well typed only becomes apparent once the argument ids is also taken into account. As a result, HMF uses a typing rule for applications that takes arbitrary numbers of arguments into account, requiring that the *overall* application is typed in a way that minimises polymorphism. By using this rule, the example term above is accepted.

Finally, in order to permit functions whose return types have toplevel quantifiers, which is required to encode System F in HMF, Leijen proposes *rigid annotations*. This means that an expression annotated with a type prevents further instantiation or generalisation of that term, meaning that the type annotation denotes exactly the type of the term. Formally, this is achieved by distinguishing two categories of terms, based on whether or not a term is type-annotated. Then, the INST and GEN rules mentioned above only apply to the category of un-annotated terms.

2.3.3.2 Boxy Types

The key idea of the Boxy Types [118] approach is to propagate typing information similar to a bidirectional type system [95, 15].

In general, the defining feature of bidirectional systems is that they use two typing judgements, reflecting in which *mode* the system is. In *type checking mode*, the type of a term is known already and must merely be verified; *type synthesis mode* is used when the type is not known. Note that this distinction between two modes in the specification of the type systems reflects how type inference may be performed: In type checking mode, the type of a term may be considered part of the inputs of the algorithm, whereas it is part of the outputs in type inference mode. The typing rules for both judgements may then occasionally switch modes. For example, a type annotation on a term results in the system treating the annotated term in type checking mode, independently from the mode in which the overall term is encountered. Propagation of type information occurs, for instance, when a function is encountered in checking mode: The type of the function's parameter can directly be obtained from the (known) type of the overall function.

The Boxy Types system does not follow this bidirectional approach directly. Rather than having dedicated typing judgements for typing and inference mode each, the Boxy Types system indicates when inference mode is entered within types themselves. *Boxes* in types denote the parts of the type of a term that were inferred, whereas the non-boxed structure of a type denotes known (i.e., checked) parts of type. The resulting syntax of types is shown in Figure 2.12. Note that boxes are not nested, meaning that a box only contains non-boxed types.

Vanilla monotypes	$\tau ::= a \mid \tau_1 \to \tau_2 \mid \ldots$
Guarded vanilla types	$\rho ::= \tau \mid \rho \to \rho \mid \ldots$
Vanilla polytypes	$\sigma ::= \forall \overline{a}.\rho$
Boxy guarded types	$\rho' ::= \tau \mid \rho' \to \rho' \mid \boxed{\rho} \mid \ldots$
Boxy polytypes	$\sigma' ::= \forall \overline{a}.\rho' \mid \boxed{\sigma}$

Figure 2.12: Type language of Boxy Types system

One critique of the Boxy Types approach is that the system has a strong algorithmic flavour, with their type system being a reflection of a particular inference algorithm. In particular, as Vytiniotis et al. point out, there are no clear guidelines to where annotations are required; they expect programmers to experiment with the type-checker in those cases where the system does not behave as desired.

An extended version of the Boxy Types approach was used as the foundation of the `ImpredicativeTypes` extension of GHC, which was subsequently abandoned due to the complexity of the implementation and its unpredictable behaviour [27]. The extension was later re-implemented based on the Quick Look system (discussed in Section 2.3.3.4).

2.3.3.3 GI

The GI system [110] is named after the idea of providing *guarded impredicative polymorphism*. Its type language is that of System F, its term language simply extends ML with type applications on function parameters as well as on arbitrary terms.

The centrepiece of the system is its typing of n-ary function applications. When applying a function M (using our syntax for types and terms), the system follows a set of rules that determine how quantified variables a_i within M's type may be instantiated. Specifically, the system may allow each quantified variable a_i to be instantiated with either an arbitrarily polymorphic type, a type without toplevel quantifiers, or a fully monomorphic type. Fully polymorphic instantiation is only permitted if the variable a_i is guarded by (i.e., appears under) a type constructor within the function's curried parameter types, hence the system's name.

GI performs instantiation followed by generalisation of function arguments such that their types may match the corresponding parameter type. Each function call is typed without relying on information from the surrounding context, as it would be the case in the Boxy Types system or a bidirectional setting. Similarly to HMF, the system requires type annotations on function parameters if they are polymorphic, and on function bodies if the function's return type should be given a type with toplevel quantifiers.

GI does not perform let generalisation, unless the let-bound term is annotated with the desired result of generalisation. As a result, the system only ever needs to perform generalisation to yield a type whose toplevel quantifiers are known, which is obtained from the function being applied or a surrounding annotation. While this means that the system is not a conservative extension of ML, it nevertheless represents an interesting design point.

2.3.3.4 QuickLook

The Quick Look (QL) system [109] is a successor of GI proposed by the same authors. It adapts the work by Peyton Jones et al. [92] on using bidirectional typing to perform type inference in the presence of polymorphism beyond the limits of the HM system. The work by Peyton Jones et al. is limited to predicative systems, where bidirectionality helps reduce the number of necessary type annotations. For example, if the type of a function is known, it may be given a polymorphic parameter type without requiring an annotation.

Quick Look combines this idea with the ability to perform impredicative instantiation. This can either by achieved by providing explicit type arguments in the style of System F, or by what Serrano et al. refer to as having a “quick look” at the arguments of a function application. Thus, the system allows inferring impredicative instantiations of function types, inspired by ideas of the GI system. However, unlike GI, QL *does* utilise contextual information.

For example, assuming $f : (\mathsf{List}\ \forall a.a \to a) \to \mathsf{Int}$, the system is able to type f $(\mathsf{singleton\ id})$, by propagating the parameter type of f inwards to become the expected type of the subterm $\mathsf{singleton\ id}$. In contrast, GI requires annotating the subterm $(\mathsf{singleton\ id})$. However, the ability to propagate typing information to aid type inference of function arguments exceeds what is typically supported by bidirectional type systems (i.e., using a function’s parameter type to infer[5] the types of arguments). For example, the system allows some degree of information propagation *between* the types of different arguments to deduce impredicative instantiations.

Serrano et al. emphasise that the Quick Look approach is entirely local to function applications, which is also reflected in their inference algorithm. This allowed the approach to be implemented in GHC without requiring pervasive changes. Their core language even eschews let bindings, making it technically not a conservative extension of ML, but they state that the locality of the approach makes it simple to add generalising let bindings and more advanced language features.

2.3.4 MLF and its descendants

Le Botlan and Rémy state that they designed MLF [56] (also stylised as ML^{F}) as a continuation of their work on Poly-ML (as discussed in Section 2.3.2.2). The defining

[5] Using the terminology of bidirectional type inference, only the function’s type is *inferred*, while the types of arguments are *checked*.

feature of MLF is that it goes beyond the expressivity of System F types, by including *bounds* on quantifiers. As an example, consider again the term singleton id, which we used in Section 1.3. There, we described the challenge of deciding whether to instantiate id (leading to type List $a \to a$ of the overall term) or not (leading to type List $(\forall a.a \to a)$). In MLF, the principal type of the term is $\forall(b \geq (\forall a.a \to a)).\mathsf{List}\ b$. Here, the system need not commit to an instantiation decision, but can represent in the resulting type that the list contains elements whose type is an instance of $\forall a.a \to a$, including that type itself. This is achieved using *flexible bounds*, denoted $\geq$, on quantifiers such as b.

The term language of MLF is simply that of ML extended with type annotations on arbitrary terms. Type annotations on binders can then be derived as syntactic sugar for the former.[6]

The syntax of types is given in Figure 2.13. Note that polytypes maintain a prenex form: Quantifiers may only appear at the outermost level and in bounds. As a result, the System F type $(\forall a.a \to a) \to (\forall a.a \to a)$ is represented as $\forall(b = (\forall a.a \to a)).b \to b$. Here, a *rigid bound* is imposed on b, asserting that it is equivalent to its bound rather than an instance of it. A bound of the form $a \geq \bot$ may be used to represent ordinary ML types.

$$\begin{array}{ll} \text{Monotypes} & \tau ::= a \mid \tau \to \tau \mid \ldots \\ \text{Polytypes} & \sigma ::= \tau \mid \bot \mid \forall(a \geq \sigma).\sigma' \mid \forall(a = \sigma).\sigma' \end{array}$$

Figure 2.13: Syntax of MLF types

Typing in MLF happens under a *prefix*, giving $=$ or $\geq$ bounds to all type variables in scope. Prefixes are closed and order-dependent, similar to the usage of contexts in the work of Gundry et al. [35] as well as Dunfield and Krishnaswami [16].

MLF enjoys a principal types property and complete type inference, requiring annotations only on binders that are used polymorphically. Like many other systems, the typing rules of MLF then impose that monotypes τ must be used in certain situations. An important aspect of MLF's usage of such monomorphism conditions is that some typing derivations must then reason abstractly about a type by using a type variable in its place, while access to the variable's bound in the context is restricted.

As a result of its powerful type system, MLF programs require few annotations,

[6]Formally, all type annotations are desugared to term applications using a dedicated constant for each type, meaning that it suffices to have constants as language primitives.

while the system's specification remains entirely declarative. Only function parameters that are *used* polymorphically need annotations; no annotations are required elsewhere. The price of this impressive feat is the use of a somewhat involved unification algorithm, as Vytiniotis observes [106].

HML HML [59] is a simplification of MLF. It only uses flexible bounds and only allows System F types for function parameters (i.e., without carrying bounds on quantifiers). Let-bound variables still use quantified type schemes (i.e., types σ in Figure 2.13, but without *rigid* bounds). The result is a simple annotation rule where, unlike MLF, *all* function parameters with polymorphic types require type annotations. The embedding of System F into HML is even simpler than the one of the former into MLF, as it is no longer necessary to translate System F types with nested quantifiers into MLF types with rigidly bound quantifiers.

Leijen suggests a further simplification, called HML^{F}, where let-bound variables receive System F types, too. Given a term $\textbf{let}\ x = M\ \textbf{in}\ N$, typing derivations must use the principal type of M (possibly using arbitrarily nested flexible quantifiers), which is then translated to a System F type, thus sacrificing generality. However, this further limits the exposure of users to types with flexibly bound quantifiers as compared to MLF and HML, but does not fully eliminate it. Such types may still occur in error messages. Further, users who do not wish to reason about the principal HML type of a let-bound term M and its translation to a System F type may have to annotate *all* let bindings where M has a higher-rank type.

FPH The authors of the Boxy Types system proposed FPH [118] as another system directly based on MLF. Unlike Boxy Types, FPH has a declarative specification. It offers simple guideline that allows overapproximating where annotations must be placed, which is the same rule as in HML^{F} (i.e., on let and lambda binders with non-HM types). Although the system is implemented internally by reusing parts of the MLF machinery, its specification does not use any bounded quantification. Instead, it uses a notion of boxed types, albeit with a different meaning than in the Boxy Types work. Leijen [59] conjectures that FPH accepts a strict subset of the programs accepted by HML^{F}.

2.3.5 Remy's FML system

Rémy's FML [101] system (stylised as $\mathrm{F_{ML}}$) utilises the type language of System F as well as a predicative variant, denoted $\leq_{\mathrm{p}}^{\eta}$, of Mitchell's *containment* relation [80]. For example, the following relations hold (using our syntax for types).

$$\begin{array}{rcl}
\forall a.\mathsf{Int} \to (a \to a) & \leq_{\mathrm{p}}^{\eta} & \mathsf{Int} \to \forall a.(a \to a) \\
\mathsf{Unit} \to (\forall a.a \to a) & \leq_{\mathrm{p}}^{\eta} & \mathsf{Unit} \to (\mathsf{Int} \to \mathsf{Int}) \\
(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Unit} & \leq_{\mathrm{p}}^{\eta} & (\forall a.a \to a) \to \mathsf{Unit}
\end{array}$$

As illustrated here, the relation allows (monomorphic) instantiation of nested quantifiers, is contravariant in function types and allows hoisting of certain quantifiers. The FML system then satisfies the subsumption property that if term M has type A and $A \leq_{\mathrm{p}}^{\eta} B$, then M also has type B in the same context.

The system relies on the existence of coercion functions of type $A \to B$ for all such types with $A \leq_{\approx}^{\mathrm{F}} B$. Here, $A \leq_{\approx}^{\mathrm{F}} B$ is a variant of the generic instance relation (the original version is shown in Figure 2.3 on page 14) that allows impredicative instantiation, but only of toplevel quantifiers, and permits the reordering of all adjacent quantifiers, including nested ones.

These coercions must explicitly be applied whenever any *impredicative* instantiation should be performed. Further, for all let bindings **let** $x = M$ **in** N and applications $M\,N$, the system requires a type annotation on M whenever its type contains any quantifiers. In the case of let bindings, this is with respect to the type of M *before* generalisation occurs, otherwise the system would not be a conservative extension of ML.

Finally, Rémy provides a sophisticated elaboration mechanism that is run on source programs prior to type-inference. It propagates user-provided annotations throughout the term to reduce the burden of having to provide redundant annotations that can be deduced from other annotations due to (partial) overlap between them.

Chapter 3

FreezeML

In Section 1.2, we described the design goals for FreezeML to be such that it should

- have predictable behaviour,
- use System F types,
- permit type inference close to that for ML,
- have low syntactic overhead, and
- accommodate the value restriction.

In Section 3.1, we describe the design of FreezeML, guided by these design goals and formalise the type system in Section 3.2. We show translation between FreezeML and System F in Section 3.3. Section 3.4 addresses the fact that the typing rules for FreezeML terms reason about all typing derivations of the term, and shows that the resulting typing relation is nevertheless well founded.

3.1 The design of FreezeML

As per the design goals mentioned above, the type language of FreezeML is that of System F. The behavior of FreezeML is made predictable by exactly following ML with respect to where instantiation and generalisation occurs.

In the following, we outline the design of FreezeML guided by examples. For reference, their types are shown in Section 3.1. The first three functions were introduced in Chapter 1, the remaining two are standard.

$\mathsf{every\_other}$	$: \forall a.\mathsf{List}\ a \to \mathsf{List}\ a$
$\mathsf{cull\_and\_combine}$	$: \forall a\,b\,c.(\forall d.\mathsf{List}\ d \to \mathsf{List}\ d) \to (a \to b \to c) \to \mathsf{List}\ a \to \mathsf{List}\ b \to \mathsf{List}\ c$
$\mathsf{list\_cull\_and\_combine}$	$: \forall a\,b\,c.\mathsf{List}\ (\forall d.\mathsf{List}\ d \to \mathsf{List}\ d) \to (a \to b \to c) \to \mathsf{List}\ a \to \mathsf{List}\ b \to \mathsf{List}\ c$
$\mathsf{singleton}$	$: \forall a.a \to \mathsf{List}\ a$
head	$: \forall a.\mathsf{List}\ a \to a$

Figure 3.1: Example functions used in this chapter

3.1.1 Implicit instantiation and freezing

In Section 2.2.1.1, we discussed that the key observation leading to the syntax-directed presentation of ML is that it is sufficient to limit instantiation to act only on variables. When typing the example term $\mathsf{singleton}\ \mathsf{every\_other}$ from Section 1.1 in ML, the quantifiers in the type of each variable are eagerly instantiated. This means that the type $\forall a.\mathsf{List}\ a \to \mathsf{List}\ a$ of $\mathsf{every\_other}$ is instantiated to $\mathsf{List}\ A \to \mathsf{List}\ A$ for some type A, whereas the variable b of $\mathsf{singleton}$'s type $\forall b.b \to \mathsf{List}\ b$ is instantiated with the same type $\mathsf{List}\ A \to \mathsf{List}\ A$. This results in the overall term having the type $\mathsf{List}\ (\mathsf{List}\ A \to \mathsf{List}\ A)$.

We choose to preserve this behaviour of variables in FreezeML, meaning that all variables eagerly instantiate their (toplevel) quantifiers. However, a crucial difference is that we allow instantiation to be with polymorphic types. As a result, in FreezeML the type A above may contain quantifiers.

This leads to the question of how to type the term $\mathsf{list\_cull\_and\_combine}\ (\mathsf{singleton}\ \mathsf{every\_other})$, where we defined the type of the function $\mathsf{list\_cull\_and\_combine}$ in Section 1.1 such that it takes a parameter of type $\mathsf{List}\ (\forall a.\mathsf{List}\ a \to \mathsf{List}\ a)$. The approach taken by some systems is to maintain sufficient context in the type system to decide that when typing the overall term $\mathsf{list\_cull\_and\_combine}\ (\mathsf{singleton}\ \mathsf{every\_other})$, the subterm $\mathsf{every\_other}$ should either re-generalise after instantiation, or instantiation should be suppressed altogether. This may, for instance, be achieved by relying on bidirectionality, or taking (possibly nested) n-ary applications into account.

The approach we take in FreezeML is different: Given that variables unconditionally perform instantiation, we would still like to use variables in the style of System F, where a variable's type is simply taken from the environment. Thus, we introduce *frozen* variables $\lceil x \rceil$, which prevent instantiation. This means that only the term

$\mathsf{list\_cull\_and\_combine}\ (\mathsf{singleton}\ \lceil \mathsf{every\_other} \rceil)$, is well typed in FreezeML. While this means requiring the programmers to be more explicit about their intent, it avoids having to give the term $(\mathsf{singleton}\ \mathsf{every\_other})$ both of the following types, based on their surrounding context.

$$\mathsf{List}\ (\mathsf{List}\, A \to \mathsf{List}\, A) \tag{3.1}$$

$$\mathsf{List}\ (\forall a.\mathsf{List}\ a \to \mathsf{List}\ a) \tag{3.2}$$

In particular, there is no more general type of the term that both (3.1) and (3.2) would be (generic) instances of, even when allowing the (generic) instance relation to perform polymorphic instantiation.[1]

3.1.2 Explicit instantiation

We now consider a list $\mathsf{cull\_funs}$ of type $\mathsf{List}\ (\forall a.\mathsf{List}\ a \to \mathsf{List}\ a)$ containing polymorphic functions for culling lists, such as $\mathsf{every\_other}$. Using the usual head function, we may then use $\mathsf{head}\ \mathsf{cull\_funs}$ to retrieve the first such culling function.

However, the term $\mathsf{head}\ \mathsf{cull\_funs}\ [1;2;3;4]$ is ill-typed. FreezeML types *are* System F types in the sense that the quantified type $\forall a.\mathsf{List}\ a \to \mathsf{List}\ a$ of the subterm $\mathsf{head}\ \mathsf{cull\_funs}$ has no direct relationship with function types of the form $\mathsf{List}\, A \to \mathsf{List}\, A$. In particular, the System F type $\forall a.\mathsf{List}\ a \to \mathsf{List}\ a$ is not an ML type scheme (which would have all types of the form $\mathsf{List}\, A \to \mathsf{List}\, A$ as generic instances).

Instead, quantifiers must be eliminated in FreezeML. As discussed in the previous subsection, the mechanism for doing so is using variables.

As a result, the generic instance relation, or a variation thereof that permits polymorphic instantiation, is not useful in FreezeML to denote when a FreezeML type is more general than another one. The type $\forall a.a \to a$ is not more general than $\mathsf{Int} \to \mathsf{Int}$, as in general we cannot use terms of the former type in place of terms of the latter type. We therefore use a notion of principal types for FreezeML similar to the one introduced in Section 2.2.1.3, where other types can be obtained from the principal type by instantiating its free type variables. We formalise this notion in Section 3.2.1.1.

In our example, we may therefore introduce an intermediate variable to force in-

[1]Of course, this does not mean that it is impossible to define a type system together with a richer relation between quantified types that can relate these two types to a shared more general type of the term $\mathsf{singleton}\ \mathsf{every\_other}$. Examples for such relations include Mitchell's undecidable *containment* relation [80] as well as the decidable variants used by Rémy [101] and Peyton Jones et al. [92].

stantiation, using the following term instead.

$$\begin{array}{l}\textbf{let } f = \mathsf{head\ cull\_funs}\\ \textbf{in}\\ f\ [1;2;3;4]\end{array}$$

Observe that there is no generalisation occurring here, the type of f is exactly the type of the let-bound term, which is *already* polymorphic. We return to the question of when generalisation is performed by a let binding in the next subsection.

The introduction of intermediate variables can be tedious for one-off usages, we therefore introduce a postfix instantiation operator @, where M@ is syntactic sugar for **let** $\mathsf{tmp} = M$ **in** tmp. We may therefore rewrite the example as $(\mathsf{head\ cull\_funs})$@ $[1;2;3;4]$.

3.1.3 Implicit generalisation

Previously, we have shown how the behaviour of FreezeML with respect to instantiation is inspired by ML. Similarly, generalisation in FreezeML behaves just as in a syntax-directed presentation of ML, meaning that it is limited to let bindings. For example, the following term creates the culling function every_fourth by composing every_other with itself and then uses it when calling cull_and_combine.

$$\begin{array}{l}\textbf{let } \mathsf{every\_fourth} =\\ \quad \lambda l.\mathsf{compose\ every\_other\ every\_other}\ l\\ \textbf{in}\\ \mathsf{cull\_and\_combine}\ \lceil \mathsf{every\_fourth} \rceil (\lambda x\, y.x + \mathsf{floor}\ y)\ [1;\ 2;\ 3;\ 4]\ [1.1;\ 2.2;\ 3.3;\ 4.4]\end{array}$$

Note that due to the value restriction, we have to eta-expand the definition of every_fourth to enable generalisation.

3.1.4 Explicit generalisation

Analogously to the instantiation operator @, we define the prefix generalisation operator \$ as syntactic sugar for a let binding. Concretely, \$$M$ is defined as **let** $x = M$ **in** $\lceil x \rceil$, meaning that it generalises M and freezes the result. Due to the value restriction, such a let binding only performs generalisation if M is part of an appropriate syntactic category. We will therefore revisit the operator when introducing the syntax of FreezeML and restrict it such that it is only applicable to terms of an appropriate syntactic category, called *guarded values* in FreezeML. This category is a strict superset of the class of syntactic values of ML (as defined in Figure 2.1 on page 13), and therefore includes terms such as functions.

Thus, using the generalisation operator allows us to define a list of type $\mathsf{List}\ (\forall a.a \to a)$ as $\mathsf{singleton}\ \$(\lambda x.x)$.

3.1.5 Function parameters

In the presence of first-class polymorphism, function parameters may have polymorphic types, as we have seen in the example functions such as list_cull_and_combine discussed earlier in this section. This raises the question of inference: While all systems of which the author is aware of reject the term $\lambda f.(f\,3, f\,\mathsf{true})$, some allow inferring polymorphic parameter types under specific circumstances, using a variety of different mechanisms.

In FreezeML, just as IFX, HMF, and GI, we choose the simplest solution that is compatible with ML: Function parameters have monomorphic types, unless a type annotation is present at the binding. This means that FreezeML only accepts the example term above if it is changed to, say, $\lambda(f : \forall a.a \to a).(f\,3, f\,\mathsf{true})$. Further, the only necessary change to the definition of cull_and_combine, shown in Section 1.1, is a type annotation on the first parameter (i.e., the polymorphic culling function to be used).

3.1.6 Type variable scoping

The fact that FreezeML allows type annotations on binders raises the question of how free type variables in such annotations are treated. The answer is that FreezeML provides *lexically scoped type variables* [91]. Concretely, type annotations on let bindings may bind type variables, but taking the value restriction into account: Given a term $\mathbf{let}\ (x : \forall \Delta.H) = M\ \mathbf{in}\ N$, the type variables Δ quantified at the outermost level are bound in M if and only if M is a term that may be generalised, such as a function. This means that in the empty context, the term $\mathbf{let}\ (f : \forall a.a \to a) = \lambda(x : a).x\ \mathbf{in}\ \ldots$ satisfies the scoping rules for type variables in FreezeML, whereas $\mathbf{let}\ (f : \forall a.a \to a) = \mathsf{id}\ (\lambda(x : a).x)\ \mathbf{in}\ \ldots$ does not. The latter term is therefore considered ill-formed.

We discuss alternative approaches towards type variable scoping in Section 6.2.1.2.

3.1.7 Type equality

As usual, we consider types equivalent modulo alpha-renaming of quantified variables. Like System F, we do *not* consider them equivalent modulo reordering of quantifiers as well as the addition and removal of superfluous type variables. As a result, the following types are all considered different in FreezeML.

$$\begin{gathered}\forall abc.a \to b \to a\\ \forall ab.\ a \to b \to a\\ \forall ba.\ a \to b \to a\end{gathered}$$

In Section 6.2.1.1, we return to this matter and discuss a variant of FreezeML that does consider these types equivalent.

In conclusion, we can observe that FreezeML is similar to System F in the sense that quantified types must be introduced and eliminated explicitly. However, the fact that the introduction and elimination forms for quantified types are variables and let bindings, respectively, means that FreezeML behaves like ML when staying within the boundaries of ML, namely using toplevel quantification only.

3.2 Formal definition

We will now introduce FreezeML formally. Its syntax is shown in Figure 3.2 on the following page. In addition to defining meta-variables, we introduce names for select syntactic classes.

Types A are defined as in System F, but we distinguish two sub-categories of types. Monomorphic types S do not contain quantifiers at all, while guarded types G may only contain quantifiers if they appear guarded under a type constructor D, but not at the toplevel.

FreezeML terms M extend the syntax of ML with three constructs, all of which we discussed in the previous section: Frozen variables, annotated λ terms, and annotated let bindings. For the purposes of the value restriction, we introduce two sub-categories of (syntactic) values. As in ML, values V arises from M by removing applications. Guarded values U arise from values by removing frozen variables. For let bindings, we allow frozen variables in the let-bound term, but not the body of the binding. The

Type variables		a, b, c		
Type constructors		D	$::=$	$\to \mid \mathsf{Unit} \mid \ldots$
Types	$\mathsf{Type} \ni$	A, B	$::=$	$a \mid D\overline{A} \mid \forall a.A$
Monotypes	$\mathsf{MType} \ni$	S, T	$::=$	$a \mid D\overline{S}$
Guarded types	$\mathsf{GType} \ni$	G	$::=$	$a \mid D\overline{A}$
Type instantiations		δ	$::=$	$\emptyset \mid \delta[a \mapsto A]$
Restrictions		R	$::=$	$\bullet \mid \star$
Term variables		x, y, z		
Terms	$\mathsf{Term} \ni$	M, N	$::=$	$x \mid \lceil x \rceil \mid \lambda x.M \mid \lambda(x : A).M \mid M\,N$
				$\mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N$
Values	$\mathsf{Val} \ni$	V, W	$::=$	$x \mid \lceil x \rceil \mid \lambda x.M \mid \lambda(x : A).M$
				$\mid \mathbf{let}\ x = V\ \mathbf{in}\ W \mid \mathbf{let}\ (x : A) = V\ \mathbf{in}\ W$
Guarded values	$\mathsf{GVal} \ni$	U	$::=$	$x \mid \lambda x.M \mid \lambda(x : A).M$
				$\mid \mathbf{let}\ x = V\ \mathbf{in}\ U \mid \mathbf{let}\ (x : A) = V\ \mathbf{in}\ U$
Type contexts		Δ	$::=$	$\cdot \mid \Delta, a$
Term contexts		Γ	$::=$	$\cdot \mid \Gamma, x : A$

Figure 3.2: FreezeML syntax

idea of guarded values is that they have guarded types.[2]

We consider terms to be equivalent modulo alpha-conversion of both the term and type variables they bind. While the former simply allows renaming lambda and let binders, type variable bindings require additional care: As discussed in Section 3.1.6, binding of type variables only occurs in terms of the form $\mathbf{let}\ (x : \forall \overline{a}.H) = U\ \mathbf{in}\ N$, where the variables $\overline{a}$ are bound in U. Thus alpha-conversion of the type annotation $\overline{a}.H$ requires appropriate renaming within U.

As opposed to ML, type instantiations δ map variables to (arbitrarily polymorphic) types A in FreezeML. Instantiations in FreezeML obey a *restriction* R on the types in their codomain, which we return to when discussing the well-formedness rules for instantiations. Type contexts Δ and term contexts Γ remain unchanged as compared to to System F.

[2]There is a single exception of this rule: Given $\mathrm{bot} : \forall a_1 \ldots a_n.a_i$, we have that the guarded value bot can be given an arbitrary, non-guarded type such as $\forall a.a \to a$. We return to this in Section 3.2.1.3.

Well-formedness of types and contexts Similarly to our presentation of ML and System F in Section 2.2, our presentation of FreezeML is explicit about the type variables currently in scope. To this end, we define the well-formedness of term contexts and types under a given type context Δ. They are shown in Figure 3.3. Concretely, well-formedness judgements $\Delta \vdash \Gamma$ **ok** state that all types in Γ are well-formed with respect to Δ. The latter is formalised using judgements $\Delta \vdash_R A$ **ok**. Here, R is either $\bullet$ or $\star$, meaning that A is a monomorphic type S, or a potentially polymorphic type, respectively. We have that $\Delta \vdash_\bullet A$ **ok** (stating that A well formed *and* monomorphic) implies $\Delta \vdash_\star A$ **ok**. This is evoked by the last rule of the judgement $\Delta \vdash_R A$ **ok** in Figure 3.3. We can therefore consider restrictions to form a two-element lattice with $\star$ as the greatest element. We write $\Delta \vdash A$ **ok** as a shortcut for $\Delta \vdash_\star A$ **ok**.

$$\boxed{\Delta \vdash_R A\ \mathbf{ok}}$$

$$\frac{a \in \Delta}{\Delta \vdash_\bullet a\ \mathbf{ok}} \qquad \frac{\begin{array}{c}\mathsf{arity}(D) = n \\ \Delta \vdash_R A_1\ \mathbf{ok} \\ \cdots \\ \Delta \vdash_R A_n\ \mathbf{ok}\end{array}}{\Delta \vdash_R D\overline{A}\ \mathbf{ok}} \qquad \frac{(\Delta, a) \vdash_\star A\ \mathbf{ok}}{\Delta \vdash_\star \forall a.A\ \mathbf{ok}} \qquad \frac{\Delta \vdash_\bullet A\ \mathbf{ok}}{\Delta \vdash_\star A\ \mathbf{ok}}$$

$$\boxed{\Delta \vdash \Gamma\ \mathbf{ok}}$$

$$\Delta \vdash \cdot\ \mathbf{ok} \qquad \frac{\Delta \vdash A\ \mathbf{ok} \qquad \Delta \vdash \Gamma\ \mathbf{ok}}{\Delta \vdash \Gamma, x : A}$$

$$\boxed{\Delta; \Gamma \vdash M\ \mathbf{ok}}$$

$$\frac{x \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil\ \mathbf{ok}} \qquad \frac{x \in \Gamma}{\Delta; \Gamma \vdash x\ \mathbf{ok}} \qquad \frac{\Delta; (\Gamma, x : A) \vdash M\ \mathbf{ok}}{\Delta; \Gamma \vdash \lambda x.M\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\Delta \vdash A\ \mathbf{ok} \\ \Delta; (\Gamma, x : A) \vdash M\ \mathbf{ok}\end{array}}{\Delta; \Gamma \vdash \lambda(x : A).M\ \mathbf{ok}} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash M\ \mathbf{ok} \\ \Delta; \Gamma \vdash N\ \mathbf{ok}\end{array}}{\Delta; \Gamma \vdash M N\ \mathbf{ok}} \qquad \frac{\Delta; \Gamma \vdash M\ \mathbf{ok} \qquad \Delta; (\Gamma, x : A) \vdash N\ \mathbf{ok}}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N\ \mathbf{ok}}$$

$$\frac{\Delta \vdash A\ \mathbf{ok} \qquad (\Delta', A') = \mathsf{split}(A, M) \qquad (\Delta, \Delta') \vdash M\ \mathbf{ok} \qquad \Delta; (\Gamma, x : A) \vdash N\ \mathbf{ok}}{\Delta; \Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N\ \mathbf{ok}}$$

Figure 3.3: FreezeML well-formedness rules

Well-formedness of terms As discussed in the previous section, the toplevel quantifiers of the annotation A of a let-bound variable are bound in the let-bound term M if and only if M is a guarded value. We formalise this using the well-formedness judgement $\Delta;\Gamma \vdash M$ **ok** on terms, also shown in Figure 3.3. It checks that all type annotations only refer to type variables that are in scope according to the aforementioned scoping rules. Conversely, it also checks that M only refers to term variables in scope. For both checks, Δ and Γ denote the ambient contexts containing potential free variables.

The only interesting rules are those for annotated binders, which check that the type annotation A is well formed in the current context Δ. In addition, the typing rule for annotated let bindings also extends the context Δ with some Δ' when checking the well formedness of the let-bound term M. Here, Δ' is determined using the auxiliary function split, which is defined as follows.

$$\mathsf{split}(\forall\Delta'.H, M) = \begin{cases} (\Delta', H) & \text{if } M \in \mathsf{GVal} \\ (\cdot, \forall\Delta'.H) & \text{otherwise} \end{cases}$$

It evokes that Δ' corresponds to the outermost quantifiers of A if M is a guarded value, and is empty otherwise. Observe that the second element of the tuple returned by split is ignored by the well-formedness rule for annotated let bindings. We re-use split for a different purpose later, where the second element is used.

Note that the well-formedness relation only uses the term context Γ to track what variables are in scope, ignoring the types therein. Thus, the rules binding an unannotated variable term variable x add $(x : A)$ to the term context, where A is arbitrarily chosen.

FreezeML instantiations Similarly to our definition of ML, we use instantiations δ in the FreezeML typing rules. We augment the well-formedness judgements $\Delta \vdash \Delta' \Rightarrow \Delta''$ for instantiations used in ML (shown in Figure 2.5 on page 16), with a restriction R that all types in the codomain must obey. Therefore, $\Delta \vdash \Delta' \Rightarrow_{\bullet} \Delta''$ states that δ instantiates variables with monomorphic types, while using $\star$ instead removes this constraint. The rules of the judgement are shown in Figure 3.4 on the next page.

Indirect dynamic semantics We will not give a dynamic semantics of FreezeML directly, instead defining it in terms of the translation from FreezeML to System F, given in Section 3.3.2. We then immediately have that FreezeML's type system, defined in the next subsection, is sound due to the soundness of System F and the type-preserving nature of the translation (stated in Theorem 3.2). A similar approach is

$\boxed{\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta''}$

$$\frac{\Delta \# \Delta'}{\Delta \vdash \emptyset : \cdot \Rightarrow_R \Delta'} \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta'' \qquad (\Delta, \Delta'') \vdash_R A \ \mathbf{ok} \qquad \Delta \# (\Delta', a)}{\Delta \vdash \delta[a \mapsto A] : (\Delta', a) \Rightarrow_R \Delta''}$$

Figure 3.4: FreezeML instantiations

taken in HMF [57], GI [110] and the Boxy Types system [117]. In the latter work, Vytiniotis et al. argue that this has the following advantage, besides trivially obtaining a type soundness result: For languages whose evaluation or subsequent compilation is based on a System F-style intermediate representation, the formal definition of the dynamic semantics can more closely resemble the actual implementation. Both GHC and Links use such intermediate representations.

However, Russo and Vytiniotis [106] consider the ability to define a simple reduction semantics to be a design goal for their QML system, making it easier to reason about dynamic behaviour directly, rather than through an intermediary.

3.2.1 Typing rules

FreezeML typing judgements are of the form $\Delta; \Gamma \vdash M : A$, the rules are shown in Figure 3.5 on the following page. In order for any FreezeML typing judgement to hold, we also implicitly require the following well-formedness conditions to hold.

1. Γ is well formed under Δ (i.e., $\Delta \vdash \Gamma \ \mathbf{ok}$).
2. M is well formed under Δ and Γ (i.e., $\Delta; \Gamma \vdash M \ \mathbf{ok}$).

Also recall that we established the convention in Section 2.2.1 that any judgement, such as $\Delta; \Gamma \vdash M : A$, further imposes that Δ and Γ each consist of pairwise different elements.

The typing rules rely on the auxiliary relations $\mathsf{principal}$ and $\Updownarrow$ (discussed subsequently in Section 3.2.1.1) and the auxiliary function split (discussed subsequently in Section 3.2.1.3). They are formally defined in Figure 3.6 on page 47. While $\mathsf{principal}$ is used to impose that some derivations use principal types of certain terms, split and $\Updownarrow$ are used to implement the value restriction without requiring separate rules.

Observe that the typing rules are syntax-directed. Frozen variables are typed by simply returning the associated type from Γ. Plain variables x are typed similarly to ML (shown in Figure 2.6 on page 17), by instantiating the toplevel quantifiers of

$$\boxed{\Delta;\Gamma \vdash M : A}$$

$$\textsc{VarFrozen}\quad \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash \lceil x \rceil : A} \qquad\qquad \textsc{VarPlain}\quad \frac{x : \forall \overline{a}.H \in \Gamma \qquad \Delta \vdash \delta : \overline{a} \Rightarrow_\star \cdot}{\Delta;\Gamma \vdash x : \delta(H)}$$

$$\textsc{App}\quad \frac{\Delta;\Gamma \vdash M : A \to B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M\,N : B}$$

$$\textsc{LamPlain}\quad \frac{\Delta;(\Gamma, x : S) \vdash M : B}{\Delta;\Gamma \vdash \lambda x.M : S \to B} \qquad\qquad \textsc{LamAnn}\quad \frac{\Delta;(\Gamma, x : A) \vdash M : B}{\Delta;\Gamma \vdash \lambda(x : A).M : A \to B}$$

$$\textsc{LetPlain}\quad \frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\overline{a});\Gamma \vdash M : A' \qquad \mathsf{principal}(\Delta,\Gamma,M,\overline{a},A') \\ (\Delta,\overline{a},M,A') \Updownarrow A \qquad \Delta;(\Gamma, x : A) \vdash N : B\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

$$\textsc{LetAnn}\quad \frac{(\overline{a},A') = \mathsf{split}(A,M) \qquad (\Delta,\overline{a});\Gamma \vdash M : A' \qquad \Delta;(\Gamma, x : A) \vdash N : B}{\Delta;\Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B}$$

Figure 3.5: FreezeML typing rules

x's type in Γ. However, as opposed to ML, the instantiation is polymorphic, as denoted by the subscript $\star$ on the instantiation judgement in VARPLAIN. Applications are also standard. Un-annotated lambda abstractions are typed by choosing a monomorphic type S for the parameter. Annotated lambdas use the user-provided type A instead. Note that the implicit precondition of typing judgements regarding the well-formedness of contexts ensures that the premise of LAMANN implies that A is well formed in Δ.

3.2.1.1 Generalisation and principality

As discussed in Section 3.1.3, let bindings perform generalisation in FreezeML, subject to the value restriction.

Recall that in ML, the expression **let** $f = \lambda x.x$ **in** f 3 can be given the type Int

using different derivations.

- The subterm $\lambda x.x$ may be given the type $\mathsf{Int} \to \mathsf{Int}$, making this the type of f.
- The same subterm may be given type $a \to a$ for some fresh a, meaning that a can be generalised to obtain the ML type scheme $\forall a.a \to a$ for f. As a result, f must be instantiated in the body of the let binding prior to being applied to 3.

However, additional restrictions are required in FreezeML when typing **let** $x = M$ **in** N. Consider the slight variation **let** $f = \lambda x.x$ **in** $(f\ 3, \lceil f \rceil)$ of the example. If we allowed both types for f (i.e., $\mathsf{Int} \to \mathsf{Int}$ and $\forall a.a \to a$), the term could be given both of the following two types.

$$\mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int})$$
$$\mathsf{Int} \times (\forall a.a \to a)$$

In Section 3.1.2 we discussed that we will express principal types in FreezeML strictly in terms of instantiating free type variables, rather than using, say, an impredicative variant of the generic instance relation. However, there is no type A that we could give the term in FreezeML such that instantiating A would yield both types above. We observe that the term **let** $f = \lambda x.x$ **in** $\lceil f \rceil$ already exhibits the same problematic behaviour. Without further restrictions, the term could be given types such as $(\mathsf{Int} \to \mathsf{Int})$ and $(\forall a.a \to a)$, which are unrelated in FreezeML, and there is no more general type of the term that could be instantiated to yield both.

In practice, the problem also affects type inference: If we allowed both types for f, we would require some kind of backtracking during type inference. Inferring a type for **let** $f = \lambda x.x$ **in** $\lceil f \rceil\ 3$ would necessitate observing that f must be given type $\mathsf{Int} \to \mathsf{Int}$ instead of $\forall a.a \to a$ after processing its definition. Only the former choice for the type of f makes the term well typed.

We now introduce the notion of the principal type A for a term M in FreezeML under a given type context Δ and term context Γ, following a similar definition for ML in Section 2.2.1.3. It must be possible to obtain all other types of M by instantiating the free type variables of A, possibly with polymorphic types. We formalise this using the relation $\mathsf{principal}$, which is defined in Figure 3.6 on the next page. In a context consisting of Δ and Γ, $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ holds if A is a type of M. Here, Δ' explicitly denotes the free type variables of A that do not appear in the ambient type context Δ. It must then be the case that any other type of M can be obtained from A by

$$\mathsf{split}(\forall\Delta'.H, M) = \begin{cases} (\Delta', H) & \text{if } M \in \mathsf{GVal} \\ (\cdot, \forall\Delta'.H) & \text{otherwise} \end{cases}$$

$$\begin{array}{l} \mathsf{principal}(\Delta, \Gamma, M, \Delta', A') = \\ \quad (\Delta, \Delta'); \Gamma \vdash M : A' \ \text{ and} \\ \quad (\text{for all } \Delta'', A'' \mid \text{if } (\Delta, \Delta''); \Gamma \vdash M : A'' \\ \qquad\qquad\qquad \text{then there exists } \delta \text{ such that} \\ \qquad\qquad\qquad \Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta'' \text{ and } \delta(A') = A'') \end{array}$$

$\boxed{(\Delta, \overline{a}, M, A') \Updownarrow A}$

$$\frac{M \in \mathsf{GVal}}{(\Delta, \overline{a}, M, A') \Updownarrow \forall\overline{a}.A'} \qquad \frac{\Delta' = \overline{a} \qquad \Delta \vdash \delta : \Delta' \Rightarrow_{\bullet} \cdot \qquad M \notin \mathsf{GVal}}{(\Delta, \overline{a}, M, A') \Updownarrow \delta(A')}$$

Figure 3.6: Auxiliary functions and relations for FreezeML typing

instantiating the variables in Δ', potentially with polymorphic types. As an example, we have $\mathsf{principal}(\Delta, \cdot, \lambda x.x, a, a \to a)$ for any Δ with $a \# \Delta$.

On principal types and type schemes In Section 2.2.1.3, we provided the closely related definitions of principal types and principal type schemes in the context of ML. While the former instantiates free type variables, the latter instantiates existing quantifiers using the generic instance relation.

The alert reader may notice that this gives rise to a similar notion of type schemes in FreezeML. If $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ holds in FreezeML, it means that the variables in Δ' are *generic* in the sense that they do not appear in the ambient type context Δ. Thus, in contexts Δ and Γ, we have that Δ' together with A could be considered as the principal type scheme of A. We could make this explicit in FreezeML by introducing a separate universal quantifier such as $\widetilde{\forall}$. After all, this is what systems like IFX and QML do, as discussed in Section 2.3.2.

We avoid extending our language of types in this way due to the fact that principal types (and thus, a potential new notion of principal type schemes) are only used when typing let bindings in FreezeML. In particular, unlike in IFX, QML and Poly-ML, FreezeML term contexts Γ would never contain such type schemes.

3.2.1.2 Plain let bindings

To enforce the value restriction, the typing of terms **let** $x = M$ **in** N differs based on whether M is a guarded value or not. Before examining the rule LETPLAIN in Figure 3.5, which handles both cases, we first consider the following specialised rule for illustrative purposes. It handles the case that M is some guarded value U equivalently to LETPLAIN.

$$\frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\overline{a});\Gamma \vdash U : A' \qquad \mathsf{principal}(\Delta,\Gamma,U,\overline{a},A') \\ A = \forall\overline{a}.A' \qquad \Delta;(\Gamma,x:A) \vdash N : B\end{array}}{\Delta;\Gamma \vdash \textbf{let } x = U \textbf{ in } N : B}\ \textsc{LetPlain-Value}$$

The need for principality conditions discussed in Section 3.2.1.1 means that when typing terms **let** $x = U$ **in** N under context Δ and Γ, we use the principal type A' for U, meaning that we require $\mathsf{principal}(\Delta,\Gamma,U,\overline{a},A')$. Note that this implies $(\Delta,\overline{a});\Gamma \vdash U : A'$, which we include as a separate premise for the sake of clarity.

The type for x is then obtained by generalising the principal type A' of U, meaning that we quantify it over the variables in $\overline{a}$. Note that the definition of principal does not enforce that $\overline{a}$ are exactly the free type variables of A' without those in Δ, neither does it impose any ordering on the variables in Δ. As a result, all of the following hold.

$$\mathsf{principal}(\cdot,\cdot,\lambda xy.x,(a,b),\quad a \to b \to a) \tag{3.3}$$
$$\mathsf{principal}(\cdot,\cdot,\lambda xy.x,(b,a),\quad a \to b \to a) \tag{3.4}$$
$$\mathsf{principal}(\cdot,\cdot,\lambda xy.x,(a,b,c), a \to b \to a) \tag{3.5}$$

However, generalising the type $a \to b \to a$ by universally quantifying it over any of the sequences (a,b), (b,a), or (a,b,c) yields different FreezeML types. Thus, we additionally require $\overline{a} = \mathsf{ftv}(A') - \Delta$ as a premise in the typing rule, which only leaves (3.3) when typing a term of the form **let** $g = \lambda xy.x$ **in** N. This is due to the fact that we define the operator $\mathsf{ftv}(A')$ to return free type variables ordered by their first occurrence in A'. Further, we assume that this ordering is preserved under standard set and vector manipulations, such as removing Δ from $\mathsf{ftv}(A')$. The result of quantifying the principal type A' over the variables $\overline{a}$ is then the type of the variable x used in N. In the example term **let** $g = \lambda xy.x$ **in** N, this means that $\forall ab.a \to b \to a$ is the only possible type given to g when typing N. Similarly, the example term **let** $f = \lambda x.x$ **in** $\lceil f \rceil$ 3 from Section 3.2.1.1 is rejected in FreezeML.

The following lemma shows that principal types are unique modulo renaming the variables that appear freely in the type and are not part of the ambient context Δ.

Lemma 3.1 (Uniqueness of principal types modulo renaming of free variables)**.**
Let $\Delta \vdash \Gamma\,\mathbf{ok}$ *and* $\Delta;\Gamma \vdash M\,\mathbf{ok}$ *hold as well as* $\mathsf{principal}(\Delta,\Gamma,M,\Delta',A)$. *Then for all* Δ'', B *we have that* $\mathsf{principal}(\Delta,\Gamma,M,\Delta'',B)$ *holds iff we have*

- $\Delta \# \Delta''$ *and*
- *there exists a bijection* δ *between* $\mathsf{ftv}(A) - \Delta$ *and* $\mathsf{ftv}(B) - \Delta$ *such that* $\delta(A) = B$.

Proof. Follows easily from the definition of principal but requires appropriate strengthening/weakening properties (formalised in Lemma A.1 in Appendix A.1) as well as a simple substitution property, such as Lemma 4.1 introduced later in Section 4.1. □

This means that all possible types that can be given to x in LetPlain-Value are alpha-equivalent. Also note that the lemma enshrines the intuition that Δ is the ambient context by requiring $\Delta \vdash \Gamma\,\mathbf{ok}$ and $\Delta;\Gamma \vdash M\,\mathbf{ok}$. We could add these conditions directly to the definition of principal, but decided to keep it standard. In typing rules using principal, such as LetPlain-Value and LetPlain, both well-formedness conditions are satisfied due to the implicit preconditions of the typing judgement.

No generalisation must occur in let bindings when the let-bound term M is not a guarded value. However, the principality condition is still necessary in the non-generalising case. Consider the following term, where $\mathsf{bot} : \forall a.a$.

$$\mathbf{let}\ f = \mathsf{bot}\ \mathsf{bot}\ \mathbf{in}\ (f\ 3,\ \lceil f \rceil) \tag{3.6}$$

Without the principality condition, we may instantiate the non-value term bot bot to yield types such as $\mathsf{Int} \to A$ and $\forall a.a \to A$ and use these as the type for f. However, there would then be no principal type for f (which can be observed using $\lceil f \rceil$) subsuming all possible choices. Note that the quantifiers in the types for f result from instantiation in bot bot, rather than generalisation at the let binding. Therefore, a binding $\mathbf{let}\ x = M\ \mathbf{in}\ N$ where M is not a guarded value is typed by giving x a type that is obtained by instantiating the principal type A' of M, rather than generalising it.

However, we observe that arbitrary instantiation leads to the same problem we encountered to motivate the need for the principality condition when discussing the term (3.6) above. We have $\mathsf{principal}(\cdot,\Gamma,\mathsf{bot}\ \mathsf{bot},a,a)$, where Γ contains the aforementioned type of bot. If we allowed *any* instantiation of a to yield the type of f, we could again choose $f : \mathsf{Int} \to A$ or $\forall a.a \to A$, but there is no principal type for f that makes the term (3.6) well typed when polymorphic instantiations (such as the latter choice) are permitted. Therefore, we must impose that the type chosen for f must be obtained from the principal type by *monomorphic* instantiation only.

To obtain syntax-direct typing rules for FreezeML, we introduce a single typing rule LETPLAIN for the generalising and non-generalising case in Figure 3.5. As described above, we need to reason about the principal type A' of M in both cases. The two cases then only differ in how they obtain the type A for x from the principal type A' of M. This is achieved using the helper judgement $(\Delta, \overline{a}, M, A') \Updownarrow A$, also defined in Figure 3.6. Intuitively, we may consider $(\Delta, \overline{a}, M, A')$ to be the inputs of the relation, the possible choices for the output A then differ based on the syntactic category of M. If M is a guarded value, then generalisation takes place and A is uniquely determined as $\forall \overline{a}.A'$. Otherwise, no generalisation must take place and A is obtained from A' by applying some arbitrary monomorphic instantiation δ, acting on the variables $\overline{a}$.

3.2.1.3 Annotated let-bindings

Annotations on let-bound variables are used for two purposes in FreezeML, beyond just providing documentation. The first is inherently coupled with the use of principal types in un-annotated let bindings. Let Γ contain the function $\mathsf{take\_choose}$ of type $(\forall a.a \to a \to a) \to A$ for some A. We have $\mathsf{principal}(\cdot, \Gamma, \lambda x y.x, (a,b), a \to b \to a)$, as discussed in (3.3) on page 48, meaning that $\mathbf{let}\ \mathsf{choose} = \lambda x y.x\ \mathbf{in}\ \mathsf{take\_choose}\ \lceil \mathsf{choose} \rceil$ is ill-typed. An appropriate type annotation that imposes a non-principal type for choose makes the term well typed.

$$\mathbf{let}\ (\mathsf{choose} : \forall a.a \to a \to a) = \lambda x y.x\ \mathbf{in}\ \mathsf{take\_choose}\ \lceil \mathsf{choose} \rceil$$

We discussed the second use of annotated let bindings in Section 3.1.6, namely the fact that we chose a scoping behaviour for type variables in FreezeML where annotated let bindings may bind type variables, if the let-bound term is a guarded value.

As a result, some let bindings need to be annotated in order to bind a type variable needed in the annotation of a polymorphic function parameter. As an example, consider the following term.

$$\begin{array}{l}
\mathbf{let}\ (\mathsf{ep} : \forall b.(\forall a.(a \times \mathsf{Int} \times \mathsf{String}) \to b) \to b) = \\
\quad \lambda(f : \forall a.(a \times \mathsf{Int} \times \mathsf{String}) \to b).f\ (\text{“hello”}, 5, \text{“hello”}) \\
\mathbf{in} \\
\mathsf{singleton}\ \lceil \mathsf{ep} \rceil
\end{array}$$

This example illustrates the usual encoding of existential packages using universal types, where $\exists a.A := \forall b.(\forall a.A \to b) \to b$. Using this abbreviation, the type of ep is $\exists a.(a \times \mathsf{Int} \times \mathsf{String})$, and the overall type of the term is $\mathsf{List}\ \exists a.(a \times \mathsf{Int} \times \mathsf{String})$. We may interpret the latter type as a heterogeneous list of elements together with the size of each element and a representation thereof as a string. Concretely, ep is then a package consisting of the string "hello" together with its size and the string itself. Crucially, the annotation on ep brings the type variable b into scope, which is then used freely in the type annotation on f.

To implement the distinction between the generalising and non-generalising case, we use the helper function split in the rule LETANN in Figure 3.5 on page 45. The function split was originally introduced in Section 3.2 for the purposes of defining well-formedness of terms and is shown again in Figure 3.6. Its usage in LETANN splits all toplevel quantifiers from the type annotation A if the let bound-term M is a guarded value. Otherwise, the result $\overline{a}$ returned by split is empty, meaning that no generalisation is performed and M must have exactly type A.

Guarded values and their types At the beginning of Section 3.2, we mentioned the intuition behind the definition of guarded values U being that their types should be guarded (i.e., not have toplevel quantifiers). This property is what enables the scoping rules of annotated let bindings, as implemented by the well-formedness judgement $\Delta;\Gamma \vdash M$ **ok** in Figure 3.3. Given **let** $(x : \forall\Delta'.H) = M$ **in** N, where M is a guarded value U, we know without type-checking U that the quantifiers Δ' cannot originate from the type of U itself, due to its type being guarded.

Instead, the user must have intended for these quantifiers to be the result of generalisation, meaning that the let binding under consideration introduces these quantifiers, rather than them already being present in the type of U itself. In other words, the sequence $\overline{a}$ in the rule LETANN in Figure 3.5 on page 45 corresponds exactly to the type variables being generalised by the let binding itself. If M is not a guarded value, then the usage of split in that rule ensures that $\overline{a}$ is empty, corresponding to the fact that no generalisation must occur. This property of LETANN is important for the translation from FreezeML to System F, defined subsequently in Section 3.3.2. The translation must be able to introduce System F explicit type abstractions exactly where FreezeML generalisation introduces quantifiers.

As mentioned before, the only guarded value that may be given a non-guarded type is a variable bot with $\mathsf{bot} : \forall a_1 \ldots a_n.a_i \in \Gamma$, or one or more nested let bindings whose body is in turn such a variable bot. In this case, the type of the term bot may have a type A with toplevel quantifiers, due to a_i being instantiated with A directly. To preserve the property that $\overline{a}$ in LETANN corresponds to the type variables being generalised, we must rule out the case where a_i is instantiated with such a type A with toplevel quantifiers. This is achieved by the function split, whose definition in Figure 3.6 dictates that if M is a guarded value, then its type, called A' in LETANN, is a guarded type H.

As a result, a term such as $\mathbf{let}\ (f : \forall abc.a) = \mathsf{bot}\ \mathbf{in} \ldots$ remains well typed in FreezeML, but its typing derivation instantiates bot's type with the variable a, which is then generalised (and syntactically brought into scope) at the let binding. Instantiating bot's type with $\forall abc.a$ directly is not possible.

Finally, we observe that when only considering principal types, the property that guarded terms have guarded type holds unconditionally. The principality condition rules out instantiating bot and its variations with a quantified type.

Lemma 3.2.
Let $\Delta \vdash \Gamma$ **ok** *and* $\Delta;\Gamma \vdash M$ **ok** *hold. If* $\mathsf{principal}(\Delta,\Gamma,U,\Delta',A)$*, then* A *has no toplevel quantifiers.*

Proof. The proof works by reasoning about the type inference algorithm presented in Chapter 4 and is therefore left until Appendix A.6. □

3.2.2 Operators as syntactic sugar

In Section 3.1, we defined the operators \$ and @ to perform generalisation and instantiation, respectively. We defined them as syntactic sugar using plain let bindings. We now add a third operator, a variation of the generalisation operator \$ that is annotated with the type that should result from generalisation. It is desugared to an annotated let binding. Just like its un-annotated counterpart \$, the new operator is only defined on guarded values U.

Altogether, the definitions of the three operators are as follows.

$$M@ \quad := \quad \mathbf{let}\ x = M\ \mathbf{in}\ x \tag{3.7}$$

$$\$U \quad := \quad \mathbf{let}\ x = U\ \mathbf{in}\ \lceil x \rceil \tag{3.8}$$

$$\$^{A}U \quad := \quad \mathbf{let}\ (x : A) = U\ \mathbf{in}\ \lceil x \rceil \tag{3.9}$$

By unfolding the syntactic sugar, we can obtain direct typing rules for each operator. One may hope that the resulting typing rule for the instantiation operator would be equivalent to the following, simple rule. It permits polymorphic instantiation of the toplevel quantifiers of M's type.

$$\textsc{InstWrong}\quad \frac{\Delta;\Gamma \vdash M : \forall\Delta'.H \qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot}{\Delta;\Gamma \vdash M@ : \delta(H)}$$

Unfortunately, this is not the case. In fact, the rule is both unsound and incomplete with respect to the typings of **let** $x = M$ **in** x.

Consider the term $(\mathsf{id}\ \mathsf{id})@$, which is desugared to **let** $x = \mathsf{id}\ \mathsf{id}$ **in** x. We have $\mathsf{principal}(\Delta,\Gamma,\mathsf{id}\ \mathsf{id},a,a \to a)$ where $a \notin \Delta$. Due to the term id id being non-generalisable, x is assigned a type that results from *monomorphically* instantiating $a \to a$. This shows that the @ operator can affect how instantiation is performed within the term it is applied to, beyond just instantiating toplevel quantifiers. For instance, this means that $\Delta;\Gamma \vdash M : H$ does not in general imply $\Delta;\Gamma \vdash M@ : H$. This is witnessed by $\Delta;\Gamma \nvdash (\mathsf{id}\ \mathsf{id})@ : (\forall b.b) \to (\forall b.b)$, as this would require polymorphic instantiation of the type variable a. However, the latter judgement would hold according to InstWrong, showing its unsoundness.

Further, we have that $\Delta;\Gamma \vdash$ **let** $x = \lambda x.x$ **in** $x : (\forall b.b) \to (\forall b.b)$ holds, but the rule InstWrong does not permit this typing of $(\lambda x.x)@$. This is due to the fact that InstWrong does not take the generalisation into account that the underlying let binding may perform. This illustrates the rule's incompleteness.

In order to reflect the typing of the binding **let** $x = M$ **in** N that the term $M@$ desugars to, its direct typing rules rely on the principal type of M. The typing rule InstQuantified in Figure 3.7 on the following page handles the case where M's principal type is $\forall\overline{a}.H$ for some non-empty $\overline{a}$ (i.e., the principal type is indeed quantified). By Lemma 3.2, this implies that M is not a guarded value. This means that any generic variables $\overline{b}$ (i.e., variables not part of the ambient context) that may appear in H are instantiated monomorphically by the let binding when obtaining a type for x.

As a result, the overall type of $M@$ must result from instantiating the variables $\overline{a}$ polymorphically (happening at the use-site of the variable x in the desugared term) while instantiating those in $\overline{b}$ monomorphically (happening at the let binding in the desugared term).

As an example, let $(f : \forall b.b \to \forall a.a \to b) \in \Gamma$ and we consider applying the instantiation operator to the term $M := \mathsf{f}\ \mathsf{id}$. We then have $\mathsf{principal}(\cdot,\Gamma,M,b,\forall a.a \to (b \to b))$. When applying the rule InstQuantified we then have $\overline{a} = a$ and $\overline{b} = b$. We may thus instantiate a polymorphically, with types such as $\forall c.c$, while b must be instantiated monomorphically. This results in the following, reflecting the behaviour of the term **let** $x = \mathsf{f\ id}$ **in** x.

$$\begin{array}{lll} \cdot;\Gamma \vdash (\mathsf{f\ id})@ & : & (\forall c.c) \to (\mathsf{Int} \to \mathsf{Int}) \\ \cdot;\Gamma \nvdash (\mathsf{f\ id})@ & : & (\forall c.c) \to ((\forall d.d) \to (\forall d.d)) \end{array}$$

$$\textsc{InstQuantified}\quad \frac{\mathsf{principal}(\Delta,\Gamma,M,\overline{b},\forall\overline{a}.H) \qquad \overline{a} \neq \cdot \qquad \Delta \vdash \delta_{\mathrm{a}} : \overline{a} \Rightarrow_{\star} \cdot \qquad \Delta \vdash \delta_{\mathrm{b}} : \overline{b} \Rightarrow_{\bullet} \cdot}{\Delta;\Gamma \vdash M@ : \delta_{\mathrm{a}}(\delta_{\mathrm{b}}(H))}$$

$$\textsc{InstGuarded}\quad \frac{\mathsf{principal}(\Delta,\Gamma,M,\Delta',H) \qquad R = \begin{cases} \star & \text{if } M \in \mathsf{GVal} \\ \bullet & \text{otherwise} \end{cases} \qquad \Delta \vdash \delta : \Delta' \Rightarrow_R \cdot}{\Delta;\Gamma \vdash M@ : \delta(H)}$$

$$\textsc{GenPlain}\quad \frac{\mathsf{principal}(\Delta,\Gamma,U,\Delta',A) \qquad \Delta' = \mathsf{ftv}(A) - \Delta}{\Delta;\Gamma \vdash \$U : \forall\Delta'.A} \qquad\qquad \textsc{GenAnn}\quad \frac{(\Delta,\Delta');\Gamma \vdash U : H}{\Delta;\Gamma \vdash \$^{\forall\Delta'.H}U : \forall\Delta'.H}$$

Figure 3.7: FreezeML typing rules for syntactic sugar

Alternatively, in the case that M's principal type is guarded, the term **let** $x = M$ **in** x resulting from expanding $M@$ may either perform generalisation (if $M \in \mathsf{GVal}$) followed by polymorphic instantiation of the temporary variable x, or instantiate monomorphically (if $M \notin \mathsf{GVal}$). Both cases are handled by the rule INSTGUARDED.

As it is somewhat unintuitive to apply the instantiation operator to a term whose (principal) type has no toplevel quantifiers to begin with, it may be advisable to emit a warning when the rule INSTGUARDED applies to a usage of @, and only consider the cases handled by INSTQUANTIFIED as appropriate usages of @.

The remaining two rules in Figure 3.7 handle both versions of the generalisation operator \$. They straightforwardly implement the expected behaviour, due to both operators being limited to apply to guarded values only. Hence, the resulting let bindings can always perform generalisation. Note that in GENPLAIN, Lemma 3.2 guarantees that the type A is actually a guarded type H.

We can then easily check the correctness of the rules in Figure 3.7.

Lemma 3.3.
The typing rules in Figure 3.7 are sound and complete with respect to the typings of the terms resulting from expanding the definitions (3.7) to (3.9).

Proof. When considering $\Delta;\Gamma \vdash M@ : A$, the typing derivation of its expansion **let** $x = M$ **in** x has the following form for some $A', \Delta', \delta, \Delta'', H''$, where $\delta(H'') = A$ must hold.

$$\dfrac{\begin{array}{c}\dfrac{(x : \forall\Delta''.H'') \in (\Gamma, x : \forall\Delta''.H'') \qquad \Delta \vdash \delta : \Delta'' \Rightarrow_{\star} \cdot}{\Delta; (\Gamma, x : \forall\Delta''.H'') \vdash x : \delta(H'')} \\ \Delta' = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\Delta');\Gamma \vdash M : A' \qquad \mathsf{principal}(\Delta,\Gamma,M,\Delta',A') \\ (\Delta,\Delta',M,A') \Updownarrow \forall\Delta''.H''\end{array}}{\Delta;\Gamma \vdash \textbf{let } x = M \textbf{ in } x : \delta(H'')}$$

We distinguish two cases.

1. If $M \in \mathsf{GVal}$, the definition of $\Updownarrow$ imposes $\forall\Delta''.H'' = \forall\Delta'.A'$. Due to Lemma 3.2, we further have that M has a guarded principal type H', yielding $\forall\Delta''.H'' = \forall\Delta'.H'$.

   We now show that the derivation above is valid if and only if $\Delta;\Gamma \vdash M : \delta(H'')$ is derivable using INSTGUARDED. The premises of the rule when $M \in \mathsf{GVal}$ are then equivalent to the following for some Δ''', and H''', with the overall type being $\delta'''(H''')$.

$$\begin{array}{c}\mathsf{principal}(\Delta,\Gamma,M,\Delta''',H''') \\ \Delta \vdash \delta''' : \Delta''' \Rightarrow_{\star} \cdot\end{array}$$

   Due to the strengthening/weakening properties of principal (see Lemma A.1 in Appendix A.1) these conditions are equivalent to the situation where $\Delta''' = \mathsf{ftv}(H''') - \Delta$ holds, which we may therefore assume w.l.o.g. We then have that these conditions hold if and only if the derivation above for **let** $x = M$ **in** x is valid, by equating $\Delta'' = \Delta'''$, $H'' = H'''$ and $\delta = \delta'''$.

2. Otherwise, if $M \notin \mathsf{GVal}$, the definition of $\Updownarrow$ imposes $\forall\Delta''.H'' = \delta'(A')$ for some δ' with $\Delta \vdash \delta' : \Delta' \Rightarrow_{\bullet} \cdot$. We further distinguish two sub-cases, based on whether the principal type A' of M has toplevel quantifiers or not.

   (a) If the type A' of M is guarded, we have that Δ'' is empty. The premises of INSTGUARDED are then again equivalent to the following (but the rule imposes a monomorphic instantiation this time), and we may again assume w.l.o.g. that $\Delta''' = \mathsf{ftv}(H''') - \Delta$.

$$\mathsf{principal}(\Delta, \Gamma, M, \Delta''', H''')$$
$$\Delta \vdash \delta''' : \Delta''' \Rightarrow_{\bullet} \cdot$$

We may therefore establish the validity of these conditions by equating $\delta' = \delta'''$, $\Delta' = \Delta'''$ and $H''' = A'$.

(b) Otherwise, if A' has a non-empty sequence $\overline{a}$ of toplevel quantifiers, (i.e., $A' = \forall \overline{a}.H$ for some H), we show that the typing derivation for the let binding shown earlier is valid if and only if the rule InstQuantified is applicable. The premises of the latter are as follows.

$$\mathsf{principal}(\Delta, \Gamma, M, \overline{b}, \forall \overline{a}.H) \quad \overline{a} \neq \cdot$$
$$\Delta \vdash \delta_{\mathsf{a}} : \overline{a} \Rightarrow_{\star} \cdot \quad \Delta \vdash \delta_{\mathsf{b}} : \overline{b} \Rightarrow_{\bullet} \cdot$$

As before, we assume w.l.o.g. that $\overline{b} = \mathsf{ftv}(\forall \overline{a}.H) - \Delta$. We may equate $H'' = \delta_{\mathsf{b}}(H)$, $\Delta' = \overline{a}$, $\Delta'' = \overline{b}$, $\delta' = \delta_{\mathsf{b}}$ and $\delta = \delta_{\mathsf{a}}$. We then have that the derivation for **let** $x = M$ **in** x is valid iff the premises of InstQuantified are satisfied and we further have $\delta(H'') = \delta_{\mathsf{a}}(\delta_{\mathsf{b}}(H))$ (i.e., both derivations yield the same type).

The correctness of the direct typing rules for the operators $\$U$ and $\$^{A}U$ follows immediately when adapting the typing rules for plain and annotated let bindings, respectively, to the generalisable case. □

3.3 From System F to FreezeML to System F

We now show that FreezeML is indeed as expressive as System F by showing an embedding of the latter in the former. We also discuss the reverse direction.

3.3.1 Embedding System F into FreezeML

The translation from System F to FreezeML relies on type information. To this end, we extend System F typing judgements $\Delta; \Gamma \vdash^{\mathsf{F}} M_{\mathrm{F}} : A$ (defined in Figure 2.8 on page 21) to $\Delta; \Gamma \vdash^{\mathsf{F}} M_{\mathrm{F}} : A \rightsquigarrow M$, where M_{F} is a System F term and M is the translated FreezeML term. Throughout this section, non-terminals standing for System F terms receive a subscript F to help distinguish them from FreezeML terms.

The translation is designed such that the translation M has the same type A as the original term M_{F} and further, it is the *only* such type of M. The rules are shown in Figure 3.8. Alternatively, we could define the translation on System F typing derivations using the original typing judgement. The typing rules themselves remain unchanged as compared to Figure 2.8.

$$\boxed{\Delta;\Gamma \vdash^{\mathrm{F}} M_{\mathrm{F}} : A \leadsto M}$$

$$\frac{x : A \in \Gamma}{\Delta;\Gamma \vdash^{\mathrm{F}} x : A \leadsto \lceil x \rceil} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash^{\mathrm{F}} M_{\mathrm{F}} : A \to B \leadsto M \\ \Delta;\Gamma \vdash^{\mathrm{F}} N_{\mathrm{F}} : A \leadsto N\end{array}}{\Delta;\Gamma \vdash^{\mathrm{F}} M_{\mathrm{F}}\, N_{\mathrm{F}} : B \leadsto M\, N}$$

$$\frac{\Delta;(\Gamma, x : A) \vdash^{\mathrm{F}} M_{\mathrm{F}} : B \leadsto M}{\Delta;\Gamma \vdash^{\mathrm{F}} \lambda x^{A}.M_{\mathrm{F}} : A \to B \leadsto \lambda(x : A).M} \qquad \frac{(\Delta, a);\Gamma \vdash^{\mathrm{F}} V_{\mathrm{F}} : B \leadsto M}{\Delta;\Gamma \vdash^{\mathrm{F}} \Lambda a.V_{\mathrm{F}} : \forall a.B \leadsto \$^{\forall a.B}(M@)}$$

$$\frac{\Delta;\Gamma \vdash^{\mathrm{F}} M_{\mathrm{F}} : \forall a.B \leadsto M}{\Delta;\Gamma \vdash^{\mathrm{F}} M_{\mathrm{F}}\, A : B[A/a] \leadsto \mathbf{let}\ x = M\ \mathbf{in}\ \$^{B[A/a]}\, x}$$

$$\frac{\Delta;\Gamma \vdash^{\mathrm{F}} M_{\mathrm{F}} : A \leadsto M \qquad \Delta;(\Gamma, x : A) \vdash^{\mathrm{F}} N_{\mathrm{F}} : B \leadsto N}{\Delta;\Gamma \vdash^{\mathrm{F}} \mathbf{let}\ x^{A} = M_{\mathrm{F}}\ \mathbf{in}\ N_{\mathrm{F}} : B \leadsto \mathbf{let}\ x = M\ \mathbf{in}\ N}$$

Figure 3.8: System F typing rules with translation to FreezeML

System F variables are translated to frozen variables. For term abstractions and applications, we simply translate their subterms. The translations of these three term forms do not rely on typing information.

The translations of System F type abstractions and applications, which work on one quantifier at a time, need to bridge the behavioural gap with FreezeML, which may in general instantiate or generalise multiple quantifiers at once. A System F term $\Lambda a.V_{\mathrm{F}}$ is translated to $\$^{\forall a.B}\,(M@)$, where M is the translation of V_{F} and B is its type. Recall that desugaring the generalisation operator yields $\mathbf{let}\ (x : \forall a.B) = M@\ \mathbf{in}\ \lceil x \rceil$.

Here, we need to instantiate M in order to ensure that the let-bound term is a guarded FreezeML value, enabling generalisation. In general, the translation preserves non-expansiveness between the two systems, meaning that System F values are turned into FreezeML values.

Lemma 3.4.

Let V_F be a System F value. If $\Delta;\Gamma \vdash^{F} V_F : A \rightsquigarrow M$, then M is a FreezeML value.

Proof. By structural induction on V_F. □

However, when applying this lemma to the translation of $\Lambda a.V_F$, we observe that it only guarantees that the translation M of V_F is a FreezeML value, but not necessarily a *guarded* value. Therefore, in the translation of System F type abstractions, we use $M@ \equiv \textbf{let}\ y = M\ \textbf{in}\ y$ as the term bound by the outer, annotated let binding. Here we rely on the fact that if M is a value, then $M@$ is a guarded value, making generalisation in FreezeML possible. Note that overall, the annotated let binding evokes that those toplevel quantifiers of B (i.e., the type of V_F), that were instantiated when writing $M@$ are immediately re-generalised, in addition to the type variable a. The type annotation on the let binding also brings a in scope in M, reflecting the fact that it may occur freely in V_F. We will show an example for this translation shortly.

The translation of type applications $M_F\ A$ similarly has to instantiate all toplevel quantifiers of the type $\forall a.B$ of M_F, in particular those potentially found in B, and immediately re-generalise the quantifiers of B. We may hope that this can be achieved analogously to type abstractions, which would yield $\textbf{let}\ (x : B[A/a]) = M@\ \textbf{in}\ \lceil x \rceil$ as the translation result, where M is the translation of M_F. However, the value restriction prevents this idea from working correctly. We cannot guarantee that M is a value, which is required to ensure that $M@$ is a guarded value and can be generalised.[3] We work around this by first introducing a separate binding for M. The uniqueness of types of the terms produced by our translation ensures that M has principal type $\forall a.B$. The overall translation of the type application is then $\textbf{let}\ x = M\ \textbf{in}\ \$^{B[A/a]}x$. Note that x receives type $\forall a.B$. Then, all quantifiers of the latter type are instantiated by the term x, including instantiating a to A, and we re-generalise potential quantifiers in B at the instantiation operator. Note that the need to let-bind M and not being able to substitute x with $M@$, let alone just M, illustrates that the standard substitution properties of ML with respect to let bindings only hold in FreezeML when additional restrictions are imposed. We will revisit this question in Section 4.1.

The translation of System F let bindings, which we introduced for the purposes of the value restriction, works by simply translating the two subterms, but drops the type annotation.

[3]We can actually characterise the syntactic category of M more accurately. The fact that M has a unique, toplevel quantified type means that by Lemma 3.2 it cannot be a guarded value. However, it may or may not be an ordinary value.

As an example for the overall translation, we consider the System F term $\Lambda c.\mathsf{mkPair}$ $c\ c$. Here, we assume that mkPair is given type $\forall ab.a \to b \to (a \times b)$ in Γ, meaning that it is a function acting like the pair data constructor. As a result, the example term merely restricts mkPair to create pairs of two elements of the *same* type. The translation then proceeds as follows.

$$\begin{array}{c}
c;\Gamma \vdash \mathsf{mkPair} \;:\; \forall ab.a \to b \to (a \times b) \;\leadsto\; \lceil \mathsf{mkPair} \rceil \\ \hline
c;\Gamma \vdash \mathsf{mkPair}\; c \;:\; \forall b.c \to b \to (c \times b) \;\leadsto\; \underbrace{\textbf{let}\; x_2 = \lceil \mathsf{mkPair} \rceil \;\textbf{in}\; \$^{\forall b.c \to b \to (c \times b)}\, x_2}_{=:\, M_2} \\ \hline
c;\Gamma \vdash \mathsf{mkPair}\; c\; c \;:\; c \to c \to (c \times c) \;\leadsto\; \underbrace{\textbf{let}\; x_1 = M_2 \;\textbf{in}\; \$^{c \to c \to (c \times c)}\, x_1}_{=:\, M_1} \\ \hline
\cdot;\Gamma \vdash \Lambda c.\mathsf{mkPair}\; c\; c \;:\; \forall c.c \to c \to (c \times c) \;\leadsto\; \$^{\forall c.c \to c \to (c \times c)}(M_1 @)
\end{array}$$

The example illustrates how the translation mediates between the instantiation behaviour of both languages: To translate the subterm $\mathsf{mkPair}\ c$ with type variable c in scope, we bind x_2 to the translation $\lceil \mathsf{mkPair} \rceil$ of the left-hand side and use it freely in the body of the binding. Of course, this instantiates both quantifiers in x_2's type $\forall ab.a \to b \to (a \times b)$, whereas the term $\mathsf{mkPair}\ c$ is only supposed to instantiate the first one. Thus, a generalisation operator restores the second quantifier b, resulting in the overall translation M_2 of the inner type application with type $\forall b.c \to b \to (c \times b)$.

The term M_2 then appears as a subterm of M_1, the result of translating $\mathsf{mkPair}\ c\ c$. Note that here, the generalisation operator annotated with $c \to c \to (c \times c)$ does not actually perform generalisation, but acts just as a type annotation. The term M_1 is in turn instantiated (which has no effect in our example) and generalised to yield the overall translation of the original example term.

Of course, the example translation also shows that the resulting terms are unnecessarily complex. After expanding all syntactic sugar, the translated example contains six let bindings, although it is equivalent to the following, much simpler FreezeML term.

$$\$^{\forall c.c \to c \to (c \times c)}\, \mathsf{mkPair} \quad \equiv \quad \textbf{let}\; (x : \forall c.c \to c \to (c \times c)) = \mathsf{mkPair} \;\textbf{in}\; \lceil x \rceil$$

We could immediately obtain a translation creating smaller terms by translating n-ary type abstractions and type applications (e.g., define the translation on $M\,\overline{A}$, where M is not a type application itself). We eschew formalising this for the sake of retaining a more compositional translation directly following the source syntax of System F.

The correctness properties of the translation mentioned earlier can be formalised as follows.

Theorem 3.1.
Let $\Delta;\Gamma \vdash^{F} M_{\mathrm{F}} : A \rightsquigarrow M$ for some System F term M_{F} and FreezeML term M. Then the following conditions hold.

1. *We have $\Delta;\Gamma \vdash M : A$ in FreezeML.*

2. *For all Δ' and B we have that $(\Delta,\Delta');\Gamma \vdash M : B$ in FreezeML implies $A = B$.*

Proof. By structural induction on the System F term under consideration, to which exactly one typing/translation rule must have applied.

- Case x, $\lambda x^{A}.M_{\mathrm{F}}$, $M_{\mathrm{F}}\ N_{\mathrm{F}}$: Straightforward. The uniqueness of the type of the translated FreezeML term follows from just being the type of the variable (case x), from the type being a sub-expression of the type of a subterm (case $M_{\mathrm{F}}\ N_{\mathrm{F}}$), possibly in combination with the unique type A provided as an annotation on the function parameter (case $\lambda x^{A}.M_{\mathrm{F}}$).

- Case $\Lambda a.V_{\mathrm{F}}$, We have $\Delta;\Gamma \vdash V_{\mathrm{F}} : \forall a.B \rightsquigarrow \$^{\forall a.B}\,(M@)$ for some B and M. By inversion on the typing rule we further have $(\Delta,a);\Gamma \vdash^{\mathrm{F}} V_{\mathrm{F}} : B \rightsquigarrow M$.

  By Lemma 3.4 we have that M is a FreezeML value, making $M@$ a guarded value. Thus, we may use the rule GenAnn from Figure 3.7 on page 54 to derive $\Delta;\Gamma \vdash \$^{\forall a.B}(M@) : \forall a.B$. Let $\overline{b}$ denote the toplevel quantifiers of B, meaning that we have $B = \forall\overline{b}.H$ for some H. GenAnn then requires us to show $(\Delta,a,\overline{b});\Gamma \vdash M@ : H$. It also requires $\forall a.B$ to be well formed in context Δ, which implicitly follows from $(\Delta,a);\Gamma \vdash M : B$. Note that the uniqueness of the type $\forall a.B$ for the overall translated term follows directly from being the type of the annotation on the generalisation operator.

  Recall that $M@$ is syntactic sugar for **let** $x = M$ **in** x; we show how to derive $(\Delta,a,\overline{b});\Gamma \vdash$ **let** $x = M$ **in** $x : H$. Let $B' = \forall\overline{c}.H'$ be alpha-equivalent to B such that all $\overline{c}$ are fresh and let $\delta = \overline{[b/c]}$. Note that this implies $\delta(H') = H$. Applying the IH to M yields $(\Delta,a);\Gamma \vdash M : \forall\overline{c}.H'$ and by $\forall\overline{c}.H'$ being the unique type of M (modulo alpha-equivalence) also $\mathsf{principal}((\Delta,a),\Gamma,M,\cdot,\forall\overline{c}.H')$.

  We can therefore derive the following, independently from whether M is a guarded value or not.

$$\dfrac{(\Delta,a,\overline{b});\Gamma \vdash M : \forall\overline{c}.H' \qquad \dfrac{\begin{array}{c}(x : \forall\overline{c}.H') \in \Gamma \\ (\Delta,a,\overline{b}) \vdash \delta : \overline{c} \Rightarrow_\star \cdot\end{array}}{\Delta;\Gamma,x : \forall\overline{c}.H' \vdash x : \delta(H')} \\ ((\Delta,a,\overline{b}),\cdot,M,\forall\overline{c}.H') \Updownarrow \forall\overline{c}.H' \\ \cdot = \mathsf{ftv}(\forall\overline{c}.H') - (\Delta,a,\overline{b}) \qquad \mathsf{principal}((\Delta,a,\overline{b}),\Gamma,M,\cdot,\forall\overline{c}.H')}{(\Delta,a,\overline{b});\Gamma \vdash \textbf{let}\ x = M\ \textbf{in}\ x : H}$$

- Case $M_\mathrm{F}\,A$, with $\Delta;\Gamma \vdash^{\mathrm{F}} M_\mathrm{F}\,A : B[A/a] \rightsquigarrow \textbf{let}\ x = M\ \textbf{in}\ \$^{B[A/a]}\,x$ and $\Delta;\Gamma \vdash^{\mathrm{F}} M_\mathrm{F} : \forall a.B \rightsquigarrow M$.

  Similarly to the previous case, we get $\Delta;\Gamma \vdash M : \forall a.B$ and $\mathsf{principal}(\Delta,\Gamma,M,\cdot,\forall a.B)$ by induction and in particular the uniqueness condition. As in the previous case, let $\overline{b}$ again denote the toplevel quantifiers of B, meaning that we have $B = \forall\overline{b}.H$ for some H. Let $B' = \forall\overline{c}.H'$ be alpha-equivalent to B such that all $\overline{c}$ are fresh. However, this time we define δ such that $\delta(a) = A$ and $\delta(c_i) = b_i$, yielding $\forall\overline{b}.\delta(H') = B[A/a]$.

  Altogether, this now allows us to derive $\Delta;\Gamma \vdash \textbf{let}\ x = M\ \textbf{in}\ \$^{B[A/a]}\,x : B[A/a]$ as follows. Note that we again use GenAnn from Figure 3.7. Regarding the uniqueness of types, we again rely on the fact that the annotated generalisation operator ensures this for the overall term.

$$\dfrac{\Delta;\Gamma \vdash M : \forall a\overline{c}.H' \qquad \dfrac{\dfrac{(\Delta,\overline{b}) \vdash \delta : (a,\overline{c}) \Rightarrow_\star \cdot}{(\Delta,\overline{b});(\Gamma,x : \forall a\overline{c}.H') \vdash x : \delta(H')}}{\Delta;(\Gamma,x : \forall a\overline{c}.H') \vdash \$^{B[A/a]}\,x : B[A/a]} \\ (\Delta,\cdot,M,\forall a\overline{c}.H') \Updownarrow \forall a\overline{c}.H' \\ \cdot = \mathsf{ftv}(\forall a\overline{c}.H') - \Delta \qquad \mathsf{principal}(\Delta,\Gamma,M,\cdot,\forall a\overline{c}.H')}{\Delta;\Gamma \vdash \textbf{let}\ x = M\ \textbf{in}\ \$^{B[A/a]}\,x : B[A/a]}$$

- Case $\textbf{let}\ x^A = M\ \textbf{in}\ N$: Straightforward application of the IH to the subterms, again relying on the fact that the unique type of M becomes the type of x in the un-annotated FreezeML let binding.

□

3.3.2 Translating FreezeML to System F

We now consider the reverse direction, translating FreezeML to System F. Recall that the dynamic semantics of FreezeML terms is then defined by this translation. This translation also relies on type information, meaning that we again extend the typing judgements of the source language, this time FreezeML, to perform the translation. The rules of this extended typing relation are shown in Figure 3.9.

$$\boxed{\Delta;\Gamma \vdash M : A \rightsquigarrow M_{\mathrm{F}}}$$

$$\textsc{VarFrozen}\quad \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash \lceil x \rceil : A \rightsquigarrow x} \qquad \textsc{VarPlain}\quad \frac{x : \forall \overline{a}.H \in \Gamma \qquad \Delta \vdash \delta : \overline{a} \Rightarrow_{\star} \cdot}{\Delta;\Gamma \vdash x : \delta(H) \rightsquigarrow x\,\overline{\delta(a)}}$$

$$\textsc{App}\quad \frac{\Delta;\Gamma \vdash M : A \to B \rightsquigarrow M_{\mathrm{F}} \qquad \Delta;\Gamma \vdash N : A \rightsquigarrow N_{\mathrm{F}}}{\Delta;\Gamma \vdash M\,N : B \rightsquigarrow M_{\mathrm{F}}\,N_{\mathrm{F}}}$$

$$\textsc{LamPlain}\quad \frac{\Delta;(\Gamma, x : S) \vdash M : B \rightsquigarrow M_{\mathrm{F}}}{\Delta;\Gamma \vdash \lambda x.M : S \to B \rightsquigarrow \lambda x^{S}.M_{\mathrm{F}}} \qquad \textsc{LamAnn}\quad \frac{\Delta;(\Gamma, x : A) \vdash M : B \rightsquigarrow M_{\mathrm{F}}}{\Delta;\Gamma \vdash \lambda(x : A).M : A \to B \rightsquigarrow \lambda x^{A}.M_{\mathrm{F}}}$$

$$\textsc{LetPlain}\quad \frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\overline{a});\Gamma \vdash M : A' \rightsquigarrow M_{\mathrm{F}} \qquad \mathsf{principal}(\Delta,\Gamma,M,\overline{a},A') \\ (\Delta,\overline{a},M,A') \Updownarrow (A,\Delta',\delta) \qquad \Delta;(\Gamma, x : A) \vdash N : B \rightsquigarrow N_{\mathrm{F}}\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B \rightsquigarrow \mathbf{let}\ x^{A} = \Lambda\Delta'.\delta(M_{\mathrm{F}})\ \mathbf{in}\ N_{\mathrm{F}}}$$

$$\textsc{LetAnn}\quad \frac{(\overline{a},A') = \mathsf{split}(A,M) \qquad (\Delta,\overline{a});\Gamma \vdash M : A' \rightsquigarrow M_{\mathrm{F}} \qquad \Delta;(\Gamma, x : A) \vdash N : B \rightsquigarrow N_{\mathrm{F}}}{\Delta;\Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B \rightsquigarrow \mathbf{let}\ x^{A} = \Lambda\overline{a}.M_{\mathrm{F}}\ \mathbf{in}\ N_{\mathrm{F}}}$$

Figure 3.9: Translation from FreezeML to System F

As expected, frozen variables are translated to ordinary System F variables. A plain variable x is translated to $x\ A_1 \dots A_n$, where the types A_i are obtained from instantiating all toplevel quantifiers of x's type according to the instantiation δ used in the typing derivation. As in the translation from System F to FreezeML, term abstractions and applications are translated homomorphically. Here, un-annotated function parameters are annotated during the translation.

To translate plain let bindings, we introduce an extended version of the $\Updownarrow$ relation used in LETPLAIN, now yielding judgements $(\Delta,\overline{a},M,A') \Updownarrow (A,\Delta',\delta)$. The relation was originally defined in Figure 3.6 on page 47. The left-hand side of the relation symbol (i.e., the "inputs" of the relation, as it acts like a function) remain unchanged, but we add a sequence of type variables Δ' and a substitution to the right-hand side (i.e., the "outputs").

$\boxed{(\Delta,\overline{a},M,A') \Updownarrow (A,\Delta',\delta)}$

$$\frac{M \in \mathsf{GVal}}{(\Delta,\overline{a},M,A') \Updownarrow (\forall\overline{a}.A',\overline{a},\emptyset)} \qquad \frac{\Delta \vdash \delta : \overline{a} \Rightarrow_{\bullet} \cdot \qquad M \notin \mathsf{GVal}}{(\Delta,\overline{a},M,A') \Updownarrow (\delta(A'),\cdot,\delta)}$$

As in the original version, A arises from A' either by quantifying over the variables $\overline{a}$ (if $M \in \mathsf{GVal}$), or otherwise by applying a monomorphic substitution δ with $\Delta \vdash \delta : \overline{a} \Rightarrow_{\bullet} \cdot$ to A'. In the extended version, the returned sequence Δ' corresponds to the variables that were newly quantified by $\Updownarrow$, meaning that Δ' is equal to $\overline{a}$ if M is generalisable and empty otherwise. For the instantiation δ the rule then ensures that $\Delta \vdash (\overline{a} - \Delta') \Rightarrow_{\bullet} \cdot$ holds. In particular, δ is the empty substitution if M is non-generalisable. Otherwise, we return the same instantiation used to create A from A'.

In the rule LETPLAIN in Figure 3.9 we then define the body of the translated System F let binding to abstract over the type variables in Δ' as returned by $\Updownarrow$, meaning that we universally quantify exactly those variables that the original let binding generalises.

The premise $(\Delta,\overline{a});\Gamma \vdash M : A'$ of LETPLAIN remains unchanged as compared to the original typing rule, but now produces a System F term M_{F}. We then have that $(\Delta,\overline{a});\Gamma \vdash M_{\mathrm{F}} : A'$ holds in System F. We rely on the standard substitution property for System F (see Lemma 2.7 on page 20) to observe that we then have $\Delta;\Gamma \vdash \delta(M_{\mathrm{F}}) : A$, for the term M_{F}, which is used under the type abstraction in the generated let body. Here, δ is the instantiation returned by $\Updownarrow$, meaning that it only has an effect if M is not guarded.

Recall that we discussed in Section 3.2.1.2 that the premise $(\Delta,\overline{a});\Gamma \vdash M : A'$ of LETPLAIN is redundant, as it is already implied by $\mathsf{principal}(\Delta,\Gamma,M,\overline{a},A')$, and only included for illustrative purposes. An alternative definition of LETPLAIN in Figure 3.9 could therefore use the premise $\Delta;\Gamma \vdash M : \delta(A') \rightsquigarrow M'_{\mathrm{F}}$ and use this term M'_{F} in place of $\delta(M_{\mathrm{F}})$ in the translation result. Note that $\Delta;\Gamma \vdash M : \delta(A')$ is also implied by the principality premise. While this alternative definition of LETPLAIN shows that we can eschew applying a substitution to terms as part of the translation, it strays further from the original typing rules.

As an example, consider the following two translations, where we again assume $\mathsf{mkPair} : \forall ab.a \to b \to (a \times b)$.

$$\begin{aligned}
\textbf{let} \, f = \ \mathsf{mkPair} \ \textbf{in} \ f\,3\,4 : (\mathsf{Int} \times \mathsf{Int}) \ &\rightsquigarrow \ \textbf{let} \ f^{\forall cd.c \to d \to (c \times d)} = \Lambda c\,d.\mathsf{mkPair}\ c\ d \\
& \ \textbf{in} \ f \ \mathsf{Int}\ \mathsf{Int}\ 3\ 4
\end{aligned}$$

$$\begin{aligned}
\textbf{let} \, f = \lceil \mathsf{mkPair} \rceil \ \textbf{in} \ f\,3\,4 : (\mathsf{Int} \times \mathsf{Int}) \ &\rightsquigarrow \ \textbf{let} \ f^{\forall ab.a \to b \to (a \times b)} = \mathsf{mkPair} \\
& \ \textbf{in} \ f \ \mathsf{Int}\ \mathsf{Int}\ 3\ 4
\end{aligned}$$

The translation of annotated let bindings in LETANN can directly use the type variables returned by the helper function split (defined in Figure 3.6 on page 47) in the original rule to determine those variables that are generalised at the let binding. These type variables are consequently abstracted over in the result term.

As a final example, we consider the FreezeML term

$$\$^{\forall c.c \to c \to (c \times c)}\ \mathsf{mkPair} \ \equiv \ \textbf{let}\ (x : \forall c.c \to c \to (c \times c)) = \mathsf{mkPair}\ \textbf{in}\ \lceil x \rceil$$

again, which was the manual translation of the System F term $\Lambda c.\mathsf{mkPair}\ c\ c$ into FreezeML discussed in Section 3.3.1. Using the rules in Figure 3.9, this term is translated to the System F term $\textbf{let}\ x^{\forall c.c \to c \to (c \times c)} = \Lambda c.\mathsf{mkPair}\ c\ c\ \textbf{in}\ x$. It corresponds exactly to the original source term, modulo an inconsequential let binding at the top-level.

The correctness of the translation depends on a property analogous to Lemma 3.4 on page 58, showing that non-expansiveness is preserved by the reverse translation as well.

Lemma 3.5.
Let V be a FreezeML value. If $\Delta;\Gamma \vdash V : A \rightsquigarrow M_{\mathrm{F}}$ for some System F term M_{F}, then M_{F} is a System F value.

Proof. By structural induction on V. □

This property is required in order to show that if the n-ary type abstractions generated by the rules LETPLAIN and LETANN in Figure 3.9 are non-vacuous (i.e., when the list of quantifiers is non-empty), then the terms under Λ are System F values.

The overall correctness property of the translation is then the following.

Theorem 3.2.
If $\Delta;\Gamma \vdash M : A \rightsquigarrow M_{\mathrm{F}}$ *then* $\Delta;\Gamma \vdash^{F} M_{\mathrm{F}} : A$

Proof. By structural induction on the FreezeML term under consideration.

- Case x, $\lceil x \rceil$, $M\,N$, $\lambda x.M$, $\lambda(x:A).M$: Straightforward.
- Case $\mathbf{let}\ x = M\ \mathbf{in}\ N$: LetPlain must have applied and we have $\Delta;\Gamma \vdash \mathbf{let}\,x = M\ \mathbf{in}\ N : B \rightsquigarrow \mathbf{let}\ x^{A} = \Lambda\Delta'.\delta(M_{\mathrm{F}})\ \mathbf{in}\ M_{\mathrm{F}}$ for some $A, B, \delta, \Delta', \overline{a}, M_{\mathrm{F}}$, and N_{F}, further subject to the following conditions.

  $$(\Delta,\overline{a});\Gamma \vdash M : A' \rightsquigarrow M_{\mathrm{F}} \qquad \overline{a} = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\overline{a},M,A') \Updownarrow (A,\Delta',\delta)$$
  $$\Delta;(\Gamma, x:A) \vdash N : B \rightsquigarrow N_{\mathrm{F}}$$

  The principality premise of LetPlain involving M and A' is not used.

  To derive $\Delta;\Gamma \vdash^{F} x^{A} = \Lambda\Delta'.\delta(M_{\mathrm{F}})\ \mathbf{in}\ N_{\mathrm{F}} : B$, we must show that $\Delta;\Gamma \vdash^{F} \Lambda\Delta'.\delta(M_{\mathrm{F}}) : A$ and $\Delta;(\Gamma, x:A) \vdash^{F} N_{\mathrm{F}} : B$ holds. The latter follows directly by induction. Note that this also implies the well-formedness of A under Δ.

  It remains to show $\Delta;\Gamma \vdash^{F} \Lambda\Delta'.\delta(M_{\mathrm{F}}) : A$, for which we distinguish two subcases.

  1. If $M \in \mathsf{GVal}$, the definition of (the extended version of) $\Updownarrow$ imposes $A = \forall\overline{a}.A'$ and $\Delta' = \overline{a}$ and that δ is empty. This makes $\delta(M_{\mathrm{F}})$ equivalent to M_{F}. By Lemma 3.5 we have that the latter is a System F value, making $\Lambda\Delta'.M_{\mathrm{F}}$ well formed. Applying the IH to $(\Delta,\overline{a});\Gamma \vdash M : A' \rightsquigarrow M_{\mathrm{F}}$ gives us $(\Delta,\overline{a});\Gamma \vdash^{F} M_{\mathrm{F}} : A'$, and repeated application of the System F typing rule for type abstraction (if $\overline{a} = \Delta'$ is non-empty) turns this into the desired premise $\Delta;\Gamma \vdash^{F} \Lambda\Delta'.M_{\mathrm{F}} : \forall\overline{a}.A'$.
  2. If $M \notin \mathsf{GVal}$, then $\Updownarrow$ imposes $\Delta' = \cdot$, $A = \delta(A')$ and $\Delta \vdash \delta : \overline{a} \Rightarrow_{\bullet} \cdot$. This makes the type abstraction vacuous (i.e., $\Lambda\Delta'.\delta(M_{\mathrm{F}})$ is just $\delta(M_{\mathrm{F}})$). We obtain $(\Delta,\overline{a});\Gamma \vdash^{F} M_{\mathrm{F}} : A'$ by induction as in the previous case. Substituting with δ gives us $\Delta;\delta(\Gamma) \vdash^{F} \delta(M_{\mathrm{F}}) : \delta(A')$ (see Lemma 2.7 on page 20). An implicit precondition of $\Delta;(\Gamma, x:A) \vdash N : B$ is $\Delta \vdash \Gamma, x:A$, which implies that δ has no effect on Γ. In total this yields the desired premise $\Delta;\Gamma \vdash^{F} \Lambda\Delta'.\delta(M_{\mathrm{F}}) : A$.

- Case $\mathbf{let}\ (x:A) = M\ \mathbf{in}\ N$, being translated to $\mathbf{let}\ x^{A} = \Lambda\overline{a}.M_{\mathrm{F}}\ \mathbf{in}\ N_{\mathrm{F}}$: The rule LetAnn imposes that if $M \in \mathsf{GVal}$ then $\overline{a}$ are the toplevel quantifiers of A, and A'

denotes the guarded remaining type. Lemma 3.5 again shows that the translation of M is a value, making the surrounding type abstraction well formed. We obtain the necessary premises about the types of the translation of M and N by induction (and repeated type abstraction) as in the un-annotated case.

The case if M is not a guarded value is even simpler because then $\overline{a}$ is empty and A' and A are identical.

$\square$

3.4 Well-definedness of typing relation

Replacing the premise $\mathsf{principal}(\Delta,\Gamma,M,\Delta',A')$ of the LetPlain typing rule in Figure 3.5 on page 45 with its definition yields the following.

LetPlain-Expanded

$$\frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\overline{a});\Gamma \vdash M : A' \qquad (\Delta,\overline{a},M,A') \Updownarrow A \qquad \Delta;(\Gamma,x:A) \vdash N : B \\ \text{(for all } \Delta'',A'' \text{ : if } (\Delta,\Delta'');\Gamma \vdash M : A'' \\ \qquad\qquad \text{then there exists } \delta \text{ such that} \\ \qquad\qquad\qquad \Delta \vdash \delta : \overline{a} \Rightarrow_\star \Delta'' \text{ and } \delta(A') = A'')\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

This reveals that the LetPlain typing rule uses typing judgements on the left-hand side of an implication, and thus in a negative position. Ruling out negative occurrences of the relation under consideration in derivation rules is a simple syntactic criterion for establishing that the rules yield a well-defined relation.

We now show despite violating this condition, our typing rules nevertheless induce a well-defined relation. Here we utilise the fact that all FreezeML typing rules only ever use typing judgements that involve proper subterms in their premises, in particular in the principality condition on let bindings. To this end, we show that there exists a well-defined function $\mathcal{J}$ from FreezeML terms to triples (Δ,Γ,A). We then show that if we were to *define* the typing relation such that $\Delta;\Gamma \vdash M : A$ holds if and only if $(\Delta,\Gamma,A) \in \mathcal{J}(M)$, then we have $\Delta;\Gamma \vdash M : A$ exactly if a FreezeML typing rule from Figure 3.5 is applicable.

More concretely, we show that each time a typing rule applies to some term M, the corresponding triple is contained in $\mathcal{J}(M)$ and we show that the rules are invertible, meaning that for each $\mathcal{J}(M)$, a single typing rule corresponding to the shape of M is applicable. We define the function $\mathcal{J}$ as follows. It is defined mutually recursively with another function $\mathcal{P}$, the counterpart to the relation $\mathsf{principal}$. Thus $\mathcal{P}$ maps terms to quadruples $(\Delta,\Gamma,\Delta',A)$, expressing that A is the principal type of M in contexts Δ and Γ using fresh variables from Δ'. Recall that FreezeML typing judgements $\Delta;\Gamma \vdash M : A$ implicitly require that Γ and M are well formed. Further, recall that *all* of our judgements implicitly require that contexts contain no duplicates, which applies to Δ and Γ here. These implicit preconditions are reflected in $\mathcal{J}$.

$$\begin{aligned}
\mathcal{J}(\lceil x \rceil) &= \{(\Delta,\Gamma,A) \mid \Delta \vdash \Gamma\ \mathbf{ok}, (x:A) \in \Gamma\} \\
\mathcal{J}(x) &= \{(\Delta,\Gamma,\delta(H)) \mid \Delta \vdash \Gamma\ \mathbf{ok} \text{ and } (x:\forall\Delta'.H) \in \Gamma \\
&\qquad\qquad \text{and } \Delta \vdash \Delta' \Rightarrow_\star \cdot\} \\
\mathcal{J}(\lambda x.M) &= \{(\Delta,\Gamma,S \to B) \mid (\Delta,(\Gamma,x:S),B) \in \mathcal{J}(M)\} \\
\mathcal{J}(\lambda(x:A).M) &= \{(\Delta,\Gamma,A \to B) \mid (\Delta,(\Gamma,x:A),B) \in \mathcal{J}(M)\} \\
\mathcal{J}(M\,N) &= \{(\Delta,\Gamma,B) \mid (\Delta,\Gamma,A \to B) \in \mathcal{J}(M) \text{ and } (\Delta,\Gamma,A) \in \mathcal{J}(N) \\
&\qquad\qquad \text{for some } A\} \\
\mathcal{J}(\mathbf{let}\ x = M\ \mathbf{in}\ N) &= \{(\Delta,\Gamma,B) \mid \Delta;\Gamma \vdash M\ \mathbf{ok} \text{ and } (\Delta,\Gamma,\Delta',A') \in \mathcal{P}(M) \\
&\qquad\qquad \text{and } \Delta' = \mathsf{ftv}(A') - \Delta \text{ and } (\Delta,\Delta',M,A') \Updownarrow A \\
&\qquad\qquad \text{and } (\Delta,(\Gamma,x:A),B) \in \mathcal{J}(N)\} \\
\mathcal{J}(\mathbf{let}\ (x:A) = M\ \mathbf{in}\ N) &= \{(\Delta,\Gamma,B) \mid (\overline{a},A') = \mathsf{split}(A,M) \text{ and } ((\Delta,\overline{a}),\Gamma,B) \in \mathcal{J}(M) \\
&\qquad\qquad \text{and } (\Delta,(\Gamma,x:A),B) \in \mathcal{J}(N)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}(M) = \{(\Delta,\Gamma,\Delta',A) \mid\ & ((\Delta,\Delta'),\Gamma,A) \in \mathcal{J}(M) \text{ and for all } \Delta'',A'' \\
& ((\Delta,\Delta''),\Gamma,A'') \in \mathcal{J}(M) \\
& \text{implies that there exists } \delta \text{ such that} \\
& \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta(A) = A''\}
\end{aligned}$$

Note that we do not claim that the functions $\mathcal{J}$ and $\mathcal{P}$ are computable. In fact, while ML enjoys a principal types property, where the context Δ,Γ is fixed (as discussed in Section 2.2.1.3), it lacks *principal typings*, where all possible types under all possible contexts of a term are taken into account [124]. This fact carries over to FreezeML, meaning that as a result, there is no guarantee that the sets $\mathcal{J}(M)$ and $\mathcal{P}(M)$ are finitely representable for all M. However, our only concern is that the functions $\mathcal{J}$ and $\mathcal{P}$ are well defined.

Lemma 3.6.
For all M, the sets $\mathcal{J}(M)$ and $\mathcal{P}(M)$ are well defined.

Proof. Both properties can easily be shown simultaneously by structural induction on M. While $\mathcal{P}$ uses $\mathcal{J}$ on the same term, $\mathcal{J}$ uses $\mathcal{P}$ only on subterms. $\square$

As a more technical property, we now show that the definition of $\mathcal{J}$ ensures the implicit well-formedness conditions imposed on typing judgements.

Lemma 3.7.
If $(\Delta,\Gamma,A) \in \mathcal{J}(M)$*, then* $\Delta \vdash \Gamma$ **ok** *and* $\Delta;\Gamma \vdash M$ **ok**.

Proof. By structural induction on M. $\square$

Note that this property also implies that Δ and Γ contain no duplicates, by virtue of appearing in a judgement.

We now introduce alternative versions of the FreezeML typing relation $\vdash$ as well as $\mathsf{principal}$, named $\vdash^{\mathcal{J}}$ and $\mathsf{principal}^{\mathcal{P}}$, respectively. They are defined as follows.

$$\begin{array}{rcl} \Delta;\Gamma \vdash^{\mathcal{J}} M : A & \text{iff} & (\Delta,\Gamma,A) \in \mathcal{J}(M) \\ \mathsf{principal}^{\mathcal{P}}(\Delta,\Gamma,M,\Delta',A) & \text{iff} & (\Delta,\Gamma,\Delta',A) \in \mathcal{P}(M) \end{array}$$

First, we make the unsurprising observation that $\mathsf{principal}^{\mathcal{P}}$ adheres to the definition of $\mathsf{principal}$ in Figure 3.6.

Lemma 3.8.
$\mathsf{principal}^{\mathcal{P}}(\Delta,\Gamma,M,\Delta',A')$ *holds iff*

$$\begin{array}{l} (\Delta,\Delta');\Gamma \vdash^{\mathcal{J}} M : A' \ \textit{and} \\ \quad (\textit{for all } \Delta'',A'' \mid \textit{if } (\Delta,\Delta'');\Gamma \vdash^{\mathcal{J}} M : A'' \\ \qquad\qquad \textit{then there exists } \delta \textit{ such that} \\ \qquad\qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \textit{ and } \delta(A') = A'') \end{array}$$

Proof. Follows immediately from the involved definitions. $\square$

We now show that the relation $\vdash^{\mathcal{J}}$ follows the rules in Figure 3.5, meaning that if a rule with its premises expressed in terms of $\vdash^{\mathcal{J}}$ and $\mathsf{principal}^{\mathcal{P}}$ applies (and the implicit well-formedness conditions are also satisfied), then the conclusion of the rule also holds in terms of $\vdash^{\mathcal{J}}$. In this sense, this shows that $\vdash^{\mathcal{J}}$ is complete with respect to the FreezeML typing rules.

Lemma 3.9.
All of the following implications hold.

1. *If* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash \lceil x \rceil$ **ok** *and* $(x : A) \in \Gamma$*, then* $\Delta;\Gamma \vdash^{\mathcal{J}} \lceil x \rceil : A$.

2. *If* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash x$ **ok** *and* $(x : \Delta'.H) \in \Gamma$ *and* $\Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot$, *then* $\Delta;\Gamma \vdash^{\mathcal{J}} x : \delta(H)$.

3. *If* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash M\,N$ **ok** *and* $\Delta;\Gamma \vdash^{\mathcal{J}} M : A \to B$ *and* $\Delta;\Gamma \vdash^{\mathcal{J}} N : A$, *then* $\Delta;\Gamma \vdash^{\mathcal{J}} MN : B$.

4. *If* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash \lambda x.M$ **ok** *and* $\Delta;(\Gamma, x : S) \vdash^{\mathcal{J}} M : B$, *then* $\Delta;\Gamma \vdash^{\mathcal{J}} \lambda x.M : S \to B$.

5. *If* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash \lambda(x : A).M$ **ok** *and* $\Delta;(\Gamma, x : A) \vdash^{\mathcal{J}} M : B$, *then* $\Delta;\Gamma \vdash^{\mathcal{J}} \lambda(x : A).M : S \to B$.

6. *If* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash$ **let** $x = M$ **in** N **ok** *and*

$$\begin{aligned}
&\Delta' = \mathsf{ftv}(A') - \Delta \textit{ and}\\
&(\Delta,\Delta');\Gamma \vdash^{\mathcal{J}} M : A' \textit{ and}\\
&\mathsf{principal}^{\mathcal{P}}(\Delta,\Gamma,M,\Delta',A') \textit{ and}\\
&(\Delta,\Delta',M,A') \Updownarrow A \textit{ and}\\
&\Delta;(\Gamma,x : A) \vdash^{\mathcal{J}} N : B,
\end{aligned}$$

then $\Delta;\Gamma \vdash^{\mathcal{J}}$ **let** $x = M$ **in** $N : B$.

7. *If* $\Delta \vdash \Gamma$ *and* $\Delta \vdash$ **let** $(x : A) = M$ **in** N **ok** *and* $(\overline{a}, A') = \mathsf{split}(A, M)$ *and* $(\Delta, \overline{a});\Gamma \vdash^{\mathcal{J}} M : A'$ *and* $\Delta;(\Gamma, x : A) \vdash^{\mathcal{J}} N : B$, *then* $\Delta;\Gamma \vdash^{\mathcal{J}}$ **let** $(x : A) = M$ **in** $N : B$.

Proof. For each term M, the corresponding property follows directly from the definition of $\mathcal{J}(M)$. □

Finally, we show that if $\Delta;\Gamma \vdash^{\mathcal{J}} M : A$ holds, then exactly one of the rules from Figure 3.5 is applicable (i.e., its premises and implicit well-formedness preconditions are satisfied), thus showing that $\vdash^{\mathcal{J}}$ is sound with respect to the typing rules in Figure 3.5 and the rules are invertible.

Lemma 3.10.
For all Δ *and* Γ *the following conditions hold.*

1. *For all* x *and* A *such that* $\Delta;\Gamma \vdash^{\mathcal{J}} \lceil x \rceil : A$, *we have* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash \lceil x \rceil$ **ok** *and* $(x : A) \in \Gamma$.

2. *For all* x *and* A *such that* $\Delta;\Gamma \vdash^{\mathcal{J}} x : A$, *there exist* Δ', H, δ *such that* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash x$ **ok** *and* $(x : \Delta'.H) \in \Gamma$ *and* $\Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot$ *and* $A = \delta(H)$.

3. *For all* M, N, B *such that* $\Delta;\Gamma \vdash^{\mathcal{J}} MN : B$, *there exists* A *such that* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash M\,N$ **ok** *and* $\Delta;\Gamma \vdash^{\mathcal{J}} M : A \to B$ *and* $\Delta;\Gamma \vdash^{\mathcal{J}} N : A$.

4. *For all* x, M, A' *such that* $\Delta;\Gamma \vdash^{\mathcal{J}} \lambda x.M : A'$*, there exist* S, B *such that* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash \lambda x.M$ **ok** *and* $\Delta;(\Gamma, x : S) \vdash^{\mathcal{J}} M : B$ *and* $A' = S \to B$*.*

5. *For all* x, M, A, A' *such that* $\Delta;\Gamma \vdash^{\mathcal{J}} \lambda(x : A).M : A'$ *there exists* B *such that* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash \lambda(x : A).M$ **ok** *and* $\Delta;(\Gamma, x : A) \vdash^{\mathcal{J}} M : B$ *and* $A' = A \to B$*.*

6. *For all* x, M, N, B *such that* $\Delta;\Gamma \vdash^{\mathcal{J}}$ **let** $x = M$ **in** $N : B$*, there exist* Δ', A, A' *such that* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash$ **let** $x = M$ **in** N **ok** *as well as the following conditions hold.*

$$\Delta' = \mathsf{ftv}(A') - \Delta \qquad (\Delta, \Delta');\Gamma \vdash^{\mathcal{J}} M : A' \qquad \mathsf{principal}^{\mathcal{P}}(\Delta, \Gamma, M, \Delta', A')$$

$$(\Delta, \Delta', M, A') \Updownarrow A \qquad \Delta;(\Gamma, x : A) \vdash^{\mathcal{J}} N : B$$

7. *For all* x, M, N, B, A *such that* $\Delta;\Gamma \vdash^{\mathcal{J}}$ **let** $(x : A) = M$ **in** $N : B$*, there exist* $\overline{a}, A'$ *such that* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash$ **let** $(x : A) = M$ **in** N **ok** *and* $(\overline{a}, A') = \mathsf{split}(A, M)$ *and* $(\Delta, \overline{a});\Gamma \vdash^{\mathcal{J}} M : A'$ *and* $\Delta;(\Gamma, x : A) \vdash^{\mathcal{J}} N : B$*.*

Proof. In each of the seven properties, we are assuming $\Delta;\Gamma \vdash^{\mathcal{J}} M : A$ for some $\Delta;\Gamma \vdash^{\mathcal{J}} M : A$, and therefore $(\Delta, \Gamma, A) \in \mathcal{J}(M)$. The satisfaction of the two well-formedness properties $\Delta \vdash \Gamma$ **ok** and $\Delta;\Gamma \vdash M$ **ok** that need to be shown in each of the seven cases follow from Lemma 3.7. The remaining conditions in each case follow from the definition of $\mathcal{J}(M)$ for the particular shape of M. □

We have thus shown that we may define the typing relation as a well-founded recursive function, using the same rules as the ones shown in Figure 3.5. We have also shown that the rules are invertible, in the sense that given a typing judgement, we can conclude that the premises of exactly one of the rules in Figure 3.5 must hold. Therefore, we can circumvent any issues that would arise from performing induction over the potentially infinite derivation trees in our system, instead performing induction on the structure of terms and inverting typing judgements. This work seems to be the first among the systems that employ some kind of principality condition in its typing rules (either in a core system [89, 58] or a variation of a proposed system [26, 117, 59]), that discusses this aspect in detail.

Note that alternative approaches exist to show that the typing relation is well defined. For example, assuming that Ψ denotes the set of all quadruples (Δ, Γ, M, A), we may interpret the rules in Figure 3.5 as a function ϕ from 2^{Ψ} to 2^{Ψ}. Proving that ϕ is monotone with respect to the complete lattice $(2^{\Psi}, \subseteq)$ would then be sufficient to show the existence of a (least) fixpoint of ϕ, which we then take as the typing relation $\vdash \subset 2^{\Psi}$. However, we have not pursued this alternative approach so far.

Chapter 4

Type inference for FreezeML based on Algorithm W

This chapter shows how Algorithm W, the type inference algorithm for ML developed by Milner and Damas, can be adapted to work for FreezeML. To this end, we first discuss type substitution in FreezeML. In particular, we show how to augment the type system to explicitly represent which type variables can and cannot be substituted with polymorphic types while preserving typeability.

4.1 Type substitution in FreezeML

In Section 2.2.1.3, we re-iterated that ML typing is stable under substitution of type variables, as formalised in Lemma 2.3 on page 18. The same property holds for FreezeML, simply due to the fact that in the context of ML, we defined instantiations to carry monomorphic types in their codomain (see Figure 2.5). We can therefore state the same property for FreezeML, but need to use an instantiation δ that is explicitly restricted to monomorphic types. Finally, we observe that the presence of free type variables in type annotations may make them subject to substitution, too.

We could state substitution properties for FreezeML that take type annotations into account, and therefore apply δ to the term under consideration, similar to the substitution property for System F (see Lemma 2.7 on page 20). However, we have no need for such a property in the remainder of this work and eschew considering type substitutions that apply to FreezeML *terms* for the sake of simplicity. Instead, we consider the free type variables in annotations to be *rigid* and not subject to substitution. This leads to the following property.

Lemma 4.1 (Stability of typing under monomorphic substitution)**.**
If $\Delta \vdash \delta : \Delta' \Rightarrow_\bullet \Delta''$ *and* $\Delta;\Gamma \vdash M$ **ok***, then* $(\Delta,\Delta'),\Gamma \vdash M : A$ *implies* $(\Delta,\Delta''),\delta(\Gamma) \vdash M : \delta(A)$.

Proof. Follows from the more general Lemma 4.4 together with Lemma 4.2, both of which are introduced later in this section. □

Observe that a typing judgement $(\Delta,\Delta'),\Gamma \vdash M : A$ allows Γ and A to contain free type variables from Δ as well as Δ', but the well-formedness premise $\Delta;\Gamma \vdash M$ **ok** of the lemma ensures that annotations in M may only refer to type variables from Δ.

4.1.1 Issues with polymorphic substitution

So far, for the sake of exposition, we have only considered monomorphic substitution. Going beyond monomorphic substitution reveals the following issue. We have both $a;\Gamma \vdash \mathsf{id} : a \to a$ and $b;\Gamma \vdash \lambda x.x : b \to b$ in FreezeML, but the two terms do not behave the same in terms of type substitution. While we have $a;\Gamma \vdash \mathsf{id} : A \to A$ for any well formed, possibly polymorphic type A, we only have $b;\Gamma \vdash \lambda x.x : S \to S$ for monomorphic S. In other words, a may be substituted with polytypes, while b may only be substituted with monotypes. To capture exactly all the possible types of a given term, we therefore need to formalise this difference.

To this end, we extend type contexts Δ to *restriction contexts* Θ. These carry a restriction R for every type variable in scope which indicates whether the type variable may be substituted with monomorphic types or polymorphic types. Further, well-formedness judgements of types are extended to $\Theta \vdash_R A$ **ok** as shown in Figure 4.1.

$$\boxed{\Theta \vdash_R A\ \mathbf{ok}}$$

$$\frac{(a : R) \in \Theta}{\Theta \vdash_R a\ \mathbf{ok}} \qquad \frac{\begin{array}{c}\mathsf{arity}(D) = n \\ \Theta \vdash_R A_1\ \mathbf{ok} \\ \cdots \\ \Theta \vdash_R A_n\ \mathbf{ok}\end{array}}{\Theta \vdash_R D\overline{A}\ \mathbf{ok}} \qquad \frac{\Theta, a : \star \vdash_R A\ \mathbf{ok}}{\Theta \vdash_\star \forall a.A\ \mathbf{ok}} \qquad \frac{\Theta \vdash_\bullet A\ \mathbf{ok}}{\Theta \vdash_\star A\ \mathbf{ok}}$$

Figure 4.1: Well-formedness of types under restriction contexts

The only difference between $\Delta \vdash_R A$ **ok** (defined in Figure 3.3 on page 42) and $\Theta \vdash_R A$ **ok** is that we only have $\Theta \vdash_\bullet a$ **ok** if $(a : \bullet) \in \Theta$, but not if $(a : \star) \in \Theta$. In other words, while $\Delta \vdash_R A$ **ok** implicitly treated all type variables as monomorphic, the new judgement uses the restrictions from Θ. Recall that we implicitly required every type context Δ appearing in a judgement to contain pairwise different variables. The same condition carries over to judgements using restriction contexts Θ, such as $\Theta \vdash_R A$ **ok** and subsequently introduced ones.

We then define substitutions θ similarly to instantiations δ, mapping type variables to types.

$$\begin{array}{lrcl} \text{Restriction contexts} & \Theta & ::= & \cdot \mid \Theta, a : R \\ \text{Type substitutions} & \theta & ::= & \emptyset \mid \theta[a \mapsto A] \end{array}$$

We will occasionally convert between type contexts Δ and restriction contexts Θ. We use the free type variable operator ftv to obtain a type context from a restriction context, by simply dropping the restrictions associated with each variable. We also define an operator for the opposite direction: Given a type context Δ we define Δ^R to be the restriction context where all variables are given restriction R. Concretely, if Δ is $\overline{a}$, then $\Delta^\bullet$ is $\overline{a : \bullet}$.

While a well-formededness judgement $\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta''$ for instantiations (defined in Figure 3.4) imposes a single restriction R on all $\delta(a)$, substitutions θ obey restrictions on a per-variable basis. If θ is a well formed substitution, denoted $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, then for all $(a : R) \in \Theta$ we have that θ maps a to a type A that obeys restriction R in restriction context $(\Delta^\bullet, \Theta')$. The well-formedness judgement for substitutions is formalised in Figure 4.2. We write ι_Θ for the substitution with domain Θ that maps all variables therein to themselves. Thus, we have $\Delta \vdash \iota_\Theta : \Theta \Rightarrow \Theta$ for all Δ disjoint from Θ.

Finally, we introduce a new version of the well-formedness judgements for term contexts, written $\Theta \vdash \Gamma$ **ok**, also shown in Figure 4.2 on the following page. It imposes that every free type variable appearing in a type in Γ must be restricted to monomorphic types/substitution by Θ. Note that this is different from requiring that all types in Γ are monomorphic. If we consider a singleton term context containing only x, we have that while $\Theta \vdash (x : \forall a.a \to a)$ **ok** holds for any Θ, we only have $\Theta \vdash (x : \forall a.a \to b)$ **ok** if $(b : \bullet) \in \Theta$.

4.1.2 The extended FreezeML typing relation

We now introduce the *extended* FreezeML typing relation, using restriction contexts instead of type contexts, resulting in judgements of the form $\Theta; \Gamma \vdash^{\mathrm{E}} M : A$. Its typing

$\boxed{\Delta \vdash \theta : \Theta \Rightarrow \Theta'}$

$$\frac{\Delta \# \Theta}{\Delta \vdash \emptyset : \cdot \Rightarrow \Theta} \qquad \frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta' \qquad (\Delta^{\bullet}, \Theta') \vdash_{R} A \ \mathbf{ok} \qquad \Delta \# (\Theta, a : R)}{\Delta \vdash \theta[a \mapsto A] : (\Theta, a : R) \Rightarrow \Theta'}$$

$\boxed{\Theta \vdash \Gamma \ \mathbf{ok}}$

$$\frac{}{\Theta \vdash \cdot \ \mathbf{ok}} \qquad \frac{\begin{array}{c}\Theta \vdash_{\star} A \ \mathbf{ok} \qquad \Theta \vdash \Gamma \ \mathbf{ok} \\ \text{for all } a \in \mathsf{ftv}(A) : (a : \bullet) \in \Theta\end{array}}{\Theta \vdash \Gamma, x : A}$$

Figure 4.2: Well-formedness of substitutions and well-formedness of term contexts under restriction contexts

rules, shown in Figure 4.3 on the next page, arise from the original FreezeML typing rules (see Figure 3.5 on page 45), by simply replacing type contexts Δ with restriction contexts Θ. The implicit preconditions of $\Theta; \Gamma \vdash^{\mathrm{E}} M : A$ now become $\mathsf{ftv}(\Theta); \Gamma \vdash M \ \mathbf{ok}$ and $\Theta \vdash \Gamma \ \mathbf{ok}$.

Note that in the rule VARPLAIN, we convert Θ back to a type context when stating the well-formedness premise for the substitution δ. In the rule LETPLAIN, we do the same to re-use the original definition of $\Updownarrow$. We also re-use the existing definition of $\mathsf{principal}$, meaning that it remains to be defined in terms of the original typing relation. We return to this issue later.

Given that the premises in the rules in Figure 4.3 are mostly carbon copies of those of the original typing relation, except for some changes to the involved contexts, the reader may wonder how the extended typing judgement differs from the original one. Recall that the goal of defining judgements $\Theta; \Gamma \vdash^{\mathrm{E}} M : B$ was that this judgement should permit a substitution property reflecting exactly when the original typing judgement would have allowed replacing a type variable with a polymorphic type.

To this end, we observe that there are only two FreezeML typing rules in Figure 3.5 that impose some kind of monomorphism condition that may hence be invalidated when a substitution with a polymorphic type has occurred.

1. In LAMPLAIN, any type variable occurring in S must not be substituted with a polytype, as this would mean that S is no longer a monomorphic type, as required for un-annotated lambdas.

2. For **let** $x = M$ **in** N where $M \notin \mathsf{GVal}$, the rule LETPLAIN dictates that the type A

$\boxed{\Theta; \Gamma \vdash^{\mathrm{E}} M : A}$

$$\textsc{VarFrozen}\quad \frac{x : A \in \Gamma}{\Theta; \Gamma \vdash^{\mathrm{E}} \lceil x \rceil : A} \qquad \textsc{VarPlain}\quad \frac{x : \forall \overline{a}.H \in \Gamma \qquad \mathsf{ftv}(\Theta) \vdash \delta : \overline{a} \Rightarrow_{\star} \cdot}{\Theta; \Gamma \vdash^{\mathrm{E}} x : \delta(H)}$$

$$\textsc{App}\quad \frac{\Theta; \Gamma \vdash^{\mathrm{E}} M : A \to B \qquad \Theta; \Gamma \vdash^{\mathrm{E}} N : A}{\Theta; \Gamma \vdash^{\mathrm{E}} M N : B}$$

$$\textsc{LamPlain}\quad \frac{\Theta; (\Gamma, x : S) \vdash^{\mathrm{E}} M : B}{\Theta; \Gamma \vdash^{\mathrm{E}} \lambda x.M : S \to B} \qquad \textsc{LamAnn}\quad \frac{\Theta; (\Gamma, x : A) \vdash^{\mathrm{E}} M : B}{\Theta; \Gamma \vdash^{\mathrm{E}} \lambda (x : A).M : A \to B}$$

$$\textsc{LetPlain}\quad \frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Theta \\ (\mathsf{ftv}(\Theta), \overline{a}, M, A') \Updownarrow A \\ (\Theta, \overline{a : \bullet}); \Gamma \vdash^{\mathrm{E}} M : A' \qquad \Theta; (\Gamma, x : A) \vdash^{\mathrm{E}} N : B \qquad \mathsf{principal}(\mathsf{ftv}(\Theta), \Gamma, M, \overline{a}, A')\end{array}}{\Theta; \Gamma \vdash^{\mathrm{E}} \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

$$\textsc{LetAnn}\quad \frac{(\overline{a}, A') = \mathsf{split}(A, M) \qquad (\Theta, \overline{a : \bullet}); \Gamma \vdash^{\mathrm{E}} M : A' \qquad \Theta; (\Gamma, x : A) \vdash^{\mathrm{E}} N : B}{\Theta; \Gamma \vdash^{\mathrm{E}} \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B}$$

Figure 4.3: FreezeML extended typing rules

must arise from A' by monomorphically instantiating the variables $\overline{a}$ in A' using some instantiation δ'. Thus, if any of the type variables in $\mathsf{ftv}(\delta'(\overline{a}))$ were to be substituted with a polymorphic type, then a typing derivation for the substituted judgement would have to use a polymorphic δ' instead, violating the rule's premises.

Thus, in the first case the extended typing relation must ensure that the free type variables of S are monomorphic in the ambient restriction context Θ. However, the typing premise $\Theta; (\Gamma, x : S) \vdash^{\mathrm{E}} M : B$ of LamPlain has $\Theta \vdash \Gamma, x : S$ **ok** as an implicit precondition, which requires exactly this property about S.

In the second case, the rule LetPlain of the extended typing judgement must ensure that all type variables in $\delta'(\overline{a})$ must be monomorphic (or rigid, i.e., contained in Δ) in the ambient restriction context. However, as A is defined to be $\delta'(A')$ in this case and all $\overline{a}$ are guaranteed to appear freely in A', a similar implicit precondition of LETPLAIN′s premise $\Theta; (\Gamma, x : A) \vdash^{\mathrm{E}} N : B$ ensures exactly that.

In conclusion, we observe that the implicit precondition $\Theta \vdash \Gamma$ **ok** of the extended typing judgement alone enables the desired substitution property by imposing monomorphism restrictions on all type variables that may not be substituted with polymorphic types. Intuitively, this condition enshrines that the polymorphism occurring in the type A of any term variable x must be fully determined before $(x : A)$ may be added to the term context Γ, which is a cornerstone for ensuring that type inference is decidable for FreezeML.

Returning to our example terms on page 72, we may now observe the following when using the extended typing relation.

$$\begin{aligned}
(a : \bullet); \Gamma &\vdash^{\mathrm{E}} \mathsf{id} &&: a \to a \\
(a : \star); \Gamma &\vdash^{\mathrm{E}} \mathsf{id} &&: a \to a \\
(b : \bullet); \Gamma &\vdash^{\mathrm{E}} \lambda x.x &&: b \to b \\
(b : \star); \Gamma &\nvdash^{\mathrm{E}} \lambda x.x &&: b \to b
\end{aligned}$$

This reflects our earlier discussion about $\lambda x.x$ not being able to be typed with $(\forall a.a \to a) \to (\forall a.a \to a)$, while id can.

4.1.2.1 Old against new

Before stating the type substitution lemma of the extended typing relation, we explore the relationship between the latter and the original typing relation. We observe that the original typing relation is a subrelation of the extended one if temporarily identify typing contexts $\Delta = \overline{a}$ with restriction contexts $\overline{a : \bullet}$. This is formalised by the following lemma.

Lemma 4.2 (Relationship between original and extended FreezeML typing relation)**.** *The following conditions hold for all $\overline{a}$, Γ, M, and A.*

1. *If $\Delta; \Gamma \vdash M : A$ then $\Delta^{\bullet}, \Gamma \vdash^{E} M : A$.*
2. *If $\Theta; \Gamma \vdash^{E} M : A$ then $\mathsf{ftv}(\Theta); \Gamma \vdash M : A$.*

Proof. Straightforward structural induction on M for both properties. The principality premises in the LETPLAIN rule of each system can be used unchanged for the corresponding derivation in the other systems, as both use the same principal relation defined in terms of the original typing relation. □

No need to change the principality relation Note that the premise $(\Theta;\overline{a:\bullet});\Gamma \vdash^{\mathrm{E}} M : A'$ of LETPLAIN is *not* implied by $\mathsf{principal}(\mathsf{ftv}(\Theta),\Gamma,M,\overline{a},A')$, which was the case for the corresponding premises in the original LETPLAIN rule. This is due to $\mathsf{principal}$ being defined in terms of the original typing relation. However, leaving the stronger condition $(\Theta;\overline{a:\bullet});\Gamma \vdash^{\mathrm{E}} M : A'$ as a standalone premise has another advantage, in addition to mirroring the original version of the rule more closely. With this premise we can establish that there is in fact no need to redefine $\mathsf{principal}$ to a different version, say $\mathsf{principal}^{\mathrm{E}}$, that would be expressed in terms of the extended typing relation. The following lemma shows that in addition to $(\Theta;\overline{a:\bullet});\Gamma \vdash^{\mathrm{E}} M : A'$, the remaining conditions of this hypothetical premise $\mathsf{principal}^{\mathrm{E}}(\Theta,\Gamma,M,\Delta,A')$ are already established by the premises in the LETPLAIN rule in Figure 4.3.

Lemma 4.3 (Relationship between $\mathsf{principal}$ and $\vdash^{\mathrm{E}}$)**.**
Let $\mathsf{principal}(\mathsf{ftv}(\Theta),\Gamma,M,\overline{a},A)$ *hold. Then for all* A' *and* Θ' *we have that if* $(\Theta,\Theta');\Gamma \vdash^{E} M : A'$ *then there exists* δ *such that* $\mathsf{ftv}(\Theta) \vdash \delta : \overline{a} \Rightarrow_{\star} \mathsf{ftv}(\Theta')$ *and* $\delta(A) = A'$.

Proof. For every such $(\Theta,\Theta');\Gamma \vdash^{\mathrm{E}} M : A'$, we can apply Lemma 4.2, yielding $(\mathsf{ftv}(\Theta),\mathsf{ftv}(\Theta'));\Gamma \vdash M : A'$. By $\mathsf{principal}(\mathsf{ftv}(\Theta),\Gamma,M,\overline{a},A')$ we then have that an appropriate δ exists. □

4.1.3 Substitution property

We may now use the extended FreezeML typing relation to state a type substitution property going beyond monomorphic substitution. The substitution lemma relies on the key property of the extended FreezeML typing relation introduced in the previous section, namely the fact that $\Theta,(a:\star);\Gamma \vdash^{\mathrm{E}} M : A$ implies that a may safely be substituted with any (well formed) polymorphic type without breaking typeability. Therefore, a substitution θ that is well-formed with respect to the restriction context used in a typing judgement involving $\vdash^{\mathrm{E}}$ preserves typeability. Further, as expected for a substitution property, the resulting type of the term results from applying θ to the original type.

Lemma 4.4 (Stability of typing under substitution)**.**
If $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta;\Gamma \vdash M$ **ok**, *then* $(\Delta^{\bullet},\Theta);\Gamma \vdash^{E} M : A$ *implies* $(\Delta^{\bullet},\Theta');\theta(\Gamma) \vdash^{E} M : \theta(A)$.

Proving this is straightforward, with one notable exception: When proving that $(\Delta^{\bullet},\Theta);\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : A$ implies $(\Delta^{\bullet},\Theta');\theta(\Gamma) \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \theta(A)$, we need to reason about the principal type of M in context $(\Delta^{\bullet}, \Theta'); \theta(\Gamma)$. Here we need to rely on the property that principality is also stable under substitution.

Lemma 4.5 (Stability of principality under substitution).
Let $(\Delta^{\bullet}, \Theta) \vdash \Gamma\, \mathbf{ok}$ *and* $\Delta; \Gamma \vdash M\, \mathbf{ok}$ *and* $\mathsf{ftv}(\Theta') \,\#\, \Delta'$. *If* $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta)), \Gamma, M, \Delta', A)$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$*, then* $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta')), \theta(\Gamma), M, \Delta', \theta(A))$.

Unfortunately, while this property is intuitively correct, it is not straightforward to prove. Crucially, according to the definition of $\mathsf{principal}$, we need to show that for any Δ'' and A'' we have that $(\Delta, \mathsf{ftv}(\Theta'), \Delta''); \theta(\Gamma) \vdash M : A''$ implies that A'' can be obtained from $\theta(A)$ using an appropriate substitution. However, the fact that $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta)), \Gamma, M, \Delta', A)$ holds does not immediately help towards this goal, as it only applies to typing judgements involving the original term context Γ.[1]

However, the reasoning behind how principal types are obtained for each term and how they are influenced by substitutions is exactly what is required to establish the correctness of the type inference algorithm introduced later in this chapter. In fact, Lemma 4.5 easily follows from several properties derived from the completeness of our type inference algorithm and the fact that it infers principal types.

The proofs of Lemmas 4.4 and 4.5 are then shown in Appendix A.3. Of course, care is required to avoid circular reasoning, as the correctness of the inference algorithm in turn relies on the overall substitution property (i.e., Lemma 4.4). We address this in Appendix A.5.

4.2 Unification

In general, unification is the process of determining for two terms Φ and Ξ whether there exists a substitution υ, called *unifier*, such that $\upsilon(\Phi)$ and $\upsilon(\Xi)$ are equivalent. Robinson [105] has shown that unification is decidable if Φ and Ξ are first-order terms and the notion of equivalence is syntactic equality. Further, if successful, his algorithm yields most general unifiers υ_g, meaning that for every other unifier υ_o there exists υ' such that $\upsilon_o = \upsilon' \circ \upsilon_g$. First-order unification is an essential ingredient of Algorithm W. Higher-order unification, where the language of terms to be unified is extended to lambda terms, is undecidable in general [43, 31], but various decidable fragments have been identified [76, 65, 107, 108, 66].

We define an extension of Robinson's algorithm that can unify quantified types and

[1]Surprisingly, the other two systems known to the author that state a type substitution property for a typing relation with a principality condition do not give any details for its proof [89, 57].

takes restrictions R imposed on type variables into account. The algorithm is shown in Figure 4.4 on the following page. In addition to the types A_{u} and B_{u} to unify, the algorithm takes two type contexts as input. The sequence Δ contains *rigid* type variables, also called *skolem variables* or *skolem constructors* [52], that are in scope but must not be substituted with other types. The second context Θ contains *flexible* variables that may be substituted with other types. These substitutions are subject to the restrictions in Θ, which we return to later.

The input types A_{u} and B_{u} are expected to be well formed under the combined restriction context $(\Delta^{\bullet}, \Theta)$. If successful, the algorithm returns an updated restriction context Θ' with $\mathsf{ftv}(\Theta') \subseteq \mathsf{ftv}(\Theta)$ and a most general unifier θ of A_{u} and B_{u}. The algorithm ensures that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ holds.

The algorithm is defined in terms of the structure of A_{u} and B_{u}; the first clause that applies is executed. Our presentation loosely follows that of Leijen [58]. In general, the algorithm may only fail if A_{u} and B_{u} do not match any of the five clauses given for $\mathcal{U}$, or if an assertion or recursive invocation of $\mathcal{U}$ fails.

If both types are the same type variable (from either Δ or Θ), the algorithm returns Θ unchanged and returns the identity on Θ as their unifier. If both types are applications of a type constructor D, the corresponding arguments are unified, subject to the intermediate unifier accumulated so far.

If two quantified types $\forall a.A$ and $\forall b.B$ are to be unified, the algorithm picks a fresh skolem variable c, and unifies the result of replacing the quantified variable a and b with c in A and B, respectively. Due to c being added to Δ it will not be substituted with other types, but a flexible variable may be substituted with c (explained below). The latter would however represent a case of a quantified variable escaping its scope. Therefore, the algorithm performs an escape check to ensure that examples such as $\mathcal{U}(\cdot, (d : \star), \forall a.a \to a, \forall b.b \to d)$ fail.

Finally, if none of these cases apply, but one of the two types to unify is a flexible variable, it may be unified with the other type A. This leads to a unifier that maps a to A and is the identity on all other variables. However, this is subject to additional checks. The idea is that unifying a with some type A always succeeds if the restriction on a is $\star$ (i.e., polymorphic) in the context. Otherwise, if a is restricted to monomorphic types, the substitution is only permitted if A is a monomorphic type after *demoting* all of its free flexible type variables to $\bullet$. This is formalised using the helper function demote.

In general, $\mathsf{demote}(R, \Theta, \overline{a})$ returns a restriction context Θ' containing exactly the variables from Θ. The sequence $\overline{a}$ must contain a subset of the variables of Θ. If R is

$\boxed{\mathcal{U}(\Delta,\Theta,A,B)}$

```
$\mathcal{U}(\Delta,\Theta,a,a) =$
  return $(\Theta,\iota_\Theta)$

$\mathcal{U}(\Delta,\Theta,D\,\overline{A},D\,\overline{B}) =$
  let $(\Theta_1,\theta_1) = (\Theta,\iota_\Theta)$
  let $n = \mathsf{arity}(D)$
  for $i \in 1\ldots n$
    let $(\Theta_{i+1},\theta_{i+1}) =$
      let $(\Theta',\theta') = \mathcal{U}(\Delta,\Theta_i,\theta_i(A_i),\theta_i(B_i))$
      return $(\Theta',\theta' \circ \theta_i)$
  return $(\Theta_{n+1},\theta_{n+1})$

$\mathcal{U}(\Delta,\Theta,\forall a.A,\forall b.B) =$
  let $c \notin \Delta,\Theta$
  let $(\Theta_1,\theta') = \mathcal{U}((\Delta,c),\Theta,A[c/a],B[c/b])$
  assert $c \notin \mathsf{ftv}(\theta')$
  return $(\Theta_1,\theta')$

$\mathcal{U}(\Delta,(\Theta,a:R),a,A) =$
$\mathcal{U}(\Delta,(\Theta,a:R),A,a) =$
  let $\Theta_1 = \mathsf{demote}(R,\Theta,\mathsf{ftv}(A)-\Delta)$
  assert $(\Delta^\bullet,\Theta_1) \vdash_R A\ \mathbf{ok}$
  return $(\Theta_1,\iota_{\Theta_1}[a \mapsto A])$
```

$\boxed{\mathsf{demote}(R,\Theta,\Delta)}$

$$\begin{aligned}
\mathsf{demote}(\star,\Theta,\overline{a}) &= \Theta \\
\mathsf{demote}(\bullet,\cdot,\overline{a}) &= \cdot \\
\mathsf{demote}(\bullet,(\Theta,b:R),\overline{a}) &= \mathsf{demote}(\bullet,\Theta,\overline{a}),b:\bullet \quad (b \in \overline{a}) \\
\mathsf{demote}(\bullet,(\Theta,b:R),\Delta) &= \mathsf{demote}(\bullet,\Theta,\overline{a}),b:R \quad (b \notin \overline{a})
\end{aligned}$$

Figure 4.4: Unification algorithm

$\star$, then Θ is left unchanged. Otherwise, the restrictions of all variables from $\overline{a}$ are set to $\bullet$ in the result Θ', and left unchanged for all variables in $\Theta - \overline{a}$.

The unification algorithm then implements the check described on page 79 by asserting that $(\Delta^{\bullet}, \Theta_1) \vdash_R A$ **ok** holds, where Θ_1 is the result of demoting free variables in A and R is the restriction of a in the original restriction context. Note that the context Θ_1 used for this well formedness check does not contain a, which implicitly performs an *occurs check*, disallowing recursive types. Note that returning Θ_1 also shows that a is not considered further after this point, the returned substitution effectively removes the variable.

Overall, the treatment of restrictions by the unification algorithm illustrates the role of restrictions on type variables during type inference. If a type variable appears freely in one of the types to be unified, no substitution for it is currently known. Demoting such a variable to $\bullet$ then corresponds to unification uncovering a constraint on the variable, specifically the fact that it must only be substituted with monomorphic types going onward.

As an example, consider $\mathcal{U}(\cdot, (a : \bullet, b : \star, c : \star), a, b \to c)$. This successfully returns $(b : \bullet, c : \bullet), [a \mapsto (b \to c)]$. Demoting b and c resulted in the type $b \to c$ being monomorphic, meaning that it can be substituted for a. However, $\mathcal{U}(\cdot, (a : \bullet, b : \star), a, \forall c.b \to c)$ fails. The type $\forall c.b \to c$ is still polymorphic, even after demoting the restriction on b to $\bullet$. Thus, the same example would succeed if we had $(a : \star)$ in the input context instead.

Finally, $\mathcal{U}(\cdot, (a : \bullet, b : \bullet,), a, a \to b)$ fails due to the occurs check.

Regarding the correctness of $\mathcal{U}$, we first observe its termination.

Theorem 4.1 (Termination of $\mathcal{U}$).
The function $\mathcal{U}$ terminates on all inputs.

Proof. We utilise a standard argument for termination of the original algorithm [81], which also applies to our version. We define the degree of a tuple (A, B) of types as a tuple (m, n), where m is the number of distinct type variables occurring freely in A or B and n is the sum of the sizes of A and B (counting all occurrences of type constructors, quantifiers, and type variables). We then observe that for all recursive calls performed by $\mathcal{U}(\Delta, \Theta, A, B)$, the degree of the arguments (A', B') is strictly lexicographically smaller than the degree of (A, B). This follows from the fact that whenever our algorithm constructs a substitution θ with a non-identity mapping $a \mapsto A''$, it removes a from the codomain of θ. $\square$

The unification algorithm is sound, meaning that if it succeeds, the result is indeed a well formed unifier of the input types.

Theorem 4.2 (Soundness of $\mathcal{U}$)**.**
Let $(\Delta^\bullet, \Theta) \vdash_\star A$ **ok** *and* $(\Delta^\bullet, \Theta) \vdash_\star B$ **ok** *hold. If* $\mathcal{U}(\Delta, \Theta, A, B) = (\Theta', \theta)$ *holds, then we have* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\theta(A) = \theta(B)$.

Proof. Let A' and B' denote the types to unify. We observe that structural induction on the input types is not sufficient, as the algorithm is recursively invoked on parts of either type *after* applying an intermediate substitution. Instead, the proof is by complete induction on the number of recursive invocations of $\mathcal{U}$. Due to Theorem 4.1, we know that this number is finite.

Since we are assuming that unification succeeds, we only consider the different shapes of A' and B' where a clause of $\mathcal{U}$ in Figure 4.4 applies.

- Case $A' = B' = a$: The desired properties follow immediately for the identity substitution ι_Θ.

- Case $A' = D\,\overline{A}$, $B' = D\,\overline{B}$, where $\overline{A} = A_1, \dots, A_n$ and $\overline{B} = B_1, \dots, B_n$: The reasoning in this case is largely standard, just accommodating our usage of explicit contexts. We perform a nested induction, showing that for all $0 \leq j \leq n$ the following holds: We have $\Delta \vdash \theta_{j+1} : \Theta \Rightarrow \Theta_{j+1}$ and $\theta_{j+1}(A_k) = \theta_{j+1}(B_k)$ for all $1 \leq k \leq j$. Due to $\Theta_1 \overset{\text{def}}{=} \Theta$ and $\theta_1 \overset{\text{def}}{=} \iota_\Theta$, this holds immediately for $j = 0$. For $j > 0$, we have $\theta_{j+1} \overset{\text{def}}{=} \theta' \circ \theta_j$, where θ' is the result of unifying $\theta_j(A_j)$ and $\theta_j(B_j)$. According to the outer induction hypothesis we then have $\theta'(A_j) = \theta'(B_j)$ and $\Delta \vdash \theta' : \Theta_j \Rightarrow \Theta_{j+1}$. Composing θ_j with θ' to yield θ_{j+1} does not change the fact that it is a unifier of all A_k and B_k for $1 \leq k < j$ (see Lemma A.5 on page 229), meaning that θ_{j+1} is indeed a unifier of A_k and B_k for all $1 \leq k \leq j$. Further, by Lemma A.4 on page 229 we have the desired well-formedness of the composition.

- Case $A' = \forall a.A$, $B' = \forall b.b$: By induction we have $\theta'(A[c/a]) = \theta'(B[c/b])$ and $(\Delta, c) \vdash \theta' : \Theta \Rightarrow \Theta_1$. The escape check allows us to strengthen the latter to the desired property $\Delta \vdash \theta' : \Theta \Rightarrow \Theta_1$.

  By alpha-equivalence, we may assume that in addition to c, a and b are also fresh. Hence, no variable capture is taking place when applying θ', and we have that $\theta'(A[c/a]) = \theta'(B[c/b])$ implies $\theta'(\forall a.A) = \theta'(\forall b.B)$.

- Case $A' = a$ or $B' = a$: By the definition of demote, we have $\mathsf{ftv}(\Theta) = \mathsf{ftv}(\Theta_1)$. This implies $a \notin \Theta_1$ and due to the assertion $(\Delta^\bullet, \Theta_1) \vdash_R A$ **ok** we have $a \notin \mathsf{ftv}(A)$. Therefore, the returned substitution has no effect on A. Together with $\Delta \vdash \iota_\Theta : \Theta \Rightarrow \Theta_1$ and the assertion, we obtain $\Delta \vdash \iota_\Theta[a \mapsto A] : (\Theta, a : R) \Rightarrow \Theta_1$.

□

Unification is also complete, meaning that if two types have a unifier θ, then $\mathcal{U}$ succeeds. In this case, the returned unifier is also more general than θ, meaning that it can be composed with another substitution to yield θ.

Theorem 4.3 (Completeness and generality of $\mathcal{U}$)**.**
Let $(\Delta^\bullet, \Theta) \vdash_\star A$ **ok** *and* $(\Delta^\bullet, \Theta) \vdash_\star B$ **ok** *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *hold. If* $\theta(A) = \theta(B)$*, then there exist* $\theta', \theta'', \Theta''$ *such that* $\mathcal{U}(\Delta, \Theta, A, B) = (\Theta'', \theta'')$ *and* $\Delta \vdash \theta' : \Theta'' \Rightarrow \Theta'$ *and* $\theta = \theta' \circ \theta''$.

Proof. The proof is by complete induction on the number of recursive calls, for the same reason as outlined in the proof of Theorem 4.2. We perform a case analysis over all possible shapes of A and B.

- Case $A = D\, A_1 \ldots A_n$, $B = D\, B_1 \ldots B_n$. We perform a nested induction, showing that for any $0 \leq j \leq n$, the following holds:

  1. The invocation $\mathcal{U}(\Delta, \Theta_j, \theta_j(A_j), \theta_j(B_j))$ succeeds.
  2. There exists θ'_j such that $\Delta \vdash \theta'_j : \Theta_{j+1} \Rightarrow \Theta'$ and $\theta = \theta'_j \circ \theta_{j+1}$

  For $j = 0$, the first condition is vacuous and we may choose $\theta'_0 = \theta$.

  For the case $j > 0$ we first show that $\mathcal{U}(\Delta, \Theta_j, \theta_j(A_j), \theta_j(B_j))$ succeeds. According to the nested induction hypothesis, we have that $\theta = \theta'_{j-1} \circ \theta_j$ and $\Delta \vdash \theta'_{j-1} : \Theta_j \Rightarrow \Theta'$. We may therefore apply the outer induction hypothesis, guaranteeing that unification succeeds. Let θ'_{U}) be the substitution returned by $\mathcal{U}(\Delta, \Theta_j, \theta_j(A_j), \theta_j(B_j))$.

  The inner induction hypothesis further yields the existence of θ'' such that $\theta'_{j-1} = \theta'' \circ \theta'_{\mathrm{U}}$ and $\Delta \vdash \theta'' : \Theta_{j+1} \Rightarrow \Theta'$. Taking θ'_j to be θ'', we then observe the following.

$$\theta \;=\; \theta'_{j-1} \circ \theta_j \;=\; \underbrace{\theta''}_{\overset{\mathrm{def}}{=} \theta'_j} \circ \underbrace{\theta'_{\mathrm{U}} \circ \theta_j}_{\overset{\mathrm{def}}{=} \theta_{j+1}}$$

  Thus, we have shown both parts of the nested induction hypothesis and observe that for $j = n$ it implies the statement of the theorem.

- Case $A = \forall a.A'$, $B = \forall b.B'$. By alpha-equivalence, we may assume that a and b are fresh, which means that $\theta(A) = \theta(B)$ implies $\theta(A') = \theta(B')$. Let $c \notin \Delta, \Theta$ be the variable picked by the algorithm. Thus, we have that c does not appear in the domain of θ and we have $\theta(A') = \theta(A'[c/a]) = \theta(B') = \theta(B'[c/b])$.

  We may assume w.l.o.g. that $c\#\Theta'$ holds.[2] We thus have $(\Delta, c) \vdash \theta : \Theta \Rightarrow \Theta'$. We observe that $A'[c/a]$ and $B'[c/b]$ are the types used in the recursive call. By induction we have that $\mathcal{U}((\Delta,c),\Theta,A'[c/a],B'[c/b])$ succeeds, returning some (Θ_1, θ') for which there exists θ'' such that $\theta = \theta'' \circ \theta'$ and $(\Delta, c) \vdash \theta'' : \Theta_1 \Rightarrow \Theta'$. Note that the latter implies $c\#\Theta_1$.

  We show $c \notin \mathsf{ftv}(\theta')$ by contradiction: Otherwise, there would exist $d \in \Theta$ such that $c \in \mathsf{ftv}(\theta'(d))$. However, by $\theta = \theta'' \circ \theta'$ and $\theta''(c) = c$ (due to $c\#\Theta_1$) this would imply $c \in \mathsf{ftv}(\theta(d)) = \mathsf{ftv}(\theta''(\theta'(d)))$. By assumption $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, this violates $c\#\Theta', \Delta$. Thus the assertion in the algorithm succeeds.

  Using the same reasoning, we can show that for all $a \in \mathsf{ftv}(\theta')$, we have $c \notin \mathsf{ftv}(\theta''(a))$. Now let $\overline{a} := \Theta_1 - \mathsf{ftv}(\delta')$ and $\theta''' := \theta''\restriction_{(\Theta_1 - \overline{a})}[\overline{a} \mapsto \mathsf{Unit}]$. This means that we turn θ'' into a substitution that behaves identically when composed with θ', but whose codomain does not contain c.

  This construction guarantees that θ''' has the necessary properties, namely $\theta''' \circ \theta' = \theta'' \circ \theta' = \theta$ and $\Delta \vdash \theta''' : \Theta_1 \Rightarrow \Theta'$.[3]

- Case $A = a$. If $B = a$, the algorithm returns $(\Theta, \iota_\Theta\Theta)$ due to its first clause. Choosing $\theta' = \theta$ yields the required properties.

  In the case $B \neq a$, we observe that the assumptions $\theta(a) = \theta(B)$ and $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ imply $a \in \Theta$. Due to types being finite syntax trees, this also implies $a \notin \mathsf{ftv}(B)$.

  Let $\Theta'' := \Theta - a$. We show that choosing θ' to be θ restricted to Θ'' (i.e., without the mapping for a) satisfies the required properties. Note that this immediately yields $\Delta\theta' : \Theta'' \Rightarrow \Theta'$.

  As the algorithm chooses $\theta'' = \iota_\Theta[a \mapsto B]$, we first need to show that $\theta = \theta' \circ \iota_\Theta[a \mapsto B]$. To this end, it suffices to show that $\theta'(B) = \theta(a)$, which holds by $\theta(B) = \theta(a)$ and $a \notin \mathsf{ftv}(B)$.

[2]Otherwise, we may perform the subsequent reasoning on $\theta_f = [c \mapsto d] \circ \theta$ for some fresh d, which gives us $\Delta \vdash \theta_f : \Theta \Rightarrow \Theta' - c, d$ and $\theta_f(A) = \theta_f(B)$. We can then revert the renaming in the end.

[3]The unification algorithm actually guarantees that θ' is surjective with respect to Θ_1, meaning that $\overline{a}$ is always empty. However, there is no need to rely on this fact here, by turning θ'' into θ''' instead.

It remains to show that the algorithm actually succeeds and that we have $\Delta \vdash \theta' : \Theta_1 \Rightarrow \Theta'$.

We first consider the case where $(a : \star) \in \Theta$. We then have that $\Theta_1 \overset{\text{def}}{=} \mathsf{demote}(\star, \Theta'', \mathsf{ftv}(B) - \Delta) = \Theta''$. As a result, the assertion is immediately satisfied by assumptions $\Delta^\bullet, \Theta \vdash B \ \mathbf{ok}$ and the result $a \notin \mathsf{ftv}(B)$ shown earlier, meaning that the algorithm succeeds. We observe that $\Delta \vdash \theta' : \Theta_1 \Rightarrow \Theta'$ follows immediately.

In the case $(a : \bullet) \in \Theta$, we have that $\Theta_1 \overset{\text{def}}{=} \mathsf{demote}(\bullet, \Theta'', \mathsf{ftv}(B) - \Delta$ contains the same variables as Θ'', but those in $\mathsf{ftv}(B) - \Delta$ (i.e., the free flexible variables of B) are monomorphic.

By $\Delta \vdash \Theta \Rightarrow \Theta$ and $(a : \bullet) \in \Theta$ we have $(\Delta^\bullet, \Theta') \vdash_\bullet \theta(a) \ \mathbf{ok}$, which implies that $\theta(a)$ contains no quantifiers and only variables marked as monomorphic in Θ'. By $\theta(a) = \theta(B)$ we thus have that θ maps all free flexible type variables of B to monomorphic types. Thus, when considering the restriction θ' of θ, we have $\Delta \vdash \theta' : \Theta_1 \Rightarrow \Theta'$.

- Case $B = b$, A is not a variable. We observe that this is the only remaining case: If A is a quantified type or type constructor application, then assumption $\theta(A) = \theta(B)$ imposes that B is either a variable, or otherwise it must have the same shape as A (i.e., also be a quantified type or an application of the same type constructor). We may then use the same reasoning as in the case $A = a$.

□

4.3 Type inference algorithm

The type inference algorithm is implemented using the function infer, which is defined in Figure 4.5 on the next page. It takes a type context Δ, a restriction context Θ, a term context Γ and a term M as input. Here, Δ and Θ play similar roles as in the unification algorithm: While Δ contains rigid variables, Θ contains flexible variables that may be solved during inference in the sense of finding substitutions for them. As mentioned before, we consider type variables in M to be rigid and not subject to substitution, which is formalised by requiring $\Delta; \Gamma \vdash M \ \mathbf{ok}$ as a precondition for invoking infer. Similarly, we must have $(\Delta^\bullet, \Theta) \vdash \Gamma \ \mathbf{ok}$. Note that as usual, this means that Γ may contain type variables that may be refined by the inference algorithm by finding substitutions for them, but the well formedness condition imposes that such type variables are restricted to monomorphic solutions. The function infer then succeeds if and

$\boxed{\mathsf{infer}(\Delta,\Theta,\Gamma,M)}$

```
infer(Δ,Θ,Γ,⌈x⌉) =
  let $A = \Gamma(x)$
  return $(\Theta,\iota_\Theta,A)$

infer(Δ,Θ,Γ,x) =
  let $\forall\overline{a}.H = \Gamma(x)$
  let $\overline{b}\#\Delta,\Theta$
  return $((\Theta,\overline{b:\star}),\iota_\Theta,H[\overline{b}/\overline{a}]))$

infer(Δ,Θ,Γ,M N) =
  let $(\Theta_1,\theta_1,A') = \mathsf{infer}(\Delta,\Theta,\Gamma,M)$
  let $(\Theta_2,\theta_2,A) = \mathsf{infer}(\Delta,\Theta_1,\theta_1(\Gamma),N)$
  let $b\#\Delta,\Theta$
  let $(\Theta_3,\theta_3[b\mapsto B]) =$
    $\mathcal{U}(\Delta,(\Theta_2,b:\star),\theta_2(A'),A\to b)$
  return $(\Theta_3,\theta_3\circ\theta_2\circ\theta_1,B)$

infer(Δ,Θ,Γ,λx.M) =
  let $a\#\Delta,\Theta$
  let $(\Theta_1,\theta[a\mapsto S],B) =$
    $\mathsf{infer}(\Delta,(\Theta,a:\bullet),(\Gamma,x:a),M)$
  return $(\Theta_1,\theta,S\to B)$

infer(Δ,Θ,Γ,λ(x:A).M) =
  let $(\Theta_1,\theta,B) =$
    $\mathsf{infer}(\Delta,\Theta,(\Gamma,x:A),M)$
  return $(\Theta_1,\theta,A\to B)$

infer(Δ,Θ,Γ,let x = M in N) =
  let $(\Theta_1,\theta_1,A) = \mathsf{infer}(\Delta,\Theta,\Gamma,M)$
  let $\Delta' = \mathsf{ftv}(\theta_1)-\Delta$
  let $\Delta'' = \mathsf{ftv}(A)-(\Delta,\Delta')$
  $\Delta''' = \begin{cases}\Delta'' & \text{if } M\in\mathsf{GVal}\\ \cdot & \text{otherwise}\end{cases}$
  let $\Theta_1' = \mathsf{demote}(\bullet,\Theta_1,\Delta'')$
  let $(\Theta_2,\theta_2,B) = \mathsf{infer}(\Delta,\Theta_1'-\Delta''',\theta_1(\Gamma),x:\forall\Delta'''.A,N)$
  return $(\Theta_2,\theta_2\circ\theta_1,B)$

infer(Δ,Θ,Γ,let (x:A) = M in N) =
  let $(\Delta',A') = \mathsf{split}(A,M)$
  assert $\Delta'\#\Delta,\Theta$
  let $(\Theta_1,\theta_1,A_1) = \mathsf{infer}((\Delta,\Delta'),\Theta,\Gamma,M)$
  let $(\Theta_2,\theta_2') = \mathcal{U}((\Delta,\Delta'),\Theta_1,A',A_1)$
  let $\theta_2 = (\theta_2'\circ\theta_1)$
  assert $\mathsf{ftv}(\theta_2)\#\Delta'$
  let $(\Theta_3,\theta_3,B) = \mathsf{infer}(\Delta,\Theta_2,(\theta_2(\Gamma),x:A),N)$
  return $(\Theta_3,\theta_3\circ\theta_2,B)$
```

Figure 4.5: Type inference function infer

only if M is well typed in the given contexts. It then returns a tuple (Θ, θ, A), where $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and θ substitutes variables in Γ such that A is a principal type of M under (Δ, Θ') and $\theta(\Gamma)$. We formalise this in Section 4.3.1.

The inference algorithm is defined in terms of the shape of the input term. If the term under consideration is a frozen variable, infer requires that the variable appears in Γ. If so, the restriction context is returned unchanged, together with an identity substitution and the type of x obtained from Γ.

In case of a plain variable, inference behaves like the original Algorithm W. The (toplevel) quantifiers in the type of x are instantiated with fresh variables $\overline{b}$. The only non-standard aspect is that we add the variables $\overline{b}$ with restrictions $\star$ to the returned restriction context. This ensures that these variables may be unified with polymorphic types later. Note that for both sorts of variables, inference fails if the variable is not found in Γ. Besides that, the algorithm may only fail if an explicit assertion, a call to $\mathcal{U}$, or a recursive call of infer fails.

The treatment of function applications $M\ N$ is also standard. We infer types the type A' of M and then the type A of N, piping the updated context and substitution obtained from the earlier step through. A fresh (polymorphic) type variable b is then used to unify the overall function type A' with the type $A \to b$, to reflect the fact that A is the inferred type of the argument N.

Type inference for un-annotated functions is standard, too. A fresh variable a is used as the type of x while inferring a type for the body of the function. The only difference is that we explicitly mark the type variable a as monomorphic, reflecting the fact that we must only infer monomorphic parameter types. Inference for annotated functions works analogously, but no fresh type variable needs to be introduced for the type of the parameter. Note that the well formedness condition on the overall term imposes that the type annotation only contains variables from Δ, hence not subject to substitution.

As examples, consider the following three inference outcomes.

$$\begin{aligned}
\mathsf{infer}(\cdot,\cdot,\cdot,\lambda x.x) &= ((a:\bullet),[],a \to a) && (4.1)\\
\mathsf{infer}(\cdot,(b_1:\bullet),(f:b_1 \to b_1, \mathsf{id}:\forall c.c \to c), f\ \ \mathsf{id}) &= (\Theta,\theta,b_2 \to b_2) && (4.2)\\
\mathsf{infer}(\cdot,(b_1:\bullet),(f:b_1 \to b_1, \mathsf{id}:\forall c.c \to c), f\ \lceil \mathsf{id} \rceil) &\quad \textit{fails} && (4.3)\\
\text{where } \Theta = (b_2:\bullet), \theta = [b_1 \mapsto (b_2 \to b_2)]
\end{aligned}$$

In the first example, we observe that performing inference on the term $\lambda x.x$ results in a restriction context with a fresh variable a. It is monomorphic, as it was introduced as the type of an un-annotated lambda parameter. The domain of the returned substitution is the initial restriction context and therefore empty. Note that the usage of x in the body of the function did not result in new type variables being created as a result of instantiation, due to the lack of (toplevel) quantifiers in the type of x, namely a.

In the second example, we perform inference for the term f id, where we have $b_1 : \bullet$ while f has type $b_1 \to b_1$ and id has its usual type. Inference for the subterm f has no effect, while inference for the subterm id instantiates its type to $b_2 \to b_2$ for a fresh polymorphic variable b_2. Inference for the application f id then involves calling $\mathcal{U}(\cdot,(b_1 : \bullet, b_2 : \star, b_3 : \star), b_1 \to b_1, (b_2 \to b_2) \to b_3)$, where b_3 is the fresh variable introduced to represent the function's return type. This results in the following.

$$((b_2 : \bullet), [b_1 \mapsto (b_2 \to b_2), b_2 \mapsto b_2, b_3 \mapsto (b_2 \to b_2)])$$

Composing all intermediate substitutions then effectively restricts the domain of the resulting overall substitution to b_1, leading to the inference result shown in (4.2).

Finally, example (4.3) fails as a monomorphism restriction is violated during unification.

Plain let bindings Inference for terms **let** $x = M$ **in** N works as follows. First, type inference is performed for M, yielding result (Θ_1, θ_1, A). The goal of the algorithm is then to identify those free type variables of A that are generalisable. Recall that the typing rule for let bindings imposes that actual generalisation only takes place if M is a guarded value. In fact, if the let-bound term M is a guarded value and generalisation is to be performed, our inference function behaves like the original Algorithm W, except for modifications needed to incorporate the fact that the contexts Δ and Θ explicitly denote the current set of type variables in scope.

In general, variables already present in the surrounding contexts Δ and Θ must not be generalised. However, while inferring a type for M, some variables from Θ may have been substituted with other types. Thus, the sequence Δ' is defined by the algorithm as those flexible variables not to be generalised, by taking the flexible variables in the codomain of θ_1.[4] The sequence Δ'' then contains the generalisable type variables of A, by removing all rigid variables and those in Δ'. The different behaviour

[4] Alternatively, instead of using $\mathsf{ftv}(\theta_1)$ in the definition of Δ', we may consider only its subset $\mathsf{ftv}(\theta_1\Gamma)$ instead. This is justified since flexible variables appearing in $\mathsf{ftv}(\theta_1)$ but not in $\mathsf{ftv}(\theta_1\Gamma)$ will not be encountered while inferring a type for M. This alternative definition of Δ' would be more in line with classic presentations of Algorithm W.

of let bindings based on whether or not M is a guarded value is then implemented by defining Δ'''. It contains the variables that will actually be quantified to yield the type of x in N.

First, the subset Δ'' of the variables Θ_1 is unconditionally demoted to be monomorphic, yielding Θ_1'. Inference for the subterm N then depends on whether or not M is a guarded value. If it is, the variables Δ'' are removed from Θ_1' prior to the recursive call to infer, making the previous demotion step void (i.e., all previously demoted variables are immediately removed). Instead, the variables in Δ''' are universally quantified in order to yield the type of x. If M is not a value, Δ''' is empty, meaning that removing it from Θ_1' has no effect and no quantification takes place.

We consider the following examples.

$$\mathsf{infer}(\cdot,\cdot,\Gamma,\mathbf{let}\ g_1 = \lambda x.x\ \mathbf{in}\ \lceil g_1 \rceil) = (\cdot,[],\forall a.a \to a) \tag{4.4}$$

$$\mathsf{infer}(\cdot,(b_1 : \bullet),\Gamma,\mathbf{let}\ g_2 = \lambda(x : \mathsf{Int}).f\ \mathsf{id}\ \ \mathbf{in}\ \lceil g_2 \rceil) = (\Theta',\theta',\mathsf{Int} \to b_2 \to b_2) \tag{4.5}$$

$$\text{where } \Gamma = (f : b_1 \to b_1, \mathsf{id} : \forall c.c \to c),\ \Theta' = (b_2 : \bullet)$$

$$\theta' = [b_1 \mapsto (b_2 \to b_2)]$$

In the first example, the let-bound term is the one used in (4.1). Assuming that a fresh variable a is again introduced for the type of x, we have $\Delta' = \cdot$ and $\Delta'' = \Delta''' = a$ once inference is performed for the let binding. Thus, we have $g_1 : \forall a.a \to a$ in the term context while performing inference for the body.

The let-bound term in the second example is the one from (4.2), but wrapped in a lambda term to yield a guarded value. We observe that the variable b_1 appears in the surrounding context and is therefore not a generalisable variable. However, the result of performing inference for f id, as shown in (4.2), indicates that b_1 is substituted with $b_2 \to b_2$. This is correctly reflected in the algorithm, where b_2 appears in the sequence Δ' produced by the algorithm, and therefore not in Δ'', which is empty. In general, note that whenever our algorithm returns a triple such as (Θ_1,θ_1,A), we have $\theta_1(A) = A$ (i.e., in the returned type A, all substitutions discovered by the algorithm have already been applied). Together with the fact that the algorithm returns idempotent substitutions, this means that a variable such as b_1 in (4.5), which is substituted with another type rather than itself, can never appear freely in the type A of the let-bound term M. Conversely, for all generalisable variables $b \in \Delta''$, we must have $\theta_1(b) = b$ in the let case of the inference algorithm.

Annotated let bindings For annotated bindings **let** $(x : A) = M$ **in** N, the inference algorithm relies on the same function split (defined in Figure 3.6 on page 47) used by the corresponding typing rule. This means that if M is a guarded value, then Δ' contains all of its toplevel quantifiers and A' is the remaining guarded type. Otherwise, Δ' is empty and A' is identical to A. The assertion $\Delta' \# \Delta, \Theta$ merely ensures that if M is a guarded value and Δ' thus represents the toplevel quantifiers of A, then A has been sufficiently alpha-converted so that these quantifiers are fresh with respect to the contexts used for inference. As a result, this assertion is always satisfiable.

The algorithm then unifies A' with A_1, the type inferred for M in a context where Δ' is added to the rigid variables. Thus, if M is a guarded value, we expect A_1 to be a guarded value, too. Otherwise, when Δ' is empty, we unify A and the inferred type for M directly. The algorithm then performs an escape check. Note that we may alternatively define $(\Theta_2, \theta_2') = \mathcal{U}(\Delta, \Theta_1, A, \forall\Delta'.A_1)$.

We again consider examples.

$$\begin{array}{llll}
\mathsf{infer}(\cdot, \cdot \quad , \cdot \ , \textbf{let}\ (h_1 : \forall b.b \to b) = \lambda x.x \quad \textbf{in}\ \lceil h_1 \rceil) & = & (\cdot, [], \forall b.b \to b) & (4.6) \\
\mathsf{infer}(\cdot, \cdot \quad , \Gamma, \textbf{let}\ (h_2 : \forall b.b \to b) = \mathsf{id}\ \lceil \mathsf{id} \rceil\ \textbf{in}\ \lceil h_2 \rceil) & = & (\cdot, [], \forall b.b \to b) & (4.7) \\
\mathsf{infer}(\cdot, (a_1 : \bullet), \Gamma, \textbf{let}\ (h_3 : \forall b.b \to b) = f \quad \textbf{in}\ \lceil h_3 \rceil) & & \textit{fails} & (4.8)
\end{array}$$

$$\text{where } \Gamma = (f : a_1 \to a_1,\ \mathsf{id} : \forall c.c \to c)$$

In the first example, we have $\Delta' = b$ and obtain the result shown in (4.1) yet again when inferring the type of the let-bound term. We then perform $\mathcal{U}(b, (a : \bullet), b \to b, a \to a$, yielding $(\cdot, [a \mapsto b])$, leading to the result shown in (4.6).

In the second example, we have $\Theta_1 = \cdot$, $A_1 = \forall c.c \to c$ and $\theta_1 = [a_1 \mapsto A_1, a_2 \mapsto A_1]$. However, split sets $\Delta' = \cdot$ and we perform $\mathcal{U}(\cdot, \Theta_1, \forall b.b \to b, \forall c.c \to c)$.

The third example fails due to a_1 being substituted with b and therefore violating the escape check.

In conclusion, we observe that the function infer in Figure 4.5 is a modest extension of Algorithm W. This is no surprise, since a design goal of FreezeML was specifically to follow the HM system, and hence Algorithm W, regarding when generalisation and instantiation occurs. The key additions are the usage of restriction contexts, indicating what unification variables may or may not stand for polymorphic types. Unification has been extended to arbitrarily quantified types, checking and updating restrictions as needed. As a result, a key deviation from the HM system, namely the *polymorphic* instantiation of all quantifiers of a term variables' types, merely requires adding freshly introduced type variables with restriction $\star$ instead of $\bullet$ to the restriction context when handling term variables. While the restriction to use principal types for let-bound terms affects the overall system's metatheory, this need not be reflected in the inference algorithm.

4.3.1 Properties of the inference algorithm

We first observe that the inference algorithm terminates.

Theorem 4.4 (Termination of infer)**.**
The function infer *terminates on all inputs.*

Proof. All recursive invocations of infer happen on strict subterms of the input term. The inference algorithm uses the function $\mathcal{U}$, which is also guaranteed to terminate (see Theorem 4.1). □

We use the extended FreezeML typing relation $\vdash^{E}$, introduced in Section 4.1.2, to express the correctness properties of the inference algorithm. We first show that infer is sound, meaning that the type A inferred for a term M is indeed a valid type of M in the context obtained from using the returned restriction context Θ' and substitution θ.

Theorem 4.5 (Soundness of infer)**.**
Let $(\Delta^{\bullet}, \Theta) \vdash \Gamma\, \mathbf{ok}$ *and* $\Delta; \Gamma \vdash M\, \mathbf{ok}$*. If* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ *then* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $(\Delta^{\bullet}, \Theta'); \theta(\Gamma) \vdash^{E} M : A$*.*

Proof. By induction on structure of M, details in Appendix A.4. □

Further, we show that the inference algorithm is complete and most general. If a type A can be given to a term M by applying a substitution to Γ, then type inference on the term succeeds. In addition, the returned substitution θ'' can be refined using a substitution θ' to obtain the original substitution and θ'' can be used to turn the inferred type into A.

Theorem 4.6 (Completeness and generality of infer)**.**
Let $\Delta; \Gamma \vdash M\, \mathbf{ok}$ *and* $(\Delta^{\bullet}, \Theta) \vdash \Gamma\, \mathbf{ok}$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$*. If* $(\Delta^{\bullet}, \Theta'); \theta(\Gamma) \vdash^{E} M : A$*, then there exist* $\Theta'', \theta', \theta'', A'$ *such that* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta'', \theta'', A')$ *and* $\Delta \vdash \theta' : \Theta'' \Rightarrow \Theta'$ *and* $\theta = \theta' \circ \theta''$ *and* $\theta'(A') = A$*.*

Proof. By induction on structure of M, details in Appendix A.4. □

Finally, we relate the output (Θ, θ, A) of the inference algorithm to the original FreezeML typing relation. Here, we consider the case where Γ only contains variables from Δ and the restriction context is empty, which is how the algorithm would typically be used to perform overall type inference. As a result, this immediately yields that the returned substitution θ is empty. In contrast, Theorems 4.5 and 4.6 are suitable for inductive reasoning for inference on subterms.

First, we observe that the type A returned by the algorithm is principal in the sense of the principal predicate (which was defined in terms of the original typing relation). Secondly, the types of M in term context Γ and any type contexts that extends Δ are exactly those that are obtained from A using a well formed substitution with domain Θ (i.e., the restriction context returned by the inference algorithm).

Theorem 4.7.
Let $\Delta \vdash \Gamma\ \mathbf{ok}$ *and* $\Delta;\Gamma \vdash M\ \mathbf{ok}$. *If* $\mathsf{infer}(\Delta,\cdot,\Gamma,A) = (\Theta,\theta,A)$ *then the following holds.*

1. *We have* $\mathsf{principal}(\Delta,\Gamma,M,\mathsf{ftv}(\Theta),A)$.
2. *For all* Δ',A' *we have* $(\Delta,\Delta'),\Gamma \vdash M : A'$ *iff there exists* θ' *such that* $\Delta \vdash \theta' : \Theta \Rightarrow \Delta'$ *and* $\theta(A) = A'$.

Proof. This is a simplified version of Lemma A.13 in Appendix A.4. □

Theorem 4.8 (Existence of principal types)**.**
If M is well typed in FreezeML in contexts Δ and Γ then there exist Δ' and A such that $\mathsf{principal}(\Delta,\Gamma,M,\Delta',A)$.

Proof. By Lemma 4.2 we have that M is also well typed using the extended FreezeML typing relation. Theorem 4.6 then guarantees that infer succeeds, which means that the existence of the principal type follows from Theorem 4.7. □

Chapter 5

Constraint-based type inference for FreezeML

The type inference algorithm presented in the previous chapter is based on Algorithm W, the original type inference algorithm devised for the HM system. This makes Algorithm W a natural basis for a type inference algorithm for an extension of ML, such as ours, and has been adapted by other extensions providing first-class polymorphism [89, 117, 58].

However, in the decades following the inception of the HM system and Algorithm W, type systems and inference systems utilising *constraints* have gained popularity. In the late 1980s and early 1990s, a multitude of systems were developed that employ some notion of constraints, in order to implement individual language features. Examples of such systems include work on type classes and overloading [120, 47, 87, 84] as well as record types [121, 98] and subtyping [1, 112, 17].

Subsequently, approaches were developed that generalise the treatment of constraints, rather than be tailored to a specific use case or constraint language. Jones [44] presented a variation of ML with *qualified types*, and showed how his general approach can be instantiated to implement the language features listed above. In his system, type inference and qualified types are largely orthogonal: During type inference, constraints are treated in a purely syntactic manner and mostly just accumulated.

Later work by Odersky et al. [86] on the HM(X) framework had a similar goal: The framework is parameterised in a concrete constraint domain X, similarly to Jones' system. Odersky et al. then present a generic algorithm that performs type inference, independent from the constraint domain X. The framework then guarantees that the resulting type systems is sound as well as the soundness and completeness of the inference algorithm. This means that users of the framework merely have to show that their choice of X satisfies the formal requirements imposed by the framework to obtain the meta-theoretic properties established by the framework.

The inference algorithm shown by Odersky et al. works by generating and solving appropriate constraints. Thus, rather than merely performing inference *for* a constraint-based type system, constraints are at the heart of their inference algorithm, performing inference *with* constraints. While their generic algorithm is oblivious of the constraint domain X, it still depends on domain-specific functionality, such as a decidable decision procedure for *normalising* constraints, which involves determining their satisfiability.

Later, Pottier and Rémy [97] presented a variant of the HM(X) framework that refines the two-stage approach of constraint generation and constraint solving. In contrast to earlier work, their approach allows complete separation between the two phases: Instead of interleaving constraint generation and constraint solving for subterms as part of an overall type inference algorithm, Pottier and Rémy show how a term can be translated to a constraint as a whole. Then, the resulting constraint is guaranteed to be satisfiable if and only if the original term was well typed, thus reducing the type inference problem to a constraint satisfiability problem. Further, the types of the term can be obtained from the solutions of the constraint. This approach offers a greater separation of concerns: Once a term has been translated to a constraint, the original term plays no further rule during the next stage of type inference, namely constraint solving. While Pottier and Rémy discuss how their work can be adapted to handle some extensions of the HM system, their solver itself is not generic in the sense of being independent of the constraint domain.

In practice, the usefulness of a constraint-based inference approach has been demonstrated by the GHC Haskell compiler, well known for its multitude of type system extensions, which relies on constraint-based type inference [72, 90].

In this chapter, we present a constraint-based type inference system for FreezeML that follows the spirit of Pottier and Rémy's approach with two independent stages: We show a translation of FreezeML terms to constraints, and provide a solver for our constraint language. While this inference process is not currently parameterised over a constraint domain, we conclude the chapter with a comparison with HM(X).

5.1 Constraint language

In this section, we present the constraint language used for our constraint-based inference approach. The key mechanism used by Pottier and Rémy [97] to reduce the type inference problem for a whole program to a constraint solving problem is the inclusion of *term* variables in their constraint language. It comprises constraint forms that bind term variables and forms that use them.

We use the same approach for our system. The resulting language of constraints C is defined as follows.

$$\begin{aligned} C ::= {} & \mathsf{true} \mid C_1 \wedge C_2 \mid A \sim B \mid \lceil x : A \rceil \mid x \preceq A \mid \forall a.C \mid \exists a.C \mid \mathsf{mono}(a) \\ \mid {} & \mathbf{def}\,(x : A)\ \mathbf{in}\ C \mid \mathbf{let}_\star\, x = \sqcap a.C_1\ \mathbf{in}\ C_2 \mid \mathbf{let}_\bullet\, x = \sqcap a.C_1\ \mathbf{in}\ C_2 \end{aligned}$$

As usual, our language contains tautological constraints (true) and logical conjunctions ($C_1 \wedge C_2$). Constraints $A \sim B$ assert that A and B are equal, meaning that after substituting the flexible type variables therein, both types are syntactically equivalent modulo alpha-conversion. The constraint forms $\forall a.C$ and $\exists a.C$ bind type variables that are universally and existentially quantified in C, respectively. Constraints $\lceil x : A \rceil$ and $x \preceq A$ reason about the type of the term variable x. Any solution of a constraints $\mathsf{mono}(a)$ must choose a monomorphic type for a.

The remaining three constraint forms, namely def constraints and two forms of let constraints, bind term variables in terms. We consider constraints equivalent modulo alpha-conversion of both the type and term variables bound within a constraint.

We define *interpretations* as quadruples $\Delta;\Xi;\Gamma;\delta$, consisting of a *rigid type context* Δ, a *flexible type context* Ξ, a term context Γ, and a substitution δ. We return to the role of each component of an interpretation shortly. We then give a semantics of constraints by defining when a constraint is satisfied by an interpretation, denoted $\Delta;\Xi;\Gamma;\delta \vdash C$, making the interpretation a *model* or *solution* of C.

Both contexts Δ and Ξ in an interpretation follow the same grammar, which is in turn equivalent to the grammar of type contexts introduced before (see Figure 3.2 on page 41). The only difference is that the former contains type variables treated as rigid, whereas the latter contains type variables considered flexible. Thus, the role of Ξ is similar to that of restriction contexts introduced in Section 4.1, in the sense of indicating which flexible type variables are in scope, but without carrying information about the restrictions on these type variables. Term contexts Γ are defined and used as before. Instantiations δ are also formally defined as before, used here to map all flexible type variables from Ξ to types well formed under Δ.

Given fixed Δ, Ξ and Γ, our constraint language is designed with the goal that every constraint satisfiable under these contexts has a principal/most general model using these contexts. We will formalise the notion of most general solutions in Section 5.1.2.

We say that an interpretation consisting of $\Delta, \Xi, \Gamma, \delta$ is well formed if we have $\Delta \vdash \Gamma\ \mathbf{ok}$ and $\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$. Note that this implies $\Delta \# \Xi$ and that both Γ and the codomain of δ only contain rigid variables from Δ. As usual in this work, we make it an implicit precondition of the constraint satisfaction judgements that the used interpretation is well formed.

The rules of the constraint satisfaction relation are shown in Figure 5.1 on the next page. The rules for true and conjunctions are straightforward. To satisfy a constraint $A \sim B$, both A and B must be well formed types under the combined restriction context $(\Delta^\bullet, \Xi)$, using the judgement defined in Figure 4.1 on page 72.[1] We then require that the interpretation δ of flexible variables indeed makes A and B equal.

Constraints of the form $\lceil x : A \rceil$ assert that the type of x *is* A. As A may contain flexible variables, the corresponding rule SEM-FREEZE applies δ to it before comparing the result with the type for x in Γ. Conversely, constraints of the form $x \preceq A$ assert that all toplevel quantifiers in the type of x may be *instantiated* to yield A. The resulting rule again needs to apply δ to A. Note that usually, in order to be the result of instantiation, $\delta(A)$ has to be a guarded type, meaning that it has no toplevel quantifiers. The only exception are constraints bot $\preceq A$, where bot $: \forall a_1, \ldots, a_n . a_i \in \Gamma$.

The rule for constraints $\forall a.C$ simply adds a to the rigid type context Δ, meaning that the satisfaction of C is independent of any further information about a. Conversely, the rule SEM-EXISTS establishes that a constraint $\exists a.C$ is satisfiable if a type A can be chosen for a such that C is satisfied by the extended interpretation. A constraint $\mathsf{mono}(a)$ is satisfied if and only if a is either a rigid type variable or one interpreted with a monomorphic type.

We now consider the constraint forms binding term variables.

5.1.1 Definition constraints

A definition constraint (or "def" constraint, directly reflecting the syntax) $\mathbf{def}\ (x : A)\ \mathbf{in}\ C$ binds the term variable x with type A in the sub-constraint C. Here, unlike for instance annotated lambda terms in FreezeML, A may contain flexible type vari-

[1]Note that the restrictions on the type variables in scope are not relevant for the satisfaction of these well-formedness premises. This means that we could equivalently require $(\Delta, \mathsf{ftv}(\Theta)) \vdash A$ and $(\Delta, \mathsf{ftv}(\Theta)) \vdash B$, using the type well-formedness judgement defined in Figure 4.1 on page 72.

$\boxed{\Delta; \Xi; \Gamma; \delta \vdash C}$

$$\textsc{Sem-True}\quad \frac{}{\Delta; \Xi; \Gamma; \delta \vdash \mathsf{true}}$$

$$\textsc{Sem-And}\quad \frac{\Delta; \Xi; \Gamma; \delta \vdash C_1 \qquad \Delta; \Xi; \Gamma; \delta \vdash C_2}{\Delta; \Xi; \Gamma; \delta \vdash C_1 \wedge C_2}$$

$$\textsc{Sem-Equiv}\quad \frac{\delta(A) = \delta(B) \qquad (\Delta, \Xi) \vdash A \ \mathbf{ok} \qquad (\Delta, \Xi) \vdash B \ \mathbf{ok}}{\Delta; \Xi; \Gamma; \delta \vdash A \sim B}$$

$$\textsc{Sem-Freeze}\quad \frac{\Gamma(x) = \delta(A)}{\Delta; \Xi; \Gamma; \delta \vdash \lceil x : A \rceil}$$

$$\textsc{Sem-Instance}\quad \frac{\Gamma(x) = \forall \overline{a}.H \qquad \Delta \vdash \delta' : \overline{a} \Rightarrow_{\star} \cdot \qquad \delta'(H) = \delta(A)}{\Delta; \Xi; \Gamma; \delta \vdash x \preceq A}$$

$$\textsc{Sem-Forall}\quad \frac{(\Delta, a); \Xi; \Gamma; \delta \vdash C}{\Delta; \Xi; \Gamma; \delta \vdash \forall a.C}$$

$$\textsc{Sem-Exists}\quad \frac{\Delta; (\Xi, a); \Gamma; \delta[a \mapsto A] \vdash C}{\Delta; \Xi; \Gamma; \delta \vdash \exists a.C}$$

$$\textsc{Sem-Mono}\quad \frac{\Delta \vdash_{\bullet} \delta(a) \ \mathbf{ok}}{\Delta; \Xi; \Gamma; \delta \vdash \mathsf{mono}(a)}$$

$$\textsc{Sem-Def}\quad \frac{\Delta; \Xi; (\Gamma, x : \delta(A)); \delta \vdash C \qquad \text{for all } a \in \mathsf{ftv}(A) : \ \Delta; \Xi; \Gamma; \delta \vdash \mathsf{mono}(a)}{\Delta; \Xi; \Gamma; \delta \vdash \mathbf{def}\ (x : A)\ \mathbf{in}\ C}$$

$$\textsc{Sem-LetGen}\quad \frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, C_1, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}}) \\ \Delta_{\mathrm{o}} = \mathsf{ftv}(\delta_{\mathrm{m}}(\Xi)) - \Delta \qquad \overline{b} = \mathsf{ftv}(\delta_{\mathrm{m}}(a)) - \Delta, \Delta_{\mathrm{o}} \\ \Delta \vdash \delta' : \Delta_{\mathrm{o}} \Rightarrow_{\bullet} \cdot \qquad A = \delta'(\delta_{\mathrm{m}}(a)) \\ (\Delta, \overline{b}); (\Xi, a); \Gamma; \delta[a \mapsto A] \vdash C_1 \qquad \Delta; \Xi; (\Gamma, x : \forall \overline{b}.A); \delta \vdash C_2\end{array}}{\Delta; \Xi; \Gamma; \delta \vdash \mathbf{let}_{\star}\ x = \sqcap a.C_1\ \mathbf{in}\ C_2}$$

$$\textsc{Sem-LetNonGen}\quad \frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, C_1, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}}) \\ \Delta \vdash \delta' : \Delta_{\mathrm{m}} \Rightarrow_{\bullet} \cdot \qquad A = \delta'(\delta_{\mathrm{m}}(a)) \\ \Delta; (\Xi, a); \Gamma; \delta[a \mapsto A] \vdash C_1 \qquad \Delta; \Xi; (\Gamma, x : A); \delta \vdash C_2\end{array}}{\Delta; \Xi; \Gamma; \delta \vdash \mathbf{let}_{\bullet}\ x = \sqcap a.C_1\ \mathbf{in}\ C_2}$$

Figure 5.1: Satisfiability of constraints

ables. However, the semantics of def constraints in Figure 5.1 impose a crucial side condition: All flexible type variables in A must be interpreted with monomorphic types by any model of the def constraint.

Intuitively, this preserves the invariant that in FreezeML, only fully determined – rather than guessed – polymorphism may appear in the types of term variables and can thus be instantiated. As an example of the concrete need for the restriction to monomorphic solutions, we consider the constraint C_{m} defined as $\mathbf{def}\ (x : a)\ \mathbf{in}\ x \preceq \mathsf{Int} \to \mathsf{Int} \wedge c \sim (a \to \mathsf{Int})$. If we allowed polymorphic solutions for a, the constraint would be satisfied by interpretations choosing $c \mapsto (\forall b.b \to b) \to \mathsf{Int})$ or $c \mapsto ((\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int})$. However, there exists no more general type that may be chosen for c and hence no most general solution of the constraint would exist.

Note that the requirement for free type variables in the annotation A to be instantiated monomorphically does not preclude A being polymorphic. The constraint $\mathbf{def}\ (\forall a.a \to b)\ \mathbf{in}\ \mathsf{true}$ is satisfiable, its solutions are all interpretations that map b to an arbitrary monomorphic type.

Finally, we observe that it is not possible to avoid the monomorphism requirement in the semantics of definition constraints by using appropriate mono constraints. The constraint $\mathsf{mono}(a) \wedge C_{\mathrm{m}}$ is logically equivalent to the constraint C_{m} itself, which we defined earlier on this page, even if we were to drop the monomorphism requirement from the semantics of def constraint. However, as discussed before, its sub-constraint C_{m} would still not have a most general solution. This means that while we can use mono constraints to suppress polymorphic solutions, removing the monomorphism requirement from definition constraints would immediately lead to the existence of satisfiable constraints without most general solutions.

5.1.2 Generalising let constraints

We now discuss the two closely related forms of let constraints. We refer to $\mathbf{let}_{\star}\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ as a *generalising* let constraint, whereas $\mathbf{let}_{\bullet}\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ is a *non-generalising* one. Both forms of constraints bind the type variable a in C_1 and the term variable x in C_2. In both cases, only the *principal* solution for a in C_1 will be used to determined the type for x. Thus, we can consider let constraints as performing a special kind of existential quantification of the type variable a in C_1, that only permits principal solutions. The details of how the type for x is determined from the solution for a differ between the two forms of let constraints. We first consider generalising let constraints, where the type of x is the result of generalising the solution for a. We return to non-generalising let constraints afterwards.

Before discussing the exact semantics of polymorphic let constraints, we make some observations that led to the overall semantics of let constraints.

Need to generalise principal solution As mentioned, a constraint $\mathbf{let}_\star\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ existentially quantifies a as a flexible type variable in C_1, but in such a way that we must choose the most general solution for a that satisfies C_1. This is similar to the principality condition imposed on FreezeML let *terms* and necessitated by similar reasons. As an example, consider the following constraint.

$$\begin{array}{l} \mathbf{let}_\star\ x = \sqcap a.\overbrace{\exists b_1.a \sim (b_1 \to b_1)}^{=:\,C'}\ \mathbf{in} \\ x \preceq (\mathsf{Int} \to \mathsf{Int}) \wedge \underbrace{\exists b_2.\lceil x : b_2 \rceil \wedge c \sim (b_2 \to \mathsf{Int})}_{=:\,C''} \end{array} \tag{5.1}$$

The only free type variable of the overall constraint is c, but models of the constraint may choose both $c \mapsto (\forall a.a \to a) \to \mathsf{Int}$ and $c \mapsto (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$, with no more general solution existing. This closely resembles the example illustrating the need for the monomorphism requirement on definition constraints shown in Section 5.1.1 and the examples motivating the principality condition on let terms in Section 3.2.1.1. Note that due to the types $\mathsf{Int} \to \mathsf{Int}$ and $\forall a.a \to a$ being incomparable in FreezeML, the previous example would already exhibit the lack of principal solutions if we took C'' to be $\lceil x : c \rceil$ instead. To have any hope of ensuring the existence of principal solutions for our constraint language, we must therefore require that let constraints $\mathbf{let}_\star\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ use the principal solution for a in C_1 as the type to be generalised to obtain the type for x.

The rule SEM-LETGEN in Figure 5.1 formalises this using the predicate $\mathsf{mostgen}$, defined in Figure 5.2 on the following page, which closely resembles the $\mathsf{principal}$ predicate used in the typing rule for let terms. We have that $\mathsf{mostgen}(\Delta, \Xi, \Gamma, C, \Delta_\mathrm{m}, \delta_\mathrm{m})$ holds if $(\Delta, \Delta_\mathrm{m}), \Xi, \Gamma, \delta_\mathrm{m}$ is a model of C and every other model based on Δ,Ξ,Γ can be obtained by refining δ_m by composition. Here, the variables in Δ_m can be considered to be rigid placeholders for variables left undetermined by C. Note that we effectively consider Δ, Ξ, and Γ as inputs of the relation, meaning that we only considering most general interpretations of C among those interpretations using Δ, Ξ, and Γ. In particular, the term context Γ is fixed. This resembles how $\mathsf{principal}$ is used to reason about

$\boxed{\mathsf{mostgen}(\Delta, \Gamma, C, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})}$

$$\begin{aligned}
&\mathsf{mostgen}(\Delta, \Xi, \Gamma, C, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}}) = \\
&\quad (\Delta, \Delta_{\mathrm{m}}); \Xi; \Gamma; \delta_{\mathrm{m}} \vdash C \text{ and} \\
&\quad (\text{for all } \Delta'', \delta'' \mid \text{if } (\Delta, \Delta''); \Xi; \Gamma; \delta'' \vdash C \\
&\qquad\qquad \text{then there exists } \delta' \text{ such that} \\
&\qquad\qquad \Delta \vdash \delta' : \Delta_{\mathrm{m}} \Rightarrow_{\star} \Delta'' \text{ and } \delta' \circ \delta_{\mathrm{m}} = \delta'')
\end{aligned}$$

Figure 5.2: Definition of mostgen

the principal *types* of a terms, as opposed to principal *typings*, a distinction discussed in Section 3.4.

Returning to the example constraint in (5.1) and its sub-constraint C', let a and Δ and Ξ be pairwise disjoint. We observe that $\mathsf{mostgen}(\Delta, (\Xi, a), \cdot, C', b, [a \mapsto (b \to b)])$ then holds for all $b \# (\Delta, \Xi, a)$.

The rule SEM-LETGEN uses the similar premise $\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, C_1, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$. It then determines the two subsets Δ_{o} and $\overline{b}$ of Δ_{m}. The sequence Δ_{o} denotes those undetermined variables that are related to the set Ξ of flexible variables from the *outer* context in the sense that they appear in $\mathsf{ftv}(\delta_{\mathrm{m}}(\Xi))$. As usual, such variables must not be generalised. Note that we explicitly exclude the mapping for a when determining Δ_{o} as a is not part of the surrounding context from the perspective of the let constraint under consideration.

In turn, $\overline{b}$ contains the type variables appearing in the most general solution for a, except those from Δ_{o} and Δ. The variables in $\overline{b}$ are then generalised when determining the type of x in SEM-LETGEN. Note that similarly to the relation principal, the definition of mostgen allows superfluous variables to appear in Δ_{m}, in the sense that they may not appear in the codomain of δ_{m}. If we disregard such superfluous variables and assume $\Delta_{\mathrm{m}} \subseteq \mathsf{ftv}(\delta_{\mathrm{m}})$, then Δ_{o} and $\overline{b}$ form a partitioning of Δ_{m}.

The mostgen premise of SEM-LETGEN results in constraint satisfaction judgements appearing in a negative position in the rule. This raises the same questions regarding the well-definedness of the constraint satisfaction relation as for the typing relation of FreezeML terms. We observe that as for let terms, the mostgen condition on let constraints only involves sub-terms/sub-constraints. We may therefore perform the same line of reasoning as in Section 3.4 to show the well-definedness of the constraint satisfaction relation, but eschew formalising this here.

Safe interaction with surrounding context We have just outlined how the rule SEM-LETGEN in Figure 5.1 splits the type variables in Δ_{m} into those appearing in the outer context, named Δ_{o}, and the remaining free type variables of $\delta_{\mathrm{m}}(a)$ (i.e., the solution for a in the most general model of the first subconstraint C_1 of the let constraint). The latter sequence is named $\overline{b}$ by the rule, and corresponds to the variables that are generalised when constructing the type for x in C_2.

We now observe that one more condition is required to ensure that no undetermined polymorphism enters the term context. As an example, we consider the following constraint.

$$\textbf{let}_\star\, x = \sqcap a.a \sim b \textbf{ in } x \preceq (\mathsf{Int} \to \mathsf{Int}) \tag{5.2}$$

For the principality condition in SEM-LETGEN, we observe that $\mathsf{mostgen}(\cdot,(a,b),\cdot,$ $a \sim b, c, \delta')$ holds, where $\delta'(a) = \delta'(b) = c$. As a result, we have $\overline{b} = \cdot$, meaning that no type variables will be generalised to obtain the type for x. Naively, the rule may then permit whatever type the surrounding context picked for b of x. However, the constraint (5.2) would then have multiple models, choosing types such as $\forall c.c$, $\forall c.c \to c$, or $(\mathsf{Int} \to \mathsf{Int})$ for b. All three types are incomparable in FreezeML, and no most general model of the constraint would exist. This is due to the fact that the different choices of potentially polymorphic types for b leak into the term context and subject it to instantiation.

To this end, we consider the free type variables of $\delta_{\mathrm{m}}(a)$ that appear in Δ_{o}, and are therefore *not* generalised. The rule SEM-LETGEN then imposes that the type chosen for x must result from monomorphically instantiating the variables in Δ_{o}. In other words, this means that the type A of x is obtained from $\delta_{\mathrm{m}}(a)$, the most general type for a permitted by the constraints in C_1, by generalising the variables in $\overline{b}$ *and* monomorphically instantiating the non-generalisable variables from Δ_{o}. For the example constraint (5.2), this leads to the *only* choice for b being the type $\mathsf{Int} \to \mathsf{Int}$.

Finally, we observe that not all choices for A in SEM-LETGEN may be compatible with the ambient instantiation δ. To this end, consider the following variation of (5.2).

$$b \sim \mathsf{Bool} \wedge \textbf{let}_\star\, x = \sqcap a.a \sim b \textbf{ in } x \preceq (\mathsf{Int} \to \mathsf{Int}) \tag{5.3}$$

For the let constraint, we still have $\mathsf{mostgen}(\cdot,(a,b),\cdot,a \sim b, c, \delta')$, using the same δ' as before, which simply maps a and b to the same fresh variable c. However, choosing A to be $\mathsf{Int} \to \mathsf{Int}$ is incompatible with the choice for b by the ambient instantiation δ, which must map b to Bool, due to the newly added constraint $b \sim \mathsf{Bool}$. To ensure that the choice of A is compatible with the ambient context, the rule SEM-LETGEN uses the premise $\Delta;\Xi;\Gamma;\delta[a \mapsto A] \vdash C_1$. Note that here, the ambient instantiation δ is used, rather than δ_m. While this premise holds when applying the rule to the let sub-constraint of (5.2), it fails in the case of (5.3), correctly determining the latter to be unsatisfiable.

In summary, we observe that the semantics of polymorphic let constraints implement two ideas that have previously occurred in FreezeML: A principality condition, here in the form of requiring the usage of a most general solution, and a monomorphism requirement that avoids undetermined polymorphism in the term context. Formalising the interplay between the two conditions then leads to the somewhat intricate rule SEM-LETGEN in Figure 5.1.

No other constraint based type inference system is known to the author which employs a principality condition in the semantics of its constraint language.

5.1.3 Non-generalising let constraints

As we have just seen, the rule SEM-LETGEN in Figure 5.1 determines the two subsets Δ_o and $\overline{b}$ of Δ_m. The type of x is then obtained by generalising the variables $\overline{b}$ and monomorphically instantiating the variables Δ_o in a way that is compatible with the ambient instantiation.

As their name suggests, non-generalising let constraints do not perform generalisation. Instead, they monomorphically instantiate the variables in $\overline{b}$, in addition to the variables in Δ_o.

We may therefore define the semantics of non-generalising let constraints as follows, with the only two changes as compared to SEM-LETGEN highlighted.

SEM-LETNONGEN-ALT

$$\frac{\begin{array}{c} \mathsf{mostgen}(\Delta,(\Xi,a),\Gamma,C_1,\Delta_\mathrm{m},\delta_\mathrm{m}) \\ \Delta_\mathrm{o} = \mathsf{ftv}(\delta_\mathrm{m}(\Xi)) - \Delta \qquad \overline{b} = \mathsf{ftv}(\delta_\mathrm{m}(a)) - \Delta,\Delta_\mathrm{m} \\ \Delta \vdash \delta' : \boxed{(\Delta_\mathrm{o},\overline{b})} \Rightarrow_\bullet \cdot \qquad A = \delta'(\delta_\mathrm{m}(a)) \\ (\Delta,\overline{b});(\Xi,a);\Gamma;\delta[a \mapsto A] \vdash C_1 \qquad \Delta;\Xi;(\Gamma,x:\boxed{A});\delta \vdash C_2 \end{array}}{\Delta;\Xi;\Gamma;\delta \vdash \mathbf{let}_\star\ x = \sqcap a.C_1\ \mathbf{in}\ C_2}$$

Thus, the domain of the instantiation δ' is extended by $\overline{b}$ and no generalisation is performed to obtain the type of x from A, instead using A directly.

As discussed in Section 5.1.2, the type variables in Δ_m not appearing in Δ_o or $\overline{b}$

are superfluous. The rule SEM-LETNONGEN in Figure 5.1 is a simplification of the rule SEM-LETNONGEN-ALT, utilising this fact. In SEM-LETNONGEN, all variables in Δ_{m} are monomorphically instantiated by δ'.

As an example, consider the following variation of (5.1) from page 99, where the only change is using a non-generalising let constraint instead.

$$\begin{gathered}\mathbf{let}_{\bullet}\ x = \sqcap a.\exists b_1.a \sim (b_1 \to b_1)\ \mathbf{in} \\ x \preceq (\mathsf{Int} \to \mathsf{Int}) \wedge \exists b_2.\lceil x : b_2 \rceil \wedge c \sim (b_2 \to \mathsf{Int})\end{gathered}$$

This constraint is still satisfiable, but all models must choose the type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ for c, as opposed to $(\forall a.a \to a) \to \mathsf{Int}$ in the original version (5.1).

5.1.4 Well-formedness of constraints

After defining when an interpretation $\Delta, \Xi, \Gamma, \delta$ satisfies a constraint, we now introduce the notion of constraints being well formed. The relation largely exists for technical reasons, to formalise what type and term variables may appear freely in a constraint during proofs.

Constraint well-formedness judgements are written $\Delta; \Xi; \Gamma \vdash C\ \mathbf{ok}$, stating that all type variables appearing in C are either bound appropriately within C or appear in (Δ, Ξ). Conversely, term variables must have been bound within C or appear in Γ.

The rules of the relation are shown in Figure 5.3 on the following page. Note that the relation is independent from an instantiation δ. Further, while a term context Γ appears in well-formedness judgements, it is only used to track what term variables are in scope, their types are ignored. We used term contexts Γ in a similar fashion in well-formedness judgements $\Delta; \Gamma \vdash M\ \mathbf{ok}$ for FreezeML terms, defined in Section 3.2. As a result, in the rule for let constraints, an arbitrary type A may be picked for x.

Finally, we observe that all satisfiable constraints are well formed under their models. This is in contrast to FreezeML typing judgements, where we *required* well-formedness of the term as a precondition in order to achieve greater separation between the typing rules and the scoping rules for type variables in annotations.

Lemma 5.1 (Constraint satisfaction implies well-formedness)**.**
If $\Delta; \Xi; \Gamma; \delta \vdash C$ *holds, then* $\Delta; \Xi; \Gamma \vdash C\ \mathbf{ok}$ *holds.*

Proof. By structural induction on C, observing that the premises of each rule explicitly or via implicit preconditions establish the necessary well-formedness conditions. □

$\boxed{\Delta;\Xi;\Gamma \vdash C \ \mathbf{ok}}$

$$\frac{}{\Delta;\Xi;\Gamma \vdash \mathsf{true}\ \mathbf{ok}} \qquad \frac{a \in (\Delta,\Xi)}{\Delta;\Xi;\Gamma \vdash \mathsf{mono}(a)\ \mathbf{ok}}$$

$$\frac{\Delta;\Xi;\Gamma \vdash C_1\ \mathbf{ok} \qquad \Delta;\Xi;\Gamma \vdash C_2\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash C_1 \wedge C_2\ \mathbf{ok}} \qquad \frac{\Delta;(\Xi,a);\Gamma \vdash C\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \exists a.C\ \mathbf{ok}} \qquad \frac{(\Delta,a);\Xi;\Gamma \vdash C\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \forall a.C\ \mathbf{ok}}$$

$$\frac{(\Delta,\Xi) \vdash A\ \mathbf{ok} \qquad (\Delta,\Xi) \vdash B\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash A \sim B\ \mathbf{ok}} \qquad \frac{x \in \Gamma \qquad (\Delta,\Xi) \vdash A\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash x \preceq A\ \mathbf{ok}}$$

$$\frac{x \in \Gamma \qquad (\Delta,\Xi) \vdash A\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \lceil x : A \rceil\ \mathbf{ok}} \qquad \frac{(\Delta,\Xi) \vdash A\ \mathbf{ok} \qquad \Delta;\Xi;(\Gamma, x:A) \vdash C\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \mathbf{def}\ (x : A)\ \mathbf{in}\ C\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\Delta;(\Xi,a);\Gamma \vdash C_1\ \mathbf{ok} \\ \Delta;\Xi;(\Gamma, x:A) \vdash C_2\ \mathbf{ok}\end{array}}{\Delta;\Xi;\Gamma \vdash \mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2\ \mathbf{ok}}$$

Figure 5.3: Well-formedness of constraints

5.2 Constraint generation

We now present a translation from FreezeML terms to our constraint language, such that the resulting constraint is satisfiable if and only if the term is well typed. Further, we show how the types of the well typed term can be obtained from the models of the constraint.

The translation is defined as a function $[\![M : A]\!]$, where M is the term to translate and A is the expected type of M. We assume that M is well formed under the given contexts Δ and Γ (i.e., we have $\Delta;\Gamma \vdash M\ \mathbf{ok}$).

In the simplest use case, the only free variables of A are rigid variables from Δ. A constraint such as $[\![\lambda x.x : \mathsf{Int} \to \mathsf{Int}]\!]$ then asserts that the given term has the given type, effectively allowing type *checking*.

However, we additionally permit A to contain *flexible* variables from some Ξ (i.e., $(\Delta,\Xi) \vdash A\ \mathbf{ok}$) in order to perform type *inference*. In general, the free type variables of the constraint $[\![M : A]\!]$ are exactly those of A. The models of the constraint $[\![M : A]\!]$ then interpret the free flexible variables of A such that A *becomes* a valid type of M.

This means that despite the expected type A appearing as an input of the translation, we may still perform type inference: We may simply pick a single variable $a \in \Xi$ as A. For example, the models of $[\![\lambda x.x : a]\!]$ are exactly those that interpret a with a valid type of $\lambda x.x$. We can therefore use constraints both to perform type checking and type inference. Since A is an arbitrary type, we can also generate constraints that simultaneously make assertions about part of a term's type, while inferring other parts of the type, as in the constraint $[\![\lambda xy.x+y : \mathsf{Int} \to a \to b]\!]$.

The translation function is shown in Figure 5.4. We assume that all variables a_i

$$\begin{array}{rcll}
[\![\lceil x \rceil : A]\!] & = & \lceil x : A \rceil & \\
[\![x : A]\!] & = & x \preceq A & \\
[\![M N : A]\!] & = & \exists a_1.([\![M : a_1 \to A]\!] \wedge [\![N : a_1]\!]) & \\
[\![\lambda x.M : A]\!] & = & \exists a_1, a_2.(a_1 \to a_2 \sim A \wedge \mathbf{def}\,(x : a_1)\ \mathbf{in}\ [\![M : a_2]\!]) & \\
[\![\lambda (x : B).M : A]\!] & = & \exists a_1.B \to a_1 \sim A \wedge \mathbf{def}\,(x : B)\ \mathbf{in}\ [\![N : a_1]\!] & \\
[\![\mathbf{let}\ x = U\ \mathbf{in}\ N : A]\!] & = & \mathbf{let}_\star\ x = \sqcap a.[\![U : a]\!]\ \mathbf{in}\ [\![N : A]\!] & \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N : A]\!] & = & \mathbf{let}_\bullet\ x = \sqcap a.[\![M : a]\!]\ \mathbf{in}\ [\![N : A]\!] & (\text{if } M \notin \mathsf{GVal}) \\
[\![\mathbf{let}\ (x : \forall \overline{b}.H) = U\ \mathbf{in}\ N : A]\!] & = & (\forall \overline{b}.[\![U : H]\!]) \wedge \mathbf{def}\,(x : \forall \overline{b}.H)\ \mathbf{in}\ [\![N : A]\!] & \\
[\![\mathbf{let}\ (x : B) = M\ \mathbf{in}\ N : A]\!] & = & [\![M : B]\!] \wedge \mathbf{def}\,(x : B)\ \mathbf{in}\ [\![N : A]\!] & (\text{if } M \notin \mathsf{GVal})
\end{array}$$

Figure 5.4: Translation from FreezeML terms to constraints

existentially bound by the translation are fresh. Unsurprisingly, both forms of variables are translated to the related constraint forms. The translation of applications is standard. A fresh variable a_1 is introduced for the type for the argument N. Given that the type of the overall application is expected to be A, the type of the function M must then be $a_1 \to A$.

For un-annotated functions, type variables a_1 and a_2 are introduced for the parameter and return type, respectively. The translation is again similar to the one used by Pottier and Rémy [97]. The variable a_1 is then the sole annotation on the variable x bound by the def constraint. Finally, an equality constraint asserts that $a_1 \to a_2$ corresponds to the expected overall type of the function. Note that the semantics of def constraints then enforce that only monomorphic types for a_1 are permitted. For annotated functions, we can simply eschew the flexible variable standing for the parameter type and reuse the type annotation directly. This illustrates the dual purpose of constraints as both performing type inference and asserting types. Note that by the assumption that the term to translate is well formed under some Δ, the annotation A may only refer to type variables from Δ, which are hence considered rigid.

The translation of un-annotated let bindings simply introduces let constraints. The value restriction is implemented at this point, by choosing the kind of let constraint based on the syntactic category of the let-bound term. Only a guarded term leads to the generation of a *generalising* let constraint.

No let constraints need to be generated for annotated terms $\textbf{let}\ (x : B) = M\ \textbf{in}\ N$. The translation again implements the value restriction. In both cases, the resulting constraint contains $\textbf{def}\ (x : B)\ \textbf{in}\ [\![N : A]\!]$ as sub-constraint, where A is the expected overall type. If M is a guarded value U, and may therefore be generalised to obtain type B, the typing rule for annotated let terms (see Figure 3.5 on page 45) imposes that M must have a guarded type, called H in the corresponding translation case. The toplevel quantifiers $\overline{a}$ of B are then universally quantified on the constraint level, resulting in the conjunct $\forall\overline{a}.[\![M : H]\!]$ being generated. Otherwise, if M is not a guarded value, we simply assert that the type of M *is* B.

As an example, we consider the translation of the following term M.

$$\lambda x.\textbf{let}\ f = \lambda y.x\ \textbf{in}\ f\ 3$$

It has type $S \to S$ for all monomorphic types S that are well formed under the given context.

The result of $[\![M : a]\!]$ is the constraint shown in Figure 5.5 on the next page. Here, for simplicity we assume that $[\![i : A]\!] \overset{\text{def}}{=} A \sim \mathsf{Int}$ for all integer literals i. Of course, we may generalise this and assume that the translation has access to an environment containing the types of built-in constants, which we eschew formalising.

We observe that due to the sub-constraint $(b_1 \to b_2) \sim a$, the overall constraint must equate b_1 and b_2 and assert that they may only be interpreted with monomorphic types. Further, note that under a term context containing $(x : b_1)$, the sub-constraint C'' is logically equivalent to $\exists b_4.(b_4 \to b_1) \sim b_3 \wedge \mathsf{mono}(b_4)$. This means that the type of f in C''' is $\forall b_4.b_4 \to b_1$. The constraint C''' is equivalent to $f \preceq \mathsf{Int} \to b_2$, which due to $f's$ type then yields the equivalence of b_1 and b_2. The monomorphism of the type is enforced by b_1 appearing in the type annotation in a def constraint, namely the one binding x. Thus, the overall constraint in Figure 5.5 is equivalent to $\exists b.a \sim (b \to b)\ \wedge\ \mathsf{mono}(b)$.

$$
\begin{array}{l}
[\![\lambda x.\mathbf{let}\ f = \lambda y.x\ \mathbf{in}\ f\ 3\ :\ a]\!]\ = \\[1em]
\exists b_1\, b_2.\, b_1 \to b_2 \sim a \wedge \\
\qquad \mathbf{def}\ (x : b_1)\ \mathbf{in} \\
\qquad \left.\begin{array}{l}
\mathbf{let}_\star\ f = \sqcap b_3.\overbrace{\exists b_4\, b_5.b_4 \to b_5 \sim b_3\ \wedge\ \big(\mathbf{def}\ (y : b_4)\ \mathbf{in}\ x \preceq b_5\big)}^{C'' := [\![\lambda y.x\, :\, b_3]\!]}\ \mathbf{in} \\
\underbrace{\exists b_6.f \preceq b_6 \to b_2\ \wedge\ b_6 \sim \mathsf{Int}}_{C''' := [\![f\, 3\, :\, b_2]\!]}
\end{array}\right\} =: C' \\[2em]
\text{where } C' = [\![\mathbf{let}\ f = \lambda x.y\ \mathbf{in}\ f\ 3\ :\ b_2]\!]
\end{array}
$$

Figure 5.5: Translation example

5.2.1 Correctness of the translation

We first observe that the translation produces a well formed constraint from any well-formed FreezeML term. Of course, the constraint may be unsatisfiable.

Lemma 5.2 (Well-formedness of constraints obtained from terms)**.**
If $\Delta;\Gamma \vdash M\ \mathbf{ok}$ *and* $(\Delta,\Xi) \vdash A\ \mathbf{ok}$ *then* $\Delta;\Xi;\Gamma \vdash [\![M : A]\!]\ \mathbf{ok}$.

Proof. By induction on the structure of M. We observe that the only free type variables of $[\![M : A]\!]$ are those appearing freely in A and in the type annotations appearing within M. By $\Delta;\Gamma \vdash M\ \mathbf{ok}$ we have that all such free type variables in the annotations in M are rigid variables from Δ. Hence, the only other type variables appearing in $[\![M : A]\!]$ are those from Ξ. $\square$

In order to prove the soundness and completeness of the translation, we need to formalise the close relationship between principal (a relation involving terms, defined in Section 3.2.1) and mostgen (a relation involving constraints, defined in Section 5.1.2) The principal type A of a term M is exactly the type that the most general solution of the constraint $[\![M : a]\!]$ must use for a.

Lemma 5.3 (Relationship between principal and mostgen)**.**
Let $a \# (\Delta,\Delta')$ *and* $\Delta;\Gamma \vdash M\ \mathbf{ok}$. *Then we have* $\mathsf{principal}(\Delta,\Gamma,M,\Delta',A)$ *iff* $\mathsf{mostgen}(\Delta,a,\Gamma,[\![M : a]\!],\Delta',[a \mapsto A])$.

Proof. Recall the definitions of principal and mostgen:

$$\begin{aligned}
&\mathsf{principal}(\Delta,\Gamma,M,\Delta',A) = \\
&\qquad (\Delta,\Delta');\Gamma \vdash M : A \ \text{ and} && (5.4)\\
&\qquad (\text{for all } \Delta'',A'' \mid \text{if } (\Delta,\Delta'');\Gamma \vdash M : A'' \\
&\qquad\qquad\qquad \text{then there exists } \delta \text{ such that} && (5.5)\\
&\qquad\qquad\qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta(A) = A'')
\end{aligned}$$

$$\begin{aligned}
&\mathsf{mostgen}(\Delta,a,\Gamma,\delta,[\![M : a]\!],\Delta',[a \mapsto A]) = \\
&\qquad (\Delta,\Delta');a;\Gamma;[a \mapsto A] \vdash [\![M : a]\!] \ \text{ and} && (5.6)\\
&\qquad (\text{for all } \Delta'',\delta'' \mid \text{if } (\Delta,\Delta'');a;\Gamma;\delta'' \vdash [\![M : a]\!] \\
&\qquad\qquad\qquad \text{then there exists } \delta \text{ such that} && (5.7)\\
&\qquad\qquad\qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta'' = \delta \circ [a \mapsto A])
\end{aligned}$$

$\Rightarrow$ We apply Theorem 5.1 to (5.4), immediately yielding the desired property (5.6).

To show (5.7), we assume $(\Delta,\Delta'');a;\Gamma;\delta'' \vdash [\![M : a]\!]$, which implies that $\delta'' = [a \mapsto A'']$ for some A''.

By Theorem 5.2 this gives us $(\Delta,\Delta'');\Gamma \vdash M : A''$. According to (5.5), there exists a δ with the desired properties.

$\Leftarrow$ We apply Theorem 5.2 to (5.6), which gives us satisfaction of property (5.4).

To show (5.7), we assume $(\Delta,\Delta'');\Gamma \vdash M : A''$. Theorem 5.1 then gives us $(\Delta,\Delta'');a;\Gamma;[a \mapsto A''] \vdash [\![M : a]\!]$. According to (5.7) there exists an appropriate δ.

□

We observe that this proof relies on the soundness and completeness of the translation, as subsequently stated in Theorems 5.1 and 5.2. The proofs of these theorems in turn use Lemma 5.3, but only on sub-constraints.

We can now formalise the soundness of the translation, stating that if M has type A, then $[\![M : a]\!]$ has a model that maps a to A.

Theorem 5.1 (Soundness of constraint generation with respect to typing relation)**.** *Let* $\Delta;\Gamma \vdash M : A$ *hold and* $a \# \Delta$*. Then we have that* $\Delta;a;\Gamma;[a \mapsto A] \vdash [\![M : a]\!]$ *holds.*

Proof. We prove the following, more general property by induction on M: Let $(\Delta,\Xi) \vdash A$ and $\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$ and $\Delta;\Gamma \vdash M : \delta(A)$. Then we have that $\Delta;\Xi;\Gamma;\delta \vdash [\![M : A]\!]$ holds. See Appendix B.2 for details. $\square$

Conversely, if $[\![M : a]\!]$ has a model, then its solution for a is a valid type of M.

Theorem 5.2 (Completeness of constraint generation with respect to typing relation)**.** *Let* $\Delta;\Gamma \vdash M$ **ok** *hold. Then* $\Delta;a;\Gamma;\delta \vdash [\![M : a]\!]$ *implies* $\Delta;\Gamma \vdash M : \delta(a)$.

Proof. We strengthen the property to the following, stronger statement: If $\Delta;\Gamma \vdash M$ **ok** and $(\Delta,\Xi) \vdash A$ **ok** hold, then $\Delta;\Xi;\Gamma;\delta \vdash [\![M : A]\!]$ implies $\Delta;\Gamma \vdash M : \delta(A)$. The proof is then by structural induction on M, see Appendix B.2 for details. $\square$

5.3 Constraint solving

We now present a solver for our constraint language. It is a stack machine, loosely based on the HM(X) solver presented by Pottier and Rémy [97]. The solver is defined in terms of states s and a transition relation between them. The transitions represent the steps taken by the solver, unless it is stuck (i.e., no transition is possible) indicating that the constraint under consideration is unsatisfiable.

Unlike the solver by Pottier and Rémy, ours relies on a deterministic transition relation, where at most one rule applies. We first discuss states of the machine before proceeding to the transition relation.

5.3.1 Machine states

The states s of the solver are the form (F,Θ,θ,C). Here, the only kind of component not previously discussed elsewhere is the *stack* F. The role of the different components in the state is as follows.

5.3.1.1 Stack and in-progress context

The first component of a machine state is the *stack* F. It is closely related to the last component of the state, the *in-progress constraint* C. While the latter denotes what constraint to process (i.e., solve) next, the stack indicates in which context C occurs. The stack not only contains bindings for type and term variables in C, but also indicates how to proceed once C has been solved.

To this end, stacks are defined as possibly empty lists of stack frames f, which directly correspond to those constraint forms with at least one sub-constraint, and are separated by the :: operator.

$$\begin{array}{ll} \text{Frames} & f ::= \square \wedge C \mid \forall a \mid \exists a \mid \mathbf{let}_R\, x = \sqcap a.\square \ \mathbf{in}\ C \mid \mathbf{def}\,(x : A) \\ \text{Stacks} & F ::= \cdot \mid F :: f \end{array}$$

We can interpret stacks as a linearisation of a constraint with a single hole. Concretely, consider the stack F defined as follows.

$$\exists a \ :: \ \mathbf{def}\,(x : \forall b.b \to b) \ :: \ \square \wedge \lceil x : a \rceil$$

It stands for the single-hole constraint $\exists a.\, \mathbf{def}\,(x : \forall b.b \to b)\ \mathbf{in}\ \square \wedge \lceil x : a \rceil$. Observe that stacks grow to the right and the first/leftmost element of the stack corresponds to the outermost part of the resulting constraint.

This gives rise to the operator $F[-]$ for plugging a constraint into the remaining hole.[2] It is defined as follows.

$$\begin{array}{rcl} \cdot[C] & = & C \\ (F :: \square \wedge C_2)[C_1] & = & F[C_1 \wedge C_2] \\ (F :: \forall\, a)[C] & = & F[\forall a.C] \\ (F :: \exists\, a)[C] & = & F[\exists a.C] \\ (F :: \mathbf{let}_R\, x = \sqcap a.\square \ \mathsf{in}\ C_2)[C_1] & = & F[\mathbf{let}_R\, x = \sqcap a.C_1 \ \mathbf{in}\ C_2] \\ (F :: \mathbf{def}\,(x : A))[C] & = & F[\mathbf{def}\,(x : A)\ \mathbf{in}\ C] \end{array}$$

As a result, we have the following for our example stack F.

$$F[x \prec \mathsf{Int} \to \mathsf{Int}] \ = \ \exists a.\, \mathbf{def}\,(x : \forall b.b \to b)\ \mathbf{in}\ x \preceq (\mathsf{Int} \to \mathsf{Int}) \wedge \lceil x : a \rceil$$

We observe that a stack F synthesises a rigid context, flexible context and term context denoting what elements are in scope at the hole due to the surrounding type and term variables bindings in F. We formalise this using functions $\mathsf{rc}(F)$, $\mathsf{fc}(F)$ and $\mathtt{tc}(F)$, respectively, which are defined in Figure 5.6 on the following page. Note that the term variables bound by let frames are not considered by $\mathtt{tc}(F)$. While x is in scope in the constraint C of a frame $\mathbf{let}_R\ x = \sqcap a.\square\ \mathbf{in}\ C$, it is not in scope in the

[2]Given this definition, the reader may wonder why we do not define stacks directly as "constraint contexts" in the style of an evaluation context (i.e., defining $F ::= [] \mid F[[] \wedge C] \mid \ldots$). We avoid this notation in order to be able to easily reason about the head *and* tail of stacks F. While our solver only operates on the the rightmost stack frame(s) of the stack, we need to reason about the left-most frames as part of our metatheory in Section 5.4.

hole. Further, note that the binders for type and term variables appearing directly in the constraint part C of frames $\square \wedge C$ and $\mathbf{let}_R\, x = \sqcap a.\square$ **in** C are ignored, as they never bind variables that are in scope in the hole.

$$\mathsf{rc}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{rc}(F'), a & \text{if } F = F' :: \forall a \\ \mathsf{rc}(F') & \text{otherwise } (F = F' :: \_) \end{cases}$$

$$\mathsf{fc}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{fc}(F'), a & \text{if } F = F' :: \exists a \text{ or} \\ & \quad F = F' :: \mathbf{let}_R\, x = \sqcap a.\square \ \mathbf{in}\ C_2 \\ \mathsf{fc}(F') & \text{otherwise } (F = F' :: \_) \end{cases}$$

$$\mathsf{tc}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{tc}(F'), (x : A) & \text{if } F = F' :: \mathbf{def}\,(x : A) \\ \mathsf{tc}(F') & \text{otherwise } (F = F' :: \_) \end{cases}$$

Figure 5.6: Functions for synthesising contexts from stacks

In addition, we define $\mathsf{btv}(F)$ and $\mathsf{bv}(F)$ to stand for all type variables (rigid or flexible) and term variables bound by the stack frames in F, respectively. Here, we again ignore type and term variables bound within the constraints appearing in conjunction and let frames. As a result, $\mathsf{btv}(F)$ contains simply the union of $\mathsf{rc}(F)$ and $\mathsf{fc}(\Delta)$. Unlike $\mathsf{tc}(F)$, we have that $\mathsf{bv}(F)$ also contains those term variables bound by let frames.

Finally, by abuse of notation, we allow writing $f :: F$ and $F_1 :: F_2$ to add a frame at the front of the stack and to concatenate two stacks, respectively. We use this to pattern-match stacks: For example, given a stack F, asserting that F is equal to $\forall a :: \exists b :: F'$ or equal to $F' :: \forall a :: \exists b$ allows us to extract all but two innermost or outermost frames, respectively, captured by F'.

5.3.1.2 Unification context

Each solver state contains a restriction context Θ and substitution θ, both of which were previously defined in Section 4.1.1. Together, we refer to Θ and θ as the *unification* *context* of the state.

The type variables in Θ must correspond exactly to those bound by the stack F and in scope in the in-progress constraint (i.e., $\mathsf{fc}(F)$ and Θ contain the same variables). The role of the restrictions in Θ remains as before: If $(a : \bullet) \in \Theta$, then a may only be interpreted with monotypes, otherwise polytypes are permitted.

The substitution θ encodes the currently known equalities imposed on the type variables in Θ. To this end, we must have that $\Delta \vdash \theta : \Theta \Rightarrow \Theta$ holds, meaning that θ contains mappings for all flexible type variables currently brought in scope by the stack F.

For example, when processing the constraint $\exists ab.a \sim (b \to b) \wedge \mathsf{mono}(b)$, right after $\exists a$ and $\exists b$ are encountered by the solver, we have $\Theta = (a : \star, b : \star)$ and $\theta = [a \mapsto a, b \mapsto b]$ in the resulting state, indicating that no further knowledge about the variables is present. After progressing further, we will eventually reach a state with $a \mapsto (b \to b)$ and $(b : \bullet)$ in its unification context.

This illustrates that the components Θ and θ can be considered to be a *residual constraint* (or *solved form*) representing the current knowledge about flexible variables. We can concretise this by directly translating Θ and θ into a corresponding constraint. This is achieved by translating all $(a : \bullet) \in \Theta$ to $\mathsf{mono}(a)$ constraints and replacing all mappings of θ with corresponding equality constraints. We refer to the former constraint as $\mathfrak{U}(\Theta)$ and the latter as $\mathfrak{U}(\theta)$, with $\mathfrak{U}(\Theta, \theta)$, the overall representation of the unification context as a constraint, simply defined as their conjunction. Formally, these constraints are defined as follows.

$$\begin{aligned}
\mathfrak{U}(\Theta) &= \textstyle\bigwedge_{(a:\bullet)\in\Theta} \mathsf{mono}(a) \\
\mathfrak{U}(\theta) &= \textstyle\bigwedge_{a\in\mathsf{ftv}(\Theta)} a \sim \theta(a) \\
\mathfrak{U}(\Theta,\theta) &= \mathfrak{U}(\Theta) \wedge \mathfrak{U}(\theta)
\end{aligned}$$

Given a state (F, Θ, θ, C), we may then consider the constraint $F[\mathfrak{U}(\Theta, \theta) \wedge C]$ to be a logical representation of the overall state as a constraint. We will revisit this perspective when discussing the correctness of the solver, by relating the state before and after each transition using the constraints obtained from each of the two states in this way.

Instead of carrying Θ and θ in states, we could alternatively carry $\mathfrak{U}(\Theta, \theta)$ as a dedicated constraint in each state, similarly to the usage of unification constraints in the solver of Pottier and Rémy.[3] Their representation is more easily extensible (as they

[3]However, Pottier and Rémy's unification constraints consist of *multi-equations*. These expose the

may extend the language of residual constraints), while our representation is slightly simpler for our purposes as we can re-use the existing well-formedness definitions from Chapter 4, relating Θ and θ.

5.3.1.3 Well-formedness of machine states

To define when an overall state (F, Θ, θ, C) is well formed, we first consider the well-formedness of its stack component F. To this end, we use well-formedness judgements $\Theta \vdash F$ **ok**, the rules for which are shown in Figure 5.7.

$$\boxed{\Theta \vdash F\ \mathbf{ok}}$$

$$\frac{}{\cdot \vdash \cdot\ \mathbf{ok}} \qquad \frac{\mathsf{rc}(F); \Theta; \mathsf{tc}(F) \vdash C\ \mathbf{ok} \qquad \Theta \vdash F\ \mathbf{ok}}{\Theta \vdash F :: \square \wedge C\ \mathbf{ok}}$$

$$\frac{\Theta \vdash F\ \mathbf{ok} \qquad a \notin \mathsf{btv}(F)}{\Theta \vdash F :: \forall\, a\ \mathbf{ok}} \qquad \frac{\Theta \vdash F\ \mathbf{ok} \qquad a \notin \mathsf{btv}(F)}{(\Theta, a : R) \vdash F :: \exists\, a\ \mathbf{ok}}$$

$$\frac{\begin{array}{c} x \notin \mathsf{bv}(F) \qquad \Theta \vdash F\ \mathbf{ok} \\ \text{for all } a \in \mathsf{ftv}(A) - \mathsf{rc}(F) : (a : \bullet) \in \Theta \end{array}}{\Theta \vdash F :: \mathbf{def}\ (x : A)\ \mathbf{ok}} \qquad \frac{\begin{array}{c} \mathsf{rc}(F); \Theta; (\mathsf{tc}(F), x : A) \vdash C\ \mathbf{ok} \\ x \notin \mathsf{bv}(F) \qquad \Theta \vdash F\ \mathbf{ok} \end{array}}{(\Theta, a : R) \vdash F :: \mathbf{let}_{R'}\, x = \sqcap a.\square\ \mathsf{in}\ C\ \mathbf{ok}}$$

Figure 5.7: Well-formedness of stacks

We have that $\Theta \vdash F$ **ok** holds if all type variables bound by frames in F are pairwise different, similarly for all term variables. Further, for all constraints C appearing somewhere in a stack frame, such as $\square \wedge C$, it must be the case that C is well formed with respect to the contexts that may be synthesised to represent the surrounding bindings on the stack. As in other well-formedness relations before, the concrete types associated with term variables are ignored by the relation, which is why the rule for let frames uses an arbitrary type A when synthesising a term context under which its sub-constraint is checked for well-formedness.

Finally, the judgement checks that Θ contains exactly the same type variables as $\mathsf{fc}(F)$, and all variables therein that appear in an annotation on a def frame must carry a monomorphic restriction in Θ.

underlying union-find structure more explicitly as compared to a simple set of equality constraints.

We may now define that an overall machine state $s = (F, \Theta, \theta, C)$ is well formed, denoted $\vdash s$ **ok**, if and only if the following conditions hold.

1. We have $\Theta \vdash F$ **ok**.
2. We have $\mathsf{rc}(F) \vdash \theta : \Theta \Rightarrow \Theta$.
3. The substitution θ is idempotent, meaning that for all $a \in \mathsf{ftv}(\Theta), b \in \mathsf{ftv}(\theta(a))$ we have $\theta(b) = b$.
4. The constraint C is well formed under the contexts induced by F, meaning that $\mathsf{rc}(F); \mathsf{fc}(F); \mathsf{tc}(F) \vdash C$ **ok** holds.

Also note that a well formed state is closed in the sense that its stack binds all type and term variables that may occur freely in C, appear in Θ, or appear in the domain and co-domain of θ.

Lemma 5.4 (Well-formedness of constraints obtained from states)**.**
Let $\vdash (F, \Theta, \theta, C)$ **ok** *hold. Then the constraints* $F[C]$ *and* $F[\mathfrak{U}(\Theta, \theta) \wedge C]$ *are well formed under empty rigid, flexible and term contexts.*

Proof. By induction on structure of F. □

5.3.2 Solving rules

As mentioned before, the solver is defined in terms of a transition relation between states. The possible transitions are defined in Figure 5.8 on the following page. At most one rule applies to any state.

We will generally invoke the solver on states of the form $(\cdot, \cdot, \emptyset, \forall\Delta.\exists\Xi.C)$. Execution of the solver then either reaches a *final state* (indicating that the original constraint C is satisfiable) or otherwise gets stuck before reaching such a state. Final states have the form $(\forall\Delta :: \exists(\Xi, \Xi'), \Theta, \theta, \mathsf{true})$, allowing a model for the constraint C to be read off. We will formalise these correctness properties in Section 5.4.

We now discuss the individual rules of the solver in detail.

5.3.2.1 Basic constraints

Type equality constraints $A \sim B$ are simply deferred to the unification algorithm $\mathcal{U}$, as defined in Figure 4.4 on page 80, by the rule S-EQ, after applying θ to A and B.

$$(F,\Theta,\theta,A \sim B) \;\to\; (F,(\Theta_{\mathrm{u}},\Theta'),\theta_{\mathrm{u}}\circ\theta,\mathsf{true}) \qquad \text{(S-Eq)}$$
$$\text{where } (\Theta_{\mathrm{u}},\theta_{\mathrm{u}}) = \mathcal{U}(\mathsf{rc}(F),\Theta,\theta(A),\theta(B))$$
$$\Theta \approx (\Theta',\Theta'') \qquad \mathsf{ftv}(\Theta'') = \mathsf{ftv}(\Theta_{\mathrm{u}})$$

$$(F,\Theta,\theta,\lceil x : A\rceil) \;\to\; (F,\Theta,\theta,\mathsf{tc}(F)(x) \sim A) \qquad \text{(S-Freeze)}$$

$$(F,\Theta,\theta,x \preceq A) \;\to\; (F,\Theta,\theta,\exists\overline{a}.H \sim A) \qquad \text{(S-Inst)}$$
$$\text{where } \forall\overline{a}.H = \mathsf{tc}(F)(x) \quad \overline{a}\,\#\,\mathsf{btv}(F)$$

$$(F,\Theta,\theta,\mathsf{mono}(a)) \;\to\; (F,\Theta',\theta,\mathsf{true}) \qquad \text{(S-Mono)}$$
$$\text{where } \overline{b} = \mathsf{ftv}(\theta(a)) - \mathsf{rc}(F) \quad \Theta' = (\Theta - \overline{b}) \cup \overline{b : \bullet}$$
$$(\mathsf{rc}(F)^{\bullet},\Theta') \vdash_{\bullet} \theta(a)\ \mathbf{ok}$$

$$(F,\Theta,\theta,C_1 \wedge C_2) \;\to\; (F :: \square \wedge C_2,\Theta,\theta,C_1) \qquad \text{(S-ConjPush)}$$

$$(F :: \square \wedge C_2,\Theta,\theta,\mathsf{true}) \;\to\; (F,\Theta,\theta,C_2) \qquad \text{(S-ConjPop)}$$

$$(F,\Theta,\theta,\exists a.C) \;\to\; (F :: \exists a,(\Theta,a:\star),\theta[a \mapsto a],C) \qquad \text{(S-ExistsPush)}$$

$$(F :: f :: \exists\overline{a},\Theta,\theta,\mathsf{true}) \;\to\; (F :: \exists\overline{c} :: f,\Theta',\theta\restriction_{\Theta'},\mathsf{true}) \qquad \text{(S-ExistsLower)}$$
$$\text{where } f \text{ is neither a } \mathbf{let} \text{ or } \exists \text{ frame} \quad \overline{b};\overline{c} = \mathsf{partition}(\overline{a},\theta,\Theta)$$
$$\Theta' = \Theta - \overline{b} \quad |\overline{a}| > 0$$

$$(F,\Theta,\theta,\forall a.C) \;\to\; (F :: \forall\, a,\Theta,\theta,C) \qquad \text{(S-ForallPush)}$$

$$(F :: \forall\, a,\Theta,\theta,\mathsf{true}) \;\to\; (F,\Theta,\theta,\mathsf{true}) \text{ where } a \notin \mathsf{ftv}(\theta(\Theta)) \qquad \text{(S-ForallPop)}$$

$$(F,\Theta,\theta,\mathbf{def}\ (x:A)\ \mathbf{in}\ C) \;\to\; (F :: \mathbf{def}\ (x:A),\Theta',\theta,C) \qquad \text{(S-DefPush)}$$
$$\text{where } \overline{b} = \mathsf{ftv}(\theta(A)) - \mathsf{rc}(F) \quad \Theta' = (\Theta - \overline{b}) \cup \overline{b : \bullet}$$
$$\text{for all } a \in \mathsf{ftv}(A) : \Theta' \vdash_{\bullet} \theta(a)\ \mathbf{ok}$$

$$(F :: \mathbf{def}\ (x:A),\Theta,\theta,\mathsf{true}) \;\to\; (F,\Theta,\theta,\mathsf{true}) \qquad \text{(S-DefPop)}$$

$$\text{(S-LetPush)}$$
$$(F,\Theta,\theta,\mathbf{let}_R\ x = \sqcap b.C_1\ \mathbf{in}\ C_2) \;\to\; (F :: \mathbf{let}_R\ x = \sqcap b.\square\ \mathbf{in}\ C_2,(\Theta,b:\star),\theta[b \mapsto b],C_1)$$

$$\text{(S-LetPop}\star\text{)}$$
$$(F :: \mathbf{let}_{\star}\ x = \sqcap b.\square\ \mathbf{in}\ C :: \exists\overline{a},\Theta,\theta,\mathsf{true}) \;\to\; (F :: \exists\overline{a''},\Theta',\theta\restriction_{\Theta'},\mathbf{def}\ (x:B)\ \mathbf{in}\ C)$$
$$\text{where } \overline{a'};\overline{a''} = \mathsf{partition}((\overline{a},b),\theta,\Theta) \quad A = \theta(b) \quad \overline{c} = \mathsf{ftv}(A) \cap \overline{a'} \quad \Theta' = \Theta - \overline{a'} \quad B = \forall\overline{c}.A$$

$$\text{(S-LetPop}\bullet\text{)}$$
$$(F :: \mathbf{let}_{\bullet}\ x = \sqcap b.\square\ \mathbf{in}\ C :: \exists\overline{a},\Theta,\theta,\mathsf{true}) \;\to\; (F :: \exists(\overline{c},\overline{a''}),\Theta',\theta\restriction_{\Theta'},\mathbf{def}\ (x:A)\ \mathbf{in}\ C)$$
$$\text{where } \overline{a'};\overline{a''} = \mathsf{partition}((\overline{a},b),\theta,\Theta) \quad A = \theta(b) \quad \overline{c} = \mathsf{ftv}(A) \cap \overline{a'} \quad \Theta' = \Theta - (\overline{a'} - \overline{c})$$

Figure 5.8: Constraint solving rules

As a minor technical concern, recall that $\mathcal{U}$ may return a restriction context Θ_u that contains only a subset of the variables of input restriction context Θ. This is due to $\mathcal{U}$ removing variables a from the returned restriction context when discovering a mapping $a \mapsto A$ where $a \neq A$. Since $\mathcal{U}$ returns – and our solver acts on – idempotent substitutions, this means that the type A mentioned above cannot contain a freely. Thus, the resulting substitution is guaranteed to remove a whenever it is applied.

However, in our solver this removal from the restriction context would mean that the resulting state created by S-EQ may have a stack F with $\Theta_u \not\vdash F$, making the state ill formed. While we may define a variant of $\mathcal{U}$ that eschews removing variables from the restriction context, thereby ensuring $\mathsf{ftv}(\Theta) = \mathsf{ftv}(\Theta_u)$, the rule S-EQ simply re-adds the removed variables Θ' to the restriction context of the result state. Recall that we use $\approx$ to indicate when two sequences, including restriction contexts, are permutations of each other.

This treatment of Θ_h highlights the fact that our constraint solver uses the stack to indicate what flexible type variables are in scope and the restriction context is merely supposed to contain the restrictions on the variables in scope. A mechanism discussed in Section 5.3.2.2 then ultimately ensures that the substituted variable a mentioned above is indeed removed at a later point, namely when returning to the stack frame binding it.

The unification algorithm is treated as a black box by the constraint solver. This essentially allows us to parameterise the constraint over the equational theory of types implemented by the unification algorithm. We return to the discussion of alternative equational theories in Section 6.2.1.1. An alternative approach would be to implement unification directly in terms of constraint solving, by adding dedicated rules for individual unification steps to the solver itself [26, 110]. For example, we may rewrite $A_1 \to A_2 \sim B_1 \to B_2$ to $A_1 \sim B_1 \wedge A_2 \sim B_2$.

Freezing constraints of the form $\lceil x : A \rceil$ are handled by introducing an equality constraint between A and the type associated with x in an enclosing def frame in the state's stack F. In the rule S-FREEZE, this is formalised by using $\mathsf{tc}(F)$, which synthesises a term context from the def frames in F.

Conversely, an instantiation constraint $x \preceq A$ is replaced by $\exists \overline{a}.(H \sim A)$, where the type associated with x in the stack is $\forall \overline{a}.H$. As expected, this corresponds to instantiating all toplevel quantifiers of x's type to obtain A.

Given an in-progress constraint $\mathsf{mono}(a)$, the solver restricts all free flexible type variables of $\theta(a)$ to monomorphic types. Afterwards, it verifies that this makes $\theta(a)$ a monomorphic type in the *resulting* restriction context. Thus, the rule verifies that the current information about a is compatible with a being monomorphic and ensures that it cannot become polymorphic during subsequent solving steps.

The rules S-CONJPUSH and S-CONJPOP are used for solving conjunctions. When encountering a constraint $C_1 \wedge C_2$, we turn its first sub-constraint into the in-progress constraint of the subsequent state. A stack frame $\square \wedge C_2$ is added to the stack. As a result, once C_1 is solved (i.e., the in-progress constraint becomes true) the rule S-CONJPOP pops the stack frame and proceeds with solving C_2. This pattern of *PUSH rules extracting a new in-progress constraint from a larger constraint and adding a stack frame, in combination with a corresponding *POP rule is exhibited by multiple other pairs of rules.

5.3.2.2 Existential quantification

The rule S-EXISTSPUSH handles constraints of the form $\exists a.C'$. It adds a corresponding stack frame and initialises a allowing it to be polymorphic.

Here and in other rules handling an in-progress constraint that binds type or term variables, we implicitly assume that the constraint has been alpha-converted to ensure that any such bindings do not clash with variables already bound by the stack. In the case of S-EXISTSPUSH, this means that we assume $a \notin \mathsf{btv}(F)$.

In general, existential quantifiers remaining on top of the stack once the in-progress constraint is solved require careful consideration. While some quantifiers may be simply dropped, others may need to be moved downwards in the stack – effectively increasing their syntactic scope – if the bound type variable is still used.

To this end, consider the following transitions from s_1 to s_3. As usual, $\rightarrow^+$ denotes the transitive closure of the transition relation.

$$\begin{array}{lll}
 & (\exists a :: \square \wedge a \sim \mathsf{Int},\ (a : \star),\ [a \mapsto a],\ \exists b_1\, b_2\, b_3.a \sim (b_1 \rightarrow b_2) \wedge b_1 \sim b_2) & =: s_1 \\
\rightarrow^+ & (\exists a :: \square \wedge a \sim \mathsf{Int} :: \exists b_1 :: \exists b_2 :: \exists b_3,\ \Theta_2,\ \theta_2, \mathsf{true}) & =: s_2 \\
\rightarrow & (\exists a :: \exists b_2 :: \square \wedge a \sim \mathsf{Int},\ (a : \star, b_2 : \star),\ [a \mapsto (b_2 \rightarrow b_2), b_2 \mapsto b_2], \mathsf{true}) & =: s_3
\end{array}$$

$$\text{where}\quad \Theta_2 = (a : \star, b_1 : \star, b_2 : \star, b_3 : \star),$$
$$\theta_2 = [a \mapsto (b_2 \rightarrow b_2), b_1 \mapsto b_2, b_2 \mapsto b_2, b_3 \mapsto b_3]$$

The steps from s_1 to s_2 represent the solving of the constraint $\exists b_1\, b_2\, b_3.a \sim (b_1 \rightarrow b_2) \wedge b_1 \sim b_2$.

In s_2, we observe that removing the stack frames for b_1 and b_3 is possible at this point, neither variable is needed anymore. However, the variable b_2 appears in the mapping for the variable a (i.e., the outer context from the perspective of the frames binding b_1 to b_3). Thus, a standard approach is to lower the existential frames for variables still required in subsequent states within the stack, while removing unnecessary ones [97]. The variable b_3 illustrates that merely appearing in the co-domain of θ is not a sufficient condition for being a variable that should be kept.

The rule S-EXISTSLOWER acts on stacks of the form $F' :: \exists\overline{a}$, a shorthand for $F' :: \exists a_1 :: \cdots :: \exists a_n$. In general, this shorthand allows n to be zero, but S-EXISTSLOWER requires $n \geq 1$ to ensure that the rule only applies to states whose stack ends with one or more existential frames.

The rule also requires that F' (i.e., the prefix of the stack before $\exists\overline{a}$) does not end with an existential or let frame. The former ensures that the rule exhaustively applies to all existential frames on top of the stack at once. The latter is due to the fact that existential frames right above a let frame must be handled directly by the rules for let frames.

The rule S-EXISTSLOWER then partitions the variables $\overline{a}$ into two subsets: Those variables that appear in the co-domain of θ when restricted to the outer stack frames (i.e., excluding the mappings for the variables $\overline{a}$) and those that do not. This is implemented using the helper function partition, which is defined as follows and returns two sequences.

$$\begin{aligned}&\mathsf{partition}(\Xi, \theta, \Theta) = \Xi'; \Xi'' \text{ where}\\ &\quad (\Xi', \Xi'') = \Xi \text{ and}\\ &\quad \text{for all } a \in \Xi : a \in \Xi'' \text{ iff } a \in \mathsf{ftv}(\theta\restriction_{(\mathsf{ftv}(\Theta)-\Xi)})\end{aligned}$$

In S-EXISTSLOWER, the variables in the first sequence returned by partition are removed from Θ and θ, while the existential frames binding the variables in the second sequence are lowered within the stack. In our example transitions, applying the rule S-EXISTSLOWER to s_2 leads to the state s_3. Here, b_2 has been lowered within the stack while b_1 and b_3 have been removed. Subsequent solving steps will eventually get stuck once unification of $b_2 \to b_2$ and Int fails.

5.3.2.3 Universal quantification

When processing a constraint $\forall a.C$, a stack frame is added for the binding before solving C. Once the stack frame is about to be popped by S-FORALLPOP, an escape check is performed, ensuring that no other existentially quantified variable is unified with (a type containing) a. Note that due to this escape check it is strictly *required* to drop unnecessary existentials in S-EXISTSLOWER, rather than merely being an optimisation. Consider the following transitions for solving the overall constraint $\exists a.\forall c.\exists b.b \sim c$.

$$\begin{array}{rll} & (\cdot,\cdot,[\,],\exists a.\forall c.\exists b.b \sim c) & \\ \rightarrow^{+} & (\exists a :: \forall c :: \exists b,\ (a:\star,b:\star),\ [a \mapsto a, b \mapsto c], \mathsf{true}) & =: s_4 \\ \overset{\text{S-ExistsLower}}{\rightarrow} & (\exists a :: \forall c,\ (a:\star),\ [a \mapsto a], \mathsf{true}) & =: s_5 \\ \overset{\text{S-ForallPop}}{\rightarrow} & (\exists a,\ (a:\star),\ [a \mapsto a], \mathsf{true}) & =: s_6 \end{array}$$

If the rule S-EXISTSLOWER had lowered the binding for b in the step from s_4 to s_5, rather than correctly identifying it to be unnecessary and removing it, this would yield the following state s'_5 instead.

$$(\exists a :: \exists b :: \forall c,\ (a:\star,b:\star),\ [a \mapsto a, b \mapsto c], \mathsf{true})$$

However, the escape check only succeeds when S-FORALLPOP is applied to s_5, but not when applied to s'_5. Thus, lowering the binding for b would have made the solver incomplete, as the original constraint is satisfiable. Instead, the solver correctly reaches the final state s_6.

5.3.2.4 Definition constraints

A constraint of the form **def** $(x:A)$ **in** C is handled by the rules S-DEFPUSH and S-DEFPOP. Similarly to the rule S-MONO, all free flexible type variables $\overline{b}$ of $\theta(A)$ are restricted to monomorphic types in the resulting restriction context Θ'. The rule then checks that all existing mappings for variables in $\overline{b}$ are compatible with these updated restrictions. As an equivalent alternative, we may instead assert that $\mathsf{rc}(F) \vdash \theta : \Theta' \Rightarrow \Theta'$ holds or add a conjunct $\bigwedge_{b \in \overline{b}} \mathsf{mono}(b)$ to the in-progress constraint C.[4]

As an example, we consider the following constraint C_{ex}. It is logically equivalent to the translation result $[\![\lambda x.\,\mathbf{let}\ f = \lambda y.x\ \mathbf{in}\ f\ 3\ :\ a]\!]$ shown earlier in Figure 5.5 on page 107, yet slightly simplified for conciseness.[5] As discussed then, the constraint

[4]Note that we must, however, update Θ to Θ' as part of adding the def frame to the stack in S-DEFPUSH. This is required to ensure that the resulting state is well formed, which imposes monomorphism conditions on free variables found in the type annotations of def frames. While $\Theta' \vdash F :: \mathbf{def}\ (x:A)\ \mathbf{ok}$ is guaranteed to hold, $\Theta \vdash F :: \mathbf{def}\ (x:A)\ \mathbf{ok}$ is not.

[5]We may further simplify the constraint C_{ex} by dropping the sub-constraint $\mathsf{mono}(b_4)$, which yields an overall constraint still logically equivalent to the one shown in Figure 5.5. This is due to the variable b_4 being generalised by the enclosing let constraint, at which point the restriction of b_4 to monotypes becomes irrelevant. However, the current version of C_{ex} ensures that its sub-constraint C''_{ex} is also equivalent to its counterpart C'' in Figure 5.5.

is equivalent to the even simpler form $\exists b.a \sim (b \to b) \wedge \mathsf{mono}(b)$, meaning that we expect b_1 and b_2 appearing in C_{ex} to be equated and restricted to monomorphic types during solving.

$$
\begin{aligned}
C_{\text{ex}} :=\ & \exists b_1\, b_2.\, b_1 \to b_2 \sim a \wedge \\
& \quad \mathbf{def}\,(x : b_1)\ \mathbf{in} \\
& \quad \left.\begin{aligned} &\mathbf{let}_\star\, f = \sqcap b_3.\overbrace{\exists b_4\, b_5.\, b_4 \to b_5 \sim b_3 \ \wedge\ \mathsf{mono}(b_4)\ \wedge\ x \preceq b_5}^{=:\, C''_{\text{ex}}}\ \mathbf{in} \\ &\underbrace{f \preceq \mathsf{Int} \to b_2}_{=:\, C'''_{\text{ex}}} \end{aligned}\right\} =:\ C'_{\text{ex}}
\end{aligned}
$$

The solver proceeds as shown in Figure 5.9. Here, we again add a toplevel existential frame for a, the variable standing for the overall type of the term, in the initial state s_7. Recall that $\Xi^\star$ stands for the restriction context Θ containing the same variables as Ξ, but with restriction $\star$ on all variables. We sometimes omit identity mappings $b \mapsto b$ when showing substitutions and indicate this with ellipses.

$$
\begin{array}{rll}
& (\exists a, (a : \star), [a \mapsto a], \overbrace{\exists b_1\, b_2.b_1 \to b_2 \sim a\ \wedge\ \mathbf{def}\,(x : b_1)\ \mathbf{in}\ C'_{\text{ex}}}^{=\, C_{\text{ex}}}) & =:\ s_7 \\
\to^+ & (\exists a\, b_1\, b_2, (a, b_1, b_2)^\star, [a \mapsto (b_1 \mapsto b_2), \ldots], \mathbf{def}\,(x : b_1)\ \mathbf{in}\ C'_{\text{ex}}) & =:\ s_8 \\
\overset{\text{S-DEFPUSH}}{\to} & (\exists a\, b_1\, b_2 :: \mathbf{def}\,(x : b_1), (a : \star, b_1 : \bullet, b_2 : \star), [a \mapsto (b_1 \mapsto b_2), \ldots], C'_{\text{ex}}) & =:\ s_9 \\
\to^+ & (\exists a\, b_1\, b_2 :: \mathbf{def}\,(x : b_1), (a : \star, b_1 : \bullet, b_2 : \bullet), \theta_{13}, \mathsf{true}) & =:\ s_{13} \\
\overset{\text{S-DEFPOP}}{\to} & (\exists a\, b_1\, b_2, (a : \star, b_1 : \bullet, b_2 : \bullet), \theta_{13}, \mathsf{true}) & =:\ s_{14} \\
& \quad \text{where } \theta_{13} = [a \mapsto (b_2 \mapsto b_2), b_1 \mapsto b_2, b_2 \mapsto b_2] &
\end{array}
$$

Figure 5.9: Solving of the example constraint C_{ex}

In state s_8, the in-progress constraint defines x using type b_1. Applying S-DEFPUSH to this restricts b_1 to monomorphic types in the successor state s_9. Further solving from the latter state onward eventually leads to the state s_{13}, where b_1 has been equated with b_2. We will show the intermediate states s_{10} to s_{12} shortly in Section 5.3.2.5, when discussing how the let constraint C'_{ex} is solved.

In state s_{13}, no further checks are required when popping the def frame, leading to the final state s_{14}. Instead, the monomorphism restriction on b_1 would prevent any polymorphic solution for this variable.

As mentioned on page 119, the constraint C_{ex} is equivalent to $\exists b.a \sim (b \to b) \wedge \mathsf{mono}(b)$. We observe that the solution $b_2 \to b_2$ for a in the final state s_{14} as well as the restriction of b_2 to monotypes therein correctly reflects this.

5.3.2.5 Generalising let constraints

Upon encountering a constraint $\mathbf{let}_R\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$, a corresponding stack frame is added by S-LETPUSH, as well as entries in the unification context for the type variable b bound by the let constraint. Note that the added frame is marked with the form of let constraint (generalising or non-generalising). The solver then proceeds with solving the sub-constraint C_1.

When applied to our previously considered example state s_9 in Figure 5.9, part of an overall execution of the solver on the constraint C_{ex}, this results in the state s_{10}, shown in Figure 5.10 on the next page. Subsequent solving of the sub-constraint C''_{ex} (i.e., the first sub-constraint of the let constraint) then leads to the state s_{11}. The intermediate steps involve solving the sub-constraint $x \preceq b_5$ of C''_{ex}, which is rewritten to $b_1 \sim b_5$ (rule S-INST) at that point and then solved (rule S-EQ). This leads to the mapping $b_1 \mapsto b_5$ in s_{11}.

Once the in-progress constraint is solved, let frames for generalising let constraints are handled by the rule S-LETPOP$\star$. Note that it applies if the in-progress constraint is solved and there is a let frame marked with $\star$ on top of the stack, or such a frame followed by a series of existential frames binding variables $\overline{a}$. The latter is the case for our example state s_{11} in Figure 5.10. Such variables may reside on top of the stack either due to appearing at the toplevel of the constraint C_1 (i.e., the first sub-constraint of the overall let constraint), or by being lowered there by rules such as S-EXISTSLOWER. Note that the latter rule may move existential frames down to, but not past, surrounding let frames on the stack.

The rule S-LETPOP$\star$ then splits the existentially bound variables using the function $\mathsf{partition}$, similarly to S-EXISTSLOWER. Here, the existentially bound variables include the variables $\overline{a}$ mentioned before as well as the variable b, bound by the let frame itself. This results in the sequence $\overline{a''}$ of variables related to type variables from the surrounding context and those variables $\overline{a'}$ that are not. The goal of the rule is to create a def frame with a binding for the term variable x that was bound by the original let constraint that led to the let frame under consideration. The type for x is obtained from $\theta(b) \stackrel{\text{def}}{=} A$, the solution for b after the solver processed the first sub-constraint of the let constraint. Since S-LETPOP$\star$ handles frames resulting from generalising let

$$
\begin{aligned}
&(\overbrace{\exists a\, b_1\, b_2 :: \mathbf{def}\,(x : b_1)}^{=:\,F_9}, \overbrace{(a : \star, b_1 : \bullet, b_2 : \star)}^{=:\,\Theta_9}, \overbrace{[a \mapsto (b_1 \mapsto b_2), \ldots]}^{=:\,\theta_9}, C'_{\mathrm{ex}}) &&= s_9 \\
=\;& (F_9, \Theta_9, \theta_9, \mathbf{let}\ x = \sqcap b_3.\overbrace{\exists b_4\, b_5.b_4 \to b_5 \sim b_3 \wedge \mathsf{mono}(b_4) \wedge x \preceq b_5}^{=C''_{\mathrm{ex}}}\ \mathbf{in}\ C'''_{\mathrm{ex}}) && \\
\xrightarrow{\text{S-LetPush}}\;& (F_9 :: \mathbf{let}_\star\ f = \sqcap b_3.\square\ \mathbf{in}\ C'''_{\mathrm{ex}}, (\Theta_{10}, b_3 : \star), \theta_{10}[b_3 \mapsto b_3], C''_{\mathrm{ex}}) &&=: s_{10} \\
\longrightarrow^{+}\;& (F_9 :: \mathbf{let}_\star\ f = \sqcap b_3.\square\ \mathbf{in}\ C'''_{\mathrm{ex}} :: \exists b_4\, b_5, \Theta_{11}, \theta_{11}, \mathsf{true}) &&=: s_{11} \\
\xrightarrow{\text{S-LetPop}\star}\;& (F_9 :: \exists b_5, \Theta_{12}, \theta_{12}, \mathbf{def}\,(f : \forall b_4.b_4 \to b_5)\ \mathbf{in}\ \overbrace{f \preceq (\mathsf{Int} \to b_2)}^{=C'''_{\mathrm{ex}}}) &&=: s_{12} \\
\longrightarrow^{+}\;& (\exists a\, b_1\, b_2 :: \mathbf{def}\,(x : b_1), \Theta_{13}, [a \mapsto (b_2 \mapsto b_2), b_1 \mapsto b_2, b_2 \mapsto b_2], \mathsf{true}) &&= s_{13}
\end{aligned}
$$

$$
\begin{aligned}
\text{where}\quad & \Theta_{11} = (\Theta_{12}, b_3 : \star, b_4 : \bullet) && \theta_{11} = \theta_{12}[b_3 \mapsto (b_4 \to b_5), b_4 \mapsto b_4] \\
& \Theta_{12} = (a : \star, b_1 : \bullet, b_2 : \star, b_5 : \bullet) && \theta_{12} = [a \mapsto (b_5 \to b_2), b_1 \mapsto b_5, \ldots] \\
& \Theta_{13} = (a : \star, b_1 : \bullet, b_2 : \bullet) &&
\end{aligned}
$$

Figure 5.10: Solving of the let sub-constraint C'_{ex} of C_{ex}

constraints, the type for x is then determined from A by quantifying the generalisable type variables therein.

The aforementioned sequence $\overline{a'}$ returned by partition contains those variables that may potentially be generalised. However, $\overline{a'}$ may contain additional variables that were created while the solver processed the first sub-constraint of the let constraint, which do not actually appear in A. Due to FreezeML's notion of type equality, adding such unnecessary quantifiers to the type of x would result in an incorrect type.

Therefore, the solver determines the sub-sequence $\overline{c}$ of $\overline{a'}$ of variables that actually appear in A. Recall that $\mathsf{ftv}(A)$ returns the free variables of A ordered by their appearance within A; we assume that this ordering is preserved under intersection when defining $\overline{c}$. The rule S-LetPop⋆ then creates an in-progress constraint $\mathbf{def}\,(x : \forall\overline{c}.A)\ \mathbf{in}\ C$, where C is the second sub-constraint of the original let constraint. Note that the type $\forall\overline{c}.A$ may contain free flexible variables (namely, those which may appear in $\overline{a''}$). As expected, these type variable are monomorphised when the def constraint is processed subsequently. The rule then removes the variables from $\overline{a'}$ in the successor state, they were either generalised in the type of x or are no longer needed. The bindings for the variables in $\overline{a''}$ are preserved but moved downwards in the stack, similarly to S-EXISTSLOWER.

In our example in Figure 5.10, S-LETPOP$\star$ is applied to state s_{11}, resulting in s_{12}. When applying the rule we have that $\overline{a'}$ contains b_3 and b_4, while $\overline{a''}$ contains b_5, due to $\theta_{11}(b_1) = b_5$. Further, A is $b_4 \to b_5$ and $\overline{c}$ is just b_4. This leads to the overall in-progress constraint **def** $(f : \forall b_4.b_4 \to b_5)$ **in** C'''_{ex} in the subsequent state s_{12}. The rule also appends a binding for b_5 to the end of the resulting stack, ensuring that the variable is bound by the stack when appearing freely in the type annotation on f in the def constraint.

Substituting new type variables with old ones Note that in the step from s_{11} to s_{12} in Figure 5.10, the need to keep the variable b_5 and lower it in the stack is due to the unification of b_5 and b_1 discussed earlier when detailing the solver steps that led from s_{10} to s_{11}. Here, when solving $b_1 \sim b_5$, the "older" (i.e., bound lower in the stack) variable b_1 was substituted with the "younger" variable b_5, simply because b_1 appears on the left-hand side of the equality constraint. This led to the mapping $b_1 \mapsto b_5$ in θ_{11} (defined in Figure 5.10).

Had we substituted b_5 with b_1 instead, we could have dropped b_5 from the stack, rather than lowering it. An optimisation that prefers substituting newer variables with older ones could easily be implemented in our solver using *ranks*, whose usage we discuss in Section 5.3.3. This potentially leads to fewer variables needing to be preserved during the lowering of existentials.

Purely coincidentally, the variable b_5 is later dropped when s_{12} is further processed, solving the constraint $f \preceq (\mathsf{Int} \to b_2)$. While doing so, b_5 is unified with b_2, which is already known to be equal to b_1 at that point, eventually leading to the state s_{13}. This state and its successor s_{14} was already shown earlier in Figure 5.9, where s_{14} concluded the solving of C_{ex}.

5.3.2.6 Non-generalising let constraints

For non-generalising let constraints, a stack frame is added as described at the beginning of Section 5.3.2.5. When removing this frame once the first sub-constraint of the let constraint is solved, the rule S-LETPOP$\bullet$ acts mostly analogously with S-LETPOP$\star$. In particular, the sequences $\overline{a'}, \overline{a''}$, and $\overline{c}$ as well as the type A are defined as in the latter rule. The only difference is that rather than universally quantifying the variables in $\overline{c}$ when creating the type annotation for x, the type A is used unchanged, meaning that the variables $\overline{c}$ appear freely in it. As a result, the bindings for the variables $\overline{c}$ must be preserved in the successor state.

Like the other free flexible variables in the annotation, the variables $\overline{c}$ are then monomorphised when the def constraint is processed.

As an example, we consider the variant $\hat{C}_{\text{ex}}$ of the constraint C_{ex}, as defined on page 120, that uses a non-generalising let constraint rather than a generalising one. We observe that this constraint is in fact logically equivalent to the original one: When using a non-generalising let constraint, f will obtain the type $b_4 \to b_5$, where b_4 appears freely and is restricted to monomorphic types, rather than being universally quantified, as in the original version. This means that when encountering the sub-constraint $f \prec \mathsf{Int} \to b_5$, no instantiation takes place, but b_4 is unified with Int. This has no further consequences, and b_5 is unified with b_2, as it is when solving the original constraint.

The solver operates on this variant of C_{ex} similarly as on the original one. The steps from s_7 to s_{11}, shown in Figures 5.9 and 5.10, are performed identically, except for the let stack frame being marked with $\bullet$ instead. The resulting state s'_{11} is shown in Figure 5.11 on the following page. Note that the sub-constraints C''_{ex} and C'''_{ex} remain unchanged as they appear in C_{ex} and are shown on page 120. Applying S-LetPop$\bullet$ then yields the state s'_{12}. Note that in contrast to the step from s_{11} to s_{12}, the variable b_4 has been lowered in s'_{12} rather than being removed. This is necessary due to b_4 appearing *freely* in the type of f this time.

Subsequent solving then leads to the same state s_{13} and final state s_{14} shown in Figure 5.9 as for the original constraint C_{ex}.

5.3.3 Relationship between partition function and ranks

In the original Algorithm W, the determination of what type variables to generalise at let bindings is based on whether or not a variable appears in the result of applying the current substitution to the term context. In our type inference algorithms, similar checks are performed.

- In our variant of Algorithm W in Figure 4.5 on page 86, defining $\Delta' = \mathsf{ftv}(\theta_1) - \Delta$ when handling let bindings performs a similar check.

- In our constraint solver, using $\mathsf{partition}(\overline{a}, \theta, \Theta)$ splits $\overline{a}$ based on their appearance in the codomain of θ.

A more efficient technique that avoids searching the codomain of a substitution

$$(\exists a, (a:\star), [a \mapsto a], \overbrace{\exists b_1\, b_2 . b_1 \to b_2 \sim a \wedge \mathbf{def}\,(x:b_1)\ \mathbf{in\ let_\bullet} f = \sqcap b_3 . C''_{\text{ex}}\ \mathbf{in}\ C'''_{\text{ex}}}^{=\hat{C}_{\text{ex}}})$$

$$\to^{+} (F_9 :: \mathbf{let_\bullet}\ f = \sqcap b_3.\square\ \mathbf{in}\ C'''_{\text{ex}} :: \exists b_4\, b_5,\ \Theta'_{11},\ \theta'_{11},\ \mathsf{true}) \qquad =: s'_{11}$$

$$\xrightarrow{\text{S-LetPop}\bullet} (F_9 :: \exists b_4 b_5,\ \Theta'_{12},\ \theta'_{12},\ \mathbf{def}\,(f : b_4 \to b_5)\ \mathbf{in}\ \overbrace{f \preceq (\mathsf{Int} \to b_2)}^{=C'''_{\text{ex}}}) \qquad =: s'_{12}$$

$$\to^{+} (\exists a\, b_1\, b_2 :: \mathbf{def}\,(x:b_1), \Theta_{13}, [a \mapsto (b_2 \mapsto b_2), b_1 \mapsto b_2, b_2 \mapsto b_2],\ \mathsf{true}) \qquad = s_{13}$$

$$\xrightarrow{\text{S-DefPop}} s_{14}$$

where

$$\begin{aligned}
F_9 &= \exists a\, b_1\, b_2 :: \mathbf{def}\,(x:b_1) & & \\
\Theta'_{11} &= (\Theta'_{12}, b_3 : \star) & \theta'_{11} &= \theta'_{12}[b_3 \mapsto (b_4 \to b_5)] \\
\Theta'_{12} &= (a:\star, b_1:\bullet, b_2:\star, b_4:\bullet, b_5:\bullet) & \theta'_{12} &= [a \mapsto (b_5 \to b_2), b_1 \mapsto b_5, \ldots] \\
\Theta_{13} &= (a:\star, b_1:\bullet, b_2:\bullet) & &
\end{aligned}$$

Figure 5.11: Solving variant $\hat{C}_{\text{ex}}$ of C_{ex} with non-generalising let constraint

function is the usage of *ranks*[6]. During inference, an integer index is associated with each type variable a, indicating the outermost term variable binder whose type mentions a. The decision whether a type variable can be generalised can then be made solely by inspecting its rank. Different variants of the approach exist, varying in terms of which term variable binders (i.e., lambda or let binders) are considered and how ranks are initialised [99, 74, 50].

While we eschew a full complexity analysis of our constraint solver, we now show how the decisions about what type variables to generalise made in our solver can directly be related to approaches using ranks. More concretely, we introduce a notion of ranks for type variables in our system and show how we may re-define the helper function partition to utilise them.

To this end, we define the function atv that returns a sequence of all type variables (flexible and rigid) bound by the frames of a stack. Here, we only consider well-formed stacks, guaranteeing that all such variables are pairwise different.

[6]This usage of the term *rank* is unrelated to the notion of ranks in the sense of higher-rank polymorphism [61].

$$\mathsf{atv}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{fc}(F'), a & \text{if } F = F' :: \exists a \qquad\qquad\qquad \text{or} \\ & \quad F = F' :: \mathbf{let}_R\, x = \sqcap a.\square \ \mathbf{in}\ C_2 \ \text{ or} \\ & \quad F = F' :: \forall a \\ \mathsf{fc}(F') & \text{otherwise } (F = F' :: \_) \end{cases}$$

Therefore, if $\mathsf{atv}(F) = (a_1, \dots, a_n)$, then a_1 is the outermost type variable bound in F and $\mathsf{atv}(F)$ is an interleaving of $\mathsf{rc}(F)$ and $\mathsf{fc}(F)$, as defined earlier in Figure 5.6.

Let a stack F with $\mathsf{atv}(F) = (a_1, \dots, a_n)$, a variable $b \in \mathsf{atv}(F)$, and a substitution θ with $\mathsf{rc}(F) \vdash \theta : \Theta \Rightarrow \Theta$ be given, where $\mathsf{ftv}(\Theta) = \mathsf{fc}(F)$. We may then define the *index* of $a_i \in \mathsf{atv}(F)$ as its (1-based) index i in $\mathsf{atv}(F)$ and its *rank* as the smallest index of any variable b such that a_i appears in $\mathsf{ftv}(\theta(b))$, or ∞ if $b \notin \mathsf{ftv}(\theta)$.

$$\begin{array}{lcl} \mathsf{index}(b, F) & = & i \in \{1, n\} \text{ such that } a_i = b \\ \mathsf{rank}(b, \theta, F) & = & \begin{cases} \min_{i \in \{1,\dots,n \,|\, b \in \mathsf{ftv}(\theta(a_i))\}} i & \text{if such an } i \text{ exists} \\ \infty & \text{otherwise} \end{cases} \end{array}$$

For convenience, we also define $\mathsf{index}(F) = n$ (i.e., the length of $\mathsf{atv}(F)$).

Note that in contrast to the usual notion of ranks, ours indexes binders of type variables, rather than term variables. Further, we have $\theta(c) = c$ for all rigid variables (i.e., $c \in \mathsf{rc}(F)$). This means that given a rigid variable c with index i, no variable other than c may have rank i. However, the rank of a rigid variable is not necessarily equal to its index, if it appears in the mapping of another existential. We are interested in these cases for the purposes of detecting escaping quantifiers.

Finally, we observe that those variables a with rank ∞ are those that are *eliminated* by θ, meaning that $\theta(a) = A$ where $a \neq A$. Note that our idempotency condition on substitutions in well formed states (see Section 5.3.1.3) then imposes $a \notin \mathsf{ftv}(\theta)$.

We can now relate ranks to the behaviour of the partition function introduced in Section 5.3.2.2. We assume that there exists a stack $F = F_1 :: F_2$ such that $\Theta \vdash F$ **ok** and $\Xi = \mathsf{fc}(F_2)$, meaning that Θ contains exactly the flexible variables bound by F and F_2 is a suffix of F binding exactly the flexible variables in Ξ (and possibly other rigid ones). We then observe that the following, alternative definition of partition yields the same results.

$$\begin{aligned}&\mathsf{partition}'(\Xi,\theta,F_1,F_2) = \Xi_g;\Xi_l \text{ where}\\&\quad\Xi_g = \{a \in \Xi \mid \mathsf{index}(F_1)+1 \leq \mathsf{rank}(a,\theta,F)\} \text{ and}\\&\quad\Xi_l \ = \{a \in \Xi \mid \mathsf{rank}(a,\theta,F) < \mathsf{index}(F_1)+1\}\end{aligned}$$

We can use this alternative version of partition directly in Figure 5.8, taking $F_1 := F :: f$ and $F_2 := \exists\overline{a}$ in S-EXISTSLOWER as well as $F_1 := F$ and $F_2 := \mathbf{let}_R\ x = \sqcap b.\square\ \mathbf{in}\ C :: \exists\overline{a}$ in both S-LETPOP rules. Note that we are adding 1 to $\mathsf{index}(F_1)$ in the definition of Ξ_g and Ξ_l to obtain the index in the combined stack $F_1 :: F_2$ of the first type variable bound by F_2. Therefore, when used by S-EXISTSLOWER, we have that $\mathsf{index}(F_1)+1$ is the index of the first variable from the sequence $\overline{a}$ in the overall stack. When used by the S-LETPOP rules, $\mathsf{index}(F_1)+1$ is the index of the variable b bound by the let frame under consideration. Using ranks, we can then perform the escape check for the universally quantified variable a in S-FORALLPOP by verifying that $\mathsf{rank}(a,\theta,F :: \forall a) = \mathsf{index}(F)+1$ holds.

We can formalise the correctness of this alternative version of partition as follows. Since neither version makes any guarantees about the order of variables within the two returned sequences, we simply state that the corresponding sequences returned by each function are permutations of each other.

Lemma 5.5 (Correspondence of partition and partition′)**.**
*Let $F = F_1 :: F_2$ and $\Theta \vdash F$ **ok** and $\mathsf{rc}(F) \vdash \theta : \Theta \Rightarrow \Theta$ and $\Xi \approx \mathsf{fc}(F_2)$. Further, let the following hold.*

$$\begin{aligned}\mathsf{partition}(\Xi,\theta,\Theta) &= \Xi_1;\Xi_2\\ \mathsf{partition}'(\Xi,\theta,F_1,F_2) &= \Xi_1';\Xi_2'\end{aligned}$$

Then we have $\Xi_1 \approx \Xi_1'$ and $\Xi_2 \approx \Xi_2'$.

Proof. We first observe that the definitions of both helper functions immediately guarantee that they return a *partitioning* of the input sequence Ξ. Now, let $\mathsf{atv}(F_1) = (b_1,\dots,b_n)$. It then suffices to show that Ξ_2 and Ξ_2' contain the same variables.

$$\begin{aligned}&a \in \Xi_2\\ &\qquad\qquad\qquad\qquad\text{(by definition of partition)}\\ \text{iff}\quad &\text{exists } b \in \Theta - \Xi \text{ s.t. } a \in \mathsf{ftv}(\theta(b))\\ &\qquad\qquad\qquad\qquad\text{(by } \mathsf{ftv}(\Theta) - \Xi \subseteq \mathsf{atv}(F_1) \text{ and}\\ &\qquad\qquad\qquad\qquad\theta(c) = c \neq a \text{ for all } c \in \mathsf{atv}(F_1) - \mathsf{ftv}(\Theta))\\ \text{iff}\quad &\text{exists } i \in \{1,n\} \text{ s.t. } a \in \mathsf{ftv}(\theta(b_i))\end{aligned}$$

(by definition of rank)

iff $\mathsf{rank}(a, \theta, F_1) \leq n$

(by F_1 being prefix of F, $\mathsf{index}(F_1)$ = n)

iff $\mathsf{rank}(a, \theta, F) \leq n$

(by definition of $\mathsf{partition}'$)

iff $a \in \Xi'_2$

□

Recall that the first set returned by partition are those variables that we *may* generalise in S-LETPOP⋆ in Figure 5.8 when determining the type of the let-bound variable x. However, we must only generalise those variables that actually appear in the type A, which is defined as $\theta(b)$ in the rule. Further, due to FreezeML adopting the same equational theory on types as System F, the variables must be quantified in the order in which they appear in A.

Using ranks, we can perform the first step directly within (a redefined version of) the partition function, returning only variables appearing in $A = \theta(a)$, by refining the $\mathsf{partition}'$ function as follows:

$$\begin{aligned}
&\mathsf{partition}''(\Xi, \theta, F_1, F_2) = \Xi_g; \Xi_l \ \text{ where} \\
&\quad \Xi_g = \{a \in \Xi \mid \mathsf{index}(F_1) + 1 = \mathsf{rank}(a, \theta, F)\} \text{ and} \\
&\quad \Xi_l \ = \{a \in \Xi \mid \mathsf{rank}(a, \theta, F) < \mathsf{index}(F_1) + 1\}
\end{aligned}$$

The only difference as compared to $\mathsf{partition}'$ is that we require equality in the rank check for Ξ_g, thus discarding the variables related to any of the variables in $\overline{a}$, but not A. Note that in a solver for a variant of FreezeML where the order of quantifiers does *not* matter, further discussed in Section 6.2.1.1, we may generalise the first set returned by this function directly.

5.3.3.1 Practical concerns

So far, we have discussed how to re-define the partition function used by our constraint solver in a way that relies on ranks instead of inspecting the codomain of θ. Of course, instead of re-calculating the rank of type variables whenever needed, the efficiency benefits of the rank-based approach stem from maintaining the ranks of all type variables during constraint solving and updating them individually as needed. This could be achieved in the standard way in our system. If the unification algorithm detects that some flexible variable a is to be unified with some type A, then the ranks of all variables in A are lowered to the minimum of their current rank and the index of a. Of course, ranks would also need to be updated when lowering existentials in the stack.

Note that an alternative definition of rank-based generalisation may group existentials together with the next enclosing $\forall$ or let frame, if such a frame exists. The choice for defining ranks and indices in a more fine-grained manner in this section was mostly to facilitate a simpler comparison with the existing definition of partition. We conjecture that using a more traditional approach where ranks refer to the indices of let frames rather than type variable binders would similarly be possible. Regardless, our definition of ranks simply increases the range of possible indices used as ranks, but does not otherwise negatively affect the efficiency of the rank-based approach.

5.4 Metatheory of the constraint solver

We now discuss the correctness of our solver and the constraint-type inference approach as a whole.

In Section 5.3.1.2 we described how a solver state (F, Θ, θ, C) may be interpreted as a more structured representation of the constraint $F[\mathfrak{U}(\Theta, \theta) \wedge C]$. Here, $\mathfrak{U}(\Theta, \theta)$ contains mono and equality constraints induced by Θ and θ.

An important correctness property of a constraint solver is that whenever it takes a step from some (well formed) state s_0 to s_1, then the constraint representations of both states should be logically equivalent. For example, such a preservation result is given for the HM(X) solver by Pottier and Rémy [97, Lemma 10.6.9].

However, the semantics of let constraints in our language require additional restrictions when stating such a property for our system. To motivate the need for these restrictions, we first make a general observation in Section 5.4.1 about let constraints in our system. We then state the preservation property in Section 5.4.2. We outline difficulties encountered when proving said property, which is why it remains as a conjecture.

In Sections 5.4.3 and 5.4.4 we then state the overall correctness of the solver and constraint-based inference approach, respectively, contingent on the preservation conjecture stated in Section 5.4.2.

5.4.1 Hoisting sub-constraints out of let constraints

Recall that in our system, constraints **let** $x = \sqcap b.C_1$ **in** C_2 require reasoning about the most general solution of C_1. We now discuss how this prevents some transformations on let constraints that would preserve logical equvialence in systems without this requirement for most general solutions.

Assuming $b \notin \mathsf{ftv}(C'_1)$, we observe that the following two constraints C and C' are *not* logically equivalent in general, irrespective of the choice of R.

$$\begin{aligned} C &:= \textbf{let}_R\, x = \sqcap b.(C_1 \wedge C'_1) \textbf{ in } C_2 \\ C' &:= C'_1 \wedge (\textbf{let}_R\, x = \sqcap b.C_1 \textbf{ in } C_2) \end{aligned}$$

This is due to the monomorphism conditions imposed on certain type variables by the semantics of let constraints (see rules SEM-LETGEN and SEM-LETNONGEN in Figure 5.1 on page 97): Given a most general solution δ_m of the first sub-constraint of a let constraint (such as $C_1 \wedge C'_1$ in the case of C) the semantics require *monomorphic* interpretations for all type variables that appear in the type $\delta_\mathrm{m}(b)$ and are bound by the surrounding context. As a result, satisfiability can differ based on whether a polymorphic solution for a type variable is imposed by a sub-constraint of the let constraint itself or outside of the let constraint.

To illustrate this, we consider two concrete examples for C and C', named C_h and C'_h. Their sub-constraints C_1 and C'_1 also shown. The choice of the sub-constraint named C_2 earlier is irrelevant, we pick true here.

$$\begin{aligned} C_\mathrm{h} &:= \exists a. \quad \textbf{let}_\star\, x = \sqcap b.(\overbrace{b \sim a}^{=:C_1} \;\wedge\; \overbrace{a \sim \forall c.c}^{=:C'_1}) \textbf{ in } \mathsf{true} \\ C'_\mathrm{h} &:= \exists a.\underbrace{(a \sim \forall c.c)}_{=C'_1} \wedge \textbf{let}_\star\, x = \sqcap b.\ \underbrace{b \sim a}_{=C_1} \quad \textbf{in } \mathsf{true} \end{aligned}$$

Here, hoisting the sub-constraint C'_1 out of the let constraint turns the satisfiable constraint C_h into the unsatisfiable constraint C'_h. We now outline the technical reasons for the satisfiability of C_h and the unsatisfiability of C'_h.

Constraint C_h is satisfiable We first consider the *most general* solution δ_m of the constraint $b \sim a \;\wedge\; a \sim \forall c.c$ (i.e., $C_1 \wedge C'_2$). Clearly, it simply maps a and b to $\forall c.c$.

We formalise this using the mostgen relation, assuming empty type and term contexts, and that no flexible variables except a and b are in scope.

$$\mathsf{mostgen}(\cdot, (a,b), \cdot, \overbrace{b \sim a}^{=C_1} \wedge \overbrace{a \sim \forall c.c}^{=C'_1}, \cdot, \overbrace{[a \mapsto \forall c.c, b \mapsto \forall c.c]}^{=:\delta_\mathrm{m}}) \tag{5.8}$$

We now consider the derivation of $\Delta;\Xi;\Gamma;\delta \vdash \mathbf{let}_\star\ x = \sqcap b.C_1 \wedge C'_1\ \mathbf{in}\ \mathsf{true}$ (i.e, satisfaction of the let sub-constraint of C_h), again assuming that Δ and Γ are empty, while Ξ contains only a, the type variable bound by the surrounding existential. The derivation then has the following form, relying on the earlier mostgen judgement.

$$\frac{\begin{array}{c}\mathsf{mostgen}(\cdot, (a,b), \cdot, C_1 \wedge C'_1, \cdot, \delta_\mathrm{m}) \\ \Delta_\mathrm{o} = \mathsf{ftv}(\delta_\mathrm{m}(b)) \qquad \overline{b} = \mathsf{ftv}(\delta_\mathrm{m}(b)) - \Delta, \Delta_\mathrm{o} \\ \Delta \vdash \delta' : \Delta_\mathrm{o} \Rightarrow_\bullet \cdot \qquad A = \delta'(\delta_\mathrm{m}(b)) \\ \cdot;(a,b);\cdot;\delta[b \mapsto A] \vdash C_1 \wedge C'_1 \qquad \cdot;a;(x : \forall \overline{b}.A);\delta \vdash \mathsf{true}\end{array}}{\cdot;a;\Gamma;\delta \vdash \mathbf{let}_\star\ x = \sqcap b.C_1 \wedge C'_1\ \mathbf{in}\ \mathsf{true}}$$

Here, Δ_o and $\overline{b}$ are empty, meaning that δ' is the empty substitution, and A is $\forall c.c$. In other words, the monomorphisation mechanism built into let constraints in our system, implemented by the monomorphic substitution δ' in the derivation above, has no effect in this example. We have thus shown that the constraint C_h is indeed satisfiable.

Constraint C'_h is unsatisfiable We now show that the constraint C'_h is unsatisfiable. Recall that it is defined as $\exists a.a \sim \forall c.c \wedge \mathbf{let}_\star\ x = \sqcap b.a \sim b\ \mathbf{in}\ \mathsf{true}$. We again consider the most general solution of the first sub-constraint of the let constraint, which is $a \sim b$ this time (i.e., just C_1 instead of $C_1 \wedge C'_1$ in the case of C_h). Formally, we have the following, expressing that the most general solution δ'_m of C_1 maps both variables a and b to a fresh skolem variable d.

$$\mathsf{mostgen}(\cdot, (a,b), \cdot, \overbrace{b \sim a}^{=C_1}, \overbrace{\ d\ }^{\Delta'_\mathrm{m}}, \overbrace{[a \mapsto d, b \mapsto d]}^{=:\delta'_\mathrm{m}}) \tag{5.9}$$

Note that any derivation of $\cdot;\cdot;\cdot;\delta \vdash C'_\mathrm{h}$ would require a sub-derivation for $\cdot;a;\cdot;[a \mapsto \forall c.c] \vdash \mathbf{let}_\star\ x = \sqcap b.C_1\ \mathbf{in}\ \mathsf{true}$. Any other interpretation for a would violate the conjunct $a \sim \forall c.c$.

To derive the latter (i.e., $\cdot;a;\cdot;[a \mapsto \forall c.c] \vdash \mathbf{let}_\star\ x = \sqcap b.C_1\ \mathbf{in}\ \mathsf{true}$), the following additional premises would need to be satisfied, somewhat analogous to the derivation for C_h above.

$$\Delta'_\mathrm{o} = \mathsf{ftv}(\delta'_\mathrm{m}(b)) = d \qquad \overline{b'} = \mathsf{ftv}(\delta'_\mathrm{m}(b)) - \Delta, \Delta_\mathrm{o} = \cdot \qquad A' = \delta''(\delta'_\mathrm{m}(b))$$

$$\Delta \vdash \delta'' : \Delta'_{\mathrm{o}} \Rightarrow_{\bullet} \cdot \tag{5.10}$$

$$\cdot; (a,b); \cdot; [a \mapsto \forall c.c, b \mapsto \forall \overline{b'}.A'] \vdash a \sim b \tag{5.11}$$

However, these conditions are unsatisfiable. As shown in (5.9), we have $\delta'_{\mathrm{m}}(b) = d$. Further, the domain Δ'_{o} of δ'' is just d. The only interpretation of b that can lead to satisfaction of (5.11) is $\forall c.c$, meaning that δ'' must map d to $\forall c.c$, so that $\forall \overline{b'}.A'$ becomes $\forall c.c$. However, (5.10) imposes that δ'' must be a *monomorphic* instantiation, ruling this choice out. Thus, no such δ'' exists, and the overall constraint C'_{h} is unsatisfiable.

In total, we observe that hoisting the sub-constraint C'_1 out of the first sub-constraint of the let binding has changed the most general solution of the first sub-constraint, as shown in the difference between (5.8) and (5.9). In C_{h}, the polymorphic solution is imposed within the let binding itself, and therefore reflected in the most general type of the first sub-constraint. After hoisting the constraint out of the let binding, leading to the constraint C'_{h}, the most general solution does not involve any polymorphism at all anymore. The monomorphisation mechanism built into let constraints – required to have any hope to guarantee the existence of most general solutions for the constraint language – then rules out the polymorphic instantiation that would yield the required type.

5.4.2 Preservation property

As mentioned in the beginning of Section 5.4, our goal is to state a property such as the following: If state $(F_0, \Theta_0, \theta_0, C_0)$ steps to some state $(F_1, \Theta_1, \theta_1, C_1)$, then the constraints $F_0[C_0 \wedge \mathfrak{U}(\Theta_0, \theta_0)]$ and $F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$ should be logically equivalent.

In the previous section, we observed that hoisting some sub-constraints out of let bindings may make them unsatisfiable. We now relate this observation about let bindings to the desired preservation property: Our solver may take steps which effectively transform the constraint representation of the involved states similarly to the hoisting transformation discussed in the previous section. Therefore, additional measures are required to ensure that these steps of the solver do not invalidate the preservation property.

As a concrete example, let s_0 be a state with the following components.

$$\begin{aligned} F_0 &= \exists a \\ \Theta_0 &= (a : \star) \\ \theta_0 &= [a \mapsto \forall c.c] \\ C_0 &= \textbf{let}_\star\, x = \sqcap b.b \sim a \textbf{ in } \mathsf{true} \end{aligned}$$

Applying the solver rule S-LetPush (see Figure 5.8 on page 115) to this state yields the successor state $s_1 = (F_1, \Theta_1, \theta_1, C_1)$ with the following components.

$$\begin{array}{rcl} F_1 & = & \exists a :: \mathbf{let}_\star\, x = \sqcap b.\square \ \mathbf{in}\ \mathsf{true} \\ \Theta_1 & = & (a : \star, b : \star) \\ \theta_1 & = & [a \mapsto \forall c.c, b \mapsto b] \\ C_1 & = & b \sim a \end{array}$$

Note that the constraint representation $F_0[C_0 \wedge \mathfrak{U}(\Theta_0, \theta_0)]$ of s_0 is $\exists a.(\mathbf{let}_\star\, x = \sqcap b.b \sim a\ \mathbf{in}\ \mathsf{true}) \wedge a \sim \forall c.c$, which is equivalent to the constraint C'_h discussed in Section 5.4.1, up to reordering of conjuncts. Similarly, the constraint representation of s_1 is identical to the constraint $\exists a.\mathbf{let}_\star\, x = \sqcap b.b \sim a \wedge a \sim \forall c.c\ \mathbf{in}\ \mathsf{true}$, which was named C_h in Section 5.4.1. We thus observe that the solver has effectively performed a lowering step (i.e., the reverse of hoisting), moving a constraint from outside of a let constraint inside of it. As discussed in Section 5.4.1, only the constraint C_h is satisfiable, while C'_h is not. The solver has thus turned a state with an unsatisfiable constraint representation into a constraint with a satisfiable one, violating the preservation property stated in the beginning of this section.

However, we observe that a state like s_0 never occurs while solving a constraint obtained from a FreezeML program using the translation defined in Section 5.2. Whenever this translation yields a constraint $\mathbf{let}_R\, x = \sqcap b.C_1\ \mathbf{in}\ C_2$, we have that the only free flexible type variable that C_1 may contain is b. We introduce the judgement $\Delta \vdash^{\mathrm{ftv}} C\ \mathbf{ok}$ to denote that this condition holds for a constraint C and its sub-constraints. As usual, Δ tracks the rigid variables in scope. It is formalised in Figure 5.12 on the following page. The only non-trivial rules are those for $\forall$ constraints, which bring additional rigid variables into scope, and the rule for let constraints, where the free flexible type variables of the first sub-constraint are inspected.

For all states (F, Θ, θ, C) encountered while solving constraints obtained through the translation from FreezeML terms, we have $\mathsf{rc}(F) \vdash^{\mathrm{ftv}} C\ \mathbf{ok}$. Recall that $\mathsf{rc}(F)$ denotes the variables bound by $\forall$ frames in F, thus binding all rigid variables that are in scope in C. We will revisit this invariant of the constraint solver and how it is established by the constraint translation in Sections 5.4.3 and 5.4.4.

Note we deliberately keep the additional relation $\Delta \vdash^{\mathrm{ftv}} C\ \mathbf{ok}$ on constraints separate from the existing well-formedness relation $\Delta; \Xi; \Gamma \vdash^{\mathrm{ftv}} C\ \mathbf{ok}$ for constraints (see Figure 5.3 on page 104). The former relation is used to restrict the *constraint components* of the states encountered by the solver. However, our we must still reason about con-

$\boxed{\Delta \vdash^{\mathsf{ftv}} C\ \mathbf{ok}}$

$$\frac{}{\Delta \vdash^{\mathsf{ftv}} \mathsf{true}\ \mathbf{ok}} \qquad \frac{}{\Delta \vdash^{\mathsf{ftv}} \mathsf{mono}(a)\ \mathbf{ok}} \qquad \frac{}{\Delta \vdash^{\mathsf{ftv}} A \sim B\ \mathbf{ok}}$$

$$\frac{}{\Delta \vdash^{\mathsf{ftv}} x \preceq A\ \mathbf{ok}} \qquad \frac{}{\Delta \vdash^{\mathsf{ftv}} \lceil x : A \rceil\ \mathbf{ok}}$$

$$\frac{\Delta \vdash^{\mathsf{ftv}} C_1\ \mathbf{ok} \qquad \Delta \vdash^{\mathsf{ftv}} C_2\ \mathbf{ok}}{\Delta \vdash^{\mathsf{ftv}} C_1 \wedge C_2\ \mathbf{ok}} \qquad \frac{\Delta \vdash^{\mathsf{ftv}} C\ \mathbf{ok}}{\Delta \vdash^{\mathsf{ftv}} \mathbf{def}\ (x : A)\ \mathbf{in}\ C\ \mathbf{ok}} \qquad \frac{\Delta \vdash^{\mathsf{ftv}} C\ \mathbf{ok}}{\Delta \vdash^{\mathsf{ftv}} \exists a.C\ \mathbf{ok}}$$

$$\frac{(\Delta, a) \vdash^{\mathsf{ftv}} C\ \mathbf{ok}}{\Delta \vdash^{\mathsf{ftv}} \forall a.C\ \mathbf{ok}} \qquad \frac{\mathsf{ftv}(C_1) - \Delta \subseteq \{a\} \qquad \Delta \vdash^{\mathsf{ftv}} C_1\ \mathbf{ok} \qquad \Delta \vdash^{\mathsf{ftv}} C_2\ \mathbf{ok}}{\Delta \vdash^{\mathsf{ftv}} \mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2\ \mathbf{ok}}$$

Figure 5.12: Definition of $\Delta \vdash^{\mathsf{ftv}} C\ \mathbf{ok}$

straints for which the $\vdash^{\mathsf{ftv}}$ condition does not hold. In particular, the overall constraint representation $F[C \wedge \mathfrak{U}(\Theta, \theta)]$ of a state (F, Θ, θ, C) need not satisfy this condition.

We now return to stating a preservation property for our solver. The following property is analogous to the one described at the beginning of this section, but incorporates the assumption that $\mathsf{rc}(F') \vdash^{\mathsf{ftv}} C\ \mathbf{ok}$ holds for the constraint component of the initial state, where F' is its stack component. The property also assumes that there exist some initial contexts Δ and Ξ, represented by the outermost frames of both states. These initial contexts and the interpretation of the variables in Ξ are used to reason about the overall correctness of the solver in Section 5.4.3.

Conjecture 5.1 (Preservation)**.**
Let $\vdash (\forall\Delta :: \exists\Xi :: F_0, \Theta_0, \theta_0, C_0)\ \mathbf{ok}$ *and* $\mathsf{rc}(\forall\Delta :: \exists\Xi :: F_0) \vdash^{\mathit{ftv}} C\ \mathbf{ok}$ *and*

$$(\forall\Delta :: \exists\Xi :: F_0, \Theta_0, \theta_0, C_0) \to (\forall\Delta :: \exists\Xi :: F_1, \Theta_1, \theta_1, C_1)$$

hold. Then we have that

$$\Delta; \Xi; \cdot; \delta \vdash F_0[C_0 \wedge \mathfrak{U}(\Theta_0, \theta_0)] \textit{ iff } \Delta; \Xi; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$$

holds.

While the author conjectures that this property holds, it does not currently have a proof. The remainder of this subsection is structured as follows: In Section 5.4.2.1, we outline a potential proof of Conjecture 5.1. It relies on two auxiliary properties, namely Conjectures 5.2 and 5.3, which would allow this proof of Conjecture 5.1 to be completed. Section 5.4.2.2 focuses on Conjecture 5.2, explaining in detail the challenges encountered when attempting to prove it. The issues affecting Conjecture 5.3 are similar. We then summarise the situation regarding the lacking proof of Conjecture 5.1 in Section 5.4.2.3.

5.4.2.1 Towards a proof of Conjecture 5.1

Before discussing the difficulties encountered while attempting to find a proof of Conjecture 5.1, we observe the following, unsurprising property, which may be used in a potential proof of Conjecture 5.1.

Intuitively, it states that given two logically equivalent constraints C and C', we have that if C appears as a sub-constraint of some C'', then replacing C with C' within C'' yields a constraint logically equivalent to C''. Instead of using some kind of substitution operator, we state the property by plugging constraints into stacks.

Lemma 5.6 (Substituting logically equivalent constraints)**.**
Let C and C' be logically equivalent. Then $F[C]$ and $F[C']$ are logically equivalent.

Proof. By induction on the structure of F. In the case of a let frame, we observe that the definition of mostgen directly implies that for all $\Delta, \Xi, \Gamma, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}}$ we have $\mathsf{mostgen}(\Delta, \Xi, \Gamma, C, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$ iff $\mathsf{mostgen}(\Delta, \Xi, \Gamma, C', \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$. □

We now return to discussing a potential proof of Conjecture 5.1. Since the second state in the statement of Conjecture 5.1 is obtained by applying one of the rules in Figure 5.8, we may expect a successful proof to perform a case analysis on the used rule. We will organise the rules into two groups. For the first group, the statement of Conjecture 5.1 can be shown to hold; we outline the necessary steps. We then consider the second group, comprising two rules, and discuss the problems encountered while attempting to prove their correctness, which ultimately prevents the proof of Conjecture 5.1 from being completed.

First group of rules The first group consists of all rules except S-FREEZE and S-INST. Recall that we are given Δ, Ξ, δ as well as states $s_0 = (F_0, \Theta_0, \theta_0, C_0)$ and $s_1 = (F_1, \Theta_1, \theta_1, C_1)$ and need to show that $\Delta; \Xi; \cdot; \delta \vdash F_0[C_0 \wedge \mathfrak{U}(\Theta_0, \theta_0)]$ iff $\Delta; \Xi; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$ holds.

For each rule in the first group, we apply the following proof strategy: We identify a shared prefix F_{p} of F_0 and F_1, which induces F_0' and F_1' such that $F_0 = F_{\mathrm{p}} :: F_0'$ and $F_1 = F_{\mathrm{p}} :: F_1'$. Showing $\Delta; \Xi; \cdot; \delta \vdash F_0[C_0 \wedge \mathfrak{U}(\Theta_0, \theta_0)]$ iff $\Delta; \Xi; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$

then becomes equivalent to showing the following:

$$\Delta;\Xi;\cdot;\delta \vdash F_{\mathrm{p}}[F_0'[C_0 \wedge \mathfrak{U}(\Theta_0,\theta_0)]] \text{ iff } \Delta;\Xi;\cdot;\delta \vdash F_{\mathrm{p}}[F_1'[C_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)]]$$

Let C_0' and C_1' be defined as $F_0'[C_0 \wedge \mathfrak{U}(\Theta_0,\theta_0)]$ and $F_1'[C_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)]$, respectively. We then show that C_0' and C_1' are logically equivalent, which immediately yields the desired property for the rule under consideration by applying Lemma 5.6.

The prefix F_{p} can easily be read off from the corresponding rule directly, it generally corresponds to the part of the stack being left unchanged by the rule in question.

For some rules, proving the equvialence of the induced constraints C_0' and C_1' is trivial, since both constraints only differ slightly. For example, for rules CONJPUSH the prefix F_{p} is the stack called F in the definition of the rule itself, meaning that the only difference between the resulting constraints C_0' and C_1' is the order of some conjuncts.

Other cases rely on hoisting properties that do *not* involve let constraints: Assuming that f is not a let frame, and C' does not contain type or term variables bound by f towards its hole, we have that $f[C \wedge C']$ and $f[C] \wedge C'$ are logically equivalent.

The following cases are particularly interesting:

- In the case of S-EQ, we rely on the fact that our unification algorithm is sound and yields most general unifiers (see Theorems 4.2 and 4.3).

- In the case of S-LETPUSH, we have that C_0 is of the form $\mathbf{let}_R\ x = \sqcap b.\hat{C}_1\ \mathbf{in}\ \hat{C}_2$ for some constraints $\hat{C}_1$ and $\hat{C}_2$. We then need to show that the constraints $(\mathbf{let}_R\ x = \sqcap b.\hat{C}_1\ \mathbf{in}\ \hat{C}_2) \wedge \mathfrak{U}(\Theta_0,\theta_0)$ and $\mathbf{let}_R\ x = \sqcap b.\hat{C}_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)\ \mathbf{in}\ \hat{C}_2$ are logically equivalent.

  This crucially relies on the additional premise that $\mathsf{rc}(\forall\Delta :: \exists\Xi :: F_{\mathrm{p}}) \vdash^{\mathsf{ftv}} C_0\ \mathbf{ok}$ holds, meaning that we have $\mathsf{ftv}(\hat{C}_1) - (\Delta, \mathsf{rc}(F_{\mathrm{p}})) \subseteq \{b\}$ (i.e., the only flexible variable in $\hat{C}_1$ may be b), while $\mathfrak{U}(\Theta_0,\theta_0)$ cannot contain b.

- The rules S-LETPOP$\star$ and S-LETPOP$\bullet$ require relating let constraints to definition constraints, which involves reasoning about the most general solution of the first sub-constraint of the let constraint. While the cases are laborious, they only involve reasoning about the most general solutions of constraints of a simple, known shape, instead of reasoning about the most general solutions of arbitrary constraints: We only need to reason about the most general solution of constraints of the form $\exists\overline{a}.\mathfrak{U}(\Theta_0,\theta_0)$ (i.e., quantified conjunctions of mono and equality constraints), for which we can read off most general solutions immediately.

Second group of rules The second group of rules includes only S-FREEZE and S-INST. We first describe the issues faced in the proof case when S-FREEZE was used and briefly describe how similar issues are faced in the other case.

Thus, let F, Θ, and θ be defined as in the definition of rule S-FREEZE; we further have $C_0 = \lceil x : A \rceil$ and $C_1 = \mathsf{tc}(F)(x) \sim A$. We define $B := \mathsf{tc}(F)(x)$. We will attempt to apply the same proof strategy as for the first group of rules, that is, identifying a prefix F_p of the stacks of both states and then proving the equivalence of C'_0 and C'_1, which are induced by the choice of F_p in the same fashion as before.

Following the approach taken for the first group of rules, a natural choice for F_p would be F (i.e., the entire stack before and after the step). This choice induces $C'_0 = \lceil x : A \rceil \wedge \mathfrak{U}(\Theta,\theta)$ and $C'_1 = B \sim A \wedge \mathfrak{U}(\Theta,\theta)$. However, we observe that these constraints are not logically equivalent in general: As an example, assume $F = \exists b :: \mathbf{def}\,(x : b \to b)$ and $A = \mathsf{Int} \to \mathsf{Int}$, which implies $B = b \to b$. We assume that there is no further knowledge about b, except that it must be monomorphic since it appears in the type of x in a well-formed state (i.e., $\Theta = (b : \bullet)$ and $\theta = [b \mapsto b]$). The conjunctions with $\mathfrak{U}(\Theta,\theta) = \mathsf{mono}(b) \wedge b \sim b$ in both C'_0 and C'_1 are largely irrelevant for the remainder of this discussion.

We first observe that both $F[C'_1] = \exists b.\, \mathbf{def}\,(x : b \to b)$ **in** $(\lceil x : \mathsf{Int} \to \mathsf{Int} \rceil \wedge \mathfrak{U}(\Theta,\theta))$ and $F[C'_1] = \exists a.\, \mathbf{def}\,(x : b \to b)$ **in** $(a \to a \sim \mathsf{Int} \to \mathsf{Int} \wedge \mathfrak{U}(\Theta,\theta))$ are tautological constraints.

However, $\lceil x : \mathsf{Int} \to \mathsf{Int} \rceil \wedge \mathfrak{U}(\Theta,\theta)$ (i.e., C'_0) and $b \to b \sim \mathsf{Int} \to \mathsf{Int} \wedge \mathfrak{U}(\Theta,\theta)$ (i.e., C'_1) are *not* logically equivalent. Both constraints have different models: Constraint C'_0 is satisfiable if and only if the term context maps x to $\mathsf{Int} \to \mathsf{Int}$, while b may be interpreted with arbitrary monomorphic types. Conversely, C'_1 is satisfied by models that interpret b with $\mathsf{Int} \to \mathsf{Int}$, while the term context is irrelevant.

This shows that the *local* reasoning that was adequate for the first group of rules, where we only consider the constraints induced by just the tip of the stack of both states and their constraint component, does not work for S-FREEZE: When only comparing the constraints $\lceil x : \mathsf{Int} \to \mathsf{Int} \rceil \wedge \mathfrak{U}(\Theta,\theta)$ and $b \to b \sim \mathsf{Int} \to \mathsf{Int} \wedge \mathfrak{U}(\Theta,\theta)$ in isolation, the relationship between x and its supposed type $b \to b$ is lost.

We observe that whenever rule S-FREEZE applies, the stack F contains a definition frame binding x.[7] Formally, this means that F is of the form $F' :: \mathbf{def}\ (x : B) :: F''$ for some F' and F''. Thus, we may still apply the overall strategy described for the first group, but choose the prefix F_p to be F' instead. This induces $C'_0 = \mathbf{def}\ (x : B)\ \mathbf{in}\ F''[\lceil x : A \rceil \wedge \mathfrak{U}(\Theta, \theta)]$ and $C'_1 = \mathbf{def}\ (x : B)\ \mathbf{in}\ F''[B \sim A \wedge \mathfrak{U}(\Theta, \theta)]$, again letting $B := \mathsf{tc}(F)(x)$. We may thus prove these choices of C'_0 and C'_1 to be logically equivalent instead. As opposed to the previous, unsuitable choice of F_p, we effectively increased the amount of context preserved in C'_0 and C'_1, thus retaining the relationship between x and B. We may state the desired equivalence of C'_0 and C'_1 as the following, standalone property:

Conjecture 5.2 (Substituting freezing constraints)**.**
The constraints

$$\mathbf{def}\,(x : B)\ \mathbf{in}\ F[\,\lceil x : A \rceil\,]$$

and

$$\mathbf{def}\,(x : B)\ \mathbf{in}\ F[B \sim A]$$

are logically equivalent.

This property directly corresponds to the intuition that a constraint $\lceil x : A \rceil$ simply asserts that the type of x *is* B, subject to instantiation of the free flexible variables in A and B.

Note that in order to apply this property in a potential proof of Conjecture 5.1, we would choose the stack F in the statement of Conjecture 5.2 to be $F'' :: \square \wedge \mathfrak{U}(\Theta, \theta)$ (i.e., we may move $\mathfrak{U}(\Theta, \theta)$ from the hole to a new conjunction frame at the top of the stack). Thus, we may prove correctness of the S-FREEZE case of Conjecture 5.1 directly by using Conjecture 5.2. We will return to the latter conjecture shortly in Section 5.4.2.2.

Meanwhile, we observe the following about the rule S-INST, the other rule in the second group of rules: We may apply the same reasoning as in the case for the rule S-FREEZE. This means choosing F_p in the same manner, inducing C'_0 and C'_1 as follows, for some $\overline{b}$ and H:

$$\begin{array}{rcl} C'_0 & = & \mathbf{def}\,(x : \forall \overline{b}.H)\ \mathbf{in}\ F''[x \preceq A \qquad \wedge \mathfrak{U}(\Theta_0, \theta_0)] \\ C'_1 & = & \mathbf{def}\,(x : \forall \overline{b}.H)\ \mathbf{in}\ F''[\exists \overline{b}.(H \sim A) \wedge \mathfrak{U}(\Theta_0, \theta_0)] \end{array}$$

Recall that H denotes guarded types (i.e., without toplevel quantifiers). We may

[7]Recall that while let frames in F may bind term variables, they do not bind them towards the hole. Thus, $\mathsf{tc}(F)$ only returns bindings due to definition constraints.

thus prove the correctness of the S-INST case of Conjecture 5.1 by using the following conjecture similarly to the previous case.

Conjecture 5.3 (Substituting instantiation constraints)**.**
The constraints

$$\mathbf{def}\ (x : \forall \overline{b}.H)\ \mathbf{in}\ F[x \preceq A]$$

and

$$\mathbf{def}\ (x : \forall \overline{b}.H)\ \mathbf{in}\ F[\exists \overline{b}.(H \sim A)]$$

are logically equivalent.

This property directly reflects the semantics of instantiation constraints: The constraint $x \preceq A$ expresses that the toplevel quantifiers of x's type can be instantiated to yield A, after instantiation of the free flexible variables in A. The semantics of definition constraints require that any model $(\Delta, \Xi, \Gamma, \delta)$ of $\mathbf{def}\ (x : \forall \overline{b}.H)\ \mathbf{in}\ \ldots$ interprets the free type variables of H monomorphically. This means that the toplevel quantifiers of $\delta(\forall \overline{b}.H)$, the type subsequently associated with x in the term context, are still just $\overline{b}$.

5.4.2.2 Towards a proof of Conjecture 5.2

We now discuss the issues faced when attempting to prove Conjecture 5.2.

Since both constraints are similar def constraints, we obtain by inversion that that it suffices to show the following property to prove Conjecture 5.2: For all $\Delta, \Xi, \Gamma, \delta$ such that $(\Delta, \Xi) \vdash_\star B\ \mathbf{ok}$ and $\Delta \vdash_\bullet \delta(b)\ \mathbf{ok}$ for all $b \in \mathsf{ftv}(B)$, we have

$$\Delta;\Xi;(\Gamma, x : \delta(B));\delta \vdash F[\lceil x : A \rceil] \quad \text{iff} \quad \Delta;\Xi;(\Gamma, x : \delta(B));\delta \vdash F[B \sim A]. \qquad (5.12)$$

Note that this directly encodes the monomorphism condition imposed by the original, toplevel definition constraint.

There are two potential approaches for proving this refined property inductively: Either by induction on the structure of the stack F, or by induction on the structure of the constraint $F[\lceil x : A \rceil]$.[8]

Note that both approaches are somewhat dual: Given that stacks are inductively defined as $F ::= F :: f \mid \cdot$, the first approach involves reasoning about the shape of the innermost part of the constraint $F[\lceil x : A \rceil]$ (i.e., the subconstraint immediately surrounding $\lceil x : A \rceil$). Conversely, reasoning about the structure of the whole constraint $F[\lceil x : A \rceil]$ involves reasoning about the overall shape of the constraint. Effectively,

[8]Of course, we may also perform induction on the structure of $F[B \sim A]$ in the second approach.

this approach also reasons about the structure of the stack F, but analyses the outermost stack frame instead. We explore the two approaches separately.

First approach: Induction on the structure of F All cases are straightforward, except for the case where F is of the shape $F' :: f$ and f is of the shape $\mathbf{let}_R\, y = \sqcap b.\square$ **in** C. Note that if it was the case that $f[\,\lceil x : A \rceil\,]$ and $f[B \sim A]$ are logically equivalent, we could apply Lemma 5.6 on page 135 to obtain logical equivalence of $F[\lceil x : A \rceil]$ and $F[B \sim A]$. However, expanding both constraints yields $\mathbf{let}_R\ y = \sqcap b.\lceil x : A \rceil$ **in** C and $\mathbf{let}_R\ y = \sqcap b.B \sim A$ **in** C, which are clearly not logically equivalent in general.

The fact that Lemma 5.6 is not applicable is unsurprising: It requires a relationship between $f[\,\lceil x : A \rceil\,]$ and $f[B \sim A]$ independent of the connection between x and B, stronger than what we are given. Proving logical equivalence of the two constraints requires considering all possible interpretations of the flexible variables in scope, instead of respecting the connection established between B and the type of x via the interpretation δ in (5.12). In turn, Lemma 5.6 would establish a stronger relationship between $F[\,\lceil x : A \rceil\,]$ and $F[B \sim A]$ than what is required.

We may thus hope to find a slightly refined version of Lemma 5.6 which does not require full logical equivalence of the given constraints, but only requires a slightly weaker relationship that only considers those interpretations that respect the connection between certain flexible type variables and term variables. However, the author has so far not succeeded in finding such an alternative version. Any attempt at proving correctness of such an alternative version of Lemma 5.6 that imposes a weaker relationship between the constraints named C and C' therein has been unsuccessful when reaching the case of a let constraint: The mostgen condition imposed by the semantics of let constraints effectively requires reasoning about all possible models of the constraint, not just those exhibiting a particular relationship with the given term context.

Second approach: Induction on the structure of $F[\,\lceil x : A \rceil\,]$ We now discuss attempts to prove (5.12) by structural induction on the constraint $F[\,\lceil x : A \rceil\,]$. Again, all cases are straightforward, except when $F[\,\lceil x : A \rceil\,]$ is of the form $\mathbf{let}_R\, y = \sqcap c.F'[\,\lceil x : A \rceil\,]$ **in** C_2, meaning that $F = f :: F'$. We focus on the case where $R = \star$ (i.e., $F[\,\lceil x : A \rceil\,]$ is a *generalising* let constraint), the problems encountered for non-generalising constraints are analogous.

Let $\mathcal{J}^1$ and $\mathcal{J}^2$ be the judgements $\Delta; \Xi; (\Gamma, x : \delta(B)); \delta \vdash \mathbf{let}_\star\, y = \sqcap c.F'[\,\lceil x : A \rceil\,]$ **in** C and $\Delta; \Xi; (\Gamma, x : \delta(B)); \delta \vdash \mathbf{let}_\star\ y = \sqcap c.F'[B \sim A]$ **in** C, respectively. We need to show that $\mathcal{J}^1$ is derivable if and only if $\mathcal{J}^2$ is. Let $C^1 := \lceil x : A \rceil$ and $C^2 := B \sim A$.

According to rule SEM-LETGEN in Figure 5.1, we have that for all $i \in \{1,2\}$, any derivation of $\mathcal{J}^i$ then has the following shape.[9]

$$\frac{\begin{array}{c} \mathsf{mostgen}(\Delta, (\Xi, c), (\Gamma, x : \delta(B)), F'[C^i], \Delta^i_{\mathrm{m}}, \delta^i_{\mathrm{m}}) \\ \Delta^i_{\mathrm{o}} = \mathsf{ftv}(\delta^i_{\mathrm{m}}(\Xi)) - \Delta \qquad \Delta^i_{\mathrm{g}} = \mathsf{ftv}(\delta^i_{\mathrm{m}}(c)) - \Delta, \Delta^i_{\mathrm{o}} \\ \Delta \vdash \delta^i : \Delta^i_{\mathrm{o}} \Rightarrow_{\bullet} \cdot \qquad A^i = \delta^i(\delta^i_{\mathrm{m}}(c)) \\ (\Delta, \Delta^i_{\mathrm{g}}); (\Xi, c); (\Gamma, x : \delta(B)); \delta[c \mapsto A^i] \vdash F'[C^i] \\ \Delta; \Xi; (\Gamma, x : \delta(B), y : \forall \Delta^i_{\mathrm{g}}.A^i); \delta \vdash C \end{array}}{\Delta; \Xi; (\Gamma, x : \delta(B)); \delta \vdash \mathbf{let}_{\star}\ y = \sqcap c.F'[C^i]\ \mathbf{in}\ C}$$

We focus on two difficulties encountered when trying to obtain a derivation of $\mathcal{J}^2$ from that of $\mathcal{J}^1$.

The first difficulty is that one would have to show that the existence of a most general solution of the constraint $F'[\lceil x : A \rceil]$ implies the existence of such a most general solution of $F'[B \sim A]$. In general, it is possible to infer the existence of most general solutions for satisfiable constraints from the overall correctness property of the solver stated later in this chapter (see Conjecture 5.4 in Section 5.4.3). However, this correctness property depends on Conjecture 5.1 and inherits the restriction regarding free type variables in let constraints imposed in Conjecture 5.1: Conjecture 5.4 only applies to constraints C' with $\Delta' \vdash^{\mathrm{ftv}} C'$ **ok** (for some initial Δ'), which does not generally hold for $F'[\lceil x : A \rceil]$ or $F'[B \sim A]$. Of course, it is also the case that relying on this kind of reasoning to establish the existence of most general solutions requires care to ensure that the dependence between the involved proofs would be well founded: If Conjecture 5.2 and Conjecture 5.4 were successfully proven this way, they would (transitively) rely on each other.

We now focus on the second difficulty encountered when trying to obtain a derivation of $\mathcal{J}^2$ from that of $\mathcal{J}^1$. We observe that the most general solutions of $F'[\lceil x : A \rceil]$ and $F'[B \sim A]$ are not guaranteed to be identical in general, which inherently causes the derivations of $\mathcal{J}^1$ and $\mathcal{J}^2$ to differ, due to the use of the most general solution δ^i_{m} in each. As an example for this difference between the most general solutions, we consider the following example for the property (5.12), where F is $f :: F'$ and C is

[9]We use superscripts instead of subscripts here to avoid interfering with the naming scheme used for the premises of let constraints throughout this work.

arbitrary:

$$\begin{aligned} F' &= \cdot \\ f' &= \mathbf{let}_\star\, y = \sqcap c.\square \ \mathbf{in}\ C \\ A &= c \\ B &= b \to b \end{aligned}$$

This means that the judgements $\mathcal{J}^1$ and $\mathcal{J}^2$ become the following:

$$\begin{array}{ll} \Delta;\Xi;(\Gamma,x:\delta(b\to b));\delta \vdash \mathbf{let}_\star\, y = \sqcap c.\ \lceil x : c \rceil & \mathbf{in}\ C \\ \Delta;\Xi;(\Gamma,x:\delta(b\to b));\delta \vdash \mathbf{let}_\star\, y = \sqcap c.\ b\to b \sim c & \mathbf{in}\ C \end{array}$$

As expected, it is indeed the case that for any choice of Δ,Ξ,Γ and δ satisfying the precondition $\Delta \vdash_\bullet \delta(B)$ **ok** (i.e., $\delta(b)$ is a monotype), we then have that if $\mathcal{J}^1$ is derivable, then $\mathcal{J}^2$ is derivable, too.[10] We will explain why this is the case by focusing on the more interesting case where b is a flexible variable (i.e., $b \in \Xi$). To simplify the reasoning, we further assume that Ξ contains no other variables and the other contexts are empty (i.e., $\Xi = b$ and $\Delta = \Gamma = \cdot$).

We then have that the following holds for the any derivation of $\mathcal{J}^1$, using the naming scheme established in the abstract derivation of $\mathcal{J}^1$ shown on the previous page.

$$\begin{gathered} \mathsf{mostgen}(\cdot,(b,c),(x:\delta(b\to b)),\lceil x:c\rceil,\Delta^1_{\mathrm{m}},\delta^1_{\mathrm{m}}) \\ \Delta^1_{\mathrm{g}} = \cdot \\ \delta^1_{\mathrm{m}}(c) = \delta(b\to b) \\ \mathsf{domain}(\delta^1) \mathbin{\#} \mathsf{ftv}(\delta^1_{\mathrm{m}}(c)) \\ A^1 = \delta^1(\delta^1_{\mathrm{m}}(c)) = \delta(b\to b) \\ \cdot;b;(x:\delta(B),y:A^1);\delta \vdash C \end{gathered}$$

This reflects that the most general solution δ^1_{m} of the constraint $\lceil x : c\rceil$ under term context $(x : \delta(b \to b))$ maps c to $\delta(b \to b)$. The latter type is then given to y in the second sub-constraint of the overall let constraint.

However, we observe that rewriting $\lceil x : c\rceil$ (the sub-constraint appearing in $\mathcal{J}^1$) to $b \to b \sim c$ (the sub-constraint appearing in $\mathcal{J}^2$), changes the most general solution. Concretely, we have that the most general solution δ^2_{m} of the constraint $b \to b \sim c$ (under any term context) maps b to some fresh b', and c to $(b' \to b')$. We may then still derive $\mathcal{J}^2$, but the derivation is somewhat different.

[10]The reverse direction also holds, but as mentioned earlier, we focus on the case of obtaining a derivation for $\mathcal{J}^2$ from one for $\mathcal{J}^1$.

$$
\dfrac{\begin{array}{c}
\mathsf{mostgen}(\cdot,(b,c),(x:\delta(b\to b)),b\to b\sim c,\ \overbrace{b'}^{=:\Delta^2_{\mathrm{m}}},\overbrace{[c\mapsto(b'\to b'),b\mapsto b']}^{=:\delta^2_{\mathrm{m}}})\\
\Delta^2_{\mathrm{o}}=\mathsf{ftv}(\delta^2_{\mathrm{m}}(b))=b' \qquad \Delta^2_{\mathrm{g}}=\mathsf{ftv}(\delta^2_{\mathrm{m}}(c))-\Delta^2_{\mathrm{o}}=\cdot\\
\Delta\vdash\delta^2:\Delta^2_{\mathrm{o}}\Rightarrow_{\bullet}\cdot \qquad A^2=\delta^2(\delta^2_{\mathrm{m}}(c))=\delta(b\to b)\\
\cdot;(b,c);(x:\delta(B));\delta[c\mapsto A^2]\vdash b\to b\sim c \qquad \cdot;b;(x:\delta(B),y:A^2);\delta\vdash C
\end{array}}{\cdot;b;(x:\delta(b\to b));\delta\vdash\mathbf{let}_{\star}\ y=\sqcap c.b\to b\sim c\ \mathbf{in}\ C}
$$

The crucial part here is the following: Despite the fact that the most general solution of the first sub-constraint of the let constraint changed as compared to the derivation of $\mathcal{J}^1$, the type ultimately given to y is guaranteed to be same as in the derivation of $\mathcal{J}^1$. Concretely, we have that $A^1 = A^2 = \delta(b \to b)$. This allows us to re-use the fact that $\cdot;b;(x:\delta(B),y:A^1);\delta\vdash C$ holds from the derivation of $\mathcal{J}^1$.

The reason why y receives the same type $\delta(b\to b)$ as in the derivation of $\mathcal{J}^1$ is as follows. While the most general solution for c is $b'\to b'$, the type A^2 ultimately used for y is obtained by applying a monomorphic substitution δ^2 to $\delta^2_{\mathrm{m}}(c)$. The second-to-last premise $\cdot;(b,c);(x:\delta(B));\delta[c\mapsto A^2]\vdash b\to b\sim c$ then enforces that $\delta^2(b')=\delta(b)$. The latter type is guaranteed to be monomorphic, as imposed on (5.12).

This is not a coincidence: In the semantics of constraints $\mathbf{let}_{\star}\ y=\sqcap c.C_1\ \mathbf{in}\ C_2$ (see rule SEM-LETGEN in Figure 5.1 on page 97), the second to last premise, regarding C_1, is specifically designed to ensure that the most general solution for c in C_1 is *refined* in a way such that the type ultimately given to y is compatible with the existing interpretations of type variables (i.e., the mappings of the ambient instantiation δ in the rule SEM-LETGEN).

We may thus summarise the second difficulty encountered when attempting to prove (5.12) by induction on the structure of $F[\lceil x:A\rceil]$ as follows: We have shown alongside an example that the most general types used in the derivations of $\mathcal{J}^1$ and $\mathcal{J}^2$ may differ. Concretely, we have observed how the most general solution of $F'[B\sim A]$ has become *more general* than the most general solution of $F'[\lceil x:A\rceil]$. However, our example has illustrated how the type ultimate given to the term variable y bound by the let binding under consideration remains unchanged: The additional condition imposed on let bindings ensuring compatibility with the ambient instantiation forces the more general type $\delta^2_{\mathrm{m}}(c)$ to be instantiated back to $\delta^1_{\mathrm{m}}(c)$ when determining the type ultimately given to y. So far, the author of this work has not succeeded in proving this aspect formally. Reasoning about the exact relationship between the most general solutions of $F'[\lceil x:A\rceil]$ and $F'[B\sim A]$ appears challenging: One of the appealing characteristics of the constraint-based inference approach is that the solver modifies constraints in a way that the involved constraints remain logically equivalent. In contrast, the reasoning needed here effectively requires formalising how transforming $F'[\lceil x : A \rceil]$ into $F'[B \sim A]$ *changes* the (most general) solutions of the constraints.

We have thus shown the obstacles faced when attempting either of the two inductive proof strategies outlined for Conjecture 5.3.

5.4.2.3 Summary

In total, we observe the following regarding a preservation property for our solver. The mostgen condition used in the semantics of let constraints leads to additional restrictions being required in the preservation property for our system, leading to the property stated in Conjecture 5.1. We outlined how this property may be proved, except for the cases when the rules S-FREEZE or S-INST were applied. For these last two cases, we have distilled the missing parts to complete the proof of Conjecture 5.1 into two simple, standalone conjectures, namely Conjectures 5.2 and 5.3.

We have then discussed different potential approaches for proving Conjecture 5.2, all of which have been unsuccessful so far, in part due to the aforementioned restrictions imposed in Conjecture 5.1. Although not discussed further, similar problems are encountered when attempting a proof of Conjecture 5.3. Altogether, this means that Conjecture 5.1 does not have a complete proof at this point and remains a conjecture.

5.4.3 Correctness of the solver

We now prove additional properties of our solver. Together with Conjecture 5.1, these then lead to an overall correctness property of the solver.

One important property is termination of the solver.

Theorem 5.3 (Termination)**.**
The constraint solver terminates when invoked on any well formed input state.

Proof. We show the existence of a well-ordering $<$ on states such that $s \rightarrow s'$ implies $s' < s$. See Appendix B.3 for details. □

We now observe that whenever the constraint representation of a well formed state s is satisfiable and the state s is not final (as defined in Section 5.3.2), then the solver can take a step.

Theorem 5.4 (Progress)**.**
Let $\vdash (F, \Theta, \theta, C)$ **ok** *and* $F[C] \neq \forall\Delta.\exists\Xi.\mathsf{true}$ *for all* Δ, Ξ. *Further, let* $\cdot;\cdot;\cdot;\emptyset \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$ *hold. Then there exists a state* s_1 *such that* $(F, \Theta, \Gamma, \theta, C) \rightarrow s_1$.

Proof. By case analysis regarding the shape of C, showing that one of the rules from Figure 5.8 is applicable. Details are provided in Appendix B.3. □

Note that combining the termination of the solver (Theorem 5.3) and the preservation property (Conjecture 5.1), implies that for every state with a satisfiable constraint representation, the solver eventually reaches a final state. Conversely, it means that when executed on a state with a non-satisfiable constraint representation, the solver must eventually get stuck, as all well formed final states have a satisfiable constraint representation.

We can formalise these findings in the following property, stating the overall correctness of the solver. Due to its dependence on Conjecture 5.1, it remains a conjecture. Given $\Gamma = (x_1 : A_1), \ldots, (x_n : A_n)$, we define **def** Γ **in** C as the constraint **def** $(x_1 : A_1)$ **in** $\ldots$ **def** $(x_n : A_n)$ **in** C. The conjecture then states that a constraint C has a model $\Delta, \Xi, \Gamma, \delta$ if and only if the solver reaches a final state from the constraint $\forall\Delta.\exists\Xi.$ **def** Γ **in** C and δ is a refinement of the substitution θ in the final state. Here, by "refinement" we mean composing with a well formed substitution θ', respecting the restriction context returned by the solver. Note that in order to apply Conjecture 5.1, we further require $\vdash^{\text{ftv}} C$ **ok**.

Conjecture 5.4 (Correctness of the constraint solver)**.**
Let $\Delta \vdash \Gamma$ **ok** *and* $\Delta;\Xi;\Gamma \vdash C$ **ok** *and* $\Delta \vdash^{\textit{ftv}} C$ **ok**. *Then we have*

$\Delta;\Xi;\Gamma;\delta \vdash C$
iff
there exist $\Theta, \theta', \theta, \Xi'$ *s.t.*
$(\cdot, \cdot, \emptyset, \forall\Delta.\exists\Xi.\ \textbf{def}\ \Gamma\ \textbf{in}\ C) \rightarrow^* (\forall\Delta :: \exists\ (\Xi, \Xi'), \Theta, \theta, \mathsf{true})$ *and*
$\Delta \vdash \theta' : \Theta \Rightarrow \cdot$ *and*
$(\theta' \circ \theta)\restriction_{\Xi} = \delta.$

Proof (dependent on Conjecture 5.1). The contingent proof relies on a slightly more general property, formalised as Conjecture B.1 in Appendix B.3, which in turn may be shown to hold by relying on Conjecture 5.1. Conjecture B.1 also imposes an additional well-formedness condition on states, which is separately shown to be preserved as an invariant during solving (Lemma B.6). See Appendix B.3 for details. △

Note that this correctness property also implies that our solver returns most general solutions. The result θ of the solver is independent from δ. Due to its deterministic nature we therefore have that *any* such δ can be obtained as a refinement of the substitution θ returned by the solver. We thus have that all satisfiable constraints C, subject to the restriction $\Delta \vdash^{\mathsf{ftv}} C \ \mathbf{ok}$, have a most general solution.

5.4.4 Correctness of constraint-based type inference

So far, we have discussed the correctness of the constraint solver in isolation, independent from FreezeML terms. We may now relate the typings of a FreezeML term M to the output of the solver when run on the constraint obtained from M, as defined in Section 5.2. Thus, we state two properties encoding the overall correctness of the constraint-based type inference approach. These two properties rely on Conjecture 5.4 defined in the last section, which is why they remain as conjectures, ultimately depending on Conjectures 5.2 and 5.3 discussed in Section 5.4.2.

These properties also rely on the fact that constraints obtained from FreezeML terms satisfy the additional condition imposed on free type variables in let constraints. We formalise this as follows.

Lemma 5.7.
If $\Delta;\Gamma \vdash M \ \mathbf{ok}$ *and* $(\Delta,\Xi) \vdash A \ \mathbf{ok}$ *then* $\Delta \vdash^{\mathit{ftv}} [\![M : A]\!] \ \mathbf{ok}$.

Proof. By induction on the structure of M. By Lemma 5.2 on page 107, we have that the only free type variables of the constraint $[\![M : A]\!]$ are those appearing freely in A and (in the type annotations appearing) in M. If M is a term $\mathbf{let}\ x = M'\ \mathbf{in}\ N'$, we then have that the first sub-constraint of the resulting let constraint is $[\![M' : a]\!]$. This constraint may thus only contain a and the variables appearing freely in M'. By assumption $\Delta;\Gamma \vdash M \ \mathbf{ok}$ we have that the latter variables are a subset of Δ. □

As a first correctness property of the overall inference approach, we state that executing the solver on a the constraint $[\![M : a]\!]$, where a is a fresh placeholder for the type of M, yields a sound type inference algorithm. More concretely, if the solver reaches a final state containing some Θ and θ, then any substitution θ' that respect the restrictions Θ may be used to refine $\theta(a)$ to a valid type of M.

Conjecture 5.5 (Constraint-based type-checking is sound)**.**
Let $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash M \ \mathbf{ok}$ *and* $a \# \Delta$. *If* $(\cdot,\cdot,\emptyset,\forall\Delta.\exists a.\ \mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \rightarrow^{*} (\forall\Delta :: \exists\,(a,\overline{b}),\Theta,\theta,\mathsf{true})$ *and* $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$ *then* $\Delta;\Gamma \vdash M : (\theta' \circ \theta)(a)$.

Proof (dependent on Conjecture 5.4). Follows from Lemma 5.7, Theorem 5.2, and Conjecture 5.4, details in Appendix B.4. △

Alternatively, we may use the extended FreezeML typing relation, defined in Section 4.1.2, and restate the conclusion of the conjecture such that $(\Delta^{\bullet}, \Theta); \Gamma \vdash^{\mathrm{E}} M : \theta(a)$ holds. By Lemma 4.4 on page 77, this then implies the original statement of Conjecture 5.5.

Conversely, we state that the inference algorithm is complete, meaning that if M has type A, then the solver reaches a final state and A can be obtained from $\theta(a)$. It is also most general, as any such A can be obtained from the same $\theta(a)$ this way.

Conjecture 5.6 (Constraint-based type-checking is complete and most general)**.**
Let $a \# \Delta$. *If* $\Delta; \Gamma \vdash M : A$ *then there exist* $\Xi, \Theta, \theta, \delta$ *such that* $(\cdot, \cdot, \emptyset, \forall \Delta . \exists a . \mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \rightarrow^{*} (\forall \Delta :: \exists\, \Xi, \Theta, \theta, \mathsf{true})$ *and* $A = \delta(\theta(a))$.

Proof (dependent on Conjecture 5.4). Follows from Lemma 5.7, Theorem 5.1, and Conjecture 5.4, details in Appendix B.4. △

We again observe that we may rephrase the conjecture, stating in its conclusion that $\mathsf{principal}(\Delta, \Gamma, M, \Xi, \theta(a))$ holds.

Note that the chain of contingent proofs provided throughout this section shows that we immediately obtain the overall correctness of the solver if Conjectures 5.2 and 5.3 were proved to hold.

5.5 Comparison with HM(X)

We now compare our constraint-based type inference approach with the work by Pottier and Rémy [97].

5.5.1 Interdefinability of constraint forms

In the constraint language by Pottier and Rémy, all def constraints can be substituted with equivalent constraints not using these forms in a purely syntactic fashion (i.e., without needing to perform any solving). Further, let bindings are simply defined as syntactic sugar for other constraints. We now compare these aspects to our constraint language and explore which related substitutions our language permits.

5.5.1.1 Definition constraints

The reader may have observed that our monomorphism constraints can be encoded by definition constraints: The constraint $\mathsf{mono}(c)$ is logically equivalent to $\mathbf{def}\ (x : a)\ \mathbf{in}\ \mathsf{true}$. Nevertheless, we keep the former as primitives in our constraint language to mirror a constraint substitution property of Pottier and Rémy's language.

Their work uses constraints of the form $\mathbf{def}\ (x : \sigma)\ \mathbf{in}\ C$, where σ is a (qualified) type scheme of the form $\forall \overline{a}\,[C'].\tau$. Here, C' constrains the possible instantiations of $\overline{a}$, analogously to qualified types, and τ cannot contain further quantifiers or constraints. Of course, $\overline{a}$ may be empty, and C' may be the trivial constraint true. Conversely, type schemes may be uninhabited, such as $\forall \overline{a}\,[\mathsf{false}].\mathsf{Int}$.

In their work, it is then the case that $\mathbf{def}\ (x : \forall \overline{a}\,[C'].\tau)\ \mathbf{in}\ C$ is logically equivalent to the constraint resulting from replacing every occurrence of $x \preceq \tau'$ inside C with $\exists \overline{a}.(C' \wedge \tau \leq \tau')$. Here, $\leq$ denotes subtype constraints, for the purposes of this discussion we may use equality constraints instead. Note that if x does not appear freely in C, then the overall constraint is satisfiable even if the type of x is uninhabited.

One advantage of using def constraints rather than directly replacing them using the equivalence above is that the size of constraints generated from terms is linear in the size of the original term. They allow marking where substitution is (logically) occurring, without needing to perform the actual syntactic manipulation of the constraint.

Using mono constraints, we can state a similar substitution property for def constraints in our language. Here, we need to reflect that all free type variables appearing in the type annotation must be monomorphic, as imposed by the semantics of def constraints.

Conjecture 5.7 (Substitution of def constraints)**.**
The constraint $\mathbf{def}\ (x : \forall\Delta.H)\ \mathbf{in}\ C$ *is logically equivalent to* $C' \wedge \bigwedge_{a \in \mathsf{ftv}(H)} \mathsf{mono}(a)$. *where* C' *results from* C *by replacing all of its sub-constraints of the form* $\lceil x : A \rceil$ *with* $(\forall\Delta.H) \sim A$ *and all sub-constraints of the form* $x \preceq A$ *with* $\exists\Delta.(H \sim A)$.

We observe that this property is a slightly more general version of what can be obtained from repeated applications of Conjectures 5.2 and 5.3. While the latter two only reason about constraints resulting from plugging $\lceil x : A \rceil$ or $x \preceq A$ into a stack, they may appear in arbitrary positions in the property above. However, this is merely a matter of presentation, successful proofs of Conjectures 5.2 and 5.3 would lead to a proof of Conjecture 5.7 as well.

5.5.1.2 Let constraints

In Pottier and Rémy's work, let constraints have the form $\mathbf{let}\ (x : \sigma)\ \mathbf{in}\ C$, where σ is a again a qualified type scheme of the form $\forall\overline{a}[C'].\tau$. The only difference with their def constraints is that let constraints require the type scheme σ to be inhabited. Thus, they define the previous let constraint to be syntactic sugar for $(\exists\overline{a}.C') \wedge (\mathbf{def}\ (x : \sigma)\ \mathbf{in}\ C)$.

No substitution for generalising let constraints However, the author believes that there is no hope of finding a similar relationship between let constraints and other constraints in our system that does not involve solving the sub-constraint C_1 of $\mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ first.

Let constraints are the only place in our system that exposes the polymorphic types arising from generalisation. We cannot obtain a constraint equivalent to the previous let constraint (choosing $\star$ for R) by replacing all $\lceil x : A \rceil$ appearing in C_2 with $\exists a.(C_1 \wedge a \sim A)$. If generalisation occurs, such as in the example $\mathbf{let}_\star\ x = \sqcap a.(\exists b.a \sim b \to b)\ \mathbf{in}\ \lceil x : \forall b.b \to b \rceil$, this substitution scheme fails.

Conversely, we cannot obtain a constraint equivalent to the previous let constraint by replacing all $x \preceq A$ in C_2 with $\exists a.(C_1 \wedge a \sim A)$. If *no* generalisation occurs, such as in the example $\mathbf{let}_\star\ x = \sqcap a.a \sim \forall b.b \to b\ \mathbf{in}\ x \preceq (\mathsf{Int} \to \mathsf{Int})$, this also fails.

One solution may be the introduction of generalisation constraints, where $\sqcap a.C : A$ asserts that the result of generalising the principal solution for a in C is A.

No substitution for non-generalising let constraints Finally, for non-generalising let constraints, no syntactic way of eliminating them seems to exist, either. A constraint equivalent to $\mathbf{let}_\bullet\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ would have to impose monomorphism constraints on exactly those type variables appearing in the most general solution for a when solving C_1. In particular, $\exists a.\big(C_1 \wedge \mathbf{def}\ (x : a)\ \mathbf{in}\ C_2\big)$ is not equivalent to the previous let constraint, as it forces the *entire* type of x to be monomorphic. This would turn $\mathbf{let}_\bullet\ x = \sqcap a.a \sim \forall b.b \to b\ \mathbf{in}\ x \preceq (\mathsf{Int} \to \mathsf{Int})$ into an unsatisfiable constraint.

One solution could be the introduction of qualified type schemes in FreezeML, where $\forall\overline{a}\,[C].A$ *is* the type resulting from generalising the most general solution for A in C with respect to $\overline{a}$. We would then introduce a form of instantiation constraint, optionally limited to *monomorphic* instantiation. We expect that this approach may subsume the introduction of dedicated generalisation constraints mentioned in the previous paragraph when discussing generalising let constraints.

However, care is required if one were to allow instantiation constraints between arbitrary types of the form $A \preceq B$ in our constraint language, rather than just allowing our current constraints of the form $x \preceq A$.

This is due to the dual purpose that term variables serve in our constraint-based approach. In addition to marking where substitution occurs, as in HM(X), they enforce the simple invariant in our constraint language that only fully known polymorphism can be instantiated. Our monomorphism conditions on def and let constraints impose that the polymorphism occurring in the types of term variables is uniquely determined, reflecting a similar invariant holding for the FreezeML term language. Weakening this property can easily lead to the loss of principal solutions.

We leave the addition of constrained types to our system as future work and briefly revisit it in Section 7.2.3.

Substituting def constraints with let constraints We now observe that the reverse direction, replacing def constraints with let constraints, is indeed possible. However, we still keep def constraints as a simpler form in the language. In particular, recall that our solver eventually replaces a constraint $\mathbf{let}_R\, x = \sqcap a.C_1$ **in** C_2 with a suitable def constraint once C_1 is solved (see the two S-LETPOP rules in Figure 5.8 on page 115).

Lemma 5.8 (Def constraints as let constraints).
Let $a \# \mathsf{ftv}(A)$. Then following constraints are logically equivalent.

$$\begin{aligned}
&\mathbf{def}\ (x : A)\ \mathbf{in}\ C\\
&\mathbf{let}_\star\ x = \sqcap a.a \sim A\ \mathbf{in}\ C\\
&\mathbf{let}_\bullet\ x = \sqcap a.a \sim A\ \mathbf{in}\ C
\end{aligned}$$

Proof. We consider a well formed interpretation $(\Delta, \Xi, \Gamma, \delta)$; due to the assumption $a \# \mathsf{ftv}(A)$ we can assume w.l.o.g. that $a \# (\Delta, \Xi)$ holds. We first examine the conditions imposed by the semantics of generalising let constraints onto the interpretation to be a model of $\mathbf{let}_\star\ \sqcap a.a \sim A$ **in** C.

Using the meta-variable names from SEM-LETGEN in Figure 5.1 on page 97, we observe that the premise $\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, a \sim A, \Delta_\mathrm{m}, \delta_\mathrm{m})$ holds if and only if we have the following: There exists a bijection δ_f between Ξ and $\overline{c}$, which is a subset of Δ_m, such that $\delta_\mathrm{m}(a) = \delta_\mathrm{f}(A)$ and $\delta_\mathrm{m}\restriction_\Xi = \delta_\mathrm{f}$. This yields $\Delta_\mathrm{o} = \overline{c}$ and thus $\overline{b} = \cdot$. The premise regarding the satisfiability of C_1 requires that $\delta'(\delta_\mathrm{m}(a)) = \delta(A)$ holds. This is equivalent to $\delta'(\delta_\mathrm{f}(A)) = \delta(A)$. As δ_f simply renames variables, the premise $\Delta \vdash \delta' : \Delta_\mathrm{o} \Rightarrow \cdot$ requires that δ maps all free flexible type variables of A to monomorphic types.

Altogether, this means that we have that $(\Delta, \Theta, \Gamma, \delta)$ is a model of the constraint $\textbf{let}_\star\, x = \sqcap a.a \sim A\ \textbf{in}\ C$ if and only if δ maps all flexible type variables of A to monomorphic ones and we have that $\Delta; \Xi, (\Gamma, x : \delta(A)); \delta \vdash C$ holds. These are exactly the conditions imposed by the semantics of def constraints for $\Delta; \Theta; \Gamma; \delta \vdash \textbf{def}\ (x : A)\ \textbf{in}\ C$ to hold.

Showing the equivalence of the latter constraint and $\textbf{let}_\bullet\, x = \sqcap a.a \sim A\ \textbf{in}\ C$ works analogously. $\square$

5.5.2 Remy's extension of HM(X) solver to support FML

To perform type inference for the FML system (discussed in Section 2.3.5), Rémy [101] shows that a simple extension of the original HM(X) solver by Pottier and Rémy is sufficient. In particular, it is *not* necessary to extend the solver to permit polymorphic interpretations of type variables.

The only change to the constraint languages is the addition of constraints of the form $A \leq' B$. Such constraints are satisfied if and only if there exists a monomorphic interpretation ϕ of all variables in $\overline{c} := \mathsf{ftv}(A) \cup \mathsf{ftv}(B)$ such that $\phi(A) \leq^{\mathrm{F}}_{\approx} \phi(B)$, where the latter relation is the System F generic instance relation modulo reordering of nested quantifiers (see Section 2.3.5).

Pottier and Rémy shows that such constraints can be solved in two steps. First, A and B are normalised (i.e., putting quantifiers in a canonical order and removing superfluous ones) to $\forall \overline{a}.H_1$ and $\forall \overline{b}.H_2$, respectively. Then, whether the resulting types satisfy the System F generic instance relation (i.e., polymorphic instantiation of toplevel quantifiers) is equivalent to checking whether the constraint $\forall \overline{b}.\exists a.(H_1 = H_2)$ holds. Here, the free variables $\overline{c}$ of H_1 and H_2, as defined earlier, must be instantiated monomorphically, while only the type variables $\overline{a}$ may be interpreted using polymorphic types. These constraints can then be solved in isolation, it is guaranteed that the polymorphic types picked for the variables $\overline{a}$ are entirely local to this sub-constraint and do not leak into (i.e., affect) the rest of the constraint solving problem.

In other words, the fact that FML requires type annotations for all polymorphic instantiations implies that the solver only has to be able to *check* that two types, whose polymorphism is fully known, satisfy the System F generic instance relation. This entirely local treatment of polymorphic unification variables is not sufficient for FreezeML, where a polymorphic interpretation of a type variable may need to spread globally. In particular, there is no counterpart in Remy's system to a constraint such as $\lceil x : a \rceil$ from ours, which would require equating a with the (qualified) type scheme associated with the term variable x.

5.5.3 Other aspects

We now collect some additional differences between our approach and the work by Pottier and Rémy.

Representation of unification context As mentioned in Section 5.3.1.2, Pottier and Rémy's solver carries the currently known constraints imposed on type variables as a dedicated constraint $\exists\overline{a}.C_{\mathrm{U}}$, where C_{U} is quantifier-free and in solved form. In particular, rather than storing a substitution θ in solver states, they maintain multi-equations, which directly expose the underlying union-find structure. Each multi-equation is of the form $a_1 = \cdots = a_n = A$, where all a_i are part of the same union-find set, and A is its representative. A similar design could be adopted for our solver, requiring to allow mono constraints in solved forms.

We would then replace the component Θ with the constraint $\mathfrak{U}(\Theta)$ (as defined in Section 5.3.1.2), which then becomes a sub-constraint of C_{U}. The component θ would be replaced by appropriate multi-equations in C_{U}.

Supporting recursive types In Pottier and Rémy's work, supporting recursive types is simply a matter of removing the occurs check in their unification algorithm. In our work, the counterpart to this check is the assertion $(\Delta^{\bullet}, \Theta_1) \vdash_R A$ **ok** from the last clause of the unification algorithm in Figure 4.4 on page 80. It ensures that when detecting that a substitution $a \mapsto A$ should be performed, a does not appear in A.

However, it is not clear to the author what the implications of supporting recursive types in our system would be. On one hand, our meta-theory relies on the fact that substitutions are idempotent (which is a direct result of the aforementioned assertion) in several instances. One solution may be the introduction of explicit μ types. Support for recursive types therefore remains as future work.

Optimisations The solver presented by Pottier and Rémy implements several optimisations, in particular related to reducing the number of distinct type variables present in solver states. For example, their solver avoids substituting old type variables with new ones, an issue we discussed in Section 5.3.2.5. While their formalism does not directly incorporate ranks (see Section 5.3.3), their system is set up to support their usage easily.

The author of this work believes that many of these optimisations could be added to our solver, at the cost of making its rules more complex.

Non-deterministic rules Pottier and Rémy's HM(X) solver relies on a non-deterministic transition relation between states. One on hand, this means that each rule serves a specific purpose, avoiding complex rules such as our S-LETPOP$\star$ in Figure 5.8, which lowers existentials, performs generalisation and turns the solved let frame into a def constraint. On the other hand, understanding the overall workings of the HM(X) solver requires an understanding of the subtle interplay between a larger number of rules.

It would nevertheless be interesting to investigate if it is possible to re-define our solver in such a fashion.

Chapter 6

Discussion

This chapter explores practical work in FreezeML, showing how two programming techniques relying on first-class polymorphism can be implemented using FreezeML. Potential extensions and variations to FreezeML's design are also sketched.

Further, we discuss some formal properties of FreezeML and two implementations of it. Finally, we compare FreezeML with the existing approaches towards first-class polymorphism discussed in Section 2.3.

Throughout this chapter, we generally assume that the following functions are in scope.

$$\begin{aligned}
\mathsf{id} &: \forall a.a \to a \\
\mathsf{auto} &: (\forall a.a \to a) \to (\forall a.a \to a) \\
\mathsf{choose} &: \forall a.a \to a \to a \\
\mathsf{singleton} &: \forall a.a \to \mathsf{List}\ a \\
\mathsf{app} &: \forall a,b.(a \to b) \to a \to b \\
\mathsf{revapp} &: \forall a,b.a \to (a \to b) \to b
\end{aligned}$$

6.1 Programming in FreezeML

We now discuss two real-world use-cases of first-class polymorphism and how they can be used in FreezeML. In order to ensure that the examples stay (mostly) within the vanilla FreezeML language introduced in this work, we picked two example use-cases that do not rely on additional advanced typing features, such as higher-kinded types or type classes.

6.1.1 Existential types

We briefly mentioned in Section 3.2.1.3 that existential types $\exists a.A$ can be encoded as System F types $\forall b.(\forall a.A \to b) \to b$.

We follow Le Botlan's example program [55, Section 12.2] of a server processing tasks, originally given in the context of MLF. Each task consists of a value of some type A and a function from A to Unit. The server then acts on a list of such tasks, applying each function to the corresponding argument, where each argument may have a different type A. To allow the server to process a list of these tasks, we represent each task with type $\exists a.a \times (a \to \mathsf{Unit})$, effectively hiding the heterogeneous nature of the tasks.

In the following, we use syntax combining let bindings with function definitions found in many functional programming languages. Concretely, **let** $f\, x_1\, \ldots\, x_n = M$ **in** N is syntactic sugar for **let** $f = \lambda x_1\, \ldots\, x_n.M$ **in** N. Potential annotations on any of the x_i are simply transformed to annotations on the corresponding lambda binder. We also use type annotations of the form $(M : A)$, which are syntactic sugar for $\lambda(x : A).\lceil x \rceil)\, M$. We omit the part **in** N of each binding, simulating toplevel let bindings without any further effects on typing.[1]

We use $\mathsf{Task}\ a$ as a parameterised type alias that expands to $a \times (a \to \mathsf{Unit})$. While such type aliases are available in Links, we only use them here to obtain a more concise presentation of the types of our examples, but they will not appear in actual terms.

Le Botlan first defines a function $\mathsf{encapsulate}$, which given a task t performs the encapsulation into an existentially typed value.

$$\textbf{let}\ \mathsf{encapsulate}\, t = \$^{\overbrace{\forall b.(\forall a.(a \times a \to \mathsf{Unit}) \to b) \to b}^{\exists a.\mathsf{Task}\ a}}\, \lambda(f : \forall a.(a \times a \to \mathsf{Unit}) \to b).f\, t$$

While both MLF and FreezeML require an annotation on f, we additionally require an annotated generalisation operator in FreezeML. The annotation on the operator is required in order to bind the type variable b, which then appears freely in the annotation on f.

Using the Task type alias and the aforementioned definition of existential types, we then have the following.

$$\mathsf{encapsulate}\ : \forall c.(\mathsf{Task}\ c \to \exists a.\mathsf{Task}\ a)$$

[1]Alternatively, the reader may imagine that each let binding here ends with a dangling **in** and continues at the next code snippet shown in the text.

As in MLF, we can define create as a curried version of encapsulate (that also swaps the order of arguments) without needing an annotation.

$$\textbf{let}\ \mathsf{create}\ f\ \mathsf{arg} = \mathsf{encapsulate}\ (\mathsf{arg}, f)$$

The function serve then performs a single task; its type is $(\exists a.\mathsf{Task}\ a) \to \mathsf{Unit}$.

$$\textbf{let}\ \mathsf{serve}\ (t : \overbrace{\forall b.(\forall a.(a \times a \to \mathsf{Unit}) \to b) \to b}^{\exists a.\mathsf{Task}\ a}) = t\ \$\big(\lambda p.((\mathsf{snd}\ p)\ (\mathsf{fst}\ p) : \mathsf{Unit})\big)$$

In contrast to MLF, we require an annotation on the task t as it has a polymorphic type. Further, we need to explicitly perform generalisation to ensure that the function passed to t is polymorphic. Note that we also need to annotate the body of the function, simply writing $\$(\lambda p.(\mathsf{snd}\ p)(\mathsf{fst}\ p))$ generalises to the type $\forall a, b.(a \times a \to b) \to b$ instead, which cannot be passed to t. Thus, the annotation Unit on the body of the lambda forces b to be Unit instead. Alternatively, we may use the annotated generalisation operator, requiring a longer annotation.

The implementation of the server then remains unchanged as compared to MLF.

$$\begin{aligned}&\textbf{let}\ \mathsf{task\_list} = [\mathsf{create}\ \mathsf{print\_int}\ 1;\ \mathsf{create}\ \mathsf{print\_string}\ \text{“hello”}]\\&\textbf{let}\ \mathsf{server} = \mathsf{iter\_list}\ \mathsf{serve}\ \mathsf{task\_list}\end{aligned}$$

Here, we use auxiliary functions with the following, unsurprising types.

$$\begin{array}{rcl}\mathsf{print\_int} & : & \mathsf{Int} \to \mathsf{Unit}\\ \mathsf{print\_string} & : & \mathsf{String} \to \mathsf{Unit}\\ \mathsf{iter\_list} & : & \forall a.(a \to \mathsf{Unit}) \to \mathsf{List}\ a \to \mathsf{Unit}\end{array}$$

6.1.2 State threads

Another use-case of first-class polymorphism that does not rely on other advanced typing features are *state threads*[2] [54, 53]. Launchbury and Peyton Jones demonstrate how this approach allows introducing mutable references into a language while preserving referential transparency by encapsulating a stateful computation that creates a value of type a inside a monad $\mathsf{ST}\ t\ a$. We will return to the role of the type parameter t shortly.

Unlike Haskell's IO monad, ST allows the result of the stateful computation to escape. This is achieved by applying runST to the computation v, which effectively executes v and returns its result. However, it is carefully designed such that executing

[2]Note that the usage of the term “thread” here is not related to parallelism, but expresses the independence of stateful operations.

v cannot read or modify references created outside of v. This can be achieved by using higher-rank types, rather than runtime checks. A phantom type variable t is used to track which *thread* each reference appears in.

To this end, we may use a type $\mathsf{sref}\ t\ a$ of safe references storing values of type a, where the variable t denotes the thread. The following operations are then defined to create, read, and update such references. Here, ! and := are prefix and infix operators, respectively. While their syntax resembles the ML operators on references, their types are more involved.

$$\begin{aligned} \mathsf{mkref} &: \forall a,t.\, a \to \mathsf{ST}\ t\ (\mathsf{sref}\ t\ a) \\ (!) &: \forall t,a.\, \mathsf{sref}\ t\ a \to \mathsf{ST}\ t\ a \\ (:=) &: \forall t,a.\, \mathsf{sref}\ t\ a \to a \to \mathsf{ST}\ t\ \mathsf{Unit} \end{aligned}$$

We observe that mkref does not return a reference directly, but yields a *computation* creating the reference. Crucially, the thread parameter t of that reference is coupled to the thread parameter of the encapsulating computation. While the ! and := operators act on references directly, and may thus be used within a computation, they return computations.

The monadic nature of ST becomes apparent when the following two operations are added.

$$\begin{aligned} \mathsf{return} &: \forall a,t.a \to \mathsf{ST}\ t\ a \\ \mathsf{then} &: \forall t,a,b.\, \mathsf{ST}\ t\ a \to (a \to \mathsf{ST}\ t\ b) \to \mathsf{ST}\ t\ b \end{aligned}$$

Here, the function then is the monad's *bind* operation, used to sequence two computations by giving access to the result of the first one. We can interpret the usage of t in then's type such that then $v\ f$ is only well typed if the thread variables of both v and f allow them to be part of the same thread.

So far we have not given the type of runST. The idea of the thread parameter is that it should precisely rule out programs such as the following. Here, we use $\lambda().M$ as syntactic sugar for $\lambda(x : \mathsf{Unit}).M$, where x is fresh.[3] We also allow () to appear in the syntactic sugar for let bindings mixed with function definitions introduced in Section 6.1.1

[3]This is of course just a special case of the ability to use general-purpose patterns in place of binders found in many functional programming languages.

$$\textbf{let } v = \mathsf{runST}\ (\mathsf{mkref}\ 0) \tag{6.1}$$

$$\textbf{let } \mathsf{setv}\ x = \mathsf{runST}\ (v := x) \tag{6.2}$$

$$\textbf{let } \mathsf{readv}\ () = \mathsf{runST}\ (!v) \tag{6.3}$$

If this program were accepted, referential transparency would be lost: The value of $\mathsf{readv}\ ()$ depends on prior invocations of setv.

Launchbury and Peyton Jones's solution is that in order for $\mathsf{runST}\ M$ to be well typed, M must not just be a state thread, but its thread variable t must be generalisable. The way the different operations that may appear in M force equality between thread variables then means that M cannot access any references that are present in the surrounding context. Thus, the type of runST is as follows.

$$\mathsf{runST} : \forall a.(\forall t.\mathsf{Unit} \to \mathsf{ST}\ t\ a) \to a$$

Here, in contrast to the canonical definition of runST, we thunk the computation passed to runST with a Unit argument simply to circumvent the FreezeML value restriction, in order to permit generalisation of t even when M is a function application (e.g., M is a usage of then at the toplevel).[4]

Note that with this type of runST, the earlier example is indeed rejected, even when performing the necessary thunking and inserting a generalisation operator. We first consider (6.1). While the term $\$(\lambda().\mathsf{mkref}\ 0)$ indeed has type $\forall t.(\mathsf{Unit} \to \mathsf{ST}\ t\ \mathsf{Int})$, it cannot be passed to runST. This is due to the the quantifier t escaping its scope in the hypothetical return type $\mathsf{sref}\ t\ \mathsf{Int}$ of $\mathsf{runST}\ \ \$(\lambda().\mathsf{mkref}\ 0)$.

We also cannot adapt (6.3) to become well typed. If we have $v : \mathsf{ST}\ t'\ \mathsf{Int}$ in the term context, where t' is free, then $\$(\lambda().\,!v)$ cannot generalise t', meaning that the term cannot be supplied to runST. While we cannot create a variable of such a type using terms such as (6.1), it is still useful to define helper functions called from within a computation to receive sref arguments. We will see such an example shortly when discussing an example in Section 6.1.2.1.

Finally, if the term context contained $v : \forall t.\mathsf{ST}\ \mathsf{Int}\ t$ instead, the term $\mathsf{runST}\ \$(\lambda().\,!v)$ becomes well typed. However, the types of the functions related to ST shown so far prevent us from creating values of this type.

[4]Note that in order for a language to offer referential transparency in the presence of threaded state, it must of course not provide other, unconstrained means of providing side effects, such as ordinary references usually found in ML. However, in that case, the value restriction would not be necessary to have in the language in the first place and we may use a variant of FreezeML without it, which we return to in Section 6.2.1.3.

6.1.2.1 Stateful Fibonacci numbers

So far, we have shown how state threads can *not* be used. We now show a working example program, namely the calculation of Fibonacci numbers using mutable state. For readability, we define **let**$*\ x = M$ **in** N as syntactic sugar for the term then $M\ (\lambda x.N)$, similarly to Haskell's **do** notation.

```
let fib n =                                          fib : Int → Int
  let rec fib′ i v₁ v₂ =      fib′ : ∀t.Int → sref Int t → sref Int t → ST Int t
    if i ≤ 0 then
      !v₂
    else
      let* x₁ = !v₁ in
      let* x₂ = !v₂ in
      let* () = (v₂ := x₁) in
      let* () = (v₁ := x₁ + x₂) in
      fib′ (i − 1) v₁ v₂
  in
  runST $(λ().
    let* v₁ = mkref 1 in
    let* v₂ = mkref 0 in
    fib′ n v₁ v₂)
```

Figure 6.1: Example function calculating Fibonacci numbers with safe mutable reference

The function fib in Figure 6.1 creates two references v_1 and v_2. The helper function fib$'$ evokes that if v_1 contains the j-th Fibonacci number and v_2 contains its predecessor, then after executing fib$'\ k\ v_1\ v_2$ we have that v_1 contains the $(j+k)$-th Fibonacci number and v_2 contains its predecessor.

The function fib$'$ is recursive, we thus assume the usual typing of recursive bindings, meaning that fib$'$ has a monomorphic type, and only receives the polymorphic type shown on the right in Figure 6.1 for subsequent use. We discuss *polymorphic recursion*, where a recursive function may receive a polymorphic type *inside* its own body, later in Section 6.2.2.2. In fact, the helper function fib$'$ is entirely typeable in the

HM system; the only part of the example in Figure 6.1 going beyond HM typeability are the type of runST and the generalisation operator around the argument to runST.

While the type hints shown in Figure 6.1 and the subsequent example in Figure 6.2 are helpful from a usability perspective, they are not necessary for the programs to type-check. Of course, all examples would remain valid if we used annotated let bindings with the shown types.

Launchbury and Peyton Jones proposed state threads in the context of Haskell. They point out that due to Haskell's lack of support for first-class polymorphism (unless using certain GHC language extensions), runST requires ad-hoc treatment by the type-checker, similarly to its treatment of the infix function application operator $. This special treatment prohibits some usages of runST as a regular function. In contrast, we can use runST as a first-class function in FreezeML without requiring any ad-hoc treatment, as illustrated in the following example in Figure 6.2. Here, we assume that fib′ has been factored out of fib in Figure 6.1 and is available as a standalone function. Of course, a corresponding version of this program *can* be written in Haskell by using appropriate GHC extensions.

```
let mkfibST n = $(λ().                    mkfibST : Int → (∀t.Unit → ST Int t)
   let∗ v1 = mkref 1 in
   let∗ v2 = mkref 0 in
   fib′ n v1 v2)
in
let fibSTs =                                  fibSTs : List (∀t.Unit → ST Int t)
   [mkfibST 3; mkfibST 5; mkfibST 10]
in
map runST fibSTs
```

Figure 6.2: Example for using runST as a first-class function

Of course, the example is inefficient in the sense that no partial results are shared between the three computations being executed.

6.2 Variations and Extensions of FreezeML

This section proposes potential variations and extensions of our type system.

6.2.1 Variations

We now discuss ideas that would change the behaviour of FreezeML in certain ways.

6.2.1.1 Unordered FreezeML

FreezeML uses not only the type language of System F, but also its equational theory for types. This means that the order of quantifiers is relevant and we cannot add or remove superfluous (i.e., not appearing freely under the quantifier) variables at will. Concretely, it means that none of the following types are equivalent in FreezeML.

$$\begin{aligned}&\forall a,b.a \to b \to a\\ &\forall b,a.a \to b \to a\\ &\forall a,b,c.a \to b \to a\end{aligned}$$

Choosing this equational theory for FreezeML is rooted in the perspective of treating programming in FreezeML as a way of seamlessly mixing ML and System F. This choice of equational theory has some advantages over an “unordered” theory, where the order of quantifiers is disregarded. For example, adopting the latter for FreezeML would mean that the term $\$(\lambda x y.x)$ could be given both of the following two types.

$$\begin{aligned}\$(\lambda x,y.x) &: \forall a,b.a \to b \to a\\ \$(\lambda x,y.x) &: \forall b,a.a \to b \to a\end{aligned}$$

Choosing between these two types leads to different System F translations, even though the types would be equivalent, unique types of the term in this variant of FreezeML. Unfolding the syntactic sugar for the \$ operator in the term and applying the translation from FreezeML to System F in Figure 3.9 on page 62 results in the following, different System F terms.

$$\begin{aligned}\cdot;\cdot \vdash \$(\lambda x,y.x) &: \forall a,b.a \to b \to a \rightsquigarrow \textbf{let } f = \Lambda a,b.\lambda x^a y^b.x \textbf{ in } f\\ \cdot;\cdot \vdash \$(\lambda x,y.x) &: \forall b,a.a \to b \to a \rightsquigarrow \textbf{let } f = \Lambda b,a.\lambda x^a y^b.x \textbf{ in } f\end{aligned}$$

Another advantage of our “ordered” equational theory is the ability to incorporate type applications directly into the language. In the context of FreezeML, we will consider this extension in Section 6.2.2.1. They play a particularly important role in *Explicit FreezeML*, discussed in Section 7.2.2. Explicit FreezeML subsumes both FreezeML (as syntactic sugar) and System F.

Of course, our choice of equational theory in FreezeML also has disadvantages. Firstly, given $\mathsf{bad}_{\mathsf{f}} : \forall b,a.a \to b \to a$ and $\mathsf{take\_f} : (\forall a,b.a \to b \to a) \to \mathsf{Unit}$, we cannot write take_f ⌈bad_f⌉ directly, but need to re-generalise it by writing take_f \$bad_f instead. If the order was swapped (i.e., we have bad_f : $\forall a, b. a \to b \to a$ and take_f : $(\forall b, a. a \to b \to a) \to \mathsf{Unit}$ instead), the re-generalisation operator would even require an annotation.

Secondly, some type languages do not naturally induce a canonical order of quantifiers (e.g., quantifying variables in their order of appearance by default). For record types, such an ordering might be obtained artificially by ordering its fields alphabetically. Other types, such as union and intersection types do not induce a canonical ordering at all.

Some mixed approaches towards these problems exist: The GI system [110] uses the same equational theory of types as FreezeML, but its ability to implicitly instantiate and re-generalise function arguments allows it to accept take_f bad_f nevertheless. This fails when the order mismatch appears nested within the involved types, meaning that in GI we cannot apply a function of type $(\mathsf{Unit} \to \forall a, b. a \to b) \to \mathsf{Unit}$ to an argument of type $\mathsf{Unit} \to \forall b, a. a \to b$.

Visible type application [18] follows a slightly different approach, introducing a distinction between ordered-agnostic type schemes $\forall\{\overline{a}\}$ and quantified types $\forall\overline{a}$ that respect the order of quantifiers. This reflects a similar distinction between two sorts of polymorphic types being made in to IFX [89] and QML [106]. However, Eisenberg et al. define a subsumption relation between the two sorts of polymorphic types, meaning that the following holds in their system.

$$\forall\{a, b\}. a \to b \to b \;\; \geq \;\; \forall a, b. a \to b \to a \;\; \leq \;\; \forall\{b, a\}. a \to b \to a$$

The subsumption relation then allows $\forall\overline{a}$-quantified types to be used where a $\forall\{\overline{a}\}$-quantified type is expected, but not vice versa. This approach is adopted in the Quick Look system [109], but its authors give few details about this particular aspect of their system.

In the context of FreezeML, the author conjectures that it is possible to define a variant of FreezeML that ignores the order of quantifiers. The resulting system should enjoy the same soundness and completeness properties of type inference as the original one. The changes to the inference procedure should be limited to the unification algorithm; an appropriate alternative algorithm is given by Garrigue and Rémy [26]. Alternatively, Leijen [58] suggests in the context of HMF that the inference algorithm can maintain the invariant that all types are in canonical form (i.e., ordering quantifiers based on their first appearance), meaning that we could even leave the unification algorithm unchanged and only canonicalise all user-provided type annotations.

6.2.1.2 Different type variable scoping

In FreezeML, an annotated binding **let** $(x : \forall\overline{a}.H) = M$ **in** N binds the type variables $\overline{a}$ in M, if M is a guarded value. In particular, all type variables appearing freely in an annotation must have been bound this way, unless they are part of some initial type context Δ. A similar approach to the lexical scoping of type variables is adopted in the Boxy Types system [117], where the toplevel quantifiers of any annotation on a let-bound variable are unconditionally bound in the first subterm. Other approaches rely on *partial type annotations*, which treat free type variables appearing in annotations as being existentially quantified [26, 58, 101, 106], possibly limiting their instantiation to monomorphic types.

The lexical scoping behaviour of our approach has both advantages and disadvantages, which become apparent in the body of the encapsulate function that we introduced as an example in Section 6.1.1. It was defined as follows, where we have expanded all syntactic sugar.

```
let encapsulate =
  λt.
    let (box : ∀b.(∀a.(a × a → Unit) → b) → b) =
      λ(f : ∀a.(a × a → Unit) → b).f t
    in
    ⌈box⌉
```

Here, the annotation on box is only required to bind the type variable b, so that it can appear freely in the annotation on f. Using partial annotations would then allow us to drop the annotation on box. However, it may be the intention of the user to be specific about the fact that the return type of f is the same b as in the annotation on box, both for documentation purposes and to spot mistakes more easily.

For example, consider the following variation of the program, where we assume that partial annotations are supported. The only differences are the removal of the annotation on box and the addition of 3 to the result of $f\ t$.

$$
\begin{aligned}
&\textbf{let } \mathsf{bad\_encapsulate} \ = \\
&\quad \lambda t. \\
&\qquad \textbf{let } \mathsf{box} = \\
&\qquad\quad \lambda(f \ : \ \forall a.(a \times a \to \mathsf{Unit}) \to b).(f\,t) + 3 \\
&\qquad \textbf{in} \\
&\qquad \lceil \mathsf{box} \rceil
\end{aligned}
$$

The function bad_encapsulate is well typed, but its type is the following, which will cause subsequent type errors when writing the remaining code shown in Section 6.1.1.

$$\forall c.(c \times c \to \mathsf{Unit}) \to (\forall a.(a \times a \to \mathsf{Unit}) \to \mathsf{Int}) \to \mathsf{Int}$$

Providing the original annotation on box would make the program fail to type-check, as it should. However, in the presence of partial annotations, the occurrence of b in the type of f is unrelated to its quantified occurrence in the annotation on box. Consequently, the type error shown to the user is regarding the mismatch between the type of $(f\,t) + 3$ compared to the expected type of box. The lexical scoping behaviour of FreezeML would instead report more accurately that the term $(f\,t) + 3$ is ill typed, as $(f\,t)$ does not have type Int.

This suggests a combined approach, where users can refer to type variables lexically bound by surrounding bindings, but may also use variables that are existentially bound in annotations. We may adapt the syntax used by QML for this purposes, where all type annotations begin with a list of variables that are existentially quantified. If the user then decides to eschew the annotation on box when defining encapsulate, they may write the definition of the former as $\lambda(f \ : \ b\,\forall a.(a \times a \to \mathsf{Unit}) \to b).f\,t$, explicitly marking b as an unknown, monomorphic type.[5]

The author conjectures that simple changes to the translation from FreezeML terms to constraints are sufficient to yield an inference algorithm for this approach. The changes to Figure 5.4 on page 105 are the following; no changes to the constraint language or its solver are required.

$$
\begin{aligned}
[\![\lambda(x : \overline{c}\,B).M : A]\!] &= \exists \overline{c}, a_1.\big(B \to a_1 \sim A \ \wedge \ \textbf{def}\,(x : B) \ \textbf{in} \ [\![N : a_1]\!]\big) \\
[\![\textbf{let}\,(x : \overline{c}\,\forall \overline{b}.H) = U \ \textbf{in} \ N : A]\!] &= \exists \overline{c}.\big((\forall \overline{b}.[\![U : H]\!]) \ \wedge \ \textbf{def}\,(x : \forall \overline{b}.H) \ \textbf{in} \ [\![N : A]\!]\big) \\
[\![\textbf{let}\,(x : \overline{c}\,B) = M \ \textbf{in} \ N : A]\!] &= \exists \overline{c}.\big([\![M : B]\!] \ \wedge \ \textbf{def}\,(x : B) \ \textbf{in} \ [\![N : A]\!]\big) \ (\text{if } M \notin \mathsf{GVal})
\end{aligned}
$$

[5]QML does permit free variables in type annotations (i.e., not mentioned in the list of existentially quantified variables part of the annotation), but it does not support *binding* such variables elsewhere, as we suggest here. As a result, free type variables in QML annotations act as if the whole program was parameterised over them, similarly to appearing in the initial type context Δ in our approach.

Note that any of the variables c actually appearing in the remainder of the annotation are implicitly monomorphised, when B or $\forall\overline{b}.H$ appear in the annotation on the def constraint. Further, the fact that all $\overline{c}$ are existentially quantified at the toplevel of the resulting sub-constraint implies that for generalising let bindings, none of the variables $\overline{c}$ may be instantiated with $\overline{b}$, the variables generalised at the let binding. Likewise, the variables $\overline{c}$ cannot be instantiated with potential quantifiers appearing in the type B in the other two cases.

This reflects the fact that the following term would be ill-typed under this proposal, to enforce the invariant that we can only quantify over known types.

$$\textbf{let}\ (\mathsf{wrong} : b\ \forall a.a \to b) = \lambda x.x\ \textbf{in}\ \ldots$$

6.2.1.3 FreezeML without the value restriction

So far, we have only considered FreezeML with the value restriction. This is due to the fact that the lexical scoping of typing variables depends on it. Recall that $\textbf{let}\ (x : \forall\overline{a}.H) = M\ \textbf{in}\ N$ binds the type variables $\overline{a}$ in M if and only if M is a guarded, and hence generalisable, value. As discussed in Section 3.2.1.3, if M is a guarded value and the overall let binding is well typed, then we can assign a guarded type to it that is generalised to $\forall\overline{a}.H$. In particular, this means that in the generalisable case, we can safely assume that *all* toplevel quantifiers appearing in the annotation are the result of generalisation *at* the let binding, rather than already being part of the type of M. Conversely, if M is not guarded, no generalisation must take place, and all such quantifiers *must* have already been part of M's type. As a result, the value restriction enables a simple syntactic criterion for where toplevel quantifiers must originate from, and thus, whether or not they should be brought into lexical scope.

Unfortunately, this criterion is invalidated when removing the value restriction from FreezeML. Consider the term $\textbf{let}\ (f : \forall a.a \to a) = \mathsf{foo}\ ()\ \textbf{in}\ \ldots$. If foo's type is $\mathsf{Unit} \to \forall a.(a \to a)$, then no generalisation takes place, the type of $\mathsf{foo}\ ()$ *is* $\forall a.a \to a$. However, if its type is $\forall a.\mathsf{Unit} \to a \to a$, then the type of $\mathsf{foo}\ ()$ is $B \to B$ for any B and may be *generalised* to type $\forall a.a \to a$.[6]

The only other system known to the author adopting a similar lexical scoping strategy is the Boxy Types approach. It does not use the value restriction, but avoids

[6]These two types of foo given here are isomorphic in the sense that there exists a transformation function between terms of each type that is $\beta\eta$-equivalent to the identity function [104]. This is reflected in the type instance/subtype/equivalence relations of some systems [56, 85, 92, 101], but they are unrelated in FreezeML.

the problem stated here by imposing that the types of let bound terms can never have toplevel quantifiers. Indeed, we may enforce a similar rule for let bindings, meaning that the toplevel quantifiers of let-bound annotations are always brought into scope. Assuming $\mathsf{foo} : \mathsf{Unit} \to \forall a.(a \to a)$, we would then have to perform instantiation manually, writing **let** $(f : \forall a.a \to a) = (\mathsf{foo}\ ())@$ **in** $\ldots$. We could also change the behaviour of annotated let bindings **let** $(x : A) = M$ **in** N such that they *implicitly* instantiate $M's$ type and re-generalise it to obtain A. However, the author considers this to subvert the spirit of FreezeML, where instantiation is limited to variables.

Finally, another solution would be to simply adopt a different strategy for lexical scoping: In Section 6.2.1.2, we discussed integrating partial annotations into FreezeML, in combination with the existing lexical scoping mechanism where let bindings may bind type variables. We may simply drop the latter, meaning that all free type variables in annotations are implicitly existentially quantified.

6.2.2 Extensions

We now discuss potential enhancements of FreezeML. These are usually assumed to occur on top of the original definition of FreezeML in Chapter 3, but we discuss the interaction of some extensions with the variations previously discussed in Section 6.2.1.

6.2.2.1 Explicit type applications

In FreezeML, users need not state how variables should be instantiated, even if the instantiation is polymorphic, as this is determined automatically. However, FreezeML always instantiates *all* toplevel quantifiers, mirroring the behaviour of ML. While this behaviour reflects what happens in ML, in the presence of first-class polymorphism, there are situations where we may want to instantiate only some quantifiers, while retaining others.

Consider the function $\mathsf{mkPair} : \forall a, b.a \to b \to (a \times b)$, acting like the data constructor for pairs. In System F, we may write $\mathsf{mkPair}\ \mathsf{Int}$ to obtain a function of type $\forall b.\mathsf{Int} \to b \to (\mathsf{Int} \times b)$. However, in FreezeML, this can only be achieved by instantiating both quantifiers, and then re-generalising against the desired result type, using the annotated generalisation operator in the term $\$^{\forall b.\mathsf{Int} \to b \to (\mathsf{Int} \times b)}\mathsf{mkPair}$. Of course, the term M to supply a type argument to may in general not be a variable already. The translation of System F type applications to FreezeML in Figure 3.8 on page 57 shows that in this case, we need to bind it to a variable first. Thus, the encoding of type applications in FreezeML can be somewhat cumbersome and requires a type annotation showing the desired type after the instantiation in full, rather than just the type argument.

We may therefore consider adding type applications as a language primitive to FreezeML. However, this changes one of the invariants of FreezeML, namely that only variables are instantiated. The system guarantees that all occurrences of polymorphism in the types of variables are uniquely determined. This is no longer the case once arbitrary terms may be applied to types. While we conjecture that the resulting system still enjoys principal types and complete type inference, the invalidation of the aforementioned invariant complicates its meta-theory. We revisit explicit type applications in the context of *Explicit FreezeML* in Section 7.2.2.

6.2.2.2 Recursive definitions

Practical programming languages usually provide programmers with means to define recursive functions. We already had to rely on them in our example program in Figure 6.1 on page 160 without discussing them in detail. We may thus add bindings of the form **let rec** $f = M$ **in** N to FreezeML, where f is in scope in M and N. However, care is required when typing such expressions. Even allowing *polymorphic recursion* in the style of Mycroft [83], where f can receive the same ML type scheme in M and N, leads to undecidable type inference [36, 49].

We therefore propose the following translation rules from recursive let bindings to constraints, augmenting the existing translation function shown in Figure 5.4 on page 105. It adopts the standard solution of forcing the type of f to be monomorphic in the first subterm of the let binding.

$$\begin{aligned}
[\![\mathbf{let\ rec}\, f = U \,\mathbf{in}\, N : A]\!] &= \mathbf{let}_\star\, f = \sqcap a.\big(\mathbf{def}\,(f : a)\ \mathbf{in}\ [\![U : a]\!]\big)\ \mathbf{in}\ [\![N : A]\!] \\
[\![\mathbf{let\ rec}\, f = M \,\mathbf{in}\, N : A]\!] &= \mathbf{let}_\bullet\, f = \sqcap a.\big(\mathbf{def}\,(f : a)\ \mathbf{in}\ [\![M : a]\!]\big)\ \mathbf{in}\ [\![N : A]\!]\ (\text{if } M \notin \mathsf{GVal})
\end{aligned}$$

Here, we add a def constraint that brings f into scope in the sub-constraints $[\![U : a]\!]$ and $[\![M : a]\!]$, respectively. Its type is the same variable a that will be generalised or instantiated (based on which sort of let constraint is generated, in line with the value restriction) to yield the type of f used in the sub-constraint $[\![N : A]\!]$. Here, by appearing freely in the annotation of a def constraint, a is implicitly monomorphised. Recall that this does not prevent it from being generalised.

As a result we have that the following constraint is satisfiable (the program itself would of course diverge when executed). For clarity, we alpha-renamed the monomorphised definition of id to $\mathsf{id_m}$.

$$
\begin{array}{l}
[\![\textbf{let rec}\ \mathsf{id} = \lambda x.\mathsf{id}\ x\ \textbf{in}\ \lceil \mathsf{id} \rceil\ :\ \forall b.b \to b]\!]\ = \\[1em]
\textbf{let}_\star\ \mathsf{id} = \sqcap a. \\
\quad \textbf{def}\ (\mathsf{id_m} : a)\ \textbf{in} \\
\qquad \exists a_1, a_2. \\
\qquad\quad (a_1 \to a_2) \sim a \wedge \\
\qquad\quad \textbf{def}\ (x : a_1)\ \textbf{in}\ \exists a_3.\mathsf{id_m} \preceq (a_3 \to a_2) \wedge x \preceq a_3 \\
\textbf{in} \\
\lceil \mathsf{id} : \forall b.b \to b \rceil
\end{array}
$$

Note that neither of the two instantiation constraints performs instantiation, as both $\mathsf{id_m}$ and x have monomorphic types with no toplevel quantifiers. The first subconstraint of the overall let constraint is logically equivalent to $\exists c.a \sim (c \to c) \wedge \mathsf{mono}(c)$.

Polymorphic recursive bindings can be allowed by requiring them to be annotated. We may therefore add a recursive counterpart to our annotated let bindings, using the following constraint translation rules.

$$
\begin{array}{rcl}
[\![\textbf{let rec}\ (f : \forall \overline{b}.H) = U\ \textbf{in}\ N : A]\!] & = & \textbf{def}\ (f : \forall \overline{b}.H)\ \textbf{in}\ \big((\forall \overline{b}.[\![U : H]\!]) \wedge [\![N : A]\!]\big) \\
[\![\textbf{let rec}\ (f : B) = M\ \textbf{in}\ N : A]\!] & = & \textbf{def}\ (f : B)\ \textbf{in}\ \big([\![M : B]\!] \wedge [\![N : A]\!]\big)\ \ (\text{if } M \notin \mathsf{GVal})
\end{array}
$$

As compared to the translation of the non-recursive variants, we merely moved the constraints $\forall \overline{b}.[\![U : H]\!]$, respectively $[\![M : B]\!]$, such that they appear under the def constraint binding f.

It remains to give typing rules for the two new flavours of recursive let bindings and prove that the correctness of the translation, as expressed in Theorems 5.1 and 5.2, still holds in presence of the new constraint generation rules shown here.

6.2.2.3 Type annotation propagation

The undecidability of full type inference in the presence of first-class polymorphism means that all approaches discussed in Section 2.3 (as well as FreezeML) require users to provide type annotations to make some programs well typed. As a result, some of these systems go to great lengths to propagate the user-provided types throughout the type system as far as possible, in order to minimise the need for annotations [117, 109].

Nevertheless, it is still possible to avoid some duplication occurring in annotations without greatly complicating the system, escaping "the siren call of mixing type annotation with type inference", as Leijen [58] puts it. Rémy [101] suggests a sophisticated technique for type annotation propagation, which acts as a preprocessing step before type-checking/inference. A simpler variant was suggested for HMF [58], which focuses on deconstructing annotations on whole terms into suitable annotations for its subterms.

For FreezeML, we may add the following, simple rule, evoking that function parameter types contained in the overall annotation on a let binding are propagated inwards onto the lambda binders.

$$\begin{gathered}\mathbf{let}\ (f : \forall \overline{a}.A_1 \to \cdots \to A_n \to B) = \lambda x_1, \ldots, x_n.M\ \mathbf{in}\ N \\ \rightsquigarrow \\ \mathbf{let}\ (f : \forall \overline{a}.A_1 \to \cdots \to A_n \to B) = \lambda (x_1 : A_1), \ldots, (x_n : A_n).M'\ \mathbf{in}\ N'\end{gathered}$$

Here, M' and N' result from applying the rule recursively.

This would already allow us to avoid the duplication of information exhibited by the two annotations needed in the definition of the example function encapsulate in Section 6.1.1. Concretely, the suggested propagation scheme would allow us to rewrite it as follows, omitting the annotation on f. Recall that $\$^A\, M$ is syntactic sugar for **let** $(x : A)$ **in** $\lceil x \rceil$, meaning that the propagation rule is applicable.

$$\mathbf{let}\ \mathsf{encapsulate}\ t = \$^{\forall b.(\forall a.(a \times a \to \mathsf{Unit}) \to b) \to b}\ \lambda f.f\ t$$

6.3 Implementations

FreezeML has been implemented in two ways, which we discuss here.

6.3.1 FreezeML in Links

As mentioned in the introduction, FreezeML was originally devised in the context of the Links programming language [11, 69]. Indeed, Jonathan Coates, while an undergraduate student intern contributing to the Links project, was able to replace Links' existing treatment of first-class polymorphism with the FreezeML approach, despite the multitude of type system features present in the language (described in Section 1.2). In line with Links' existing approach towards type inference loosely based on Algorithm W (but using mutable union-find structures), this implementation is inspired by the description of unification and inference in Chapter 4. We believe that this suggests that FreezeML interacts well with a variety of other, orthogonal language features and is a suitable candidate for integration into a variety of programming languages.

FreezeML is implemented in Links largely following the description of the system in Chapter 3. Following Links' existing behaviour, this implementation of FreezeML obeys the value restriction. It also uses the equational theory of System F types with respect to the ordering of quantifiers in types. The only notable difference between FreezeML as described in this work and its implementation in Links are the following.

1. In order to be compatible with Links' existing scoping behaviour of type variables, it allows free type variables to appear in terms which are implicitly bound at the enclosing toplevel binding.

2. Rather than treating the restrictions R on type variables as a standalone concept, they are integrated into Links' kinding system.[7] For example, this kind system is also used to indicate when a type variable is a row variable, or to indicate that type variable may be instantiated with linear types. The kinding system is also used by Links' session typing and language-integrated query features.

The second difference above is the only friction point between FreezeML and other language features. Consider the well typed Links example program in Figure 6.3 on the following page. It contains two definitions of the identity function and the function `auto`, which expects the polymorphic identity function as its parameter and returns it.

The keyword **`fun`** denotes the beginning of a function definition, which may optionally be equipped with a signature, stating its name and whole type, using the keyword **`sig`**. Function parameters in Links are always enclosed in parentheses, which is also reflected in how function types are printed. Function bodies are enclosed in braces; the language is expression-based and the expression appearing in the function body is its return value.

The example contains two identical definitions of the identity function, with and without a signature. Both definitions are accepted, we return to the role of the annotation on `id2` shortly. The function `auto`'s parameter must be annotated, since it has a

[7]For this reason, what we call the *restriction* of a type variable in this work was originally referred to as its *kind* in an earlier publication [20].

```
fun id1(x) {
  x
}

sig id2 : forall a, e::Row. (a) -e-> a
fun id2(y) {
  y
}

fun auto(f : (forall a, e::Row. (a) -e-> a)) {
 ~f
}
```

Figure 6.3: Example showing Links version of auto and two alternative definitions of id

polymorphic type.

The only notable difference between the types id : $\forall a.a \to a$ and auto : $(\forall a.a \to a) \to (\forall a.a \to a)$ that the reader may expect and those appearing in Figure 6.3 are the following: Due to Links' type effect system and resulting support for effect polymorphism, the function arrows appearing in `id2`'s type and the parameter type of `auto` additionally contain a row variable `e`. In each case, the variable `e` has kind `Row` and is universally quantified, in addition to `a`. This reflects that the identity function as well as the parameter `f` of `auto` are polymorphic not just in their argument type, but also their effects. Links' type system can infer effects, and effect polymorphism integrates seamlessly into FreezeML.

Typing the identity function in Links For reasons unrelated to effect types, the inferred type of `id1` in Figure 6.3 is not the same type as the one of `id2`. This is due to Links detecting that the parameter `x` of `id1` is used linearly (i.e., exactly once), which is reflected in its kind. Conversely, the annotation on `id2` implicitly states that the type `a` of `y` may not be instantiated with linear types. This means that the inferred type of `id1` is the following.

```
forall b::Type(Any,Any), e::Row. (b) -e-> b
```

Here, the kind `Type(Any,Any)` of `b` is different from the kind `Type(Unl,Any)` implicitly given to `a` in the other annotations. Here, `Type` followed by `Any` is the kind of proper types (e.g., not a row) that can be instantiated with any type, including linear ones. Conversely, `Unl` indicates that the overall kind of `b` does not permit instantiation with linear types. The second component of the tuple following `Type` in either kind is not relevant for this discussion.

While Links supports notions of subtyping and subkinding, its type system does not have any kind of subsumption rule based thereon (i.e., it does not support implicit coercion). As a result, we have that the application `auto(~id1)` fails due to a type mismatch, while `auto(~id2)` is accepted. Here, $\sim$ is the syntax for the freeze operator in Links, while function application uses C-style parentheses.

Alternatively, we could change the annotation on the parameter `f` to restrict `a` to linear types in order to accept the un-annotated `id1` as an argument, but not `id2`. This is, of course, a reasonable restriction to impose on the identity function. In general, this example does however illustrate how Links' eager desire to infer linearity, which is generally a reasonable design choice, can increase the need for annotations on functions that should be passed around polymorphically.

The Links team has so far avoided implementing implicit sub-typing coercions, as many of the traditional use-cases of sub-typing are covered by using row polymorphism in Links. However, the team could consider implementing a simple subsumption rule for subtypes as long as the types only differ in their kinds.

6.3.2 Standalone implementation of FreezeML using constraint-based inference approach

The author has created a standalone implementation of the constraint-based type inference approach from Chapter 5 in OCaml. The implementation of the translation from terms to constraints and the constraint solver closely follow the corresponding definitions in Figures 5.4 and 5.8.

However, the solver is parameterised in the unification algorithm on which it relies. The author has implemented both the unification algorithm shown in Figure 4.4 as well as the unification algorithm proposed by Garrigue and Rémy [26]. The former algorithm is used by default, whereas the latter algorithm is used to achieve the behaviour described in Section 6.2.1.1, namely ignoring the order of quantifiers as well as superfluous ones.

We have validated this implementation (including both notions of type equivalence) using a test suite containing a collection of examples exhibiting first-class polymorphism found in existing literature. This collection will be further discussed in Section 6.5.6.

6.4 Properties of FreezeML

We already formalised the relationship between FreezeML and System F by showing translations between the two systems in Section 3.3. Further, we gave a type substitution property for FreezeML in Section 4.1.

We now discuss further formal properties of FreezeML. First, we show that the latter is indeed a conservative extension of ML in the sense that all well typed ML programs are accepted in FreezeML (with the same types), while FreezeML does not accept ML programs that are rejected in ML itself.

We also discuss additional properties of ML and other systems supporting first-class polymorphism and whether they hold in FreezeML.

6.4.1 Relating ML and FreezeML

We now show that FreezeML is indeed a conservative extension of ML. To formalise this, we refer to types of the form $\forall \overline{a}.S$ as ML type schemes and ML terms correspond to the subset of FreezeML terms shown in Figure 2.1 on page 13 (i.e., not containing frozen variables and no annotated let or lambda binders).

We observe that FreezeML might give some ML terms unnecessarily polymorphic types. For example, assuming that id is assigned its usual type in the context, which is an ML type scheme, we may give the term id the type $(\forall a.a) \to (\forall a.a)$ in FreezeML, which is neither a valid ML type scheme nor derivable using the ML types rules in Figure 2.6 on page 17.

However, we observe that the *principal* type of any ML term is always monomorphic in FreezeML.

Lemma 6.1.
*Let M be an ML term and let $\Delta \vdash \Gamma$ **ok** hold, where Γ only contains ML type schemes. Then $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ implies that A is a monotype.*

Proof. By reasoning about the type inference algorithm in Figure 4.5 on page 86, details in Appendix C.1. □

We also observe that FreezeML type derivations for ML programs may assign non-HM types to some subterms, but this is not necessary to type them, as there always exists a derivation that only assigns monotypes to each term.

Lemma 6.2.
*Let M be an ML term and let $\Delta \vdash \Gamma$ **ok** hold, where Γ only contains ML type schemes. Then $\Delta;\Gamma \vdash M : S$ implies that there exists a derivation of the latter judgement where every subterm of M is assigned a monomorphic type.*

Proof. All cases are straightforward by induction on the structure of M, except for term applications, where we reason about the type inference algorithm again. See Appendix C.1 for details. □

We use the previous two lemmas to prove the main theorem about the relationship between ML and FreezeML.

Theorem 6.1 (Correspondence of ML and FreezeML on ML subset)**.**
*Let M be an ML term and let $\Delta \vdash \Gamma$ **ok** hold, where Γ only contains ML types. Then the following holds.*

1. *We have $\Delta;\Gamma \vdash^{ML} M : S$ iff $\Delta;\Gamma \vdash M : S$.*

2. *S is the principal type of M under Δ and Γ in ML iff $\mathsf{principal}(\Delta,\Gamma,M,\mathsf{ftv}(S) - \Delta, S)$ holds.*

Proof. We show both properties simultaneously by induction on the structure of the term under consideration.

- First part, $\Rightarrow$ direction: All cases are straightforward, except for the case **let** $x = M$ **in** N. According to the ML type derivation we have $(\Delta,\Delta');\Gamma \vdash^{\mathrm{ML}} M : S'$ and $\Delta;(\Gamma, x : \forall\Delta'.S') \vdash^{\mathrm{ML}} N : S$. Here, Δ' is defined by the gen helper function. In ML, there must exist a principal type S_{p} of M under Δ and Γ; let us define $\Delta_{\mathrm{p}} := \mathsf{ftv}(S_{\mathrm{p}}) - \Delta$. By the second part of this theorem we then have $\mathsf{principal}(\Delta, \Gamma, M, \Delta_{\mathrm{p}}, S_{\mathrm{p}})$. Note that the latter immediately implies $(\Delta,\Delta_{\mathrm{p}});\Gamma \vdash M : S_{\mathrm{p}}$.

  If $M \in \mathsf{GVal}$ we have that $\Delta' = \mathsf{ftv}(S') - \Delta$. We have that $\forall\Delta'.S'$ is a generic instance of the principal type scheme $\forall\Delta_{\mathrm{p}}.S_{\mathrm{p}}$ and therefore have that $\Delta;(\Gamma, x : \forall\Delta'.S') \vdash^{\mathrm{ML}} N : S$ implies $\Delta;(\Gamma, x : \forall\Delta_{\mathrm{p}}.S_{\mathrm{p}}) \vdash^{\mathrm{ML}} N : S$ (Lemma 2.2). We then get $\Delta;(\Gamma, x : \forall\Delta_{\mathrm{p}}.S_{\mathrm{p}}) \vdash N : S$ using the induction hypothesis.

  Otherwise, if $M \notin \mathsf{GVal}$, we have $\Delta' = \cdot$. Due to S_{p} being principal, we have that $\forall\Delta'.S' = S'$ can be obtained from S_{p} using a monomorphic instantiation, as required by the $\Updownarrow$ premise of the FreezeML derivation. We then get $\Delta;(\Gamma,x : S') \vdash N : S$ directly using the induction hypothesis.

- First part, $\Leftarrow$ direction: In the (plain) variable case, we have that an ML type scheme is instantiated to the monomorphic type S, meaning that the instantation is monomorphic and we can re-use it in the corresponding ML derivation. Term abstractions are straightforward.

  For term applications $M\,N$, we rely on Lemma 6.2 to show that there exists S' such that M and N can be given types $S' \to S$ and S', respectively. This is crucial in order to be able to apply the induction hypothesis.

  For the case **let** $x = M$ **in** N, we use Lemma 6.1 to show that the principal type A' of M is monomorphic. Let $\Delta' := \mathsf{ftv}(A') - \Delta$. We can then apply the induction hypothesis to $(\Delta,\Delta');\Gamma \vdash M : A'$ and $\Delta;(\Gamma,x : A) \vdash N : S$. If $M \in \mathsf{GVal}$, $\Updownarrow$ dictates $A = \forall\Delta'.A'$ and we can directly use the two ML typing judgements obtained above for the corresponding derivation of the overall let term in ML. Otherwise, if $M \notin \mathsf{GVal}$, we have that A results from A' by using a monomorphic instantiation δ with $\Delta \vdash \Delta' \Rightarrow_{\star} \cdot$, making it a monotype. To obtain the typing premise for M in the ML derivation, we can apply δ to $\Delta;\Gamma \vdash^{\mathrm{ML}} M : A'$ (using Lemma 2.3), yielding the desired judgement $\Delta;\Gamma \vdash^{\mathrm{ML}} M : A$.

- Second part, $\Leftarrow$ direction: Let $\Delta' := \mathsf{ftv}(S) - \Delta$. By assumption $\mathsf{principal}(\Delta,\Gamma, M,\Delta',S)$ we have $(\Delta,\Delta');\Gamma \vdash M : S$ and thus by the first part of this theorem, $(\Delta,\Delta');\Gamma \vdash^{\mathrm{ML}} M : S$. We now assume $(\Delta,\Delta'');\Gamma \vdash^{\mathrm{ML}} M : S''$ for some Δ'',S''. We need to show that there exists an ML instantiation δ such that $S'' = \delta(S)$ and $\Delta \vdash \delta : \Delta' \Rightarrow_{\bullet} \Delta''$. (Note that the ML and FreezeML well-formedness relation for instantiations coincide for *monomorphic* ones). By $\mathsf{principal}(\Delta,\Gamma,M,\Delta',S)$ we have that such an instantiation exists, but it may be polymorphic (i.e., we only have $\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta''$). However, the fact that all variables in Δ' appear in S and $\delta(S)$ is monomorphic implies that δ is indeed monomorphic.

- Second part, $\Rightarrow$ direction: Let $\Delta' := \mathsf{ftv}(S) - \Delta$. By the existence of principal types (Theorem 4.8) we may assume that there exist A_{p} and Δ_{p} such that $\mathsf{principal}(\Delta,\Gamma,M,\Delta_{\mathrm{p}},A_{\mathrm{p}})$. Further, we assume that, A_{p} and S cannot be obtained from each other using a bijective renaming of free type variables (i.e., we assume that there exists a principal type of M that is different from S). By Lemma 6.1 we then have that A_{p} is a monotype and by the right-to-left direction of the second part of this theorem we then have that A is a principal type of M under Δ and Γ in ML. As we assumed that S is such a principal type in ML, this violates the uniqueness of principal types in ML (see Lemma 2.6), meaning S can in fact be obtained from A by renaming free variables which in turn makes $\mathsf{principal}(\Delta, \Gamma, M, \Delta', S)$ hold (see Lemma 3.1).

Note that although individual parts of the proof directly rely on each other for the same term, the reasoning is not circular: When proving the first part of the theorem, we apply the second part of the theorem only to subterms. The proofs of the two directions of the second part of the theorem use the first part on the same term. For the second part of the theorem, the proof of the left-to-right direction relies on the right-to-left direction, but not vice versa. □

Note that using a monotype S in Theorem 6.1 does not weaken the right-to-left-direction of the theorem: Every well typed FreezeML term has a principal type (see Theorem 4.8), and we know that ML terms have monomorphic principal types (see Lemma 6.1). Thus, every well typed ML term has a monomorphic (principal) type S in FreezeML, even if it may *also* be typed with other, more polymorphic types such as $(\forall a.a) \to (\forall a.a)$ being a type of the term id.

6.4.2 Stability of typing under simple program transformations

FreezeML has no special heuristics built into the typing rules of term applications $M\,N$, typing them just as System F does. As a result, such terms retain their original types when adding an application of app or revapp (i.e., which simply perform application or reverse application, respectively). This is a useful property when defining operators such as Haskell's application operator \$, or piping operators such as $|>$ and $<|$ to reduce the number of parentheses in a term.

Lemma 6.3 (Insertion of app and revapp)**.**

1. *If* $\Delta;\Gamma \vdash M\,N : A$*, then* $\Delta;\Gamma \vdash \mathsf{app}\,M\,N : A$.
2. *If* $\Delta;\Gamma \vdash M\,N : A$*, then* $\Delta;\Gamma \vdash \mathsf{revapp}\,N\,M : A$.

Proof. Inverting $\Delta;\Gamma \vdash M\,N : A$ immediately yields the necessary premises to construct appropriate derivations using the FreezeML typing rule App. □

The same property holds for many other systems, including ML, MLF, HML, FPH, HMF, GI, QL, Poly-ML and QML. Notable exceptions are Boxy Types and IFX. Systems that *do* treat applications specially often need to type-check applications by considering more than one argument at once (e.g., GI, QL, HMF).

We now observe that like other systems that only infer monomorphic parameter types (e.g., HMF, IFX, GI, HML and FPH), FreezeML only allows eta-expansion of a function term M if the parameter has a monomorphic type. Of course, any function M may be eta-expanded by providing a type annotation. Systems such as QL (when in type-checking mode), Poly-ML and MLF can guarantee that eta expansion never requires a type annotation due to their sophisticated mechanisms for propagating type information. In QML, this is achieved with a much simpler mechanism, namely requiring full type annotations at all instantiation sites of a polymorphic parameter.

Lemma 6.4 (Eta expansion)**.**

1. *If* $\Delta;\Gamma \vdash M : S \to B$*, then* $\Delta;\Gamma \vdash \lambda x.M\,x : S \to B$*.*
2. *If* $\Delta;\Gamma \vdash M : A \to B$*, then* $\Delta;\Gamma \vdash \lambda(x : A).M\,x : A \to B$*.*

Proof. We can immediately construct the necessary derivations. □

6.4.2.1 Let abstraction and let substitution

In ML, the following transformations are guaranteed to preserve the types of terms:

Any subterm M of a term can be replaced by a fresh variable x by inserting a let term binding x to M. Conversely, any term **let** $x = M$ **in** N can be replaced by the second subterm after replacing all occurrences of x by M. In the following, we refer to the first transformation as *let abstraction* and to the second as *let substitution*.

The properties of these transformations get more complicated in the presence of the value restriction and side effects. Let abstraction is only guaranteed to succeed if a *single* instance of M is hoisted out of the program, otherwise the value restriction may step in to prevent different required instantiations of x. Also, if M has side effects and occurs multiple times, then either transformation may change program *behaviour*, even if the program remains well typed.

Before discussing these transformations in the context of FreezeML, we make another observation. In ML, types of terms are preserved when replacing the type of a variable in the environment with a more general one according to the ML generic instance relation defined in Figure 2.3 on page 14. This property is formalised in

Lemma 2.2 on page 18. However, this property does not hold in FreezeML, even when only considering predicative instantiation (as the ML generic instance relation does). While $\lceil f \rceil\ 3$ is well typed in a context giving f the type $\mathsf{Int} \to \mathsf{Int}$, it becomes ill typed when assigning the type $\forall a.a \to a$ to f, even though the former is a generic instance of the latter. We conjecture that the property *does* hold in FreezeML if no freezing operations occur in the term, but have not proven this. After all, Theorem 6.1 implies that the transformations do work when staying within the ML subset of FreezeML. In particular, it is *not* sufficient to require that only the variable that receives a more general type is never frozen, while allowing other variables to be frozen. The term $\mathbf{let}\ g = f\ \mathbf{in}\, \lceil g \rceil\ 3$ exhibits the same problem as before when changing the type of f.

We can relate this observation to let abstraction in FreezeML: Hoisting arbitrary terms may change the syntactic categories of let-bound terms and thus influence whether generalisation occurs. For example, consider the artificial term in Figure 6.4. Note that

$$\begin{array}{l} \mathbf{let}\ \mathsf{f} = \\ \quad \underbrace{\mathbf{let}\ y = \mathsf{id}\ \mathsf{id}\ \mathbf{in}\ \mathsf{id}}_{=:\, M'} \\ \mathbf{in} \\ \lceil f \rceil\ 3 \end{array}$$

Figure 6.4: Example term defining f before applying let abstraction

the variable y is unused, but the fact that the term M' is not a (guarded) value prevents generalisation, thus allowing f to receive type $\mathsf{Int} \to \mathsf{Int}$, making the overall term well typed.

As a result, hoisting the subterm $\mathsf{id}\ \mathsf{id}$ out and replacing it with the variable x leads to the ill typed term in Figure 6.5. This is due to the fact that the only possible type of f now becomes $\forall a.a \to a$, and applying it frozen to 3 fails.

Of course, this example relies on the fact that the original term freezes a variable with a monomorphic type. This is perfectly legal in FreezeML and the author considers potential variations of the language that would disallow this to contradict the spirit of FreezeML: Frozen variables are just System F variables, with no further strings (or side-conditions) attached. However, this suggests that we may state a let abstraction property for FreezeML that imposes some conditions regarding frozen variables.

Note that we can also use the earlier examples to illustrate an issue with let *substitution*. We may consider Figure 6.5 to be the original term, in which case the term in

$$
\begin{array}{l}
\textbf{let } \mathsf{x} = \mathsf{id}\ \mathsf{id}\ \textbf{in} \\
\textbf{let } f = \\
\quad \textbf{let } y = x \textbf{ in } \mathsf{id} \\
\textbf{in} \\
\lceil f \rceil\ 3
\end{array}
$$

Figure 6.5: Example term defining f after performing let abstraction

Figure 6.4 shows the result of performing let substitution to x. Thus, the transformation has again changed typeability.

These examples show that the value restriction makes stating general let abstraction and substitution properties for FreezeML somewhat subtle. The author conjectures that the following let abstraction and substitution properties hold, but has not attempted detailed proofs.

- *Guarded* values U can always be hoisted and replaced by a newly introduced variable x. The fact that U is replaced with the guarded term x means that the value categories of any surrounding term subject to the replacement do not change. Further, while the newly created let binding may generalise U, this can subsequently be reverted by instantiating x. This even works when substituting multiple occurrences of U within the same term. Note that the category of guarded values covers lambda functions. In the author's experience, anonymous functions are typical candidates for being hoisted and named when writing functional programs.

- We now outline criteria for a let abstraction property for the case where the term M to hoist is not a guarded value. In this case, M is let-bound and replaced by the *frozen* variable x. Concretely, the term M has to satisfy both of the following properties.

  1. We impose that M does not appear under the left-hand side of a let-binding. This is to avoid the problems described in Figures 6.4 and 6.5. Recall that in general, a let binding **let** $y = M'$ **in** N' is a guarded value if N' is itself a guarded value. However, M' only has to be a *value* (see category V in Figure 3.2 on page 41), which form a superset of guarded values. This means that replacing, say, a term application with a frozen variable may

change the syntactic category of the overall let binding if it is performed within M', but leaves the syntactic category unchanged if the replacement occurs within N'.

2. We require M to have a unique type. This is required to avoid a problem illustrated by the well typed term auto (id id). Here, id id requires polymorphic instantiation to obtain type $(\forall a.a \to a) \to (\forall a.a \to a)$, as expected by auto. However, allowing let abstraction of this subterm would lead to the following, ill typed term.

$$\textbf{let } x = \mathsf{id}\ \mathsf{id} \textbf{ in } \mathsf{auto}\ \lceil x \rceil$$

Here, the fact that id id is not a guarded value forces monomorphic instantiation to yield the type for x (see rule LETPLAIN in Figure 3.5 on page 45), making it impossible to give it the required type.

We could alternatively restate this requirement such that M must only be hoisted if it is not subject to *polymorphic* instantiation. However, requiring M to have a unique type means that we can hoist multiple occurrences of M at once, leading to the problem already described for ML, where different occurrences would require different instantiations.

- Regarding let *substitution* in a term **let** $x = M$ **in** N, we conjecture that it is sufficient to require that M does not appear on the left-hand side of a let binding within N. We would then substitute plain occurrences of x in N with $M@$ and occurrences of $\lceil x \rceil$ in N with $\$M$. Here, we assume that the generalisation operator \$ is defined on arbitrary terms, even though we only defined it on guarded values in Section 3.2.2. Concretely, after expanding the syntactic sugar, this means replacing all x with **let** $y = M$ **in** y and replacing all $\lceil x \rceil$ with **let** $y = M$ **in** $\lceil y \rceil$, where y is fresh.

The author considers achieving certainty about these properties an important piece of future work.

6.5 Related work

We now compare FreezeML to the existing approaches discussed in Section 2.3, focusing on the systems related to FreezeML in some particular way.

6.5.1 IFX

IFX [89] is similar to FreezeML in that it allows introducing first-class polymorphism without type annotations by using the principal type of the term. However, FreezeML avoids the distinction between ML type schemes and System F-style quantified types. In IFX, the term $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$ assigns the principal type scheme of e_1 of the form $\tilde{\forall}\overline{\alpha}.\tau$ to x in e_2, whereas the term $\mathbf{close}\, e$ receives the principal quantified type $\forall\overline{a}.\tau$. Recall that the former sort of polymorphic types is instantiated implicitly, whereas the latter requires explicit instantiation.

O'Toole and Gifford observe that type schemes and let bindings can be made superfluous in their design by replacing every $\mathbf{let}\ x = e_1\, \mathbf{in}\, e_2$ by the following.

$$(\lambda(x:\tau).e_2[(\mathbf{open}\ x)/x])(\mathbf{close}\ e_1)$$

Note that this comes at the cost of a type annotation τ if e_1 has a polymorphic principal type. However, they do not pursue this idea or its implications further. In FreezeML, we go the opposite direction: Let bindings introduce System F-style polymorphism in our system, which is the only sort of polymorphism used in FreezeML.

The **close** operator in IFX can only be used to introduce first-class polymorphism using principal types. To use non-principal types, explicit type abstractions must be used, which also bring type variables into scope to be used in annotations. As a result, the definition of the choose function in IFX requires writing $\Lambda a.\lambda(x:a)(y:a).x$. In FreezeML, this is achieved by writing $\mathbf{let}\ (\mathsf{choose} : \forall a.a \to a \to a) = \lambda x,y.x\ \mathbf{in}\ \ldots$ instead.

Another difference between FreezeML and IFX is how instantiation is performed. In the latter, instantiation of type schemes is performed with monomorphic types at variables, just as in ML. Conversely, $\forall$ quantifiers can be eliminated in three different ways, as described in Section 2.3.2.1.

1. Terms $\mathbf{open}\ e$ can be used to instantiate all toplevel $\forall$ quantifiers of e's type with monomorphic types.

2. System F-style explicit type applications can be used to perform instantiation.

3. To reduce the number of necessary **open** operations, IFX allows monomorphically instantiating toplevel $\forall$ quantifiers appearing in the type of e_1 in applications $e_1\ e_2$.

As a result, polymorphic instantiations always require type applications. To remedy this, O'Toole and Gifford consider an extension of their system by incorporating a typing rule originally proposed by McCracken [75].

$$\textsc{IPA}\ \frac{A \vdash e_1 : \forall \overline{a_i}.\tau_\mathrm{f} \to \tau_\mathrm{r} \qquad A \vdash e_2 : \tau_\mathrm{a} \qquad \{\overline{\tau_i}\} \subseteq \mathsf{ftv}(\tau_\mathrm{f}) \qquad \tau_\mathrm{a} = \tau_\mathrm{f}[\overline{\tau_i}/\overline{a_i}]}{A \vdash e_1\ e_2 : \tau_\mathrm{r}[\overline{\tau_i}/\overline{a_i}]}$$

This rule permits instantiating $\forall$ quantifiers in e_1's type with polymorphic types as long as the instantiations of all quantifiers are fully determined by the type τ_a of the argument e_2.

O'Toole and Gifford state uncertainty about the completeness of the resulting inference algorithm. While the author of this thesis believes that adding this rule is unproblematic from the perspective of type inference, it does have two disadvantages.

1. The rule only acts on $\forall$ quantified types rather than type schemes. This means that in order to change the term singleton id to produce a list of polymorphic identity functions, we must not only convert id to have a $\forall$-quantified type, but also singleton. Thus, we need to write (**close** singleton) (**close** id) to make the IPA rule applicable.

2. The rule makes simple modifications fragile. Changing the earlier term (**close** singleton) (**close** id) to (**close** app)(**close** singleton) (**close** id) retains typeability.

   However, changing it to (**close** revapp) (**close** id) (**close** singleton) results in the rule IPA not being applicable anymore. This is due to the subterm (**close** revapp) (**close** id) no longer determining the instantiation of the type variable b in the type $\forall a, b.a \to (a \to b) \to b$ of (**close** revapp). Therefore, an explicit type application is needed to type the following term in IFX even when using the IPA rule.

   $$(\textbf{close}\ \mathsf{revapp})\ (\forall a.a \to a)\ (\mathsf{List}\ (\forall a.a \to a))\ (\textbf{close}\ \mathsf{id})\ \mathsf{singleton}$$

   In order to support terms like this without annotations, the rule IPA would have to be extended to take n-ary applications into account. This somewhat ad-hoc treatment of applications is used in (or considered as an extension for) a variety of other systems [58, 117, 110, 109].

In summary, O'Toole and Gifford's work seems to originate the idea of introducing first-class polymorphism into an HM-based type system by using the principal type of a given term. However, unlike FreezeML, they keep the worlds of HM-style type schemes and first-class polymorphic types separated, only suggesting a heuristic in the typing rule of application rules as convenience to narrow the gap between the two.

In terms of instantiation, IFX does not allow *polymorphic* instantiation of arbitrary terms, which is possible in FreezeML (but may requiring the introduction of a variable first).

Note that the reliance of IFX on explicit type applications for some instantiations makes the system inherently dependent on an equational theory of types that respects the order of quantifiers. In contrast, we discussed in Section 6.2.1.1 that disregarding the order of quantifiers should not cause problems for FreezeML.

6.5.2 Poly-ML

Among all existing systems combining ML and first-class polymorphism, FreezeML is most closely related to Poly-ML [26]. The latter differentiates ML-style type schemes σ and first-class polymorphic types (in the form of *boxed* polymorphic types $[\sigma]$), similarly to IFX and QML. However, it allows impredicative instantiation of boxed polymorphism without specifying the instantiation further, just using the $\langle \_ \rangle$ operator. In particular, the user need not use explicit type abstractions (as is sometimes required in IFX), or state the original type of the term *before* instantiation, as in QML. As we describe in Section 2.3.2.2, Poly-ML uses *labels* to track the origins of first-class polymorphic types, meaning that type schemes ς in Poly-ML can not only quantify type variables, but also label variables.

Garrigue and Rémy [26] describe their systems as providing *semi-explicit* polymorphism because of where type annotations are needed: In their system, the $\langle \_ \rangle$ operator is always required to indicate *when* to eliminate first-class polymorphism, but without needing to provide further information on *how* to instantiate. The operator $[\_]$ used to introduce boxed (i.e., first-class) polymorphism always requires an annotation.

In FreezeML, all polymorphism is first-class, and we integrate its introduction and elimination into the same mechanisms used for this in vanilla ML, namely let bindings and variables. A key insight is then that this is sufficient to achieve the same expressiveness as System F; we can use these mechanisms to encode the explicit instantiation and generalisation operators defined as language primitives in IFX, Poly-ML, and QML.

Further, since the introduction and elimination of polymorphism occurs in FreezeML exactly where it does in ML, it allows us to type the ML subset of the language using first-class polymorphic types, rather than needing a separate notion of ML type schemes. However, we will see in Section 6.5.2.1 that it is easy to translate FreezeML into Poly-ML programs.

Another perspective on the difference between FreezeML and systems distinguishing different sorts of polymorphic types, as IFX, Poly-ML and QML do, is the following: FreezeML uses syntax to distinguish whether a variable should be instantiated (using id) or not (using $\lceil\mathsf{id}\rceil$). Systems like Poly-ML use different types to ensure this by distinguishing between $\mathsf{id} : \forall a. \to a$ and $\mathsf{id}' : [\forall a. \to a]$. While id is implicitly instantiated, id' must be explicitly instantiated by writing $\langle \mathsf{id}' \rangle$.[8]

However, our emphasis on the fact that FreezeML uses *only* System F types leads to a more accurate characterisation of the relationship: In IFX, QML, and Poly-ML, if a variable x has a first-class polymorphic type (i.e., a type with a toplevel $\forall$ quantifier in QML and IFX, and a type $\forall \varepsilon.[\sigma]^\varepsilon$ in Poly-ML), then usages of the variable x retain the first-class polymorphism, while an explicit instantiation form is required in each system (i.e., **open** in IFX, $\langle\cdot\rangle$ in Poly-ML, $\{\overline{\alpha}\, \forall\beta.\tau\}\ x$ in QML).

In this sense, the instantiation behaviour in FreezeML is the dual of the behaviour found in these three other systems: Variables are instantiated by default, and a special operator, namely freezing, must be used to retain first-class polymorphism. The fact that this resembles the behaviour of ML is what allows us to combine the two sorts of polymorphism found in the other systems into one, and type the ML subset of the language using it.

6.5.2.1 Translation from FreezeML to Poly-ML

The duality between the instantiation behaviour of FreezeML and Poly-ML becomes apparent when defining a translation from FreezeML to Poly-ML.

This is enabled by the fact that unlike the **open** operator of IFX, the Poly-ML operator $\langle\cdot\rangle$ may instantiate with (boxed) polymorphic types. Conversely, the instantiation operator in QML requires an explicit type annotation.

We first define two translations from FreezeML types (but allowing only $\to$ as a type constructor) to Poly-ML types.

[8]Note that in order for $\langle \mathsf{id}' \rangle$ to be well typed, id' must actually have the type scheme $\forall \varepsilon.[\forall a.a \to a]^\varepsilon$ in the environment. However, this is orthogonal from the discussion here and not the case in IFX or QML.

Let ε_0 be a fixed label. Then $[\![\_]\!]_\tau$ is defined as follows, yielding a simple type τ of Poly-ML. The translation simply boxes all adjacent quantifiers, including at the toplevel.

$$\begin{aligned}
[\![a]\!]_\tau &= a \\
[\![A_1 \to A_2]\!]_\tau &= [\![A_1]\!]_\tau \to [\![A_2]\!]_\tau \\
[\![\forall \Delta.H]\!]_\tau &= [\forall \Delta.[\![H]\!]_\tau]^{\varepsilon_0} \qquad\qquad \text{if } \Delta \neq \cdot
\end{aligned}$$

The translation $[\![\_]\!]_\varsigma$ is defined as $\forall \varepsilon_0.[\![A]\!]_\tau$, yielding a Poly-ML type scheme ς. It is applied to term contexts by applying $[\![\_]\!]_\varsigma$ to the types therein.

Our translation from FreezeML to Poly-ML relies on type information. We therefore extend FreezeML typing judgements to translation judgements of the form $\Delta;\Gamma \vdash^{\mathrm{P}} M : A \rightsquigarrow a_{\mathrm{P}}$, where a_{P} is a Poly-ML term. Its definition is shown in Figure 6.6 on the following page.

Frozen variables are translated to ordinary variables in Poly-ML. For plain variables, the translation depends on their types: Polymorphic variables x are opened, resulting in $\langle x \rangle$, while variables whose type does not have toplevel quantifiers are translated to just x.

Applications and abstractions are translated by inductively translating their parts. In the annotated case, we have that the translation, $\lambda(x : [\![A]\!]_\tau).a$, is syntactic sugar for $\lambda x.\,\mathbf{let}\, x = (x : [\![A]\!]_\tau)\,\mathbf{in}\, a$ in Poly-ML.

For un-annotated let bindings, we use the extended version of the $\Updownarrow$ judgement, defined in Section 3.3.2. Thus, Δ' denotes the variables to be quantified at the let binding, which are either $\overline{b}$ if M is a guarded value or empty otherwise. If no generalisation happens (i.e., M is not a guarded value, or there are simply no variables to quantify over), the let bound term in the translation is simply the translation a_1 of M. Otherwise, it is $[a_1 : \forall \Delta'.[\![A']\!]_\tau]$, meaning that we introduce first-class polymorphism in Poly-ML.

The translation for annotated let bindings works analogously: Here, the function split, defined in Figure 3.6 on page 47, is used to determine the variables Δ' that are generalised at the binding from the annotation A, based on whether M is a guarded value or not. We then insert the box operator if actual generalisation happens, or just a type annotation otherwise.

We may summarise the invariants of the translation as follows:

- Whenever an un-annotated lambda binding causes a binding $(x : S)$ to enter the term context in FreezeML, then the corresponding Poly-ML translation ensures

$$\boxed{\Delta;\Gamma \vdash^{\mathrm{P}} M : A \leadsto a}$$

$$\frac{x : A \in \Gamma}{\Delta;\Gamma \vdash^{\mathrm{P}} \lceil x \rceil : A \leadsto x} \qquad \frac{x : H \in \Gamma}{\Delta;\Gamma \vdash^{\mathrm{P}} x : H \leadsto x} \qquad \frac{\begin{array}{c} x : \forall\Delta'.H \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow_{\star} \cdot \\ \Delta' \neq \cdot \end{array}}{\Delta;\Gamma \vdash^{\mathrm{P}} x : \delta(H) \leadsto \langle x \rangle}$$

$$\frac{\Delta;\Gamma \vdash^{\mathrm{P}} M : A \to B \leadsto a_1 \qquad \Delta;\Gamma \vdash^{\mathrm{P}} N : A \leadsto a_2}{\Delta;\Gamma \vdash^{\mathrm{P}} M\,N : B \leadsto a_1\, a_2}$$

$$\frac{\Delta;(\Gamma, x : S) \vdash^{\mathrm{P}} M : B \leadsto a}{\Delta;\Gamma \vdash^{\mathrm{P}} \lambda x.M : S \to B \leadsto \lambda x.a} \qquad \frac{\Delta;(\Gamma, x : A) \vdash^{\mathrm{P}} M : B \leadsto a}{\Delta;\Gamma \vdash^{\mathrm{P}} \lambda(x : A).M : A \to B \leadsto \lambda(x : [\![A]\!]_{\tau}).a}$$

$$\frac{\begin{array}{c} \overline{b} = \mathsf{ftv}(A') - \Delta \\ (\Delta, \overline{b}, M, A') \Updownarrow (A, \Delta', \delta) \\ (\Delta, \overline{b});\Gamma \vdash^{\mathrm{P}} M : A' \leadsto a_1 \qquad \Delta;(\Gamma, x : A) \vdash^{\mathrm{P}} N : B \leadsto a_2 \\ \mathsf{principal}(\Delta, \Gamma, M, \overline{b}, A') \qquad a_{\mathrm{P}} = \begin{cases} a_1 & \text{if } \overline{b} = \cdot \\ [a_1 : \forall\Delta'.[\![A']\!]_{\tau}] & \text{otherwise} \end{cases} \end{array}}{\Delta;\Gamma \vdash^{\mathrm{P}} \mathbf{let}\ x = M\ \mathbf{in}\ N : B \leadsto \mathbf{let}\ x = a_{\mathrm{P}}\ \mathbf{in}\ a_2}$$

$$\frac{\begin{array}{c} (\overline{b}, A') = \mathsf{split}(A, M) \qquad (\Delta, \overline{b});\Gamma \vdash^{\mathrm{P}} M : A' \leadsto a_1 \\ \Delta;(\Gamma, x : A) \vdash^{\mathrm{P}} N : B \leadsto a_2 \qquad a_{\mathrm{P}} = \begin{cases} (a_1 : [\![A]\!]_{\tau}) & \text{if } \overline{b} = \cdot \\ [a_1 : \forall\Delta'.[\![A']\!]_{\tau}] & \text{otherwise} \end{cases} \end{array}}{\Delta;\Gamma \vdash^{\mathrm{P}} \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B \leadsto \mathbf{let} x = a_{\mathrm{P}}\ \mathbf{in}\ a_2}$$

Figure 6.6: Translation from FreezeML to Poly-ML

that $(x : [\![S]\!]_\tau)$ may enter the environment. Observe that this type does not contain ε_0 freely, thus meaning that it does not prevent ε_0 from being generalised.

- For all other forms of binders that result in $(x : A)$ entering the term context, the corresponding translation ensures that a type equivalent to $(x : \forall\varepsilon_0.[\![A]\!]_\tau)$ enters the Poly-ML environment. Note that if A does not contain quantifiers, then ε_0 does not appear freely in $[\![A]\!]_\tau$. In that case, the Poly-ML type scheme $\forall\varepsilon_0.[\![A]\!]_\tau$ is equivalent to just $[\![A]\!]_\tau$.

As an example, consider the following FreezeML term.

$$\begin{array}{l}\textbf{let}\ \mathsf{inc} = \lambda x.x+1\ \textbf{in}\\ \textbf{let}\ \mathsf{id} = \lambda y.y\ \textbf{in}\\ \mathsf{auto}\ \lceil\mathsf{id}\rceil, \mathsf{id}\ (\mathsf{inc}\ 3)\end{array}$$

It is translated to the following Poly-ML term.

$$\begin{array}{l}\textbf{let}\ \mathsf{inc} = \lambda x.x+1\ \textbf{in}\\ \textbf{let}\ \mathsf{id} = [\lambda y.y : \forall\alpha.\alpha \to \alpha]\ \textbf{in}\\ \mathsf{auto}\ \mathsf{id}, \langle\mathsf{id}\rangle\ (\mathsf{inc}\ 3)\end{array}$$

Here, for the variable id with a (boxed) toplevel quantified type, we observe the duality between plain variables and frozen variables in FreezeML, compared with opened variables and plain variables in Poly-ML. The variables x and inc with monomorphic types remain without a surrounding $\langle\_\rangle$ operator. Note that this would even be the case if the latter two variables appeared frozen in the source term. The type of the variable auto is translated to $\forall\varepsilon.([\forall\alpha.\alpha \to \alpha]^\varepsilon \to [\forall\alpha.\alpha \to \alpha]^\varepsilon)$. Since it has no toplevel quantifiers in the source type and thus no toplevel box in the Poly-ML type, it is not opened by the translation.

We conjecture that the translation always yields well typed terms of the corresponding translated types, using the second variant of Poly-ML that incorporates the value restriction [26, Section 5], in order to make the systems comparable. Concretely, we conjecture that if $\Delta;\Gamma \vdash^{\mathsf{P}} M : A \rightsquigarrow a_{\mathrm{P}}$ holds, then $[\![\Gamma]\!]_\varsigma \vdash a_{\mathrm{P}} : [\![A]\!]_\tau$ holds in Poly-ML. Recall that we use a_{P} for Poly-ML terms instead of just a to avoid confusion with (FreezeML) type variables.

We also observe that in the translation in Figure 6.6, there is only one case where the resulting terms contain additional type information not contained in the FreezeML source term: When creating a box operator in the $\Delta' \neq \cdot$ cases of both let translation rules, a type annotation is required in Poly-ML that is not required in FreezeML. We may therefore define Poly-ML†, a variant of Poly-ML extended with a new boxing operator. The existing boxing operator of the form $[a_{\mathrm{P}} : \sigma]$ asserts that the Poly-ML term a_{P} has the polymorphic type σ (which may contain toplevel quantifiers), the resulting type is $[\sigma]^{\varepsilon}$ for any ε. We suggest adding the form $[a_{\mathrm{P}}]$ that does not require the user to provide σ, but uses the principal σ instead.

We would then use this operator in the aforementioned two cases, and never need to insert additional type annotations not already present in the source term. The converse direction certainly does not hold: Poly-ML's label system allows inferring (boxed) polymorphism in function parameter types without requiring an annotation in certain cases. This is not possible in FreezeML.

Desugaring FreezeML to Poly-ML? The similarities between FreezeML and Poly-ML raise the question of whether it is possible to desugar FreezeML to Poly-ML (i.e., defining a translation that is entirely syntactic and does not rely on typing information). Of course, this desugaring would need to target Poly-ML†, the variant of Poly-ML we defined earlier that allows introducing first-class polymorphism using principal types rather than an annotation.

We investigate the following translation idea, relying on the duality between plain variables in FreezeML and the open operator of Poly-ML.[9] For the sake of this discussion, we only consider variables bound by let bindings without an annotation. Thus, we focus on terms of the form $\mathbf{let}\ x = M\ \mathbf{in}\ N$. We may translate all plain (i.e., not frozen) occurrences of x in N to $\langle x \rangle$, and translate $\lceil x \rceil$ to just x. In order for this to work, we box the type of any let-bound term M, rather than just boxing terms with polymorphic types. Thus, the translation of the let binding would be $\mathbf{let}\ x = [a_{\mathrm{M}}]\ \mathbf{in}\ a_{\mathrm{N}}$, where a_{M} and a_{N} are the translation of M and N, respectively. Note the Poly-ML† box operator enclosing a_{M}.

To state an invariant for the translation, we define a third translation function $[\![\_]\!]_\sigma$ from a FreezeML type to a polymorphic type σ in Poly-ML. It behaves like $[\![\_]\!]_\tau$ but leaves quantifiers at the toplevel unboxed.

$$\begin{aligned}
[\![a]\!]_\sigma &= [\![a]\!]_\tau \\
[\![A_1 \to A_2]\!]_\sigma &= [\![A_1 \to A_2]\!]_\tau \\
[\![\forall \Delta. H]\!]_\sigma &= \forall \Delta. [\![H]\!]_\tau && \text{if } \Delta \neq \cdot
\end{aligned}$$

[9] Didier Rémy suggested this desugaring idea to the author of this work.

The translation invariant is that if a FreezeML let binding adds $(x : A)$ to the term context, then the corresponding Poly-ML let binding gives x the type $\forall \varepsilon_0.[\llbracket A \rrbracket_\sigma]^{\varepsilon_0}$. Note that this is different from the type-directed translation discussed earlier. The invariant here states that the type of x is unconditionally boxed at the toplevel, even if the type within the box is not polymorphic. For example, we may end up with a binding $(x : \forall \varepsilon_0.[\mathsf{Int}]^{\varepsilon_0})$.

Using this translation idea, the FreezeML term **let** $\mathsf{id}_1 = \mathsf{id}$ **in** $\mathsf{id}_1\ 4$ is desugared to **let** $\mathsf{id}_1 = [\langle \mathsf{id} \rangle]$ **in** $\langle \mathsf{id}_1 \rangle\ 4$. Here, we indeed have $\mathsf{id}_1 : \forall \varepsilon_0.[\forall \alpha.\alpha \to \alpha]^{\varepsilon_0}$ when typing $\langle \mathsf{id}_1 \rangle\ 4$, making the term well typed.[10] Note that under this desugaring scheme, the Poly-ML type of id assumed to be present in the environment would be $\forall \varepsilon_0.[\forall \alpha.\alpha \to \alpha]^{\varepsilon_0}$.

However, this translation scheme seems to fail when the let-bound term *already* has a polymorphic type. Consider the following two FreezeML terms (on the left) and their Poly-ML translations (on the right).

$$\begin{array}{lcl} \textbf{let}\ \mathsf{id}_2 = \mathsf{id}\ \mathsf{id}\ \textbf{in}\ \mathsf{id}_2\ 4 & \rightsquigarrow & \textbf{let}\ \mathsf{id}_2 = [\langle \mathsf{id} \rangle\ \langle \mathsf{id} \rangle]\ \textbf{in}\ \langle \mathsf{id}_2 \rangle\ 4 \\ \textbf{let}\ \mathsf{id}_3 = \mathsf{id}\ \lceil \mathsf{id} \rceil\ \textbf{in}\ \langle \mathsf{id}_3 \rangle\ 4 & \rightsquigarrow & \textbf{let}\ \mathsf{id}_3 = [\langle \mathsf{id} \rangle\ \mathsf{id}]\ \textbf{in}\ \langle \mathsf{id}_3 \rangle\ 4 \end{array}$$

Both FreezeML terms are well typed, but only the translation of the first yields a well typed Poly-ML term. In the second Poly-ML translation result (binding id_3), the principal type scheme of $\langle \mathsf{id} \rangle\ \mathsf{id}$ is $\forall \varepsilon_0.[\forall \alpha.\alpha \to \alpha]^{\varepsilon_0}$, which would then be boxed *again*, leading to the perfectly legal type scheme $\forall \varepsilon_0.[[\forall \alpha.\alpha \to \alpha]^{\varepsilon_0}]^{\varepsilon_0}$ for id_3. Note that it exhibits two nested levels of boxing, meaning that the single open operator in the subterm $\langle \mathsf{id}_3 \rangle\ 4$ leads to an ill typed function application. It may be possible to redefine the un-annotated boxing operator $[a_P]$ suggested for Poly-ML† such that it would *only* perform boxing if the principal type of a_P is not *already* boxed, leading to the variant Poly-ML‡.

In our example, the definitions of id_2 and id_3 both belong to the category of non-guarded values. However, the double boxing issue described here seems to be unrelated to the value restriction:

- If we keep the current translation, but change the $[\_]$ operator to only generalise the type prior to boxing if the enclosed term is a value, then the problem persists.

[10]The reader may wonder why we chose the lone variable id as the let-bound term. This is to avoid discussing the desugaring of terms such as un-annotated lambdas, which requires a slightly different desugaring approach. Since this is orthogonal from the issue subsequently discussed here, we avoid detailing the desugaring of such terms.

This change only affects whether id_2 receives type $\forall\varepsilon_0.[\forall\alpha.\alpha \to \alpha]$ or $\forall\varepsilon_0.[\mathsf{Int} \to \mathsf{Int}]^{\varepsilon_0}$, while the doubly boxed type of id_3 is unaffected.

- If we changed the translation so that no box operator is inserted when the let-bound term is not a guarded FreezeML value, then the types assigned to id_2 and id_3 would again differ in that the type of the former contains no box, while the type of the latter contains a box. This would again affect the typeability of the body of each binding.

As a result, the author of this work is uncertain whether or not it is possible to define a desugaring between FreezeML and Poly-ML. In particular, it seems to the author that the variant Poly-ML‡ suggested to address the introduction of duplicate boxes does not enjoy principal types. While the principal type scheme of $\lambda x.[x]$ in Poly-ML† would be $\forall\alpha,\varepsilon.\alpha \to [\alpha]^\varepsilon$, this does not hold for Poly-ML‡. Instantiating α with a boxed type such as $[\mathsf{Int}]^{\varepsilon_0}$ results in the return type of the function having two immediately nested boxes, which the $[\_]$ operator of Poly-ML‡ would not allow. It may be possible to address this problem by refining the equational theory of types in Poly-ML‡ to disregard such duplicate boxes.

The existence of a desugaring between FreezeML and Poly-ML deserves further investigation in the future. On one hand, the similarities between the two systems suggests that a desugaring between them may exist. On the other hand, the fact that plain variables in FreezeML do not syntactically indicate whether they actually perform instantiation of any toplevel quantifiers or not may make this impossible.

6.5.3 QML

Like IFX and Poly-ML, QML [106] relies on a distinction between ML type schemes and first-class polymorphic types. Unlike the former two and our system, QML also incorporates existential types in addition to universally quantified ones.

The system requires type annotations at the introduction and elimination forms of first-class polymorphism, making the system somewhat verbose to use. However, the authors of QML state that their goal is the simplicity of the underlying formalism and its meta-theory. Indeed, QML always allows eta-expansion without requiring annotations, while not relying on complex mechanisms, such as Poly-ML's label system. Further, the language permits a simple, direct reduction semantics.

One of the suggestions the authors make to ease the burden of annotations is to define syntactic sugar for annotations on function parameters with the following effect.

$$\lambda(x : \forall\alpha.\alpha \to \alpha).e \;\equiv\; \lambda y.\,\mathbf{let}\; x = y\;\{\forall\alpha.\alpha \to \alpha\}\;\mathbf{in}\, e$$

Here, x is annotated with the polymorphic type $\forall\alpha.\alpha \to \alpha$, but the elaboration of this sugar means that x receives the type scheme $\Pi(\alpha).\alpha \to \alpha$ in e. This is achieved by eliminating the first-class polymorphism first (using $y\;\{\forall\alpha.\alpha \to \alpha\}$), and then performing ML-style generalisation at the let binding. Mixing first-class polymorphic types and ML type schemes, in particular in such a way that makes it opaque to the user, is exactly what FreezeML tries to avoid.

Russo and Vytiniotis later reflect that "the fact that HMF does not require two distinct universal quantifiers (as we do) may be simpler for programmers" [106]. They subsequently discuss the potential effects of merging the two types of universal quantifiers in their system. Two of their resulting comments are highly relevant in the context of FreezeML.

1. They observe the typical difficulty of needing to make decisions about whether or not to instantiate certain term variables or not, similarly to what we discussed in Section 1.3.

2. They consider the option of using the existing ML constructs for the introduction and elimination of first-class polymorphism (i.e., using let bindings and variables for this), just as FreezeML does. However, they observe that this requires the user to "laboriously follow the typing rules" [106, Section 7.3] to conclude where instantiation of non-variable terms must be forced by inserting explicit operators.

In FreezeML, the answer to the first issue is the freezing operator. With respect to the second issue, the author of this work believes that it is reasonable to ask programmers to have a clear picture of where exactly polymorphism occurs in their programs based on simple rules, even if it requires users to determine where additional operators are required to achieve the desired typing.

We conjecture that it would be possible to add an introductory form for first-class polymorphism to QML that uses the principal type of the given term, similarly to the **close** operator of IFX and the operator we suggest in Section 6.5.2 to obtain Poly-ML† from Poly-ML. However, we do not see a straightforward way to change the elimination of first-class polymorphism in QML such that it would not require an annotation.

6.5.4 HMF

Like FreezeML, HMF [58] just uses a single sort of quantified type. However, in contrast to our system, HMF is able to account for instantiation and generalisation in a declarative fashion, providing standalone typing rules for them, analogously to the INST and GEN rules of ML (as shown in Figure 2.2). In FreezeML both operations are inherently tied to individual syntactic forms.

The price that HMF pays for this is the need for a minimality condition on types that is imposed while typing term applications and treating them as n-ary forms, as discussed in Section 2.3.3.1, going beyond the principality condition on let bindings shared with FreezeML. Further, the type system cannot give toplevel polymorphic return types to functions without annotations. However, the resulting type system seems to require fewer annotations than FreezeML. One aspect shared between FreezeML and HMF is that the typing restrictions it imposes on let bindings as well as applications make some terms ill typed that would be accepted if the restrictions were removed from the typing rules. This is in contrast to the value-restricted variant of Poly-ML [26, Section 5], where the principality condition on let bindings never prevents typing a term that would otherwise be accepted. Instead, it only affects the possible derivations of the same judgements. Nevertheless, we conjecture that this would no longer be the case in Poly-ML†, the extension we proposed in Section 6.5.2.

Finally, we observe that Leijen's proposal of *rigid* annotations is similar in spirit to frozen variables in FreezeML. As discussed in Section 2.3.3.1, a rigidly annotated term $(e : \sigma)$ prevents the ubiquitous generalisation and instantiation rules from being applicable. In this sense, the FreezeML term $\lceil x \rceil$ is equivalent to a rigidly annotated term $(x : \sigma)$ in HMF, where σ is the type of x. This illustrates that in FreezeML, we do not require type annotations to perform freezing, but need to bind the term under consideration to a variable first. This in turn requires a type annotation for non-principal types (assuming the variable is let-bound). Thus, a variant of HMF containing a freezing operator on arbitrary terms using their principal types would be closely related to FreezeML, similarly to the idea of extending Poly-ML to Poly-ML†.

6.5.5 GI

The GI system [110] and FreezeML both rely on System F types. Other than that, the systems are rather different in spirit, due to GI's type system trying to minimise the number of annotations using heuristics.

One similarity is that GI uses three syntactic forms of unification variables during inference, denoting whether they can unify with fully polymorphic types, monomorphic types, or what we call guarded types (i.e., types without toplevel quantifiers). The first two forms correspond to unification variables restricted with restrictions $\star$ and $\bullet$ in our systems, respectively. We believe that both representations are largely equivalent, but prefer ours, where a restriction context Θ directly corresponds to residual constraints consisting only of mono constraints (as discussed in Section 5.3.1.2).

One interesting difference is that the various forms of restrictions imposed by type variables are only present during inference in GI. Their substitution property for types does however only permit monomorphic instantiation. This means that while id may be given any type $A \to A$ in FreezeML for arbitrarily polymorphic A, GI only permits monomorphic types in place of A. In other words, the distinction between allowing monomorphic and polymorphic types is visible in the possible types of FreezeML terms, whereas in GI it is only used to check the validity of certain polymorphic instantiations that are strictly required to make a term well typed.

6.5.6 Example-guided comparison

We now compare most of the systems discussed throughout this work based on a set of examples collected from existing literature. The examples were originally collected by Serrano et al. [110, 109] comparing Quick Look, GI, MLF, HMF, FPH and HML. We restate their results for these systems here and add IFX, Poly-ML, QML and FreezeML to the comparison.

The example terms assume that the functions with corresponding types shown in Figure 6.7 on the following page are in scope, containing many of the example functions we have already used earlier in this work.

For IFX, Poly-ML and QML we assume that all the types given in Figure 6.7 that have toplevel quantifiers use the notion of ML type schemes of the respective systems. Of course, all these systems dictate that for nested quantifiers, we must use first-class polymorphic types. For example, this means that the type of id from Figure 6.7 becomes the QML *type scheme* $\Pi(\alpha).\alpha \to \alpha$ in QML, while the type of ids is $\mathsf{List}\,(\forall\alpha.\alpha \to \alpha)$. Notice the different quantifiers used in each case.

The examples are divided into four collections, called A, B, C, and E. Note that there is no collection D, we follow the naming scheme introduced by Serrano et al. [109]. We only deviate from their naming scheme in a few cases where they gave a

head	$: \forall a.\mathsf{List}\,a \to a$	id	$: \forall a.a \to a$
tail	$: \forall a.\mathsf{List}\,a \to \mathsf{List}\,a$	ids	$: \mathsf{List}\,(\forall a.a \to a)$
$[\,]$	$: \forall a.\mathsf{List}\,a$	inc	$: \mathsf{Int} \to \mathsf{Int}$
$(::)$	$: \forall a.a \to \mathsf{List}\,a \to \mathsf{List}\,a$	choose	$: \forall a.a \to a \to a$
$\mathsf{singleton}$	$: \forall a.a \to \mathsf{List}\,a$	poly	$: (\forall a.a \to a) \to \mathsf{Int} \times \mathsf{Bool}$
$(+\!\!+)$	$: \forall a.\mathsf{List}\,a \to \mathsf{List}\,a \to \mathsf{List}\,a$	auto	$: (\forall a.a \to a) \to (\forall a.a \to a)$
length	$: \forall a.\mathsf{List}\,a \to \mathsf{Int}$	auto'	$: \forall b.(\forall a.a \to a) \to (b \to b)$

map	$: \forall a\,b.(a \to b) \to \mathsf{List}\,a \to \mathsf{List}\,b$
app	$: \forall a\,b.(a \to b) \to a \to b$
revapp	$: \forall a\,b.a \to (a \to b) \to b$
runST	$: \forall a.(\forall s.ST\ s\ a) \to a$
argST	$: \forall s.ST\ s\ \mathsf{Int}$
pair	$: \forall a\,b.a \to b \to a \times b$
pair'	$: \forall b\,a.a \to b \to a \times b$

Figure 6.7: Type signatures for example functions

single number to multiple related terms that exhibit differences in the systems that we add to the comparison.

The tables shown in Figures 6.8 to 6.10 then contain one column per system. In each such column, we use ● to indicate that the term in that row is accepted by the system without changes and ○ if it is not, for example because a type annotation must be added, or the term cannot be accepted even after adding annotations. We use ◐ for cases where the original term is not accepted, but it is sufficient to add some operator to the example term to make it type-check if said operator does *not* include any sort of type annotation. Thus, ◐ can only appear in the columns for IFX (adding **open** or **close**), Poly-ML (adding the $\langle \_ \rangle$ operator) or FreezeML (adding $\lceil \_ \rceil$, \$ or @).

In some rows of the table, we state how the corresponding example would have to be modified in order to be accepted by a particular system. This is usually done to show how to modify terms to become well typed in FreezeML, but we sometimes show the necessary modifications required for other systems, in particular if an example is accepted by all but one system.

We generally consider the original version of each system, as defined by the corresponding authors, without any of the language extensions they might propose. We sometimes indicate how a given result may change when using particular extensions. For the four systems added to the comparison by the author of this work, the results were established manually by using the typing rules of each system, rather than relying on potential implementations.

While our tables contain results for all terms collected by Serrano et al., we eschew going through every single one, instead focusing on examples illustrating differences between the systems. Further, we do not consider this comparison to be a quantitative analysis of how many annotations each system may typically require in practice. After all, the examples were often presented in existing literature to illustrate different aspects of typing terms in the presence of first-class polymorphism, but are not necessarily representative of real-world programs.

Collection A The first subset A of examples is shown in Figure 6.8 on the next page, containing examples performing polymorphic instantiation.

We observe that the term $\mathsf{choose}\ []\ \mathsf{id}$ (example A3) cannot be typed in IFX, even when using the IPA rule proposed by the authors (see Section 6.5.1) to implicitly instantiate first-class polymorphism in application positions. This is due to the necessary impredicative instantiation not being deducible just from the subterm $\mathsf{choose}\ []$ and illustrates the need for n-ary application rules in other systems, such as HMF, GI and QL. Poly-ML and FreezeML can type the same term without using n-ary application rules by allowing arbitrary polymorphic instantiation.

The term $\lambda(x : \forall a.a \to a).x\,x$ (example A4) can be typed in IFX only when using the IPA rule, which we indicate with the superscript I. In Poly-ML, the first occurrence of x in the body of the function must be replaced with $\langle x \rangle$ to open it. Note that the annotation on x must also be boxed. As the need for this boxing within polymorphic annotations is entirely mechanical, we implicitly assume that it takes place for all annotations involving annotations with polymorphic types in Poly-ML.

In example A5, we observe an issue already discussed in Section 6.5.1: In order for the IPA rule to become applicable, we must convert id from having a type scheme to having a first-class polymorphic type.

Example A8 illustrates that almost no system can handle the isomorphic types of auto and auto'.[11]

[11]Even in MLF, these types are not equivalent, but the type of auto' is more general. Only FML (not part of the comparison table, see Section 2.3.5) treats these types as equivalent.

		QL	GI	MLF	HMF	FPH	HML	IFX	Poly-ML	QML	FreezeML
A	Polymorphic instantiation										
A1	$\lambda x y . x$	●	●	●	●	●	●	●	●	●	●
A2	choose id	●	●	●	●	●	●	●	●	●	●
A3	choose [] ids	●	●	●	●	●	●	○	●	●	●
A4	$\lambda(x : \forall a . a \to a) . x x$	●	●	●	●	●	●	●$^{\text{I}}$	◐	●	●
A5	id auto IFX: (**close** id) auto	●	●	●	●	●	●	◐$^{\text{I}}$	●	●	●
A6	id auto$'$	○	●	●	●	●	●	◐$^{\text{I}}$	●	●	●
A7	choose id auto	●	●	●	●	●	●	○	●	●	●
A8	choose id auto$'$	○	○	●	○	○	●	○	○	○	○
A9	f (choose id) ids FreezeML: f (choose $\lceil$id$\rceil$) ids	●	○	●	○	●	●	○	○$^{\text{P}}$	○$^{\text{P}}$	◐
A10a	poly id FreezeML: poly $\lceil$id$\rceil$	●	●	●	●	●	●	◐$^{\text{I}}$	○$^{\text{P}}$	○$^{\text{P}}$	◐
A10b	poly $(\lambda x . x)$ FreezeML: poly $\$(\lambda x . x)$	●	●	●	●	●	●	◐$^{\text{I}}$	○$^{\text{P}}$	○$^{\text{P}}$	◐
A10c	id poly $(\lambda x . x)$ FreezeML: id poly $\$(\lambda x . x)$	●	●	●	●	●	●	◐$^{\text{I}}$	○$^{\text{P}}$	○$^{\text{P}}$	◐
A11	k $(\lambda f . f\ 1, f$ true) xs	●	○	○	○	○	○	○	○	○	○
A12a	poly id FreezeML: poly $\lceil$id$\rceil$	●	●	●	●	●	●	◐$^{\text{I}}$	○$^{\text{P}}$	○$^{\text{P}}$	◐
A12b	app poly id FreezeML: app poly $\lceil$id$\rceil$	●	●	●	●	●	●	◐$^{\text{I}}$	○$^{\text{P}}$	○$^{\text{P}}$	◐
A12c	revapp id poly FreezeML: revapp$\lceil$id$\rceil$ poly	●	●	●	●	●	●	○	○$^{\text{P}}$	○$^{\text{P}}$	◐
A13a	app runST argST FreezeML: app runST $\lceil$argST$\rceil$	●	●	●	●	●	●	◐$^{\text{I}}$	○$^{\text{P}}$	○$^{\text{P}}$	◐
A13b	revapp argST runST FreezeML: revapp $\lceil$argST$\rceil$ runST	●	●	●	●	●	●	○	○$^{\text{P}}$	○$^{\text{P}}$	◐
where f : $\forall a . (a \to a) \to$ List $a \to a$ k : $\forall a . a \to$ List $a \to$ Int xs : List(($\forall a . a \to a) \to$ (Int $\times$ Bool))		QL	GI	MLF	HMF	FPH	HML	IFX	Poly-ML	QML	FreezeML

Figure 6.8: Example collection A

The term f (choose id) ids (see example A9) illustrates some interesting differences between the different systems. Here, the type of f is $\forall a.(a \to a) \to \mathsf{List}\ a \to a$ (as shown in the bottom left corner of the table in Figure 6.8). First, we observe the limits of HMF's rule to imposing the use of the least polymorphic type that makes an application well typed. Even applying the rule across all arguments of f (as the n-ary application rule of HMF allows) does not help, as the system is forced to pick an insufficiently polymorphic type when typing the subterm choose id first. FreezeML would pick the same type as HMF for the latter subterm, rather than keeping id polymorphic. Thus, we must freeze id to make the term well typed in FreezeML.

In Poly-ML and QML, we must use the explicit introductory forms for first-class polymorphism to lift id from having a type scheme to having a first-class polymorphic type. Thus, we must replace id with $[\mathsf{id} : \forall\alpha.\alpha \to \alpha]$ in Poly-ML and $\{\forall\alpha.\alpha \to \alpha\}$ id in QML. However, as mentioned in Sections 6.5.2 and 6.5.3, the author of this work conjectures that it is possible to equip both systems with introductory forms for first-class polymorphism that could use the *principal* type instead of requiring an annotation. We indicate this with $\bigcirc^{\mathsf{P}}$ in the table (i.e., with our proposed extensions of both systems, the entry would become ◐ instead).

Example A12 again shows the limits of IFX, the term needs to be rewritten to the following to become well typed.

$$(\textbf{close}\ \mathsf{revapp})\ (\forall a.a \to a)\ \big((\forall a.a \to a) \to (\mathsf{Int} \times \mathsf{Bool})\big)(\textbf{close}\ \mathsf{id})\ \mathsf{poly}$$

Collections B Collection B contains functions on polymorphic lists, it is shown in Figure 6.9 on the following page.

All examples can be adapted to become well typed without annotations in FreezeML, using the freezing, instantiation, or generalisation operator. The same holds for almost all examples in QML and Poly-ML when using the extensions discussed in the context of collection A.

One outlier is example B7b, which QML requires to be changed to the following.

$$((\mathsf{head}\ \mathsf{ids})\ \{\forall a.a \to a\})\ \mathsf{true}$$

As mentioned before, we do not believe that an un-annotated version of this elimination form can be added to QML without requiring additional changes to the system.

Example B5, where the lists (singleton id) and ids are concatenated, illustrates some differences between the systems using heuristics. In GI and HMF, the subterm (singleton id) is typed in isolation, without taking the surrounding context into account.

		QL	GI	MLF	HMF	FPH	HML	IFX	Poly-ML	QML	FreezeML
B	Functions on polymorphic lists										
B1a	length ids; tail ids; head ids IFX needs (**close** f) for all functions f	●	●	●	●	●	●	◐$^{\mathrm{I}}$	●	●	●
B1b	singleton id	●	●	●	●	●	●	●	●	●	●
B2	id :: ids FreezeML: $\lceil$id$\rceil$ :: ids	●	●	●	●	●	●	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
B3	$(\lambda x.x)$ :: ids FreezeML: $\$(\lambda x.x)$ :: ids	●	●	●	●	●	●	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
B4	singleton inc $+\!\!+$ singleton id	●	●	●	●	●	●	●	●	●	●
B5	singleton id $+\!\!+$ ids FreezeML: singleton $\lceil$id$\rceil$ $+\!\!+$ ids	●	○	●	○	●	●	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
B6	map poly (singleton id) FreezeML: map poly (singleton $\lceil$id$\rceil$)	●	○	●	●	●	●	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
B7a	map head (singleton ids) IFX needs type application	●	●	●	●	●	●	○	●	●	●
B7b	head ids true FreezeML: (head ids)@ true	●	●	●	●	●	●	◐$^{\mathrm{I}}$	◐	○	◐
C	Inference of polymorphic lambda binders and generalisation points										
C1a	$\lambda f.(f\ 1, f$ true)	○	○	○	○	○	○	○	○	○	○
C1b	$\lambda(f : \forall a.a \to a).(f\ 1, f$ true) Poly-ML: $\lambda(f : [\forall a.a \to a]).(\langle f \rangle\ 1, \langle f \rangle$ true)	●	●	●	●	●	●	●	◐	●	●
C1c	$g\ (\lambda f.(f\ 1, f$ true))	●	○	○	○	○	○	○	○	○	○
C2	$r\ (\lambda x\,y.y)$ FreezeML: $r\ \$(\lambda x.\$(\lambda y.y))$	●	○	●	○	○	○	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
where $g : ((\forall a.a \to a) \to (\mathsf{Int} \times \mathsf{Bool}) \to \mathsf{Char})$ $r : (\forall a.a \to \forall b.b \to b) \to \mathsf{Int}$		QL	GI	MLF	HMF	FPH	HML	IFX	Poly-ML	QML	FreezeML

Figure 6.9: Example collections B and C

The guardedness condition in GI and the requirement to use least polymorphic types in HMF then impose that id must be instantiated. As a result, both systems require a type annotation to accept the example. Only the more elaborate heuristics of QL *do* take the surrounding context into account when typing this subterm, meaning that no annotation is required. Of course, the example does require some adaption to be accepted by any of the four explicit systems.

Collection C Collection C, also shown in Figure 6.9, contains examples where polymorphic function parameter types must be inferred or it must be inferred where exactly generalisation should happen.

No system can type $\lambda f.(f\ 1, f\ \mathsf{true})$, only Quick Look's bidirectionality can do so if sufficient context is present (see example C1c). While Poly-ML's label system can be used to infer some polymorphic function parameters, it cannot do so in example C1c (after adding appropriate open operators around both usages of f) : At the point of typing $\langle f \rangle\ 1$ and $\langle f \rangle\ \mathsf{true}$, the label associated with f's boxed polymorphic type appears in the environment, making it illegal to open f, even though the label is freshened when typing the application $g\ (\lambda f.\dots)$.

All systems in the table can type the original example when an annotation is given on f, but Poly-ML requires opening f's boxed polymorphic type first.

Example C2 illustrates the difficulties of determining generalisation points. In the explicit systems, this can be mitigated by inserting appropriate operators, rather than type annotations. In fact, all of IFX, Poly-ML, QML and FreezeML reject even the simpler term $\mathsf{poly}\ (\lambda x.x)$ as they never perform generalisation of an arbitrary term that would lead to a polymorphic function argument unless explicitly instructed to do so. The same term is accepted by the first six systems compared in Figure 6.9.

Collection E Collection E in Figure 6.10 on the next page contains examples related to eta-expansion and other small program transformations.

Of the systems shown in the table, only MLF, Poly-ML and QML unconditionally guarantee stability of typing under eta-expansion without requiring an annotation even if the eta-expanded function has a polymorphic parameter type. This coincides exactly with the results for example E2a.

Adding an annotation on the overall function allows QL to propagate it to the binder x, as shown in E2b. Syntactic annotation propagation could also be used to type the example, for example when using the corresponding extension of HMF. We mark this

		QL	GI	MLF	HMF	FPH	HML	IFX	Poly-ML	QML	FreezeML
E	η-expansion										
E1a	k h lst	○	○	○	○	○	○	○	○	○	○
E1b	k $(\lambda x.$h $x)$ lst FreezeML: k $(\lambda x.$h $x)$ lst	●	●	●	●	●	●	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
E2a	$\lambda x.$poly x	○	○	●	○	○	○	○	●	●	○
E2b	$(\lambda x.$poly $x) : (\forall a.a \to a) \to (\mathsf{Int} \times \mathsf{Bool})$	●	○	●	●$^{\mathrm{A}}$	○	○	○	●	●	○
E3a	app poly id FreezeML: app poly $\lceil$id$\rceil$	●	●	●	●	●	●	◐$^{\mathrm{I}}$	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	◐
E3b	app $(\lambda x.$poly $x)$ id FreezeML: app $(\lambda(x : \forall a.a \to a).$poly $\lceil x \rceil)$ $\lceil$id$\rceil$	○	○	●	○	○	○	○	○$^{\mathrm{P}}$	○$^{\mathrm{P}}$	○
E4a	map poly ids IFX: (**close** map) poly ids	●	●	●	●	●	●	◐$^{\mathrm{I}}$	●	●	●
E4b	map $(\lambda x.$poly $x)$ ids FreezeML: map $(\lambda(x : \forall a.a \to a).$poly $\lceil x \rceil)$ ids	●	○	●	○	○	○	○	●	●	○
E5a	compose poly head IFX needs 3 type applications	●	●	●	●	●	●	○	●	●	●
E5b	$\lambda xs.$ poly (head xs) FreezeML: $\lambda(xs : \forall a.a \to a).$ poly (head $\lceil xs \rceil$)	○	○	●	○	○	○	○	●	●	○
where k : $\forall a.a \to \mathsf{List}\ a \to a$ h : $\mathsf{Int} \to \forall a.a \to a$ lst : $\mathsf{List}\ (\forall a.\mathsf{Int} \to a \to a)$		QL	GI	MLF	HMF	FPH	HML	IFX	Poly-ML	QML	FreezeML

Figure 6.10: Example collection E

reliance on an extension by writing $\bullet^{A}$.

Note that Poly-ML and QML fail on example E3b not due to the eta expansion therein, but simply because id needs to be lifted to have a first-class polymorphic type.

Example E5b effectively eta-expands the composition of head and poly, the only examples succeeding are the same ones that support general eta-expansion, as in example E2a.

Summary As expected, we observe that the systems relying on heuristics or some degree of MLF types (shown as the first six systems in the tables in Figures 6.8 to 6.10) require fewer annotations than the explicit systems (shown in the last four columns of the tables). The systems in the latter group are designed such that users need to be more explicit about their intentions. In particular, IFX, Poly-ML and QML require the usage of dedicated operators at *all* introduction and elimination sites of their notions of first-class polymorphism. In FreezeML, this is not the case for all introduction and elimination sites of polymorphism, which is always first-class, due to the coupling with the existing behaviour of ML. However, users may still need to freeze variables as well as insert instantiation and generalisation operators. Nevertheless, the results in the table indicate that those situations where FreezeML requires the insertion of a type annotation (e.g., on a function parameter or when using the annotated generalisation operator) often coincide with situations where many of the six systems in the first group also require annotations.

We observe that Poly-ML and QML require many more type annotations than FreezeML in order to accept the examples in Figures 6.8 to 6.10. This may be alleviated by using the language extensions that the author of this work proposed for both systems that rely on principal types, as indicated with $\bigcirc^{P}$ in the tables. However, while the author of this work conjectures that these extensions preserve the existence of principal types in each system, no attempts have been made to formalise them and to prove the correctness of the updated inference algorithms. It would also be interesting to investigate how our proposed extensions would affect any of the other properties given about each system by their original developers.

Independently from these potential extensions, Poly-ML and QML can type some examples without type annotations where FreezeML does require annotations, due to the ability of the former two systems to infer polymorphic types for function parameters in certain situations. This is never possible in FreezeML (as well as GI, HMF, FPH and HML). We observe that even when allowing the use of the IPA rule in IFX, the system still requires more user intervention than the other three explicit systems.

Of course, the results in Figures 6.8 to 6.10 do not necessarily present a realistic picture of the differences between the systems when writing real-world programs. In addition, the comparison does not take into account the ability of users to reason about their programs and understand potential typing errors in each system.

Chapter 7

Conclusions and future work

We now summarise the findings of this thesis and outline future work.

7.1 Concluding remarks

This work introduced FreezeML, an approach for integrating first-class polymorphism into the HM typing discipline. It embraces the existing mechanisms for generalisation and instantiation found in the HM system, namely coupling them with let bindings and variables, respectively. While the automatic instantiation of variables is all that is needed when staying within the limits of the HM system, the presence of first-class polymorphism necessitates the ability to keep values polymorphic. FreezeML answers this by equipping users with the ability to sidestep the *magic* that is implicit instantiation in the form of the freezing operator, and provide variables that simply reflect their associated types.

This thesis shows that combining freezing with the existing mechanisms for generalisation and instantiation mentioned above are all the ingredients required to support first-class polymorphism. The result is a type system where all of its rules are taken from either ML or System F directly, except for let bindings.

Completeness of type inference is ensured by two invariants.

I-1. Ensuring that all polymorphism occurring in the types of term variables is uniquely determined

I-2. Limiting instantiation to term variables

To ensure the first condition for un-annotated binders, FreezeML impose that lambda-bound variables only receive monomorphic types, while let bindings must determine the type of the variable based on the principal type of the let-bound term.

FreezeML incorporates the value restriction by default. Based on the observation that the types of generalisable terms do not have toplevel quantifiers[1], we can use a simple rule for bringing *type* variables into scope using annotations on let bindings. However, the value restriction imposes that additional monomorphisation is required when handling non-generalising let bindings without annotations in order to maintain invariant I-1.

As we have seen in Section 6.5.6, these conditions result in a type system that only requires a moderately increased number of necessary type annotations as compared to other systems when considering typical example terms from the literature. While users still need to add freezing operators as well as instantiation and generalisation hints, the author considers this to be an acceptable compromise compared to other systems whose specification is more complex, or that rely on a more complex language of types.

Two type inference algorithms We have presented two type inference algorithms in this work. The first one is a modest modification of Algorithm W. The key additions are the usage of restriction contexts, indicating what unification variables may or may not stand for polymorphic types as well as the extension of unification to arbitrarily quantified types. During unification, restrictions need to be to checked and updated as needed.

Presenting a type inference algorithm based on Algorithm W shows that FreezeML imposes few requirements on the underlying inference algorithm. Further, being able to directly relate the principal types of FreezeML to the output of the algorithm made certain proofs about principal types (e.g., Lemma 3.2 on page 52) trivial. However, time has not stood still since the inception of Algorithm W in the 1970s. Type inference approaches based on constraint solving have been developed since, whose benefits have been illustrated in theory by the existence of extendable frameworks such as HM(X) [86] and OutsideIn(X) [116] as well as in practice by enabling the extensibility of GHC.

We have adapted Pottier and Rémy's approach [97] for FreezeML, where the entire type inference problem can be reduced to a constraint solving problem, allowing the translation and solving stages to be completely independent. However, the overall correctness of the approach remains as a conjecture due to its dependence on Conjec-

[1]As pointed out at the end of Section 3.2.1.3, there is a single exception to this rule, which causes no problems.

ture 5.1. We have discussed the obstacles faced when attempting to close this gap in detail in Section 5.4.2.

Section 6.2 offered a glimpse of the benefits of the constraint-based approach, where some extensions can be achieved simply by changing the translation function, but keeping the solver as is.

7.2 Future work

Of course, one important piece of future work is proving that Conjecture 5.1 holds, thus obtaining an overall correctness result for the constraint-based inference approach. We now explore additional avenues for future work, some of which have already been mentioned earlier.

7.2.1 Performing type inference in context

FreezeML typing judgements are explicit about the type variables in scope, which are left implicit in many traditional presentations of ML. This is particularly useful in FreezeML where we need to track what type variables can and cannot be substituted with polymorphic types.

This makes the *type inference in context* approach [35, 34] a natural candidate for adaption. In that work, contexts are not only used to track what type and term variables are in scope, but also record constraints imposed on them. For example, contexts can record known equalities between types and type variables. Crucially, contexts are ordered, where the entry for the nth variable in the context can only mention the $n-1$ previous variables in the context. This idea is similar to the usage of *telescopes* [14] in dependently typed systems. In the setting of first-class polymorphism, the idea of enriching ordered contexts with additional information is reminiscent of the order-dependent prefixes of MLF. They also appear in the work of Dunfield and Krishnaswami [16], which considers higher-rank polymorphism, but only predicative instantiation.

Thus, we consider an approach based on type inference in context for FreezeML to be an interesting third way, as a middle-ground between Algorithm W and constraint-based inference. It avoids the gap between terms and constraints naturally present in constraint-based inference, while still collecting constraints in a disciplined manner during inference, resolving the “dependency panic” that Gundry attributes to Algorithm W [34].

We return to this approach when discussing Explicit FreezeML next.

7.2.2 Explicit FreezeML

In FreezeML, generalisation and instantiation are inherently coupled to let bindings and (plain) variables, respectively. As a consequence, the typing rules for both constructs are concerned with more than one aspect of typing: As in a syntax-direct presentation of ML, the rule for $\textbf{let}\ x = M\ \textbf{in}\ N$ couples generalising the type of M with binding the variable x, while the typing rule for the term x looks up its type and performs instantiation.

The reader may therefore wonder whether there exists a declarative presentation of FreezeML, where each rule only has a single role, avoiding the somewhat algorithmic flair of the FreezeML typing rules in Figure 3.5 on page 45. After all, such a declarative presentation does exist for ML, as shown in Section 2.2.1, where generalisation and instantiation are performed by standalone rules.

Unfortunately, it is easy to see that the notion of type equivalence and principal types in FreezeML prevents this. If we added suitable versions of INST and GEN to FreezeML, performing polymorphic instantiation and generalisation to the principal type, we observe the following problem. Returning to our example singleton id from Section 1.3, we observe that in this modified version of the system, we could assign the types $\mathsf{List}\ (\forall a.a \to a)$ and $\forall a.\mathsf{List}\ (a \to a)$ to the term, with no principal type existing. This is not surprising: After all, systems that do employ standalone instantiation and generalisation rules, such as HMF, rely on careful heuristics elsewhere in the type system to limit their application.

One way to resolve this issue without complicating the typing rules is by denoting explicitly in syntax where instantiation and generalisation may occur. The resulting system, called Explicit FreezeML, then enjoys a simple, declarative specification. In the following, we use P and Q to refer to Explicit FreezeML terms.

Concretely, the FreezeML term $\textbf{let}\ x = U\ \textbf{in}\ N$, where U is a guarded value and hence generalisable, is translated to the Explicit FreezeML term $\textbf{let}\ \lceil x \rceil = \Lambda\bullet.P\ \textbf{in}\ Q$, where P and Q are the translated versions of the corresponding subterms. The Explicit FreezeML term relies on two constructs: Let bindings in Explicit FreezeML do not perform generalisation or impose a principality condition, but simply bind variables, similarly to let bindings in our version of System F (see Figure 2.8 on page 21). Note that the "frozen" let bindings of Explicit FreezeML are written using $\lceil x \rceil$ to distinguish them from those let bindings found in FreezeML (but absent in Explicit FreezeML). When applied to a term, the operator $\Lambda\bullet.P$ generalises it, using its *principal* type.

Explicit FreezeML also introduces a postfix instantiation operator $P\,\star$, which instantiates all toplevel quantifiers of P's type with polymorphic types. We can then translate a plain (i.e., instantiating) FreezeML variable x to the Explicit FreezeML term $\lceil x \rceil\,\star$. As a result, plain variables are not required in Explicit FreezeML, it only uses frozen ones. A second instantiation operator $P\bullet$ performs monomorphic instantiation instead. It can be used to obtain an explicit variant of ML itself, translating all terms x to $\lceil x \rceil\bullet$ instead.

In order to translate annotated let bindings of FreezeML to Explicit FreezeML, we may introduce a generalisation primitive $\Lambda^A.P$ that generalises P *to* type A, rather than to the principal one. However, we observe that this operator need not be a language primitive if we add System F style type abstractions to the language. Using our earlier translate of type annotations using annotated lambda terms, we may then define the following syntactic sugar within Explicit FreezeML.

$$\begin{array}{rcl} (P : A) & \equiv & (\lambda(x : A).\lceil x \rceil)\, P \\ \Lambda^{\forall \overline{a}.H}.P & \equiv & \Lambda\overline{a}.(P : H) \;\equiv\; \Lambda\overline{a}.((\lambda(x : H).\lceil x \rceil)\, P) \end{array}$$

Using this, we may then translate FreezeML terms $\textbf{let}\ (x : A) = M\ \textbf{in}\ N$ to $\textbf{let}\ \lceil x \rceil = \Lambda^{\overline{A}}.P\ \textbf{in}\ Q$.

The syntactic sugar used within Explicit FreezeML and the translation from FreezeML to Explicit FreezeML (denoted $\leadsto$) is shown in Figure 7.1 on the following page. Here, we use the convention that P and Q denote the translation of the first and second subterm of the original term (if existent), respectively. We observe that the translation is in fact a desugaring, in the sense that it can be performed on untyped terms and does no rely on type-checking the source program.

For the sake of simplicity, we do not enforce the value restriction within Explicit FreezeML, but implement it as part of the translation. However, we could also introduce a notion of syntactic values in Explicit FreezeML itself.

7.2.2.1 Retaining principal types

So far, we have not presented the full syntax of Explicit FreezeML. Based on our informal description so far, one may assume that it is defined as follows.

$$P, Q ::= \lceil x \rceil \mid \lambda x.P \mid \lambda(x : A).P \mid P\,Q \mid \Lambda a.P \mid P\,A \mid \textbf{let}\ \lceil x \rceil = P\ \textbf{in}\ Q \mid P\bullet \mid P\star$$

$$
\begin{array}{rcl}
(P : A) & \equiv & (\lambda(x : A).\lceil x \rceil)\, P \\
\Lambda^{\forall \overline{a}.H}.P & \equiv & \Lambda \overline{a}.(P : H)
\end{array}
$$

$$
\begin{array}{rcll}
x & \rightsquigarrow & \lceil x \rceil \star & \\
\lceil x \rceil & \rightsquigarrow & \lceil x \rceil & \\
\lambda x.M & \rightsquigarrow & \lambda x.P & \\
\lambda(x : A).M & \rightsquigarrow & \lambda(x : A).P & \\
M\,N & \rightsquigarrow & P\,Q & \\
\mathbf{let}\ x = U\ \mathbf{in}\ N & \rightsquigarrow & \mathbf{let}\ \lceil x \rceil = \Lambda\bullet.P\ \mathbf{in}\ Q & \\
\mathbf{let}\ x = M\ \mathbf{in}\ N & \rightsquigarrow & \mathbf{let}\ \lceil x \rceil = (\Lambda\bullet.P)\bullet\ \mathbf{in}\ Q & \text{if } M \notin \mathsf{GVal} \\
\mathbf{let}\ (x : A) = U\ \mathbf{in}\ N & \rightsquigarrow & \mathbf{let}\ \lceil x \rceil = \Lambda^{A}.P\ \mathbf{in}\ Q & \\
\mathbf{let}\ (x : A) = M\ \mathbf{in}\ N & \rightsquigarrow & \mathbf{let}\ \lceil x \rceil = (P : A)\ \mathbf{in}\ Q & \text{if } M \notin \mathsf{GVal}
\end{array}
$$

Figure 7.1: Explicit FreezeML syntactic sugar and translation from FreezeML

However, this system would lack principal types for two reasons.

1. Allowing System F style type abstractions without further restrictions can lead to terms having different, incomparable types. As an example, consider $\Lambda a.\lambda x.x$. We may pick a as the type for x, the principal type of the overall term is then $\forall a.a \to a$. However, not picking a leads to the principal type $\forall a.b \to b$ for some fresh b. This issue is what led to the incompleteness of McCracken's system [75], as identified by O'Toole and Gifford [89].

2. In Section 3.2.1.2 we used the following FreezeML term as an example for why monomorphic instantiation is required in FreezeML when the let-bound term is a not a value and therefore not generalised (see (3.6) on page 49)

$$\mathbf{let}\ f = \mathsf{bot}\ \mathsf{bot}\ \mathbf{in}\ (f\,3,\ \lceil f \rceil)$$

   A similar issue arises in Explicit FreezeML, where *all* let bindings are non-generalising. For example, consider the following related term, which we may naively consider to be the translation of the previous FreezeML term.

$$\mathbf{let}\ f = (\lceil \mathsf{bot} \rceil\ \star)\ (\lceil \mathsf{bot} \rceil\ \star)\ \mathbf{in}\ (f\,3,\ \lceil f \rceil)$$

   However, this term would have types such as $(\mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}))$ and $(\mathsf{Int} \times (\forall a.a \to a))$, but there would exist no principal one. As usual, the problem is that we would allow arbitrary polymorphism to enter the term context.

$$\begin{array}{lll}
\mathsf{ITerm} \ni & \mathsf{MTerm} \ni & \mathsf{PTerm} \ni \\
\quad I, J ::= \lceil x \rceil & \quad M, N ::= \lceil x \rceil & \quad P, Q ::= \lceil x \rceil \\
\qquad\quad \mid \lambda(x : A).I & \qquad\quad \mid \lambda(x : A).M & \qquad\quad \mid \lambda(x : A).P \\
\qquad\quad \mid I\,Q & \qquad\quad \mid M\,Q & \qquad\quad \mid P\,Q \\
\qquad\quad \mid \Lambda a.I & \qquad\quad \mid \Lambda a.I & \qquad\quad \mid \Lambda a.I \\
\qquad\quad \mid I\,A & \qquad\quad \mid M\,A & \qquad\quad \mid P\,A \\
\qquad\quad \mid \mathbf{let}\ \lceil x \rceil = I\ \mathbf{in}\ J & \qquad\quad \mid \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N & \qquad\quad \mid \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ Q \\
\qquad\quad \mid \Lambda\bullet.P & \qquad\quad \mid \Lambda\bullet.P & \qquad\quad \mid \Lambda\bullet.P \\
 & \qquad\quad \mid \lambda x.M & \qquad\quad \mid \lambda x.P \\
 & \qquad\quad \mid M\bullet & \qquad\quad \mid P\bullet \\
 & & \qquad\quad \mid P\star
\end{array}$$

Figure 7.2: Syntax of Explicit FreezeML

Both problems can be addressed in a similar way, by defining subsets of terms that are more restricted in what types they may have. Of course, our goal is that Explicit FreezeML terms P have principal types.

However, we also define terms M that guarantee that all their types can be obtained from the principal one using *monomorphic* instantiation only. The key idea is to disallow the polymorphic instantiation operator in M-terms.

We also define terms I that are guaranteed to have unique types. Here, we disallow both forms of instantiation operators, meaning that only System F style type applications remain to eliminate quantifiers, as well as removing un-annotated lambda functions.

This leads to the overall syntax of Explicit FreezeML shown in Figure 7.2. Note that the first problem we described is solved by only allowing I-terms under type abstractions. The second problem is solved by only allowing M-terms as let-bound terms in the last two categories. This means that Explicit FreezeML preserves the invariant I-2 mentioned on page 205. However, the invariant I-1 is not maintained in Explicit FreezeML, which is also the case when adding explicit type applications to FreezeML, as discussed in Section 6.2.2.1.

While this does not preclude the existence of principal types, it makes proving their existence more challenging as compared to the original design of FreezeML, where only known polymorphism is instantiated. For example, the term $\lceil \mathsf{bot} \rceil \star \mathsf{Int}$ allows any type of the form $\forall a.A$ for the subterm $\lceil \mathsf{bot} \rceil \star$, where A may contain a freely. However, while none of these types are most general, the overall term has principal type a for some fresh a.

Returning to the syntax of Explicit FreezeML in Figure 7.2, we further observe that the generalisation operator is allowed in all term forms (as it guarantees a unique type of the resulting term) and function arguments can always be P terms, as the overall function type dictates the type of the argument.

7.2.2.2 Summary

Explicit FreezeML can be considered to be an alternative, rationalised perspective on FreezeML. Effectively, the generalisation and instantiation operators, defined as syntactic sugar in FreezeML (see Section 3.2.2), become language primitives in Explicit FreezeML. This allows us to state a declarative specification of FreezeML. The careful definition of three different classes of terms denote a purely *syntactic* criterion to ensure the existence of principal types.

The author of this thesis has designed a type inference algorithm for Explicit FreezeML that is based on type inference in context (described in Section 7.2.1). However, this work is not finished at the time of writing.

We observe that the instantiation and generalisation primitives of Explicit FreezeML are closely related to operators found in the systems discussed in Section 2.3.2, but our system eschews the introduction of two separated forms of quantified types. The generalisation operator $\Lambda\bullet$ of Explicit FreezeML is similar to the **close** operator of IFX. Its annotated counterpart Λ^A is similar to the $[-]$ operator of Poly-ML and the prefix version of the $\{-\}$ operator in QML. In turn the monomorphic instantiation operator $\bullet$ is similar to the **open** operator of IFX, while polymorphic instantiation resembles Poly-ML's $\langle - \rangle$.

7.2.3 FreezeML(X)

Our constraint-based inference algorithm introduced in Chapter 5 performs type inference using an intermediary, namely the constraint language. While this on its own increases modularity, the natural end goal would be to fully embrace the constraint-based approach by allowing constraints to appear in types in the style of qualified types. Abstracting over the concrete constraint domain would then allow us to define "FreezeML(X)", a framework in the style of HM(X) or OutsideIn(X) for defining systems with first-class polymorphism using the ideas of FreezeML.

This would involve formalising requirements imposed upon the constraint domain X to ensure the existence of a complete inference algorithm, possibly by defining such an algorithm with individual parts of it being specific to X.

To the best of the author's knowledge, no such framework exists presently that supports first-class polymorphism. The definition of HM(X) only considers prenex polymorphism and polymorphic instantiation. While Serrano et al. state that their work on the GI [110] and QL [109] systems is in the context of the OutsideIn(X) framework, they do not explain the relationship in detail or formalise it. Further, their work does not include completeness results of their inference algorithms.

Concrete evidence for an amicable interaction of qualified types and first-class polymorphism exists. While Jones' work on qualified types is mostly focused on integrating it into an ML-style language and offering type inference, he also presents a version of System F with qualified types [44]. Leijen and Löh show that even a system as expressive as MLF can be extended to support qualified types [60].

However, in the presence of qualified types, the ordered treatment of quantifiers in FreezeML and Explicit FreezeML may pose problems. Jones provides an example where a field l is projected from a record r twice, using a constraint language for a record type systems. In his system, the concrete example term $\lambda r.(r.l, r.l)$ could be given both of the following two types.

$$\begin{array}{lll} \forall r.\forall a.\forall b.\, r \text{ has } l : a, r \text{ has } l : b & \Rightarrow & (a,b) \\ \forall r.\forall a. \quad r \text{ has } l : a & \Rightarrow & (a,a) \end{array}$$

Here, constraints r has $l : a$ indicate that record type r has a field l of type a. In the first type, the underlying equational theory of record types then implies that a and b must be equal.

In Jones' system, the two types are principal and equivalent. However, type-checking for two terms with these types would behave differently in the presence of explicit type applications, due to the different number of quantified variables.

7.2.4 Formalising variations and extensions

In Section 6.2, we discussed some variations and extensions of FreezeML. For most extensions, we discussed how to extend either the type inference algorithm based on Algorithm W or the constraint-based approach to support the extension. Nevertheless, we have not proved any of these extensions to be correct.

We expect this to be straightforward for some extensions, such as allowing free type variables in annotations (Section 6.2.1.2), removing the value restriction (Section 6.2.1.3), and recursive function definitions (Section 6.2.2.2), in particular in those cases where it suffices to change the translation from terms to constraints. We expect some variations and extensions to be more difficult to prove correct.

The most elegant way to prove the correctness of Unordered FreezeML (Section 6.2.1.1) would be to abstract our correctness proofs over the equational theory of types used. We would then abstractly rely on the fact that the unification algorithm $\mathcal{U}$ is correct with respect to that theory. While this is mostly the case (i.e., the proofs rarely reason about type equality and if so, in contexts where the unification algorithm is used), this is not formalised in our current presentation of the proofs.

We may also be able to make use of the future work on Explicit FreezeML when reasoning about FreezeML itself. Once we have proved that the former system enjoys principal types and that the translation between the two system preserves types, we may be able to use these results to show that adding System F style type applications to FreezeML itself leaves its key properties intact.

Bibliography

[1] Aiken, A. and Wimmers, E. L. (1993). Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA. Association for Computing Machinery.

[2] Baars, A. I. and Swierstra, S. D. (2002). Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 157–166, New York, NY, USA. Association for Computing Machinery.

[3] Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science.

[4] Barendregt, H. (1992). Lambda calculi with types. In *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*. Oxford University Press.

[5] Boehm, H.-J. (1985). Partial polymorphic type inference is undecidable. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 339–345, USA. IEEE Computer Society.

[6] Böhm, C. and Berarducci, A. (1985). Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154.

[7] Cheney, J., Lindley, S., and Wadler, P. (2013). A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*, pages 403–416. ACM.

[8] Cheney, J., Lindley, S., and Wadler, P. (2014). Query shredding: Efficient relational evaluation of queries over nested multisets. In *Proceedings of the 2014 ACM*

SIGMOD International Conference on Management of Data - SIGMOD '14, pages 1027–1038, New York, New York, USA. ACM Press.

[9] Church, A. (1941). *The Calculi of Lambda-Conversion*. Annals of Mathematics Studies. Princeton University Press.

[10] Clément, D., Despeyroux, T., Kahn, G., and Despeyroux, J. (1986). A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 13–27, New York, NY, USA. Association for Computing Machinery.

[11] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2006). Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO'06)*, pages 266–296. Springer.

[12] Damas, L. (1984). *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh.

[13] Damas, L. and Milner, R. (1982). 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, New York, NY, USA. Association for Computing Machinery.

[14] de Bruijn, N. G. (1991). Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204.

[15] Dunfield, J. and Krishnaswami, N. (2021). Bidirectional Typing. *ACM Computing Surveys*, 54(5):1–38.

[16] Dunfield, J. and Krishnaswami, N. R. (2013). Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 429–442, New York, NY, USA. Association for Computing Machinery.

[17] Eifrig, J., Smith, S., and Trifonov, V. (1995). Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132–153.

[18] Eisenberg, R. A., Weirich, S., and Ahmed, H. G. (2016). Visible Type Application. In *ESOP*, Lecture Notes in Computer Science, pages 229–254, Berlin, Heidelberg. Springer.

[19] Emrich, F., Lindley, S., and Stolarek, J. (2020a). The virtues of semi-explicit polymorphism. Extended abstract presented at the ML workshop @ ICFP 2020.

[20] Emrich, F., Lindley, S., Stolarek, J., Cheney, J., and Coates, J. (2020b). FreezeML: Complete and easy type inference for first-class polymorphism. In Donaldson, A. F. and Torlak, E., editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 423–437. ACM.

[21] Emrich, F., Stolarek, J., Cheney, J., and Lindley, S. (2022). Constraint-based type inference for FreezeML. *Proceedings of the ACM on Programming Languages*, 6(ICFP):570–595.

[22] Fehrenbach, S. and Cheney, J. (2018). Language-integrated provenance. *Science of Computer Programming*, 155:103–145.

[23] Fowler, S., Galpin, V., and Cheney, J. (2022). Language-integrated query for temporal data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2022, pages 5–19, New York, NY, USA. Association for Computing Machinery.

[24] Fowler, S., Lindley, S., Morris, J. G., and Decova, S. (2019). Exceptional asynchronous session types: Session types without tiers. *Proceedings of the ACM on Programming Languages*, 3(POPL):28:1–28:29.

[25] Garrigue, J. (2004). Relaxing the Value Restriction. In *7th International Symposium on Functional and Logic Programming*, pages 196–213. Springer, Berlin, Heidelberg.

[26] Garrigue, J. and Rémy, D. (1999). Semi-explicit first-class polymorphism for ML. *Inf. Comput.*, 155(1-2):134–169.

[27] GHC Guide (2023). GHC 9.6.1 user's guide: Documentation of `ImpredicativeTypes` extension. `https://downloads.haskell.org/~ghc/9.6.1/docs/users_guide/exts/impredicative_types.html`.

[28] Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA. Association for Computing Machinery.

[29] Girard, J.-Y. (1971). Une extension de l'interprétation de Gödel à l'analyse et son application à l'elimination des coupures dans l'analyse et dans la théorie des types. In Fenstad, J. E., editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. Elsevier.

[30] Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris Diderot.

[31] Goldfarb, W. D. (1981). The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230.

[32] Gordon, M. J., Milner, A. J., and Wadsworth, C. P. (1979). *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg.

[33] Greiner, J. (1996). Weak polymorphism can be sound. *Journal of Functional Programming*, 6(1):111–141.

[34] Gundry, A. (2013). *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde.

[35] Gundry, A., McBride, C., and McKinna, J. (2010). Type inference in context. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming - MSFP '10*, pages 43–54, New York, New York, USA. ACM Press.

[36] Henglein, F. (1993). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289.

[37] Hillerström, D. and Lindley, S. (2016). Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*, pages 15–27, New York, New York, USA. ACM Press.

[38] Hillerström, D. and Lindley, S. (2018). Shallow effect handlers. In Ryu, S., editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 415–435, Cham. Springer International Publishing.

[39] Hillerström, D., Lindley, S., Atkey, R., and Sivaramakrishnan, K. C. (2017). Continuation passing style for effect handlers. In Miller, D., editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*,

volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:19, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[40] Hindley, J. R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60.

[41] Hoang, M., Mitchell, J., and Viswanathan, R. (1993). Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25.

[42] Horn, R., Perera, R., and Cheney, J. (2018). Incremental relational lenses. *Proceedings of the ACM on Programming Languages*, 2(ICFP):74:1–74:30.

[43] Huet, G. P. (1973). The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267.

[44] Jones, M. P. (1994). A theory of qualified types. *Science of Computer Programming*, 22(3):231–256.

[45] Jones, M. P. (1997). First-class polymorphism with type inference. In *POPL '97*, POPL '97, pages 483–496, New York, NY, USA. ACM.

[46] Kaes, S. (1988). Parametric overloading in polymorphic programming languages. In Ganzinger, H., editor, *ESOP '88*, Lecture Notes in Computer Science, pages 131–144, Berlin, Heidelberg. Springer.

[47] Kaes, S. (1992). Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 193–204, New York, NY, USA. Association for Computing Machinery.

[48] Kfoury, A. J. and Tiuryn, J. (1992). Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257.

[49] Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. (1993). Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311.

[50] Kuan, G. and MacQueen, D. (2007). Efficient type inference using ranked type variables. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 3–14, New York, NY, USA. Association for Computing Machinery.

[51] Lämmel, R. and Jones, S. P. (2003). Scrap your boilerplate: A practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37.

[52] Läufer, K. and Odersky, M. (1994). Polymorphic type inference and abstract data types. *TOPLAS*, 16(5):1411–1430.

[53] Launchbury, J. and Peyton Jones, S. L. (1994). Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA. Association for Computing Machinery.

[54] Launchbury, J. and Peyton Jones, S. L. (1995). State in Haskell. *LISP and Symbolic Computation*, 8(4):293–341.

[55] Le Botlan, D. (2004). *MLF: An Extension of ML with First-Class Polymorphism and Implicit Instantiation*. PhD thesis, Ecole Polytechnique.

[56] Le Botlan, D. and Rémy, D. (2003). MLF: Raising ML to the power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 27–38, New York, NY, USA. Association for Computing Machinery.

[57] Leijen, D. (2007). HMF: Simple type inference for first-class polymorphism. Technical Report MSR-TR-2007-118, Microsoft Research.

[58] Leijen, D. (2008). HMF: Simple Type Inference for First-class Polymorphism. In *Proc. ICFP'08*, ICFP '08, pages 283–294, New York, NY, USA. ACM.

[59] Leijen, D. (2009). Flexible Types: Robust type inference for first-class polymorphism. In *Proc. POPL'09*, POPL '09, pages 66–77, New York, NY, USA. ACM.

[60] Leijen, D. and Löh, A. (2005). Qualified types for MLF. In Danvy, O. and Pierce, B. C., editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 144–155. ACM.

[61] Leivant, D. (1983a). Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 88–98, New York, NY, USA. Association for Computing Machinery.

[62] Leivant, D. (1983b). Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science (Sfcs 1983)*, pages 460–469.

[63] Leroy, X. (1993). Polymorphism by name for references and continuations. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 220–231, New York, NY, USA. Association for Computing Machinery.

[64] Leroy, X. and Weis, P. (1991). Polymorphic type inference and assignment. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 291–302, New York, NY, USA. Association for Computing Machinery.

[65] Levy, J. (1996). Linear second-order unification. In Ganzinger, H., editor, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pages 332–346, Berlin, Heidelberg. Springer.

[66] Libal, T. (2015). Regular patterns in second-order unification. In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 557–571. Springer.

[67] Lindley, S. and Cheney, J. (2012). Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 91–102.

[68] Lindley, S. and Morris, J. G. (2017). Lightweight Functional Session Types. In Simon Gay and António Ravara, editors, *Behavioural Types: From Theory to Tools*. River Publishers.

[69] Links (2022). Links language web page. https://links-lang.org/.

[70] MacQueen, D. (1984). Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, New York, NY, USA. Association for Computing Machinery.

[71] MacQueen, D., Harper, R., and Reppy, J. (2020). The history of Standard ML. *Proceedings of the ACM on Programming Languages*, 4(HOPL):86:1–86:100.

[72] Marlow, S. and Peyton Jones, S. (2012). The Glasgow Haskell Compiler. In Brown, A. and Wilson, G., editors, *The Architecture of Open Source Applications*, volume 2. Lulu.com.

[73] Matthews, D. C. J. (1988). An Overview of the Poly Programming Language. In Atkinson, M. P., Buneman, P., and Morrison, R., editors, *Data Types and Persistence*, Topics in Information Systems, pages 43–50, Berlin, Heidelberg. Springer.

[74] McAllester, D. (2003). A logical algorithm for ML type inference. In Nieuwenhuis, R., editor, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pages 436–451, Berlin, Heidelberg. Springer.

[75] McCracken, N. (1984). The typechecking of programs with implicit type structure. In *Proc. of the International Symposium on Semantics of Data Types*, pages 301–315, Berlin, Heidelberg. Springer-Verlag.

[76] Miller, D. (1992). Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358.

[77] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.

[78] Milner, R., Harper, R., MacQueen, D., and Mads, T. (1997). *The Definition of Standard ML (Revised Edition).* MIT Press.

[79] Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

[80] Mitchell, J. C. (1988). Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249.

[81] Mitchell, J. C. (1996). *Foundations for Programming Languages.* Foundations of Computing. MIT Press.

[82] Morris, J. G. and McKinna, J. (2019). Abstracting extensible data types: Or, rows by any other name. In *Proceedings of the ACM on Programming Languages*, pages 1–28. ACM.

[83] Mycroft, A. (1984). Polymorphic type schemes and recursive definitions. In Paul, M. and Robinet, B., editors, *International Symposium on Programming*, Lecture Notes in Computer Science, pages 217–228, Berlin, Heidelberg. Springer.

[84] Nipkow, T. and Prehofer, C. (1995). Type reconstruction for type classes. *J. Funct. Program.*, 5:201–224.

[85] Odersky, M. and Läufer, K. (1996). Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 54–67, New York, NY, USA. Association for Computing Machinery.

[86] Odersky, M., Sulzmann, M., and Wehr, M. (1999). Type Inference with constrained types. *TAPOS*, 5:35–55.

[87] Odersky, M., Wadler, P., and Wehr, M. (1995). A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 135–146, New York, NY, USA. Association for Computing Machinery.

[88] Odersky, M., Zenger, C., and Zenger, M. (2001). Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 41–53, New York, NY, USA. Association for Computing Machinery.

[89] O'Toole, J. W. and Gifford, D. K. (1989). Type reconstruction with first-class polymorphic values. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 207–217, Portland, Oregon, USA. Association for Computing Machinery.

[90] Peyton Jones, S. (2019). Type inference as constraint solving: How GHC's type inference engine actually works. Keynote at Zurihac 2019, Zurich, `https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/`.

[91] Peyton Jones, S. and Shields, M. (2004). Lexically scoped type variables. Technical report, Microsoft Research.

[92] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. (2007). Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82.

[93] Pfenning, F. (1988). Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 153–163, New York, NY, USA. Association for Computing Machinery.

[94] Pfenning, F. (1993). On the undecidability of partial polymorphic type reconstruction. *Fundam. Informaticae*, 19(1/2):185–199.

[95] Pierce, B. C. and Turner, D. N. (2000). Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44.

[96] Poly/ML (2022). Poly/ML language web page. https://www.polyml.org.

[97] Pottier, F. and Rémy, D. (2004). The essence of ML type inference. In Pierce, B. C., editor, *Advanced Topics in Types and Programming Languages*, pages 389–489. MIT Press.

[98] Rémy, D. (1989). Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 77–88. ACM Press.

[99] Rémy, D. (1992). Extension of ML type system with a sorted equation theory on types. Technical Report RR-1766, INRIA.

[100] Rémy, D. (1994). Programming objects with ML-ART, an extension to ML with abstract and record types. In *TACS*, pages 321–346. Springer-Verlag.

[101] Rémy, D. (2005). Simple, partial type-inference for System F based on type-containment. *ACM SIGPLAN Notices*, 40(9):130–143.

[102] Rémy, D. and Vouillon, J. (1998). Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50.

[103] Reynolds, J. C. (1974). Towards a theory of type structure. In Robinet, B. J., editor, *Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *LNCS*, pages 408–423. Springer.

[104] Roberto Di Cosmo (1995). *Isomorphisms of Types: From Lambda Calculus to Information Retrieval and Language Design*. Progress in Theoretical Computer Science. Birkhauser.

[105] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41.

[106] Russo, C. V. and Vytiniotis, D. (2009). QML: Explicit First-class Polymorphism for ML. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, pages 3–14, New York, NY, USA. ACM.

[107] Schmidt-Schauß, M. (2002). A decision algorithm for stratified context unification. *Journal of Logic and Computation*, 12(6):929–953.

[108] Schmidt-Schauß, M. and Schulz, K. U. (2002). Solvability of context equations with two context variables is decidable. *Journal of Symbolic Computation*, 33(1):77–122.

[109] Serrano, A., Hage, J., Peyton Jones, S., and Vytiniotis, D. (2020). A quick look at impredicativity. *Proceedings of the ACM on Programming Languages*, 4(ICFP):89:1–89:29.

[110] Serrano, A., Hage, J., Vytiniotis, D., and Peyton Jones, S. (2018). Guarded Impredicative Polymorphism. In *PLDI*, PLDI 2018, pages 783–796, New York, NY, USA. ACM.

[111] Shan, C.-c. (2004). Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15–22.

[112] Smith, G. S. (1993). Polymorphic type inference with overloading and subtyping. In Gaudel, M. C. and Jouannaud, J. P., editors, *TAPSOFT'93: Theory and Practice of Software Development*, Lecture Notes in Computer Science, pages 671–685, Berlin, Heidelberg. Springer.

[113] Talpin, J. P. and Jouvelot, P. (1994). The Type and Effect Discipline. *Information and Computation*, 111(2):245–296.

[114] Tofte, M. (1987). *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh.

[115] Tofte, M. (1990). Type inference for polymorphic references. *Information and Computation*, 89(1):1–34.

[116] Vytiniotis, D., Peyton Jones, S., Schrijvers, T., and Sulzmann, M. (2011). Outsidein(x) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412.

[117] Vytiniotis, D., Weirich, S., and Peyton Jones, S. (2006). Boxy types: Inference for higher-rank types and impredicativity. In *ICFP*.

[118] Vytiniotis, D., Weirich, S., and Peyton Jones, S. (2008). FPH: First-class polymorphism for Haskell. In *ICFP*.

[119] Wadler, P. (1990). Recursive types for free! Unpublished note. Revised 2008 and 2014.
`https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.

[120] Wadler, P. and Blott, S. (1989). 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, New York, New York, USA. ACM Press.

[121] Wand, M. (1991). Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15.

[122] Wells, J. B. (1994). Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. In *LICS*, pages 176–185. Society Press.

[123] Wells, J. B. (1999). Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111–156.

[124] Wells, J. B. (2002). The essence of principal typings. In Widmayer, P., Eidenbenz, S., Triguero, F., Morales, R., Conejo, R., and Hennessy, M., editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 913–925, Berlin, Heidelberg. Springer.

[125] Wright, A. K. (1992). Typing references by effect inference. In Krieg-Brückner, B., editor, *ESOP '92*, Lecture Notes in Computer Science, pages 473–491, Berlin, Heidelberg. Springer.

[126] Wright, A. K. (1995). Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355.

Appendix A

Proofs for Chapters 3 and 4

This appendix contains proofs for properties stated in Chapters 3 and 4, but not given there.

We use the following definitions and notations in this appendix and subsequent ones.

- Given $(a : R) \in \Theta$, by abuse of notation, we write $\Theta(a)$ for R.
- We define $\subseteq$ to be order-agnostic on any kind of context (i.e., Δ, Ξ, Θ), meaning that we have $\Delta \subseteq \Delta'$ iff the *set* of variables in Δ is a subset of those in Δ'. Note that for restriction contexts, this means that $\Theta \subseteq \Theta'$ holds iff for all $(a : R) \in \Theta$ we have $(a : R) \in \Theta'$ (i.e., restrictions are preserved).
- Restrictions form a lattice whose ordering relation $\leq$ satisfies exactly $R \leq R$ for all R and $\bullet \leq \star$.

A.1 Implicitly used properties

We now state some basic auxiliary properties that we may use in subsequent proofs without explicitly referencing the corresponding lemma.

The following lemma collects some straightforward strengthening and weakening properties: We may add and remove otherwise unused type variables from contexts.

Lemma A.1 (Strengthening and weakening)**.**
Let Δ_F and Θ_F each contain no duplicate type variables. Further, for all Δ'_F in the two-element set $\{\Delta_\mathrm{F}, \mathsf{ftv}(\Theta_\mathrm{F})\}$, let $\Delta'_\mathrm{F} \# \Delta, \Delta', \Delta'', \mathsf{ftv}(\Theta), \mathsf{ftv}(\Theta'), \mathsf{ftv}(\Theta'')$ and $\Delta'_\mathrm{F} \# \mathsf{ftv}(A)$ and $\Delta'_\mathrm{F} \# \mathsf{ftv}(\Gamma)$ and $\Delta'_\mathrm{F} \# \mathsf{ftv}(M)$ and $\Delta'_\mathrm{F} \# \mathsf{ftv}(\delta)$ and $\Delta'_\mathrm{F} \# \mathsf{ftv}(\theta)$ hold. Note that no requirements are imposed on Δ_1 and Θ_1.

Then the following holds.

1. $\Delta \vdash A$ **ok** *iff* $(\Delta, \Delta_F) \vdash_R A$ **ok**
2. $\Theta \vdash A$ **ok** *iff* $(\Theta, \Theta_F) \vdash_R A$ **ok**
3. $\Delta \vdash \Gamma$ **ok** *iff* $(\Delta, \Delta_F) \vdash \Gamma$ **ok**
4. $\Theta \vdash \Gamma$ **ok** *iff* $(\Theta, \Theta_F) \vdash \Gamma$ **ok**
5. $\Delta;\Gamma \vdash M$ **ok** *iff* $(\Delta, \Delta_F);\Gamma \vdash M$ **ok**
6. $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$ *iff* $(\Delta, \Delta_F) \vdash \delta : \Delta' \Rightarrow \Delta''$
7. $\Delta \vdash \delta : \Delta_1 \Rightarrow \Delta''$ *iff* $\Delta \vdash \delta : \Delta_1 \Rightarrow (\Delta'', \Delta_F)$
8. $\Delta \vdash \theta : \Theta' \Rightarrow \Theta''$ *iff* $(\Delta, \Delta_F) \vdash \theta : \Theta' \Rightarrow \Theta''$
9. $\Delta \vdash \theta : \Theta_1 \Rightarrow \Theta''$ *iff* $\Delta \vdash \theta : \Theta_1 \Rightarrow (\Theta'', \Theta_F)$
10. $\Delta;\Gamma \vdash M : A$, *iff* $(\Delta, \Delta_F);\Gamma \vdash M : A$
11. $\Theta;\Gamma \vdash^{E} M : A$ *iff* $(\Theta, \Theta_F);\Gamma \vdash^{M} : A$
12. $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ *iff* $\mathsf{principal}((\Delta, \Delta_F), \Gamma, M, \Delta', A)$
13. $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ *iff* $\mathsf{principal}(\Delta, \Gamma, M, (\Delta', \Delta_F), A)$.

Proof. Each property can be shown by induction on the element under consideration. The proofs for properties 10 to 13 are mutually dependent, but in a well founded fashion: Properties 10 and 11 rely only on property 12 and this reliance only involves strict subterms. $\square$

The following lemma relates the two well-formedness judgements for types, $\Delta \vdash_R A$ **ok** and $\Theta \vdash_R A$ **ok** as well as stating some manipulations of restriction contexts that preserve them. Recall that Δ^R stands for a restriction context where all variables from Δ receive restriction R.

Lemma A.2 (Relationships between type well-formedness judgements)**.**
The following properties hold:

1. $\Delta \vdash_R A$ **ok** *implies* $\mathsf{ftv}(\Delta)^{\bullet} \vdash_R A$ **ok**
2. $\Theta \vdash_R A$ **ok** *implies* $\mathsf{ftv}(\Theta) \vdash_R A$ **ok**

3. $(\Theta, \Theta') \vdash_R A$ **ok** *implies* $(\Theta, \mathsf{ftv}(\Theta)^\bullet) \vdash_R A$ **ok**

4. $(\Theta, \mathsf{ftv}(\Theta')^\bullet) \vdash_\star A$ **ok** *implies* $(\Theta, \Theta') \vdash_\star A$ **ok**

Proof. Follows directly from the definitions of both well-formedness judgements. □

The following lemma relates the two well-formedness judgements for term contexts, $\Delta \vdash \Gamma$ **ok** and $\Theta \vdash \Gamma$ **ok**.

Lemma A.3 (Relationships between term context well-formedness judgements)**.**
The following properties hold:

1. $\Delta \vdash \Gamma$ **ok** *implies* $\Delta^\bullet \vdash \Gamma$ **ok**

2. $\Theta \vdash \Gamma$ **ok** *implies* $\mathsf{ftv}(\Theta) \vdash \Gamma$ **ok**

Proof. Follows directly from the definitions of both well-formedness judgements. □

Lemma A.4 (Properties of substitutions)**.**
Let $\Delta \vdash \theta : \Theta_1 \Rightarrow \Theta_2$ *hold. Then we have the following:*

1. $(\Delta^\bullet, \Theta_1) \vdash_R A$ **ok** *implies* $(\Delta^\bullet, \Theta_2) \vdash_R \theta(A)$ **ok**

2. $(\Delta^\bullet, \Theta_1) \vdash \Gamma$ **ok** *implies* $(\Delta^\bullet, \Theta_2) \vdash \theta(\Gamma)$ **ok**

3. $\Delta \vdash \theta' : \Theta_2 \Rightarrow \Theta_3$ *implies* $\Delta \vdash \theta' \circ \theta : \Theta_1 \Rightarrow \Theta_3$

4. $\Delta \vdash \theta' : \Theta_1, \mathsf{ftv}(\Theta_2)^\bullet \Rightarrow \Theta_3$ *implies* $\Delta \vdash \theta' : \Theta_1, \Theta_2 \Rightarrow \Theta_3$

5. $\Delta \vdash \theta' : \Theta_1 \Rightarrow \Theta_2, \Theta_3$ *implies* $\Delta \vdash \theta' : \Theta_1 \Rightarrow, \Theta_2, \mathsf{ftv}(\Theta_3)^\bullet$

Proof. Follows straightforwardly from the involved definitions and Lemma A.2. □

The following property is trivial, but important for the proof of Theorem 4.2 and referenced from there.

Lemma A.5.
Let $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta \vdash \theta' : \Theta' \Rightarrow \Theta''$*. Further, let* A *and* B *be well formed under* (Δ, Θ)*. If* $\theta(A) = \theta(B)$ *then* $(\theta' \circ \theta)(A) = (\theta' \circ \theta)(B)$*.*

Proof. Follows immediately from premises. □

A.2 Auxiliary lemmas

We now state additional properties used in subsequent proofs.

So far, the type substitution properties discussed in this work (i.e., Lemma 4.1 and Lemma 4.4) always assumed that the term under consideration is well formed under a type context Δ (i.e., only uses variables from Δ in annotations) that is *not* subject to substitution. General substitution properties that do substitute type variables in terms are not required in this work. Nevertheless, we define the following property that does take type variables in terms into account. However, it is limited to the trivial case where type variables are simply renamed, rather than substituted with arbitrary types.

Lemma A.6 (Renaming of type variables bound in terms)**.**
*Let $\overline{a};\Gamma \vdash M$ **ok** and $\overline{b}\#\Theta$. Further, let δ be a bijection between $\overline{a}$ and $\overline{b}$ such that $\delta(\overline{a}) = \overline{b}$.*

Then we have that $(\overline{a}^{\bullet},\Theta);\Gamma \vdash^{E} M : A$ implies $(\overline{b}^{\bullet},\Theta);\delta(\Gamma) \vdash^{E} M[\overline{b}/\overline{a}] : \delta(A)$.

Proof. We prove the statement of the lemma in parallel with the following, additional property: If $\overline{a},\mathsf{ftv}(\Theta) \vdash \Gamma$ **ok** and $\overline{a};\Gamma \vdash M$ **ok** and $\overline{b}\#\Delta'$, then $\mathsf{principal}((\overline{a},\mathsf{ftv}(\Theta)),\Gamma,M,\Delta',A)$ implies $\mathsf{principal}((\overline{b},\mathsf{ftv}(\Theta)),\delta(\Gamma),M[\overline{b}/\overline{a}],\Delta',\delta(A))$. We may prove this property using the original statement of this lemma together with Lemma 4.1 (to perform some freshening of variables *not* apearing in the term), and then utilise this additional statement on subterms when proving the original statement of the lemma by structural induction on M. □

The following lemma states that the original and extended FreezeML typing judgements both ensure that any type given to a term is well-formed under the given contexts.

Lemma A.7 (Well-formedness of accepted types)**.**
The following properties hold.

1. *If $\Delta;\Gamma \vdash M : A$, then $\Delta \vdash A$ **ok**.*

2. *If $\Theta;\Gamma \vdash^{E} M : A$, then $\Theta \vdash A$ **ok**.*

Proof. The first property can easily be proved by structural induction on M. The second part of the lemma follows immediately by Lemma 4.2 and the fact that for all Θ' and A', we have $\Theta' \vdash_{\star} A'$ **ok** iff $\mathsf{ftv}(\Theta') \vdash_{\star} A'$ **ok** (i.e., the restrictions in Θ' are irrelevant for the *polymorphic* version of the well-formedness judgement). □

Lemma A.8 (Principal types contain no spurious variables)**.**
Let $\Delta;\Gamma \vdash M$ **ok** *and* $(\Delta,\Delta') \vdash \Gamma$ **ok**. *If* $\mathsf{principal}((\Delta,\Delta'),\Gamma,M,\Delta'',A)$*, then* $\mathsf{ftv}(A) \subseteq (\Delta, \mathsf{ftv}(\Gamma) - \Delta, \Delta'')$ *holds.*

Proof. The principality premise directly implies $(\Delta,\Delta',\Delta'');\Gamma \vdash M : A$. Note that according to Lemma A.7, we then have $\mathsf{ftv}(A) \subseteq \Delta,\Delta',\Delta''$.

We now assume that there exists an $a \in \mathsf{ftv}(A) - \Delta - \mathsf{ftv}(\Gamma) - \Delta''$, which implies $a \in \Delta'$. Let S_g be a monomorphic ground type (e.g., Unit). Further, let δ_g such that $\delta_\mathrm{g}(b) = b$ for all $b \in \Delta' - a$ and $\delta_\mathrm{g}(a) = S_\mathrm{g}$. This definition implies $(\Delta,\Delta'') \vdash \delta_\mathrm{g} : \Delta' \Rightarrow_\bullet \Delta'$.

By Lemma 4.1, applying this substitution to the earlier typing judgement for M yields $(\Delta,\Delta',\Delta'');\delta_\mathrm{g}(\Gamma) \vdash M : \delta_\mathrm{g}(A)$. Since δ_g maps all variables in Γ to themselves, this is equivalent to $(\Delta,\Delta',\Delta'');\Gamma \vdash M : \delta_\mathrm{g}(A)$. Note that by assumption $a \in \mathsf{ftv}(A)$, the types $\delta_\mathrm{g}(A)$ and A are not the same. We then observe that there exists no δ such that $(\Delta,\Delta') \vdash \delta : \Delta'' \Rightarrow \Delta''$ and $\delta(A) = \delta_\mathrm{g}(A)$, since δ maps a to itself. This violates the assumption $\mathsf{principal}((\Delta,\Delta'),\Gamma,M,\Delta'',A)$, □

A.2.1 Auxiliary properties of unification algorithm

Lemma A.9 (Properties of returned restriction context)**.**
Let $(\Delta^\bullet,\Theta) \vdash A$ **ok** *and* $(\Delta^\bullet,\Theta) \vdash B$ **ok**. *Then* $\mathcal{U}(\Delta,\Theta,A,B) = (\Theta',\theta')$ *implies the following.*

1. *We have* $\mathsf{ftv}(\Theta') \subseteq \mathsf{ftv}(\Theta)$.

2. *We have* $\mathsf{ftv}(\theta') - \Delta \approx \mathsf{ftv}(\Theta')$ *(i.e.,* θ *is surjective with respect to* Θ'*).*

Proof. Like other proofs about the unification algorithm, by induction on the number of recursive calls to $\mathcal{U}$.

All modifications of the restriction context originate from the last two clauses in the definition of the algorithm in Figure 4.4, where a flexible variable a is unified with a type A other than itself. As per the assertion we have that A does not contain a. The returned substitution contains identity mappings for all variables in $\mathsf{ftv}(\Theta) - a$, meaning that we have $\mathsf{ftv}(\theta') - \Delta \approx \mathsf{ftv}(\Theta) - a$ Of course, this also means that the variables in Θ_1 are a (strict) subset of those in Θ. □

The following lemma states that the substitutions returned by $\mathcal{U}$ are idempotent. Further, we have that the returned unifier contains identity mappings for all variables found in the input restriction context, but not in the input types. The restrictions on these variables are preserved in the output unchanged.

Lemma A.10 (Properties of returned substitution)**.**
Let $(\Delta^{\bullet},\Theta) \vdash A$ **ok** *and* $(\Delta^{\bullet},\Theta) \vdash B$ **ok**. *Then* $\mathcal{U}(\Delta,\Theta,A,B) = (\Theta_{\mathrm{u}},\theta_{\mathrm{u}})$ *implies the following.*

- *For all* $a \in \mathsf{ftv}(\theta_{\mathrm{u}})$ *we have* $\theta_{\mathrm{u}}(a) = a$.
- *For all* $a \in \mathsf{ftv}(\Theta) - \mathsf{ftv}(A) - \mathsf{ftv}(B)$ *we have* $\theta_{\mathrm{u}}(a) = a$ *and* $\Theta(a) = \Theta_{\mathrm{u}}(a)$.

Proof. By induction on number of recursive calls to $\mathcal{U}$. Note that by Theorem 4.2 we have that Θ is indeed the domain of θ_{u}.

The first interesting case arises in the last two clauses in the definition of the algorithm in Figure 4.4, where a flexible variable a is unified with a type A other than itself. These clauses represent the origin of non-identity mappings in the resulting substitution. Crucially, the assertion $(\Delta^{\bullet},\Theta_1) \vdash_R A$ then ensures $a \notin \mathsf{ftv}(A)$, as a is not present in Θ_1. Since *a is* one of the input types, this immediately gives us the second statement of the lemma.

The other interesting case is the unification of data constructors. We perform a nested induction, assuming the following:

$$\Delta \vdash \theta_i : \Theta \Rightarrow \Theta_i \qquad \textbf{(1)}$$

$$\mathsf{ftv}(\theta_i) - \Delta \approx \mathsf{ftv}(\Theta_i) \qquad \textbf{(2)}$$

$$\mathsf{ftv}(\Theta) \supseteq \mathsf{ftv}(\Theta_i) \qquad \textbf{(3)}$$

θ_i is idempotent (as defined in the first statement of this lemma) **(4)**

In each step, we obtain $\Delta \vdash \theta' : \Theta_i \Rightarrow \Theta'$ from Theorem 4.2 and the surjectivity of θ' w.r.t. Θ' from the second part of this lemma. We have $\mathsf{ftv}(\Theta_i) \supseteq \mathsf{ftv}(\Theta')$ according to Lemma A.9. We can then easily obtain properties (1) to (3) for the composition θ_{i+1}.

To show that θ_{i+1} is idempotent, we assume $a \in \Theta$ and $b \in \mathsf{ftv}(\theta_{i+1}(a))$, which implies $b \in \Delta, \mathsf{ftv}(\Theta')$. If $b \in \Delta$, we immediately have the desired property $\theta_{i+1}(b) = b$.

Otherwise, there exists $c \in \mathsf{ftv}(\theta_i(a))$ such that $b \in \mathsf{ftv}(\theta'(c))$. By induction, θ' is idempotent, meaning that $\theta'(b) = b$ holds. By $\mathsf{ftv}(\Theta_i) \supseteq \mathsf{ftv}(\Theta')$, we further have that $b \in \Theta'$ implies $b \in \Theta_i$. Due to (2), we then have that $b \in \mathsf{ftv}(\theta_i)$. By (3) we have $b \in \Theta$. The idempotence of θ_i then gives us $\theta_i(b) = b$. Altogether, this implies $\theta_{i+1}(b) = b$, which was required to show that θ_{i+1} is idempotent. □

A.2.2 Auxiliary properties of inference algorithm

The following lemma states that the results of the type inference function infer are unique up to the choices of type variables picked by the algorithm.

Lemma A.11 (Equivalence of inference results modulo renaming)**.**
Let $\mathsf{infer}(\Delta,\Theta,M,A)$ *return* (Θ_1,θ_1,A_1) *and* (Θ_2,θ_2,A_2)*. Then there exists a bijective substitution* θ *such that* $\cdot \vdash \theta : \Theta_1 \Rightarrow \Theta_2$ *and* $\cdot \vdash \theta^{-1} : \Theta_2 \Rightarrow \Theta_1$ *and* $A_2 = \theta(A_1)$ *and* $\theta \circ \theta_1 = \theta_2$.

Proof. The unification algorithm $\mathcal{U}$ shown in Figure 4.4 is deterministic except for the choice of the skolem variable c in the case where quantified types are unified. The variable c does however not appear in the result and is only used for the escape check.

The type inference algorithm in Figure 4.5 is deterministic except for the choices of the following variables:

- The choice of $\overline{b}$ when instantiating a plain variable
- The choice of b used as a placeholder for the return type of M' when inferring a type for $M'\ N'$
- The choice of a used for the type of an un-annotated lambda parameter

Thus, we can easily obtain the desired property for infer by structural induction on M. □

The following lemma states that whenever a term context Γ already contains sufficient information to type a term M, then the inference algorithm will return a substitution that may only rename variables, but not perform other changes.

Lemma A.12 (Inferred substitutions for valid contexts)**.**
Let $\Delta;\Gamma \vdash M$ **ok** *and* $\Delta;\Theta \vdash \Gamma$ **ok** *hold. If* $(\Delta^\bullet,\Theta);\Gamma \vdash^{E} M : A$*, then* $\mathsf{infer}(\Delta,\Theta,\Gamma,M) = (\Theta',\theta,A')$ *implies that* θ *maps all* $a \in \Theta$ *to pairwise different variables in* Θ'.

Proof. By Theorem 4.5, we have $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

We have $\Delta \vdash \iota_\Theta : \Theta \Rightarrow \Theta$ for the identity substitution on Θ. We may thus apply Theorem 4.6 and Lemma A.11 to $(\Delta,\Theta);\iota_\Theta(\Gamma) \vdash M : A$, yielding the existence of θ' such that $\Delta \vdash \theta' : \Theta' \Rightarrow \Theta$ and $\iota_\Theta = \theta' \circ \theta$. This implies the desired property. □

A.3 Stability of typing and principality under substitution

We may now prove the stability of FreezeML typing under substitution, as stated in Lemma 4.4. As discussed in Section 4.1.3, this property crucially relies on the fact that principality is also preserved under substitution, which we prove first.

Lemma 4.5 (Stability of principality under substitution; *lemma originally stated on page 78).*
Let $(\Delta^\bullet, \Theta) \vdash \Gamma\,\mathbf{ok}$ *and* $\Delta; \Gamma \vdash M\,\mathbf{ok}$ *and* $\mathsf{ftv}(\Theta') \# \Delta'$*. If* $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta)), \Gamma, M, \Delta', A)$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$*, then* $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta')), \theta(\Gamma), M, \Delta', \theta(A))$.

Proof. Let $\Delta_{\mathrm{g}} := \mathsf{ftv}(\Gamma) - \Delta$ and $\Delta_{\mathrm{A}} := \mathsf{ftv}(A) - (\Delta, \Delta_{\mathrm{g}})$. By Lemma A.8, we then have $\Delta_{\mathrm{A}} \subseteq \Delta'$. Note that by assumption $(\Delta, \Theta) \vdash \Gamma\,\mathbf{ok}$, we have $(a : \bullet) \in \Theta$ for all $a \in \Delta_{\mathrm{g}}$. We define θ_{g} as the restriction of θ to Δ_{g}. Further, let $\Delta'_{\mathrm{g}} := \mathsf{ftv}(\theta_{\mathrm{g}}) - \Delta$.

We therefore have $\Delta \vdash \theta_{\mathrm{g}} : \Delta_{\mathrm{g}}^\bullet \Rightarrow \Delta_{\mathrm{g}}'^{\bullet}$ **(1)**. and $\Delta \vdash \theta_{\mathrm{g}} : \Delta_{\mathrm{g}} \Rightarrow_\bullet \Delta'_{\mathrm{g}}$ **(2)**. Note that in the former, we ignore the restrictions that Θ' may carry for the variables in Δ'_{g} and replaced them all with $\bullet$ instead. In the latter, we treat the substitution θ as an instantiation.

We may then strengthen the assumption of A being the principal type of M to $\mathsf{principal}((\Delta, \Delta_{\mathrm{g}}), \Gamma, M, \Delta_{\mathrm{A}}, A)$ **(3)**. We will prove that this implies $\mathsf{principal}((\Delta, \Delta'_{\mathrm{g}}), \theta_{\mathrm{g}}(\Gamma), M, \Delta_{\mathrm{A}}, \theta_{\mathrm{g}}(A))$. This is equivalent to $\mathsf{principal}(\Delta, \Delta'_{\mathrm{g}}, \theta(\Gamma), M, \Delta_{\mathrm{A}}, \theta(A))$, which we may in turn weaken to the principality property stated by the lemma.

Proving $\mathsf{principal}((\Delta, \Delta'_{\mathrm{g}}), \theta_{\mathrm{g}}(\Gamma), M, \Delta_{\mathrm{A}}, \theta_{\mathrm{g}}(A))$ requires showing the following.

P-1. We have $(\Delta, \Delta'_{\mathrm{g}}, \Delta_{\mathrm{A}}); \theta_{\mathrm{g}}(\Gamma) \vdash M : \theta_{\mathrm{g}}(A)$.

P-2. For any arbitrarily chosen $A_{\mathrm{c}}, \Delta_{\mathrm{c}}$, we have that $(\Delta, \Delta'_{\mathrm{g}}, \Delta_{\mathrm{c}}); \theta_{\mathrm{g}}(\Gamma) \vdash M : A_{\mathrm{c}}$ implies that there exists δ_{c} such that $\Delta, \Delta'_{\mathrm{g}} \vdash \delta : \Delta_{\mathrm{A}} \Rightarrow_\star \Delta_{\mathrm{c}}$ and $A_{\mathrm{c}} = \delta_{\mathrm{c}}(\theta_{\mathrm{g}}(A))$.

We have that (3) implies $(\Delta, \Delta_{\mathrm{g}}, \Delta_{\mathrm{A}}); \Gamma \vdash M : A$, we can then immediately show P-1 by applying Lemma 4.1, using (2). The remainder of this proof is concerned with showing P-2, which is achieved by relying on the completeness of type inference and the principality of inferred types.

We assume that A_{c} and Δ_{c} are given and the aforementioned typing judgement to hold. We apply the completeness theorem (Theorem 4.6) to the judgement $(\Delta^\bullet, \Delta_{\mathrm{g}}'^{\bullet}, \Delta_{\mathrm{c}}^\bullet)$; $\theta_{\mathrm{g}}(\Gamma) \vdash^{\mathrm{E}} M : A_{\mathrm{c}}$ (obtained from using Lemma 4.2) and the substitution θ_{g} (after weakening (1) to $\Delta \vdash \theta_{\mathrm{g}} : \Delta_{\mathrm{g}}^\bullet \Rightarrow \Delta_{\mathrm{g}}'^{\bullet}, \Delta_{\mathrm{c}}^\bullet$), which then guarantees that invoking $\mathsf{infer}(\Delta, \Delta_{\mathrm{g}}^\bullet,$

$\Gamma, M)$ succeeds, returning some $(\Theta_\mathrm{i}, \theta_\mathrm{i}, A_\mathrm{i})$, and there exists θ' such that we have the following.

$$\Delta \vdash \theta' : \Theta_\mathrm{i} \Rightarrow (\Delta_\mathrm{g}'^{\bullet}, \Delta_\mathrm{c}^{\bullet}) \qquad \textbf{(4)}$$

$$\theta_\mathrm{g} = \theta' \circ \theta_\mathrm{i} \qquad \textbf{(5)}$$

$$A_\mathrm{c} = \theta'(A_\mathrm{i}) \qquad \textbf{(6)}$$

Note that by soundness of inference (Theorem 4.5), we have $(\Delta^{\bullet}, \Theta_\mathrm{i}); \theta_\mathrm{i}(\Gamma) \vdash M : A_\mathrm{i}$, which implies $\mathsf{ftv}(A_\mathrm{i}) \subseteq \Delta, \mathsf{ftv}(\Theta_\mathrm{i})$ (see Lemma A.7). The same theorem also implies $\Delta \vdash \theta_\mathrm{i} : \Delta_\mathrm{g}^{\bullet} \Rightarrow \Theta_\mathrm{i}$. We assume w.l.o.g. that all variables in $\mathsf{ftv}(\Theta_\mathrm{i}) - \Delta_\mathrm{g}$ are fresh with respect to all previously mentioned contexts, in particular $\big(\mathsf{ftv}(A_\mathrm{i}) - \Delta, \mathsf{ftv}(\theta_\mathrm{i}(\Delta_\mathrm{g}))\big)$ $\# \Delta_\mathrm{g}$ **(7)** and $\mathsf{ftv}(\Theta_\mathrm{i}) \# \Delta_\mathrm{A}$ **(8)**. Formally, this freshness assumption is justified as we may otherwise apply a renaming substitution δ_f to A_i that performs the necessary freshening, while preserving the existence of an appropriate θ' satisfying the properties (4) to (6) for $\delta_\mathrm{f}(A_\mathrm{i})$.[1]

By Lemma A.12 we have that θ_i is merely a renaming of type variables. Concretely, θ_i is a bijection between Δ_g and $\mathsf{ftv}(\theta_\mathrm{i}) =: \Delta_\mathrm{r}$, a subset of Θ_i. We observe that by (1), (4) and (5), we have that $\Delta \vdash \theta' \restriction_{\Delta_\mathrm{r}} : \Delta_\mathrm{r}^{\bullet} \Rightarrow \Delta_\mathrm{g}'^{\bullet}$ **(9)** holds.

The high-level idea of the remainder of the proof is to relate $\theta'(A_\mathrm{i})$, which we know to be equal to A_c, and $\delta_\mathrm{c}(\theta_\mathrm{g}(A))$, which we must show to be equal to A_c by providing an appropriate choice for δ_c. We observe that θ' plays two separate roles: On the codomain of θ_i, θ' is the same as θ_g, except that it must undo the renaming performed by θ_i (see (5)). On the generalisable variables within A_i (i.e., those *not* in the codomain of θ_i), θ' performs the necessary instantiation to obtain A_c from A_i (see (6)). Here, we have that A_i is a principal type of M in term environment $\theta_\mathrm{i}(\Gamma)$. Since θ_i simply renames variables, this allows us to relate A_i and the type A, which by assumption is a principal type of M in the original term context Γ prior to the renaming. Thus, we have that A_i and A are equal modulo a bijection of a subset of the free type variables in each type. As a result, we may establish $\theta'(A_\mathrm{i}) = A_\mathrm{c} = \delta_\mathrm{c}(\theta_\mathrm{g}(A))$ by choosing δ_c such that it first renames the generalisable variables of A (namely, Δ_A) to the corresponding ones in A_i, and then uses θ' to perform the necessary instantiation to obtain A_c from a principal type of M. Accounting for the various renamings makes showing these steps

[1]This assumption directly reflects the behaviour of the inference algorithm: The output restriction context Θ_i may re-use some variables from the input restriction context, which corresponds to the variables Δ_g in our case. All other variables in Θ_i are chosen fresh by the algorithm. The reasoning why the assumption is formally justified may equivalently be used to justify the assumption that *all* of Θ_i is disjoint from any of the contexts used beforehand, but we eschew making this stronger assumption as it contradicts what the algorithm does.

in full detail in the remainder of this proof somewhat tedious.

Concretely, the principality of inferred types (Lemma A.13) implies that we have $\mathsf{principal}(\Delta, \Delta'_{\mathrm{p}}, \theta_{\mathrm{i}}(\Gamma), M, \Delta_{\mathrm{p}}, A_{\mathrm{i}})$, where $\Delta_{\mathrm{p}} = \mathsf{ftv}(\Theta_{\mathrm{i}}) - \Delta_{\mathrm{r}}$ and $\Delta'_{\mathrm{p}} = \mathsf{ftv}(\Theta_{\mathrm{i}}) - \Delta'_{\mathrm{p}}$. We may strengthen this to $\mathsf{principal}(\Delta, \theta_{\mathrm{i}}(\Delta_{\mathrm{g}}), \theta_{\mathrm{i}}(\Gamma), M, \Delta''_{\mathrm{p}}, A_{\mathrm{i}})$, where $\Delta''_{\mathrm{p}} = \mathsf{ftv}(A_{\mathrm{i}}) - \Delta, \theta_{\mathrm{i}}(\Delta_{\mathrm{g}})$. Inverting the renaming performed by θ_{i} then yields $\mathsf{principal}((\Delta, \Delta_{\mathrm{g}}), \Gamma, M, \Delta''_{\mathrm{p}}, \theta_{\mathrm{i}}^{-1}(A_{\mathrm{i}}))$, utilising the fact that θ_{i}^{-1} has no effect on Δ and Δ''_{p}. Since we already know that A is a principal type of M in the same contexts, the uniqueness of principal types (Lemma 3.1) guarantees the existence of a renaming δ such that $\Delta, \Delta_{\mathrm{g}} \vdash \delta : \Delta_{\mathrm{A}} \Rightarrow_{\bullet} \Delta''_{\mathrm{p}}$ and $\theta_{\mathrm{i}}^{-1}(A_{\mathrm{i}})) = \delta(A)$ as well as $\delta^{-1}(\theta_{\mathrm{i}}^{-1}(A_{\mathrm{i}}))) = A$ **(10)**.

We choose δ_{c} to be $\theta' \circ \delta$, which immediately yields the required property $\Delta, \Delta'_{\mathrm{g}} \vdash \delta_{\mathrm{c}} : \Delta_{\mathrm{A}} \Rightarrow_{\star} \Delta_{\mathrm{c}}$. It remains to show that this choice results in the desired relationship $\delta_{\mathrm{c}}(\theta_{\mathrm{g}}(A)) = A_{\mathrm{c}}$.

We require two additional properties before being able to show this. First, note that the earlier freshness assumption (7) is equivalent to $\Delta''_{\mathrm{p}} \# \Delta_{\mathrm{g}}$ **(11)**. Secondly, we have that $\delta_{\mathrm{c}}\theta'\delta^{-1}(A_{\mathrm{i}}) = \theta'(A_{\mathrm{i}})$ **(12)** holds. To show the latter, we consider the different cases for $a \in \mathsf{ftv}(A_{\mathrm{i}}) \subseteq \Delta, \Delta_{\mathrm{r}}, \Delta''_{\mathrm{p}}$. For $a \in \Delta$, this follows immediately.

1. If $a \in \Delta_{\mathrm{r}}$, we have that δ^{-1} has no effect and thus $\theta'(\delta^{-1}(a)) = \theta'(a)$. The free type variables of the latter type are a subset of $\Delta, \Delta'_{\mathrm{g}}$ (due to (9)), which is disjoint from Δ_A, meaning that applying δ_c has no effect.

2. If $a \in \Delta''_{\mathrm{p}}$, we have that $\delta^{-1}(a)$ is a single type variable b from Δ_{A}, meaning that applying θ' has no effect. Thus, we have $\theta'(\delta^{-1}(a)) = b$. By definition of δ_{c} and the fact that δ inverts the earlier renaming from a to b, we have that $\delta_{\mathrm{c}}(b) = \theta'(\delta(b)) = \theta'(a)$.

Equipped with these properties, we may finally prove $\delta_{\mathrm{c}}(\theta_{\mathrm{g}}(A)) = A_{\mathrm{c}}$ as follows.

$$
\begin{aligned}
& \delta_c\theta_g(A)) \\
& \qquad\qquad\qquad\qquad \text{(by (5) and (8))} \\
= \;& \delta_c\theta'\theta_i(A) \\
& \qquad\qquad\qquad\qquad \text{(by (10)} \\
= \;& \delta_c\theta'\theta_i(\delta^{-1}\theta_i^{-1}(A_i)) \\
& \qquad\qquad\qquad\qquad \text{(by } \Delta_A \# \Delta_g \text{ and } \Delta''_p \# \Delta_r \text{ we have that} \\
& \qquad\qquad\qquad\qquad\; \theta_i \text{ and } \delta^{-1} \text{ may be applied in any order)} \\
= \;& \delta_c\theta'\delta^{-1}\theta_i\theta_i^{-1}(A_i) \\
& \qquad\qquad\qquad\qquad \text{(by (11) and } \theta_i \circ \theta_i^{-1} = \iota_{\Delta_r}) \\
= \;& \delta_c\theta'\delta^{-1}(A_i) \\
& \qquad\qquad\qquad\qquad \text{(by (12))} \\
= \;& \theta'(A_i) \\
& \qquad\qquad\qquad\qquad \text{(by (6))} \\
= \;& A_c
\end{aligned}
$$

□

Lemma 4.4 (Stability of typing under substitution; *lemma originally stated on page 77)*.
If $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta;\Gamma \vdash M$ **ok**, *then* $(\Delta^\bullet,\Theta);\Gamma \vdash^E M : A$ *implies* $(\Delta^\bullet,\Theta');\theta(\Gamma) \vdash^E M : \theta(A)$.

Proof. By structural induction on the term under consideration. All cases are straightforward, except plain let bindings where Lemma 4.5 is required to show that *principal* types are also stable under substitution. □

A.4 Correctness of type inference algorithm

We may now prove the main correctness theorems of our solver, namely Theorems 4.5 and 4.6.

First, we prove the following lemma, which is a more general version of Theorem 4.7 on page 92. As opposed to the theorem, Lemma A.13 shown here supports the case where infer may be invoked with a non-empty restriction context Θ. We then immediately obtain the statement of Theorem 4.7 by picking $\Theta = \cdot$.

Lemma A.13 (Principality of inferred types)**.**
Let $(\Delta^{\bullet}, \Theta) \vdash \Gamma\ \mathbf{ok}$ *and* $\Delta; \Gamma \vdash M\ \mathbf{ok}$ *and* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ *hold. Further, let* $\Delta' = \mathsf{ftv}(\Theta') - \mathsf{ftv}(\theta)$ *and* $\Delta'' = \mathsf{ftv}(\Theta') - \Delta'$*. Then the following holds.*

1. *We have* $\mathsf{principal}((\Delta, \Delta''), \theta(\Gamma), M, \Delta', A)$.

2. *For all* $\overline{a}$ *and* A' *we have* $(\Delta, \Delta'', \overline{a}), \theta(\Gamma) \vdash M : A'$ *iff there exists* θ' *such that* $(\Delta, \Delta'') \vdash \theta' : \Theta'' \Rightarrow \overline{a}^{\bullet}$ *and* $\theta'(A) = A'$*, where* $\Theta'' \subseteq \Theta'$ *and* $\mathsf{ftv}(\Theta'') = \Delta'$*.*

3. *We have* $\Delta'' \approx \mathsf{ftv}(\theta) - \Delta$*.*

Proof. We first observe that the second statement of the lemma implies the first one: According to the left-to-right direction of the second statement, any type A' of M in the shown contexts can be obtained from A using a substitution θ' whose domain is Δ'. Thus, we may use θ' as the *instantiation* δ that is required by the definition of $\mathsf{principal}((\Delta, \Delta''), \theta(\Gamma), M, \Delta', A)$, since $(\Delta, \Delta'') \vdash \theta' : \Theta'' \Rightarrow \overline{a}$ implies $(\Delta, \Delta'') \vdash \delta : \Delta' \Rightarrow_{\star} \overline{a}$. The only aspect remaining in order to prove $\mathsf{principal}((\Delta, \Delta''), \theta(\Gamma), M, \Delta', A)$ is that $(\Delta, \Delta'', \Delta'); \theta(\Gamma) \vdash M : A$ holds. This follows from the right-to-left direction of the second statement of the lemma: We may choose $\overline{a} = \Delta'$ and $A' = A$, in which case θ' is simply the identity.

By Theorem 4.5, we have $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ **(1)**, which implies the third statement of the lemma.

We now focus on proving the second statement of the lemma, which simply amounts to applying the soundness and completeness theorems of the inference algorithm.

- $\Rightarrow$: We may simultaneously strengthen and weaken (1) to obtain $\Delta \vdash \theta : \Theta \Rightarrow \Delta''^{\bullet}, \overline{a}^{\bullet}$ **(2)**. Lemma 4.2 allows us to state the typing judgement assumed for A' in this direction of the proof as $(\Delta^{\bullet}, \Delta''^{\bullet}, \overline{a}^{\bullet}); \theta(\Gamma) \vdash^{\mathrm{E}} M : A'$ (i.e., using the extended typing relation).

  We may then apply Theorem 4.6 to this typing judgement and (2). Since inference results are unique up to renaming (see Lemma A.11), we can assume w.l.o.g. that the result of infer mentioned in Theorem 4.6 is identical to (Θ', θ, A). The theorem then states that there exists θ'' such that the following conditions hold.

$$\Delta \vdash \theta'' : \Theta' \Rightarrow (\Delta''^{\bullet}, \overline{a}^{\bullet})$$
$$\theta = \theta'' \circ \theta$$
$$\theta''(A) = A'$$

By $\theta = \theta'' \circ \theta$, we have that the restriction of θ'' to Δ'' is the identity substitution. We may therefore define θ' as the restriction of θ'' to those variables in A but *not* in Δ'' (i.e., the domain of θ' is Δ') while preserving $\theta''(A) = A' = \theta'(A)$. This definition of θ' also gives us $(\Delta, \Delta'') \vdash \theta' : \Theta'' \Rightarrow \overline{a}^{\bullet}$, where Θ'' is like Θ', but contains only entries for the variables in Δ'.

- $\Leftarrow$: By Theorem 4.5 we have $(\Delta^{\bullet}, \Theta'); \theta(\Gamma) \vdash^{\mathrm{E}} M : A$ **(3)**. We extend θ' to θ'' by adding mappings $a \mapsto a$ for all $a \in \Delta''$. Due to $\mathsf{ftv}(\Theta') \approx (\Delta', \Delta'')$, this yields $\Delta \vdash \theta'' : \Theta' \Rightarrow (\Delta''^{\bullet}, \overline{a}^{\bullet})$. By Lemma 4.4, applying θ'' to (3) yields $(\Delta^{\bullet}, \Delta''^{\bullet}, \overline{a}^{\bullet}); \theta''(\theta(\Gamma)) \vdash^{\mathrm{E}} M : \theta''(A)$. By $\Delta' \# \mathsf{ftv}(\theta(\Gamma))$ as well as assumption $\theta'(A) = A'$, this is equivalent to $(\Delta^{\bullet}, \Delta''^{\bullet}, \overline{a}^{\bullet}); \theta(\Gamma) \vdash^{\mathrm{E}} M : A'$. Applying Lemma 4.2 then yields the desired judgement $(\Delta, \Delta'', \overline{a}); \theta(\Gamma) \vdash M : A'$.

□

Theorem 4.5 (Soundness of infer; *theorem originally stated on page 91)*.
Let $(\Delta^{\bullet}, \Theta) \vdash \Gamma\ \mathbf{ok}$ *and* $\Delta; \Gamma \vdash M\ \mathbf{ok}$. *If* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ *then* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $(\Delta^{\bullet}, \Theta'); \theta(\Gamma) \vdash^{E} M : A$.

Proof. In the following, to avoid name clashes with other meta-variables, let M_0 and A_0 denote the term under consideration and its inferred type, respectively. The proof is then by induction on M_0. In other words, we assume $(\Delta^{\bullet}, \Theta) \vdash \Gamma\ \mathbf{ok}$ **(1)** and $\Delta; \Gamma \vdash M_0\ \mathbf{ok}$ **(2)** and as well as $\mathsf{infer}(\Delta, \Theta, \Gamma, M_0) = (\Theta', \theta, A_0)$ **(3)** and show the following.

$$\Delta \vdash \theta : \Theta \Rightarrow \Theta' \tag{I}$$

$$(\Delta^{\bullet}, \Theta'); \theta(\Gamma) \vdash^{\mathrm{E}} M_0 : A_0 \tag{II}$$

In each case, we directly use the meta-variables that the algorithm introduced (e.g., in the case for $\lceil x \rceil$ we assume that A is defined as $\Gamma(x)$). We provide the largest amount of detail for the let cases.

- Cases x and $\lambda\lceil x \rceil$:

  Straightforward.

- Case $\lambda x.M$ and $\lambda(x : A).M$

  Can easily be shown by applying the induction hypothesis to the recursive call to infer.

In the un-annotated case, the fact that infer succeeded on M_0 implies that deconstructing the substitution returned by the recursive call into $\theta[a \mapsto S]$ succeeded, meaning that a is indeed substituted with a monomorphic type, allowing us to construct a typing derivation the an un-annotated function.

In the annotated case, we rely on the fact that the inferred substitution θ has no effect on A, due to (2), which implies $\Delta \vdash A$. This yields $\theta(\Gamma, x : A) = (\theta(\Gamma), x : A)$

- Case $M\,N$:

  By induction, we have the following:

  $$(\Delta^\bullet, \Theta_1); \theta_1(\Gamma) \vdash^{\mathrm{E}} M : A' \qquad \textbf{(4)}$$

  $$(\Delta^\bullet, \Theta_2); \theta_2(\Gamma) \vdash^{\mathrm{E}} N : A \qquad \textbf{(5)}$$

  Let θ'_3 denote the substitution obtained from $\mathcal{U}$, which implies $\theta'_3 = \theta_3[b \mapsto B]$.

  By soundness of unification (Theorem 4.2), as well as the earlier applications of the induction hypothesis, we have the following

  $$\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1 \qquad \Delta \vdash \theta_2 : \Theta_1 \Rightarrow \Theta_2 \qquad \Delta \vdash \theta'_3 : (\Theta_2, b : \star) \Rightarrow \Theta_3$$

  which implies $\Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$.

  The soundness theorem also gives us $\theta'_3(\theta_2((A'))) = \theta'_3(A \to b)$, which is equivalent to $\theta_3(\theta_2((A')) = \theta_3(A) \to B$,

  We may then apply Lemma 4.4 to (4) and (5), using substitutions $\theta_3 \circ \theta_2$ and θ_3, respectively. Together with $\theta_3(\theta_2((A')) = \theta_3(A) \to B$, this allows us to derive $(\Delta^\bullet, \Theta_3); \theta_3\theta_2\theta_1(\Gamma) \vdash^{\mathrm{E}} M\,N : B$.

- Case **let** $x = M$ **in** N:

  By definition of infer, we have $(\Theta_1, \theta_1, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$ **(6)**. By induction, this implies $(\Delta^\bullet, \Theta_1); \theta_1(\Gamma) \vdash^{\mathrm{E}} M : A$ **(7)** and $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(8)**. By Lemma A.7, the former implies $\mathsf{ftv}(A) \subseteq \Delta, \Theta_1$.

  By definition of infer we further have

  $$\begin{aligned} \Delta' &= \mathsf{ftv}(\theta_1) - \Delta \\ \Delta'' &= \mathsf{ftv}(A) - \Delta, \Delta' \\ &= (\mathsf{ftv}(A) - \Delta) - \mathsf{ftv}(\theta_1) \end{aligned} \qquad \textbf{(9)}$$

Together with (8), these definitions yield $\mathsf{ftv}(A) \subseteq \Delta, \Delta', \Delta'' \subseteq \Delta, \mathsf{ftv}(\Theta_1)$ **(10)**. By applying Lemma A.13 to (6), we obtain $\mathsf{principal}((\Delta, \Delta'), \theta_1(\Gamma), \mathsf{ftv}(\Theta_1) -$

$\mathsf{ftv}(\theta_1),A)$ which we may simultaneously weaken (in the first component) and strengthen (in the fourth component) to $\mathsf{principal}((\Delta,\Theta_1-\Delta''),\theta_1(\Gamma),\Delta'',A)$ **(11)**.[2]

Next, infer defines $\Theta_1' = \mathsf{demote}(\bullet,\Theta_1,\Delta'')$ and $(\Theta_2,\theta_2,B) = \mathsf{infer}(\Delta,\Theta_1'-\Delta''',$ $(\theta_1(\Gamma),x:\forall\Delta'''.A),N)$ **(12)**.

By definition of Δ'', we have $\Delta''\#\mathsf{ftv}(\theta_1)$ and thus $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1 - \Delta''$ **(13)**.

Let Δ'' have the form $(a_1,\dots,a_n)$. We then define Δ''_G as $(b_1,\dots,b_n)$, for fresh b_i, meaning that in particular we have $\Delta''_G\#\Theta_2$ **(14)**. We define θ_2' as follows, which implies $\theta_2\restriction_{\Delta'} = \theta_2'\restriction_{\Delta'}$**(15)**, and thus $\theta_2'\circ\theta_1 = \theta_2\circ\theta_1$ **(16)**.

$$\theta_2'(c) = \begin{cases} b_i & \text{if } c = a_i \in \Delta'' \\ \theta_2(c) & \text{if } c \in \Theta_1 - \Delta'' \end{cases}$$

We distinguish between M being a guarded value or not. In each case, we show that there exists A' such that the following conditions are satisfied:

$$\Delta \vdash \theta_2'\circ\theta_1 : \Theta \Rightarrow \Theta_2 \quad \textbf{(17)}$$
$$\Delta''_G = \mathsf{ftv}(\theta_2'(A)) - \Delta,\mathsf{ftv}(\Theta_2) \quad \textbf{(18)}$$
$$(\Delta^\bullet,\Theta_2,\Delta''_G{}^\bullet);\theta_2'(\theta_1(\Gamma)) \vdash^{\mathrm{E}} M : \theta_2'(A) \quad \textbf{(19)}$$
$$\big((\Delta,\mathsf{ftv}(\Theta_2)),\Delta''_G,M,\theta_2'(A)\big) \Updownarrow A' \quad \textbf{(20)}$$
$$(\Delta^\bullet,\Theta_2);\big(\theta_2'(\theta_1(\Gamma)),x:A'\big) \vdash^{\mathrm{E}} N : B \quad \textbf{(21)}$$
$$\mathsf{principal}\big((\Delta,\mathsf{ftv}(\Theta_2)),\theta_2'(\theta_1(\Gamma)),M,\Delta''_G,\theta_2'(A)\big) \quad \textbf{(22)}$$

– Sub-case $M \in \mathsf{GVal}$: By definition of infer, we have $\Delta''' = \Delta''$.

  In order to apply the induction hypothesis to (12), we need to show $\Delta,\Theta_1' - \Delta''' \vdash \theta_1(\Gamma),x:\forall\Delta''.A$ **ok**. First, by (1) and (13) and the stability of well-formedness of term contexts under substitution (see Lemma A.4), we have $\Delta,\Theta_1-\Delta''' \vdash \theta_1(\Gamma)$ **ok**.

  By (10) we have $(\Delta^\bullet,\Theta_1) \vdash A$ **ok** and thus $(\Delta^\bullet,\Theta_1-\Delta'') \vdash \forall\Delta''.A$ **ok**. It remains to show that for all $a \in \mathsf{ftv}(A)-\Delta''$ we have $(\Delta,\Theta_1) \vdash_\bullet a$ **ok**. For $a \in \Delta$, this follows immediately. Otherwise, by applying Lemma A.8 to

[2]Applying Lemma A.13 here to obtain the principality judgement required to derive the well-typedness of the overall term M_0 later on somewhat obfuscates the unavoidable dependence between our main correctness theorems for infer (namely, Theorems 4.5 and 4.6): The proof of Lemma A.13 crucially relies on Theorem 4.6 (i.e., the fact that inference is complete and yields most general types). As discussed in Appendix A.5, these dependencies are well founded as they are via subterms.

(11), we obtain $a \in \mathsf{ftv}(\theta_1(\Gamma))$, meaning that there exists $b \in \Gamma$ such that $a \in \mathsf{ftv}(\theta_1(b))$. By $(\Delta^\bullet, \Theta) \vdash \Gamma\ \mathbf{ok}$ we have $(\Delta^\bullet, \Theta) \vdash_\bullet b\ \mathbf{ok}$, which implies $(\Delta^\bullet, \Theta_1) \vdash_\bullet \theta_1(b)\ \mathbf{ok}$, which further implies $(\Delta^\bullet, \Theta_1) \vdash_\bullet a\ \mathbf{ok}$. By $\Theta_1 - \Delta''' = \Theta_1' - \Delta'''$ we then have $\Delta, \Theta_1' - \Delta''' \vdash \theta_1(\Gamma), x : \forall\Delta'''.A\ \mathbf{ok}$

In summary, we can apply the induction hypothesis by which we then have $(\Delta^\bullet, \Theta_2); \theta_2(\theta_1(\Gamma), x : \forall\Delta'''.A) \vdash^{\mathrm{E}} N : B$ **(23)** and $\Delta \vdash \theta_2 : \Theta_1 - \Delta'' \Rightarrow \Theta_2$ **(24)**, the latter of which implies $\Delta \vdash \theta_2' : \Theta_1 \Rightarrow \Theta_2, \Delta_G''^{\bullet}$ **(25)** due to our definition of θ_2'.

We choose $A' = \forall\Delta_G''.\theta_2'(A)$, therefore satisfying (20). By (13), (16) and (24), condition (17) is also satisfied.

Due to our choice of θ_2' we have $\theta_2'(\theta_1(\Gamma) = \theta_2(\theta_1(\Gamma))$ and by (14) and (24) also $\theta_2(\forall\Delta'''.A) = \theta_2(\forall\Delta_G''.A[\Delta_G''/\Delta'']) = \forall\Delta_G''.\theta_2'(A)$. Together, this means that (23) is equivalent to (21).

By Lemma 3.1, we may convert (11) to $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta_1) - \Delta''), \theta_1(\Gamma), \Delta_G'', A[\Delta_G''/\Delta''])$. By applying Lemma 4.5 to the latter principality judgement as well as (14) and (24) we obtain, $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta_2)), \theta_2(\theta_1(\Gamma)), \Delta_G'', \theta_2(A [\Delta_G''/\Delta'']))$, which is equivalent to (22).

We now show (18): Recall that $\mathsf{ftv}(A) \subseteq \Delta, \mathsf{ftv}(\Theta_1)$ holds. Thus, together with (25), we then obtain $\mathsf{ftv}(\theta_2'(A)) \subseteq \Delta, \mathsf{ftv}(\Theta_2), \Delta_G''$. Further, by definition of Δ'' we have $\Delta'' \subseteq \mathsf{ftv}(A)$. Together with $\theta_2'(\Delta'') = \Delta_G''$, this results in the desired equation $\Delta_G'' = \mathsf{ftv}(\theta_2'(A)) - \Delta, \mathsf{ftv}(\Theta_2)$ (i.e., (18)).

By applying Lemma 4.4 to (7) and (25), we obtain (19).

- Sub-case $M \notin \mathsf{GVal}$: By definition of infer, we have $\Delta''' = \cdot$.

  We show that the induction hypothesis is applicable to (12). To this end, we show $(\Delta^\bullet, \Theta_1' - \Delta''') \vdash \theta_1(\Gamma), x : \forall\Delta'''.A\ \mathbf{ok}$. We have $(\Delta^\bullet, \Theta_1) \vdash \theta_1(\Gamma)\ \mathbf{ok}$ and and $\mathsf{ftv}(A) \subseteq \Delta, \mathsf{ftv}(\Theta_1)$. It remains to show that for all $a \in \mathsf{ftv}(A)$ we have $(a : \bullet) \in (\Delta^\bullet, \Theta_1')$. If $a \in \Delta''$, then by definition of Θ_1' as resulting from Θ_1 by demoting all variables in Δ'' we have $(a : \bullet) \in \Theta_1'$. Otherwise, we may use the same reasoning as in the case $M \in \mathsf{GVal}$.

  By induction, we then have $(\Delta^\bullet, \Theta_2); \theta_2(\theta_1(\Gamma), x : A) \vdash^{\mathrm{E}} N : B$ **(26)** and $\Delta \vdash \theta_2 : \Theta_1' \Rightarrow \Theta_2$ **(27)**.

  This, together with the definition of θ_2' then implies that we have $\Delta \vdash \theta_2' : \Theta_1 \Rightarrow (\Theta_2, \Delta_G''^{\bullet})$ **(28)** and due to (15) also satisfaction of (17).

  We may then show that (18), (19) and (22) hold as in the case $M \in \mathsf{GVal}$.

Recall that $\theta_2'(\Delta'') = \Delta_G''$. Let δ be a substitution with domain Δ_G'' such that for all $b_i \in \Delta_G''$ we have $\delta(b_i) = \theta_2(a_i)$, where a_i is the corresponding variable from Δ'' (i.e., $\theta_2'(a_i) = b_i$). All variables from Δ'' are monomorphic in Θ_1' (as these are precisely the variables that were subject to demotion), meaning that by (27) we have $(\Delta^\bullet, \Theta_2) \vdash_\bullet \theta_2(a_i)$ **ok** for all such a_i. As a result, we have $(\Delta, \mathsf{ftv}(\Theta)) \vdash \delta : \Delta_G'' \Rightarrow_\bullet \cdot$ and $\delta(\theta_2'(A)) = \theta_2(A)$. We define A' to be this type, yielding satisfaction of (20).

Together with (17) this means that (26) is equivalent to the desired typing judgement (21) for N.

We have thus shown that (16) to (22) hold in each case.

We can now derive the following, showing (II), observing that each premise directly corresponds to one of (18) to (22).

$$\frac{\begin{array}{c}\Delta_G'' = \mathsf{ftv}(\theta_2'(A)) - \Delta, \mathsf{ftv}(\Theta_2) \\ (\Delta^\bullet, \Theta_2, \Delta_G''^{\bullet}); \theta_2'(\theta_1(\Gamma)) \vdash^{\mathrm{E}} M : \theta_2'(A) \\ \big((\Delta, \mathsf{ftv}(\Theta_2)), \Delta_G'', M, \theta_2'(A)\big) \Updownarrow A' \\ (\Delta^\bullet, \Theta_2); (\theta_2'(\theta_1(\Gamma)), x : A') \vdash^{\mathrm{E}} N : B \\ \mathsf{principal}\big((\Delta, \mathsf{ftv}(\Theta_2)), \theta_2'(\theta_1(\Gamma)), M, \Delta_G'', \theta_2'(A)\big)\end{array}}{(\Delta^\bullet, \Theta_2); \theta_2'(\theta_1(\Gamma)) \vdash^{\mathrm{E}} \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

The implicit precondition $(\Delta^\bullet, \Theta_2) \vdash \theta_2'(\theta_1(\Gamma))$ **ok** of this judgement follows from (19), which has a similar precondition that we may strengthen appropriately due to $\Delta_G'' \# \theta_2'(\theta_1(\Gamma))$.

By (16) and (17) we have also shown (I).

- Case **let** $(x : A) = M$ **in** N:

  Let $A = \forall \Delta''.H$ for appropriate Δ'' and H. According to (2), we have $\Delta \vdash A$ **(29)**.

  We distinguish two cases, between whether M is a guarded value or not. We show that the following conditions hold for the choice of A' and Δ' imposed by $(\Delta', A') = \mathsf{split}(A, M)$ **(30)** in each case.

$$(\Delta, \Delta') \vdash A' \qquad (31)$$

$$\mathsf{ftv}(A) \# \Delta' \qquad (32)$$

$$\Delta' \# \Theta \qquad (33)$$

- Sub-case $M \in \mathsf{GVal}$: We have $\mathsf{split}(A,M) = (\Delta'',H)$ (i.e, $\Delta' = \Delta''$ and $A' = H$).

  Together with (29) we have $(\Delta,\Delta') \vdash H$ (satisfying (31)). The assertion $\Delta' \# \Delta, \Theta$ guarantees (33). By $A = \forall \Delta'.A'$ we further have $\mathsf{ftv}(A) \# \Delta'$.

- Sub-case $M \notin \mathsf{GVal}$: We have $\mathsf{split}(A,M) = (\cdot,A)$ (i.e, $\Delta' = \cdot$ and $A' = A$). This immediately satisfies (32) and (33). It further makes (29) equivalent to (31).

Moreover, by (2), we have $\Delta,\Delta' \vdash M$ **ok** **(34)** using inversion.

We show that $(\Delta^\bullet,\Theta_1,\Delta'^\bullet);\theta_1(\Gamma) \vdash^{\mathrm{E}} M : A_1$ **(35)** holds. By (1) and since $\Delta' \# \Theta$, we have $(\Delta^\bullet,\Delta'^\bullet,\Theta) \vdash \Gamma$ **ok**. Together with (34), we then have $(\Delta^\bullet,\Delta'^\bullet,\Theta_1);\theta_1(\Gamma) \vdash^{\mathrm{E}} M : A_1$ and $(\Delta,\Delta') \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(36)** by induction. Further, this implies $\Delta' \# \Theta_1$.

By (35) we also have $(\Delta^\bullet,\Delta'^\bullet,\Theta_1) \vdash A_1$ **ok**. Recall $(\Delta,\Delta') \vdash A'$ **ok** and therefore $(\Delta^\bullet,\Delta'^\bullet,\Theta_1) \vdash A'$ **ok**. Thus, by Theorem 4.2, we have $\theta'_2(A_1) = \theta'_2(A')$ **(37)** and $(\Delta,\Delta') \vdash \theta'_2 : \Theta_1 \Rightarrow \Theta_2$ **(38)**.

According to the assertion, we have $\mathsf{ftv}(\theta'_2 \circ \theta_1) \# \Delta'$ **(39)**. By definition of infer, we have $\theta_2 = \theta'_2 \circ \theta_1$ **(40)**, yielding $(\Delta,\Delta') : \theta_2 : \Theta \Rightarrow \Theta_2$, which further implies $\Delta' \# \Theta_2$ **(41)**. By (39), we can strengthen the previous well-formedness judgement to $\Delta \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(42)**.

By (34), (35) and (38), and Lemma 4.4, we have $(\Delta^\bullet,\Delta'^\bullet,\Theta_2);\theta'_2(\theta_1(\Gamma)) \vdash^{\mathrm{E}} M : \theta'_2(A_1)$. By (36), (38) and (40), this is equivalent to $(\Delta^\bullet,\Delta'^\bullet,\Theta_2);\theta_2(\Gamma) \vdash^{\mathrm{E}} M : \theta'_2(A_1)$ **(43)**.

By (31) and (38) we have $\theta'_2(A') = A'$. Together with (37), this makes (43) equivalent to $(\Delta^\bullet,\Delta'^\bullet,\Theta_2);\theta_2(\Gamma) \vdash^{\mathrm{E}} M : A'$ **(44)**.

By definition of infer, we have $(\Theta_3,\theta_3,B) = \mathsf{infer}(\Delta,\Theta_2,(\theta_2\Gamma,x:A),N)$. Due to (2), we have $\Delta \vdash N$ **ok**. By (29) and $\Theta_2 \# \Delta$, we have $(\Delta^\bullet,\Theta_2) \vdash x : A$ **ok**. Together with (1) and (42) we then have $\Delta^\bullet,\Theta_2 \vdash (\theta_2(\Gamma),x:A)$ **ok**. Therefore, by induction, we have $(\Delta^\bullet,\Theta_3);\theta_3(\theta_2(\Gamma),x:A) \vdash^{\mathrm{E}} N : B$ **(45)** and $\Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$ **(46)**.

By (42), (46), and composition, we have $\Delta \vdash \theta_3 \circ \theta_2 : \Theta \Rightarrow \Theta_3$ (I).

We now perform freshening of Δ' due to the fact that in general, we may not have

$\Delta' \# \Theta_3$.[3] Let $\overline{a} = \Delta'$ and $\delta = [\overline{a} \mapsto \overline{c}]$ for fresh $\overline{c}$. We now define Δ''' to be empty if $M \notin \mathsf{GVal}$ and otherwise we define it to be $\overline{c}$. Further, let A''' be identical to A if $M \notin \mathsf{GVal}$ and otherwise we define it to be the result of alpha-converting the toplevel quantifiers Δ' of A to Δ'''. Together, these definitions yield the following. Observe that in the case $M \notin \mathsf{GVal}$, applying δ to A' simply has no effect.

$$(\Delta''', \delta(A')) = \mathsf{split}(A''', M) \tag{47}$$

$$A''' = \forall \Delta'''.\delta(A') \tag{48}$$

We may now apply Lemma A.6 to (44), yielding $(\Delta^\bullet, \Delta'''^\bullet, \Theta_2); \delta(\theta_2(\Gamma)) \vdash^{\mathrm{E}} M[\overline{c}/\overline{a}] : \delta(A')$. By $\Delta' \# \Delta, \Theta_2$, this is equivalent to $(\Delta^\bullet, \Delta'''^\bullet, \Theta_2); \theta_2(\Gamma) \vdash^{\mathrm{E}} M[\overline{c}/\overline{a}] : \delta(A')$

We apply Lemma 4.4 again, this time to the previous typing judgement and (46), yielding $(\Delta^\bullet, \Theta_3, \Delta'''^\bullet); \theta_3(\theta_2(\Gamma)) \vdash^{\mathrm{E}} M[\overline{c}/\overline{a}] : \theta_3(\delta(A'))$.

We have $\mathsf{ftv}(\delta(A')) \subseteq \Delta, \overline{c}$, meaning that the previous judgement is equivalent to $(\Delta^\bullet, \Theta_3, \Delta'''^\bullet); \theta_3(\theta_2(\Gamma)) \vdash^{\mathrm{E}} M[\overline{c}/\overline{a}] : \delta(A')$ **(49)**.

Together with (1), we obtain $(\Delta^\bullet, \Theta_3) \vdash \theta_3(\theta_2(\Gamma))$ **ok** and can derive the following:

$$\frac{\begin{array}{c}(\Delta''', \delta(A')) = \mathsf{split}(A''', M) \\ (\Delta^\bullet, \Theta_3, \Delta'''^\bullet); \theta_3(\theta_2(\Gamma))) \vdash^{\mathrm{E}} M[\overline{c}/\overline{a}] : \delta(A') \\ (\Delta^\bullet, \Theta_3); \theta_3(\theta_2(\Gamma)), x : A''' \vdash^{\mathrm{E}} N : B\end{array}}{(\Delta^\bullet, \Theta_3); \theta_3(\theta_2(\Gamma)) \vdash^{\mathrm{E}} \mathbf{let}\ (x : A''') = M[\overline{c}/\overline{a}]\ \mathbf{in}\ N : B}$$

Here, the first two premises follow from (47) and (49), respectively. The final premise follows from (45), observing that the types assigned to x in both judgements are alpha-equivalent. For the same reason, the term in the judgement we just derived is alpha-equivalent to M_0, meaning that we have shown (II).

□

Theorem 4.6 (Completeness and generality of infer; *theorem originally stated on page 91)*.
Let $\Delta; \Gamma \vdash M$ **ok** *and* $(\Delta^\bullet, \Theta) \vdash \Gamma$ **ok** *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$. *If* $(\Delta^\bullet, \Theta'); \theta(\Gamma) \vdash^{E} M : A$, *then there exist* $\Theta'', \theta', \theta'', A'$ *such that* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta'', \theta'', A')$ *and* $\Delta \vdash \theta' : \Theta'' \Rightarrow \Theta'$ *and* $\theta = \theta' \circ \theta''$ *and* $\theta'(A') = A$.

[3] In practice, this disjointness condition does of course hold: We have $\Delta' \# \Theta_2$, and all variables in Θ_3 not already in Θ_2 were chosen fresh by the algorithm.

Proof. In the following, to avoid name clashes with other meta-variables, we refer to the term under consideration as M_0, the inferred type as A'_0 and the substitution used to type M_0 as well as its type as θ_0 and A_0, respectively. As mentioned at the beginning of this section, the proof is then by induction on M_0. In other words, we assume that we have $\Delta;\Gamma \vdash M_0$ **ok (1)** and $(\Delta^{\bullet},\Theta) \vdash \Gamma$ **ok (2)** and $\Delta \vdash \theta_0 : \Theta \Rightarrow \Theta'$ **(3)** and $(\Delta^{\bullet},\Theta);\theta_0(\Gamma) \vdash^{\mathrm{E}} M_0 : A_0$ **(4)**. We then show the existence of A'_0, θ', θ'' and Θ'' with the following properties.

$$\mathsf{infer}(\Delta,\Theta,\Gamma,M_0) = (\Theta'',\theta'',A'_0) \tag{I}$$

$$\Delta \vdash \theta' : \Theta'' \Rightarrow \Theta' \tag{II}$$

$$\theta_0 = \theta' \circ \theta'' \tag{III}$$

$$\theta'(A'_0) = A_0 \tag{IV}$$

We provide the greatest amount of detail for the let cases.

- Case x:

  This case crucially relies on the premise (2), which implies that the type of x in Γ only contains *monomorphic* type variables.

  Let x have some type $\forall\Delta'.H'$ (for fresh Δ') in Γ and thus $\theta_0(\Gamma(x)) = \theta_0(\Delta'.H')$. However, the fact that all free variables of Γ are monomorphic in Θ implies that $\theta_0(\Delta'.H') = \forall\Delta'.\theta_0(H')$, where $\theta_0(H')$ is guaranteed to be a guarded type (i.e., the substitution preserves the quantifier structure). Thus, by (4) we have $A_0 = \delta(\theta_0(H'))$ for some instantiation δ with $\Delta, \mathsf{ftv}(\Theta') \vdash \Delta' \Rightarrow_{\star} \cdot$. The inference algorithm succeeds, choosing fresh $\overline{b} = (b_1,\ldots,b_n)$ for the quantifiers $\overline{a} = (a_1,\ldots,a_n) = \Delta'$ of $\Gamma(x)$. We may then define θ' to be a substitution with domain $(\mathsf{ftv}(\Theta),\overline{b})$ such that $\theta'(c) = \theta_0(c)$ for all $c \in \Theta$ and $\theta'(b_i) = \delta(a_i)$, yielding the desired properties.

- Case $\lceil x \rceil$:

  We immediately get the desired result by choosing $\theta' := \theta_0$.

- Case $\lambda x.M$:

  Straightforward by applying the induction hypothesis. Let θ_{i} be the substitution obtained from the recursive call. Theorem 4.5 guarantees that we have $\Delta \vdash \theta_{\mathrm{i}} : (\Theta, a : \bullet) \Rightarrow \Theta_1$.

  This guarantees that θ_{i} does indeed contain a mapping for a and it must be a monotype S (i.e., the deconstruction of θ_{i} in the algorithm succeeds). We may then choose θ' to be the substitution obtained by induction.

- Case $\lambda x.M$:

  Similar to the previous case. The only important difference is that by (1), we have that A only contains variables from Δ, which implies $\theta_0(\Gamma), x : A = \theta_0(\Gamma, x : A)$, which is required to apply the induction hypothesis.

- Case $M\,N$:

  By inversion on (4) we have $(\Delta^\bullet, \Theta'); \theta_0(\Gamma) \vdash^{\mathrm{E}} M : A_N \to A_0$ and $(\Delta^\bullet, \Theta'); \theta_0(\Gamma) \vdash^{\mathrm{E}} N : A_N$ for some type A_N.

  We apply the induction hypothesis to both recursive calls of infer.

  The first use of the induction hypothesis yields the existence of θ'_1 such that

  $$\Delta \vdash \theta'_1 \vdash \Theta_1 \Rightarrow \Theta' \qquad \textbf{(5)}$$
  $$\theta_0 = \theta'_1 \circ \theta_1$$
  $$\theta'_1(A') = A_{\mathrm{N}} \to A_0$$

  The second application of the induction hypothesis (using (5)) yields the existence of θ'_2 such that

  $$\Delta \vdash \theta'_2 \vdash \Theta_2 \Rightarrow \Theta'$$
  $$\theta'_1 = \theta'_2 \circ \theta_2$$
  $$\theta'_2(A) = A_{\mathrm{N}}$$

  Now, let θ_{b} the substitution with domain $\mathsf{ftv}(\Theta_2), b$ such that $\theta_{\mathrm{b}}(c) = \theta'_2(c)$ for all $c \in \Theta_2$ and $\theta_{\mathrm{b}}(b) = A_0$.

  This definition yields $\Delta \vdash \theta_{\mathrm{b}} : (\Theta_2, b : \star) \Rightarrow \Theta'$. We may then show that $\theta_{\mathrm{b}}(\theta_2(A')) = A_{\mathrm{N}} \to A_0 = \theta_{\mathrm{b}}(A \to b)$ holds.

  This allows us to apply Theorem 4.3 to obtain the existence of an instantiation θ' with the required properties.

- Case **let** $x = M$ **in** N: Inverting (4) gives us the existence of $\Delta_{\mathrm{p}}, A_{\mathrm{p}}, A_x$ such that the following holds.

  $$\Delta_{\mathrm{p}} = \mathsf{ftv}(A_{\mathrm{p}}) - \Delta \qquad \textbf{(6)}$$
  $$((\Delta, \mathsf{ftv}(\Theta')), \Delta_{\mathrm{p}}, M, A_{\mathrm{p}}) \Updownarrow A_x \qquad \textbf{(7)}$$
  $$(\Delta^\bullet, \Theta', \Delta_{\mathrm{p}}^\bullet); \theta_0(\Gamma) \vdash^{\mathrm{E}} M : A_{\mathrm{p}} \qquad \textbf{(8)}$$
  $$(\Delta^\bullet, \Theta'); (\theta_0(\Gamma), x : A_x) \vdash^{\mathrm{E}} N : A_0 \qquad \textbf{(9)}$$

$$\mathsf{principal}((\Delta,\mathsf{ftv}(\Theta')),\theta_0(\Gamma),M,\Delta_p,A_p) \qquad \textbf{(10)}$$

We may assume w.l.o.g. that Δ_p is fresh with respect to Θ and Θ_1 **(11)**. Otherwise, we may apply a renaming substitution to the variables in Δ_p, which of course preserves principality (Lemma 3.1), while the substitution property on typings (Lemma 4.4) ensures that the statements (7) to (9) still hold. Note that (10) already implies $\Delta_p \# \Delta, \mathsf{ftv}(\Theta')$.

We weaken (3) to $\Delta \vdash \theta_0 : \Theta \Rightarrow (\Theta', \Delta_p^\bullet)$ **(12)**. The well-typedness of M (see (8)) then allows us to apply the induction hypothesis, guaranteeing that the first recursive call of the inference algorithm succeeds, returning (Θ_1, θ_1, A). It further guarantees the existence of θ'_M such that we have the following.

$$\Delta \vdash \theta'_M : \Theta_1 \Rightarrow (\Theta', \Delta_p) \qquad \textbf{(13)}$$

$$\theta_0 = \theta'_M \circ \theta_1 \qquad \textbf{(14)}$$

$$A_p = \theta'_M(A) \qquad \textbf{(15)}$$

We now show that Δ_p, the generalisable type variables according to the typing derivation, and Δ'', the generalisable type variables as determined by the inference algorithm are identical, modulo a bijective renaming performed by θ'_M. Formally, we will show that $\theta'_M(\Delta'') = \Delta_p$ holds.

According to Lemma A.13, the type A inferred for M is principal. Concretely, we have $\mathsf{principal}((\Delta,\Delta_{ng}),\theta_1(\Gamma),M,\Delta_g,A)$ **(16)**, where $\Delta_g = \mathsf{ftv}(\Theta'') - \mathsf{ftv}(\theta_1)$ and $\Delta_{ng} = \mathsf{ftv}(\Theta'') - \Delta_g \approx \mathsf{ftv}(\theta_1) - \Delta$ **(17)**. We therefore have $\mathsf{principal}((\Delta,\Delta'),$ $\theta_1(\Gamma),M,\Delta_g,A)$ **(18)**, where Δ' is defined by the algorithm.

We now strengthen (18) by replacing Δ_g with the set of free variables appearing in A, minus those that are non-generalisable. To this end, we take $\Delta'' = \mathsf{ftv}(A) - \Delta,\Delta'$, as defined in the inference algorithm. As a result, we have that Δ'' contains the set of variables obtained from intersecting Δ_g and $\mathsf{ftv}(A)$, ordered by their first appearance within A. We then observe that for any $a \in \Delta_g$ such that $a \notin \Delta''$, we have $a \notin \Delta$ and $a \notin \mathsf{ftv}(A)$ (both by definition of Δ'') and $a \notin \mathsf{ftv}(\theta_1(\Gamma))$ (due to $a \notin \mathsf{ftv}(\theta_1)$ and (2), which implies $\mathsf{ftv}(\Gamma) \subseteq \Delta, \mathsf{ftv}(\Theta)$) and $a \notin \mathsf{ftv}(M)$ (due to $a \# \Delta$ and (1), which implies $\mathsf{ftv}(M) \subseteq \Delta$). Thus, we may indeed strengthen (18) to $\mathsf{principal}((\Delta,\Delta'),\theta_1(\Gamma),M,\Delta'',A)$ **(19)**.

Next, we define $\theta_r := \theta'_M{\restriction_{\Delta'}}$. Let Θ_r be the subset of Θ_1 containing exactly the variables in Δ'. By (13), this gives us $\Delta \vdash \theta_r : \Theta_r \Rightarrow (\Theta', \Delta_p)$. However, since the domain of Θ_r contains exactly those flexible variables in the codomain of θ_1, we can obtain $\theta_0 = \theta_r \circ \theta$ **(20)** from (14). The knowledge about the codomain of θ_0 (see (3)) then allows us to strengthen the earlier well formedness judgement regarding θ_r to $\Delta \vdash \theta_r : \Theta_r \Rightarrow \Theta'$.

Altogether, this allows us to apply Lemma 4.5 to (19), yielding $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta')), \theta_r(\theta_1(\Gamma)), M, \Delta'', \theta_r(A))$ which by (20) is equivalent to $\mathsf{principal}((\Delta, \mathsf{ftv}(\Theta')), \theta_0(\Gamma), M, \Delta'', \theta_r(A))$.

The uniqueness of principal types now allows us to relate A_p (a principal type of M, see (10)) and $\theta_r(A)$, stating that there must exist a bijection between Δ_p and Δ'', equating the two types. By (15) and the fact that θ_r is like θ'_M except acting like the identity on Δ'', we have that θ'_M is such a bijection when restricted to Δ''. Let n be the number of elements in Δ''. Since both Δ_p and Δ'' are ordered with respect to the first appearance of each variable in the respective type, we have that for all $i \in \{1, \dots, n\}$, θ'_M maps the i-th variable of Δ'' to the i-th variable in Δ_p). Thus, we have shown the desired property $\theta'_M(\Delta'') = \Delta_p$ **(21)**.

We observe that we have $\mathsf{ftv}(A) \subseteq \Delta, \Delta', \Delta''$ due to the definition of the latter two sequences by infer **(22)**.

We now distinguish two sub-cases, based on the shape of M', and show that in each case, there exists a substitution θ_c such that the following conditions hold.

$$\Delta \vdash \theta_c : (\Theta'_1 - \Delta''') \Rightarrow \Theta' \qquad \textbf{(23)}$$

$$(\Delta^\bullet, \Theta'); \theta_c\big(\theta_1(\Gamma), x : \forall \Delta'''.A\big) \vdash^{\mathrm{E}} N : A_0 \qquad \textbf{(24)}$$

$$\theta_0 = \theta_c \circ \theta_1 \qquad \textbf{(25)}$$

– First sub-case, $M \in \mathsf{GVal}$: By definition of infer and $\Updownarrow$ we have $\Delta'' = \Delta'''$ and $A_x = \forall \Delta_p.A_p$. Since $\mathsf{demote}(\bullet, \Theta_1, \Delta''')$ only affects variables in Δ''', we have $\Theta'_1 - \Delta''' = \Theta_1 - \Delta'''$ (i.e., all demoted variables are removed). We define θ_c such that it acts like θ'_M on the subset Δ' (i.e., the flexible variables in the co-domain of θ_1) of its domain. For all other b (i.e., $b \in \mathsf{ftv}(\Theta_1) - \Delta - \mathsf{ftv}(\theta_1)$), let $\theta_c(b)$ be an arbitrary, monomorphic ground type, such as Unit.[4]

[4]Note that the mappings for variables in $b \in \mathsf{ftv}(\Theta_1) - \Delta - \mathsf{ftv}(\theta_1)$ are irrelevant for anything other

By (3), (13) and (14), our choice of θ_c yields $\theta_0 = \theta_c \circ \theta_1$ and $\Delta \vdash \theta_c : (\Theta'_1 - \Delta''') \Rightarrow \Theta'$ and $\theta_0(\Gamma) = \theta_c(\theta_1(\Gamma))$. The first two properties are those we intend to show for our choice of θ_c. The latter property allows us to show that the earlier typing judgement for N (see (9)) is equivalent to the desired judgement $(\Delta^\bullet, \Theta'); \theta_c\big(\theta_1(\Gamma), x : \forall\Delta'''.A\big) \vdash^{\mathrm{E}} N : A_0$. All that remains to show is that both judgements assign the same type to x.

To this end, we prove $\theta_c(\forall\Delta'''.A) = A_x$ as follows.

$$\begin{array}{cll}
 & \theta_c(\forall\Delta'''.A) & \\
= & \theta_c(\forall\Delta''.A) & \\
= & \theta_c(\forall\Delta_p.A[\Delta_p/\Delta'']) & (\text{by } |\Delta''| = |\Delta_p| \text{ and } \Theta_1 \# \Delta_p \# \Theta', \text{see (11)}) \\
= & \forall\Delta_p.\theta_c(A[\Delta_p/\Delta'']) & (\text{by (21) and (22) and } \theta'_M\restriction_{\Delta'} = \theta_c\restriction_{\Delta'}) \\
= & \forall\Delta_p.(\theta'_M(A)) & (\text{by (15)}) \\
= & \forall\Delta_p.A_p & \\
= & A_x &
\end{array}$$

– Second sub-case, $M \in \mathsf{GVal}$: In this case we have $\Delta''' = \cdot$ and $A_x = \delta_x(A_p)$ for some instantiation δ_x with $\Delta, \mathsf{ftv}(\Theta') \vdash \delta_x : \Delta_p \Rightarrow_\bullet \cdot$ **(26)**. We define θ_c similarly to the previous sub-case. Its domain $\Theta'_1 - \Delta'''$ additionally contains the variables from Δ'' this time. For those variables $b \in \Delta''$, we define θ_c to be $\delta_x(\theta''(b))$, meaning that we first perform the renaming from Δ'' to Δ_p (see (21)), and then perform the same monomorphic instantiation as δ_x. Altogether, this yields the following definition for θ_c.

$$\theta_c(b) = \begin{cases} \theta'_M(b) & \text{if } b \in \Delta' \\ \mathsf{Unit} & \text{if } b \in \Theta_1 - \Delta', \Delta'' \\ \delta_x(\theta'_M(b)) & \text{if } b \in \Delta'' \end{cases}$$

Note that the first two cases yield the definition of θ_c in the previous sub-case and we obtain $\theta_0 = \theta_c \circ \theta_1$ using the same reasoning. To show satisfaction of $\Delta \vdash \theta_c : (\Theta'_1 - \Delta''') \Rightarrow \Theta'$, we observe that Θ_1 and Θ'_1 (which is identical to $\Theta'_1 - \Delta'''$) differ only in that all variables in Δ'' now have restriction $\bullet$. However, by (21) and (26) we have $\Delta \vdash (\delta_x \circ \theta'_M\restriction_{\Delta''}) : \Delta'' \Rightarrow_\bullet \mathsf{ftv}(\Theta')$, guaranteeing that θ_c maps all variables in Δ'' to monomorphic types.

than reasoning about the codomain of θ_c. Just re-using the mappings from θ'_M (i.e., $b \mapsto \theta'_M(b)$) for these variables b is not possible as per (13), $\theta'_M(b)$ may contain variables from Δ_p, which is not permitted in order for θ_c in order for $\Delta \vdash \theta_c : (\Theta'_1 - \Delta''') \Rightarrow \Theta'$ to hold.

As in the previous case, we show that $(\Delta^\bullet,\Theta');\theta_c\big(\theta_1(\Gamma),x:\forall\Delta'''.A\big)\vdash^{E} N: A_0$ holds due to (9) and $\theta_c(\forall\Delta'''.A)=A_x$, which we show as follows.

$$\begin{array}{rll} & \theta_c(\forall\Delta'''.A) & \\ = & \theta_c(A) & \text{(by (21) and (22) and the def. of } \theta_c) \\ = & \delta_x(\theta'_M(A)) & \text{(by (15))} \\ = & \delta_x(A_p) & \\ = & A_x & \end{array}$$

We have thus shown that in each sub-case, our choice of θ_c satisfies the conditions (23) to (25).

Applying the induction hypothesis to (23) and (24) then yields that the second recursive call to infer succeeds, returning (Θ_2,θ_2,B), and there exists θ'_N satisfying the following.

$$\Delta\vdash\theta'_N:\Theta_2\Rightarrow\Theta' \quad \textbf{(27)}$$

$$\theta_c=\theta'_N\circ\theta_2 \quad \textbf{(28)}$$

$$A_0=\theta'_N(B) \quad \textbf{(29)}$$

This also means that inference for the overall let binding succeeds.

Since infer returns $(\Theta_2,\theta_2\circ\theta_1,B)$, we have $\Theta''=\Theta_2$ and $\theta''=\theta_2\circ\theta$ and $A'_0=B$. We choose θ' to be θ'_N, means that by (27) and (29) we immediately have satisfaction of (II) and (IV), respectively.

By (25) and (28), we also have $\theta_0=\theta_c\circ\theta_1=(\theta'_N\circ\theta_2)\circ\theta_1=\theta'_N\circ(\theta_2\circ\theta_1)=\theta'\circ\theta''$, showing satisfaction of (III).

- Case **let** $(x:A)=M$ **in** N:

  By (4) and inverting LETANN, there exist Δ_G and A_M such that we have

  $$\Delta_G,A_M=\mathsf{split}(A,M) \quad \textbf{(30)}$$

  $$(\Delta^\bullet,\Theta',\Delta_G^\bullet);\theta_0(\Gamma)\vdash^{E} M:A_M \quad \textbf{(31)}$$

  $$A=\forall\Delta_G.A_M \quad \textbf{(32)}$$

  $$(\Delta^\bullet,\Theta');\big(\theta_0(\Gamma),(x:A)\big)\vdash^{E} N:A_0 \quad \textbf{(33)}$$

  By alpha-equivalence, we assume $\Delta_G\#\Theta$.

  Note that by definition of infer and split, we have $\Delta_G=\Delta'$ and $A'=A_M$ **(34)**. By (31), we have $\Delta'\#\Theta'$ **(35)**. We weaken (3) to $(\Delta,\Delta')\vdash\theta_0:\Theta\Rightarrow\Theta'$.

The assertion $\Delta' \# \Delta, \Theta$ is immediately satisfied in the case where $M \notin \mathsf{GVal}$. Otherwise, it is satisfied due the assumption $\Delta_G \# \Theta$ and the fact that judgements such as (31) further imply $\Delta' \# \Delta$.

By inversion on (1), we have $(\Delta, \Delta'); \Gamma \vdash M$ **ok** and $\Delta; (\Gamma, x : A) \vdash N$ **ok** and $\Delta \vdash A$ **ok** **(36)**, which implies $(\Delta, \Delta') \vdash A'$ **ok** **(37)**.

Together with (31) we then have the following by induction: $\mathsf{infer}((\Delta, \Delta'), \Theta, \Gamma, M)$ succeeds, returning $(\Theta_1, \theta_1, A_1)$ and there exists θ'_{M} such that

$$(\Delta, \Delta') \vdash \theta'_{\mathrm{M}} : \Theta_1 \Rightarrow \Theta' \qquad \textbf{(38)}$$

$$\theta_0 = \theta'_{\mathrm{M}} \circ \theta_1 \qquad \textbf{(39)}$$

$$\theta'_{\mathrm{M}}(A_1) = A_{\mathrm{M}} \qquad \textbf{(40)}$$

Theorem 4.5 yields $(\Delta, \Delta') \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(41)** and $(\Delta^\bullet, \Delta'^\bullet, \Theta_1); \theta_1(\Gamma) \vdash^{\mathrm{E}} M : A_1$, which implies $(\Delta^\bullet, \Delta'^\bullet, \Theta_1) \vdash A_1$ **(42)** (see Lemma A.7).

We then have

$$\begin{array}{rll} & \theta'_{\mathrm{M}}(A_1) & \\ = & A_{\mathrm{M}} & \text{(by (40))} \\ = & A' & \text{(by (34))} \\ = & \theta'_{\mathrm{M}}(A') & \text{(by (37) and (38))} \end{array}$$

In addition to this equality and (38) as well as (42), we have $\Delta^\bullet, \Delta'^\bullet, \Theta_1 \vdash A'$ by weakening (37). Hence, Theorem 4.3 yields the following: $\mathcal{U}((\Delta, \Delta'), \Theta_1, A', A_1)$ succeeds, returning (Θ_2, θ'_2), and there exists θ'_{U} such that

$$(\Delta, \Delta') \vdash \theta'_{\mathrm{U}} : \Theta_2 \Rightarrow \Theta' \qquad \textbf{(43)}$$

$$\theta'_{\mathrm{M}} = \theta'_{\mathrm{U}} \circ \theta'_2 \qquad \textbf{(44)}$$

By Theorem 4.2, we have $(\Delta, \Delta') \vdash \theta'_2 : \Theta_1 \Rightarrow \Theta_2$. Together with (41) and composition, we then have $(\Delta, \Delta') \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(45)**.

By (39) and (44), we have $\theta_0 = \theta'_{\mathrm{U}} \circ \theta'_2 \circ \theta_1 = \theta'_{\mathrm{U}} \circ \theta_2$ **(46)**. We show $\mathsf{ftv}(\theta_2) \subseteq \Delta, \Theta_2$: Otherwise, if $a \in \Theta$ and $b \in \Delta'$ such that $b \in \mathsf{ftv}(\theta_2(a))$, then by (43), $\theta'_{\mathrm{U}}(b) = b$ and $b \in \mathsf{ftv}(\theta'_{\mathrm{U}}(\theta_2(a))) = \mathsf{ftv}(\theta_0(a))$, violating (3).

Therefore, the assertion $\mathsf{ftv}(\theta_2) \# \Delta'$ succeeds, allowing us to strengthen (45) to $\Delta \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(47)**.

By (36) we have $\mathsf{ftv}(A) \subseteq \Delta$, and together with (43) this yields $\theta'_{\mathrm{U}}(A) = A$ **(48)**.

We then have the following, where we refer to the final judgement as **(49)**.

$$\begin{array}{lll} & \Delta, \Theta'; \theta_0(\Gamma), x : A \vdash^{\mathrm{E}} N : A_0 & \text{(by (33))} \\ \text{implies} & \Delta, \Theta'; \theta'_{\mathrm{U}}(\theta_2(\Gamma)), x : A \vdash^{\mathrm{E}} N : A_0 & \text{(by (43), (45) and (46))} \\ \text{implies} & \Delta, \Theta'; \theta'_{\mathrm{U}}(\theta_2(\Gamma), x : A) \vdash^{\mathrm{E}} N : A_0 & \text{(by (48))} \end{array}$$

By (2) and (47), we have $(\Delta, \Theta_2) \vdash \theta_2(\Gamma)$ **ok**. Together with (36), we then have $(\Delta, \Theta_2) \vdash \theta_2(\Gamma), x : A$ **ok**.

Hence, induction on (47) and (49) shows that $\mathsf{infer}(\Delta, \Theta_2, (\theta_2(\Gamma), x : A), N)$ succeeds, returning (Θ_3, θ_3, B) and there exists θ'_{N} such that

$$\Delta \vdash \theta'_{\mathrm{N}} : \Theta_3 \Rightarrow \Theta' \qquad \textbf{(50)}$$

$$\theta'_{\mathrm{U}} = \theta'_{\mathrm{N}} \circ \theta_3 \qquad \textbf{(51)}$$

$$\theta'_{\mathrm{N}}(B) = A_0 \qquad \textbf{(52)}$$

We have shown that all steps of the algorithm succeed. According to the return values of infer, we have $\Theta'' = \Theta_3$, $\theta'' = \theta_3 \circ \theta_2$, and $A'_0 = B$. Let $\theta' := \theta'_{\mathrm{N}}$. By (50), this choice immediately satisfies (II).

We show (III):

$$\begin{array}{lll} & \theta_0 & \\ = & \theta'_{\mathrm{U}} \circ \theta_2 & \text{(by (46))} \\ = & \theta'_{\mathrm{N}} \circ \theta_3 \circ \theta_2 & \text{(by (51))} \\ = & \theta' \circ \theta'' & (\text{by } \theta' := \theta'_{\mathrm{N}} \text{ and } \theta'' := \theta_3 \circ \theta_2\ ) \end{array}$$

By $\theta' = \theta'_{\mathrm{N}}$, (52) yields (IV).

□

A.5 Dependencies between proofs

As mentioned before, the correctness proofs of our inference algorithm exhibit some mutual dependencies. For example, the proof of Lemma 4.5 – the key property we use to prove stability of typing under substitution (Lemma 4.4) – relies on the completeness of inference (Theorem 4.6). Likewise, the soundness proof of our inference algorithm (Theorem 4.5) depends on the principality of inferred types (Lemma A.13), which in turn depends on the completeness theorem.

However, we now outline that the overall proof strategy is nevertheless well founded. Intuitively, this is due to the fact that for all groups of potentially circular dependencies it is always the case that some usages of other properties are through strict subterms.

Concretely, Figure A.1 on page 256 shows all lemmas and theorems defined in Chapters 3 and 4 as well as Appendix A that exhibit some non-trivial dependencies between each other. As subsequently explained, our goal is to detect potential cyclic dependencies between proofs. Therefore, we exclude properties whose proofs have no dependencies at all and some properties where it is clear that they cannot be part of a cycle in the graph, because they only rely on other properties without further dependencies. For instance, we need not consider any of the properties related to unification further.

We may interpret each node in the graph as a property (i.e., a lemma or theorem) $P[M]$ of terms M. An edge $P \longrightarrow Q$ between nodes P and Q in the graph then denotes that the proof of property $P[M]$ depends on property $Q[M]$. In other words, these edges denote when a proof directly uses another property on the *same* term M.

Figure A.1 does not include edges for cases where a property $P[M]$ relies on some $Q[M']$ where M' is a strict subterm of M. In particular, the proofs of Lemma 4.4 and Theorems 4.5 and 4.6 *only* exhibit such dependencies, and therefore the corresponding nodes in Figure A.1 have no outgoing edges.

We then observe that the graph in Figure A.1 contains no cycles. As a result, we observe that the properties in the graph can – and in fact have to – be proved using a single, mutual induction. Thus, the individual proofs of these properties shown earlier merely constitute the parts of this overall induction, rather than being standalone proofs.

A.6 Properties of principal types

The following lemma was stated in Section 3.2.1.3, but no proof was given as it relies on reasoning about the type inference function infer, which had not been introduced at that point.

Lemma 3.2 *(as originally stated on page 52).*
Let $\Delta \vdash \Gamma$ **ok** *and* $\Delta; \Gamma \vdash M$ **ok** *hold. If* $\mathsf{principal}(\Delta, \Gamma, U, \Delta', A)$*, then* A *has no toplevel quantifiers.*

Proof. The principality premise implies $(\Delta, \Delta'); \Gamma \vdash U : A$. According to Lemma 4.2, we then have $(\Delta^{\bullet}, \Delta'^{\bullet}); \Gamma \vdash^{\mathrm{E}} U : A$. By completeness of inference (Theorem 4.6) we thus have that $\mathsf{infer}(\Delta, \cdot, \Gamma, U)$ succeeds and returns some (Θ, θ, A'). Due to the principality of inferred types (Lemma A.13), we have that A' is a principal type of M in term context $\theta(\Gamma)$, where Lemma A.12 further states that θ merely renames type variables. Together with the uniqueness of inferred types (Lemma A.11) we then have that A and A' are the same modulo a renaming of type variables therein.

It is therefore sufficient to prove that for all $\Delta_{\mathrm{i}}, \Theta_{\mathrm{i}}, M_{\mathrm{i}}, U_{\mathrm{i}}$ we have that if $\mathsf{infer}(\Delta_{\mathrm{i}}, \Theta_{\mathrm{i}}, M_{\mathrm{i}}, U_{\mathrm{i}})$ returns some $(\Theta'_{\mathrm{i}}, \theta_{\mathrm{i}}, A_{\mathrm{i}})$, then A_{i} has no toplevel quantifiers. Proving this is a straightforward by induction on U.

For plain variables and lambda abstractions, this follows immediately from the shape of the type returned by infer. The only other possible forms of guarded values are (possibly annotated) let bindings. For both sorts of let bindings, the definition of guarded values guarantees that the second subterm is a guarded value, and the type returned for the overall binding is the type inferred for that subterm. Thus, the desired result follows immediately by induction. □

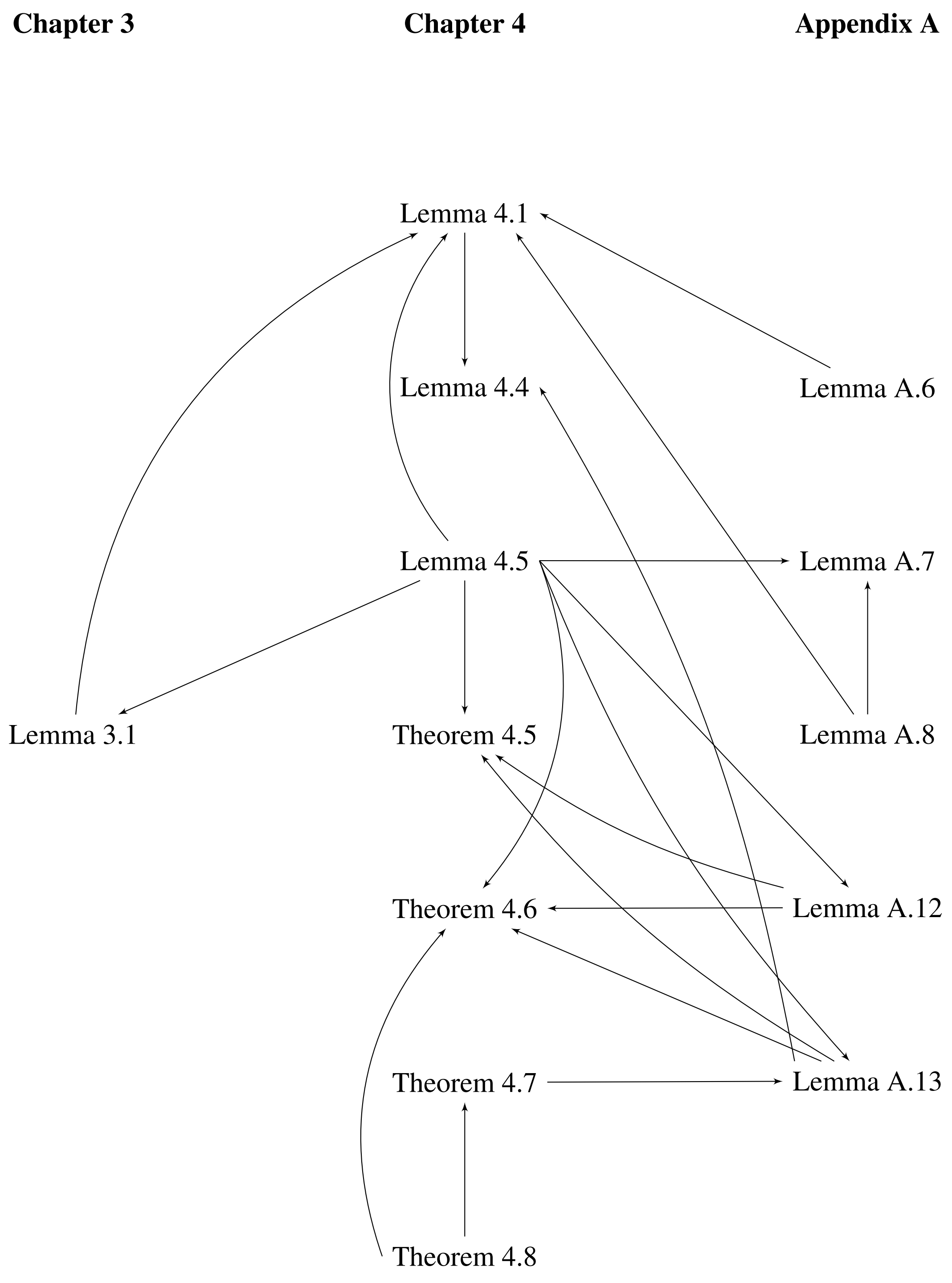

Figure A.1: Dependencies between properties in Chapters 3 and 4 as well as Appendix A

Appendix B

Proofs for Chapter 5

This appendix contains the remaining proofs for properties stated in Chapter 5.

B.1 Auxiliary properties

We provide four auxiliary properties that are used in subsequent proofs.

The following lemma states that we may weaken any $\mathsf{mostgen}$ judgement by adding additional flexible variables to Ξ that are mapped to pairwise different, fresh variables.

Lemma B.1 (Weakening $\mathsf{mostgen}$)**.**
Let $\mathsf{mostgen}(\Delta,\Xi,\Gamma,C,\Delta_{\mathrm{m}},\delta_{\mathrm{m}})$ *and* $(\Delta,\Xi,\Delta_{\mathrm{m}})\,\#(\overline{a},\overline{b})$*. Then we have* $\mathsf{mostgen}(\Delta,(\Xi,\overline{a}),$ $\Gamma,C,(\Delta_{\mathrm{m}},\overline{b}),\delta_{\mathrm{m}}[\overline{a}\mapsto\overline{b}])$.

Proof. By $\mathsf{mostgen}(\Delta,\Xi,\Gamma,C,\Delta_{\mathrm{m}},\delta_{\mathrm{m}})$ we have $(\Delta,\Delta_{\mathrm{m}});\Xi;\Gamma;\delta_{\mathrm{m}}\vdash C$ and therefore $(\Delta,$ $\Delta_{\mathrm{m}});\Xi;\Gamma\vdash C$ **ok** (see Lemma 5.1). This means that none of the variables in $\overline{a}$ are constrained in any way by C, meaning that the most general solution maps them to pairwise disjoint, fresh variables, like $\overline{b}$. □

The following lemma describes solutions for constraints $\mathfrak{U}(\Theta,\theta)$. The lemma is somewhat specialised for the specific places where we use it, by introducing an extra type context Δ' and an existential quantifier around $\mathfrak{U}(\Theta,\theta)$.

Lemma B.2.
Let $\Delta\vdash\theta:\Theta\Rightarrow\Theta$ *and* $\Delta,\Delta'\vdash\Gamma$ **ok** *and* $\mathsf{ftv}(\Theta)=\Xi,\overline{a}$*. Further, let* θ *be idempotent.*

Then we have

$$(\Delta, \Delta'); \Xi; \Gamma; \delta \vdash \exists \overline{a}.\mathfrak{U}(\Theta, \theta)$$
$$\textit{iff}$$
$$\textit{there exists } \theta' \textit{ such that } (\Delta, \Delta') \vdash \theta' : \Theta \Rightarrow \cdot \textit{ and } \delta = (\theta' \circ \theta)\restriction_{\Xi}.$$

Proof. By definition, the sub-constraint $\mathfrak{U}(\theta)$ of $\mathfrak{U}(\Theta, \theta)$ contains constraints of the form $a \sim \theta(a)$ for all a in Θ. These constraints are satisfied by exactly those substitutions δ refining θ. The idempotency premise ensures that we have $\delta(\theta(a)) = \delta(a)$.[1] In addition, the $\bullet$ sub-constraints in $\mathfrak{U}(\Theta)$ are satisfied iff the refinement respects the restrictions in Θ. □

The next lemma states how *most general* solutions of constraints $\mathfrak{U}(\Theta, \theta)$ look like.

Lemma B.3.
Let the following conditions hold:

- $\Delta \vdash \theta : \Theta \Rightarrow \Theta$
- $\mathsf{ftv}(\Theta) = \Xi, \overline{a}$
- $\Delta \vdash \Gamma\ \mathbf{ok}$
- $\overline{b} = \mathsf{ftv}(\theta\restriction_{\Xi}) - \Delta$
- θ *is idempotent*

Then we have

$$\mathsf{mostgen}((\Delta, \Delta'), \Xi, \Gamma, \exists \overline{a}.\mathfrak{U}(\Theta, \theta), \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$$
$$\textit{iff}$$
$$\textit{there exist pairwise different } \overline{c} \subseteq \Delta_{\mathrm{m}} \textit{ s.t. } \delta_{\mathrm{m}} = [\overline{b} \mapsto \overline{c}] \circ (\theta\restriction_{\Xi})$$

Proof. Follows directly from Lemma B.2 and the observation that the most general solution of $\mathfrak{U}(\Theta, \theta)$ is the one that maps all flexible variables in $\mathsf{ftv}(\theta)$ to fresh, pairwise disjoint rigid variables. Observe that by assumption $\Delta \vdash \theta : \Theta \Rightarrow \Theta$ and the fact that rigid variables are considered monomorphic we have $\Delta \vdash [\overline{b} \mapsto \overline{c}] \circ \theta : \Theta \Rightarrow \cdot$ as well. □

[1]Note that if only $\Delta \vdash \theta : \Theta \Rightarrow \Theta$ holds, but θ is not idempotent, $\mathfrak{U}(\Theta, \theta)$ may even be unsatisfiable!

B.2 Correctness of constraint generation

We now prove the soundness and completeness theorems for the translation from FreezeML terms to constraints defined in Section 5.2.

Theorem 5.1 (Soundness of constraint generation with respect to typing relation; *theorem originally stated on page 108).*
Let $\Delta;\Gamma \vdash M : A$ *hold and* $a \# \Delta$. *Then we have that* $\Delta; a; \Gamma; [a \mapsto A] \vdash [\![M : a]\!]$ *holds.*

Proof. We prove the following, more general property by induction on M, focusing on the cases for let bindings. If $(\Delta, \Xi) \vdash A$ and $\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$ and $\Delta;\Gamma \vdash M : \delta(A)$ hold, then we have that $\Delta;\Xi;\Gamma;\delta \vdash [\![M : A]\!]$ holds.

- Case **let** $x = M'$ **in** N', where $M' \in \mathsf{GVal}$:

  The derivation of $\Delta;\Gamma \vdash M : \delta(A)$ has the following form, for some B, B', Δ_p.

$$\frac{\begin{array}{c}\Delta_\mathrm{p} = \mathsf{ftv}(B') - \Delta \qquad (\Delta, \Delta_\mathrm{p}, M', B') \Updownarrow B \\ (\Delta, \Delta_\mathrm{p});\Gamma \vdash M' : B' \qquad \Delta;\Gamma, x : B \vdash N : \delta(A) \qquad \mathsf{principal}(\Delta, \Gamma, M', \Delta_\mathrm{p}, B')\end{array}}{\Delta;\Gamma \vdash \textbf{let}\ x = M'\ \textbf{in}\ N : \delta(A)}$$

  By $M' \in \mathsf{GVal}$ we have $B = \forall \Delta_\mathrm{p}.B'$ and $[\![M : A]\!] = \textbf{let}_\star\ x = \sqcap b.[\![M' : b]\!]\ \textbf{in}\ [\![N' : A]\!]$. We assume w.l.o.g. that $\Xi \# (\Delta, \Delta_\mathrm{p})$ and $b \# (\Delta, \Delta_\mathrm{p}, \Xi)$.

  By induction we have $(\Delta, \Delta_\mathrm{p}); (\Xi, b); \Gamma; \delta[b \mapsto B'] \vdash [\![M' : b]\!]$ and $\Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]$. By Lemma 5.3, $\mathsf{principal}(\Delta, \Gamma, M', \Delta_\mathrm{p}, B')$ implies $\mathsf{mostgen}(\Delta, b, \Gamma, [\![M' : b]\!], \Delta_\mathrm{p}, [b \mapsto B'])$. Let $\overline{a} := \Delta_\mathrm{p}$ and $\delta_\mathrm{m} := [b \mapsto B'][\overline{a} \mapsto \overline{c}]$ for fresh $\overline{c}$ and $\Delta_\mathrm{m} := (\Delta_\mathrm{p}, \overline{c})$.

  According to Lemma B.1 we can thus weaken the earlier judgement involving $\mathsf{mostgen}$ to $\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m})$.

  Let $\Delta_\mathrm{o} := \overline{c}$ and $\delta' := [\overline{c} \mapsto \mathsf{Unit}]$. Due to $\overline{c} \# \mathsf{ftv}(B')$ we have $\delta'(\delta_\mathrm{m}(b)) = \delta'(B') = B'$.

  We can then derive the following.

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m}) \\ \Delta_\mathrm{o} = \mathsf{ftv}(\delta_\mathrm{m}(\Xi)) - \Delta \qquad \Delta_\mathrm{p} = \mathsf{ftv}(\delta_\mathrm{m}(b)) - \Delta, \Delta_\mathrm{o} \\ \Delta \vdash \delta' : \Delta_\mathrm{o} \Rightarrow_\bullet \cdot \qquad B' = \delta'(\delta_\mathrm{m}(b)) \\ (\Delta, \Delta_\mathrm{p}); (\Xi, b); \Gamma; \delta[b \mapsto B'] \vdash [\![M' : b]\!] \qquad \Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]\end{array}}{\Delta; \Xi; \Gamma; \delta \vdash \textbf{let}_\star\ x = \sqcap b.[\![M' : b]\!]\ \textbf{in}\ [\![N' : A]\!]}$$

- Case **let** $x = M'$ **in** N', where $M' \notin \mathsf{GVal}$:

  We have a derivation of the same shape as in the previous case. However, by $M' \notin \mathsf{GVal}$ we have $B = \delta(B')$ and $[\![M : A]\!] = \mathbf{let}_{\bullet}\ x = \sqcap b.[\![M' : b]\!]\ \mathbf{in}\ [\![N' : A]\!]$, for some δ' such that $\Delta \vdash \delta' : \Delta_{\mathrm{p}} \Rightarrow_{\bullet} \cdot$.

  Let $\Delta_{\mathrm{o}}, \overline{a}, \overline{c}, \Delta_{\mathrm{m}}$ and δ_{m} be defined as in the previous case and we extend δ' to δ'' such that $\delta'' := \delta'[\overline{c} \mapsto \mathsf{Unit}]$. This implies $\Delta \vdash \delta'' : \Delta_{\mathrm{m}} \Rightarrow_{\bullet} \cdot$ and $B = \delta'(B') = \delta''(B') = \delta''(\delta_{\mathrm{m}}(b))$.

  Using the same reasoning as in the previous case we obtain $\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$.

  By Lemma 4.1, we can apply δ' to $(\Delta, \Delta_{\mathrm{p}}); \Gamma \vdash M' : B'$ and obtain $\Delta; \delta'(\Gamma) \vdash M' : \delta'(B')$. By $\Delta_{\mathrm{p}} \# \mathsf{ftv}(\Gamma)$ and the definition of B this is equivalent to $\Delta; \Gamma \vdash M' : B$. By induction, we then have $\Delta, (\Xi, b); \Gamma; \delta[b \mapsto B] \vdash [\![M' : b]\!]$. We also obtain $\Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]$ by induction, as in the previous case. We can then derive the following.

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_{\mathrm{m}}, \delta_{\mathrm{m}}) \\ \Delta \vdash \delta'' : \Delta_{\mathrm{m}} \Rightarrow_{\bullet} \cdot \qquad B = \delta''(\delta_{\mathrm{m}}(b)) \\ \Delta; (\Xi, b); \Gamma; \delta[b \mapsto B] \vdash [\![M' : b]\!] \qquad \Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]\end{array}}{\Delta; \Xi; \Gamma; \delta \vdash \mathbf{let}_{\bullet}\ x = \sqcap b.[\![M' : b]\!]\ \mathbf{in}\ [\![N' : A]\!]}$$

- Case **let** $(x : B) = M'$ **in** N', where $M' \in \mathsf{GVal}$:

  Let $\overline{a}$, H such that $B = \forall \overline{a}.H$.

  The derivation of $\Delta; \Gamma \vdash M : \delta(A)$ has the following form, for some Δ', B, B':

$$\frac{(\Delta', B') = \mathsf{split}(B, M') \qquad (\Delta, \Delta'); \Gamma \vdash M' : B' \qquad \Delta; (\Gamma, x : B) \vdash N' : \delta(A)}{\Delta; \Gamma \vdash \mathbf{let}\ (x : B) = M'\ \mathbf{in}\ N' : \delta(A)}$$

  By $M' \in \mathsf{GVal}$, we have $\Delta' = \overline{a}$, $B' = H$ and $[\![M : A]\!] = (\forall \overline{a}.[\![M' : H]\!]) \wedge \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N' : A]\!]$. By induction, we have $(\Delta, \Delta'); \Xi; \Gamma; \delta \vdash [\![M' : H]\!]$ **(5)** and $\Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]$ **(6)**.

  Recall that an implicit precondition of $\Delta; \Gamma) \vdash M : \delta(A)$ we have $\Delta \vdash M$ **ok**, which implies $\Delta \vdash B$ **ok**.

  We now show that $\Delta; \Xi; \Gamma; \delta$ is a model for both conjuncts of $[\![M : A]\!]$. For the first conjunct, $(\forall \overline{a}.[\![M' : H]\!])$, we can obtain a derivation from a series of applications of rule SEM-FORALL from Figure 5.1 on page 97 once for each variable in Δ', starting from (5). For the second conjunct, we construct the following derivation.

$$\frac{\begin{array}{c}\Delta;\Xi;(\Gamma,x:\delta(B));\delta\vdash [\![N':A]\!]\\ \text{for all } a\in \mathsf{ftv}(B) \mid \Delta;\Xi;\Gamma;\delta\vdash \mathsf{mono}(a)\end{array}}{\Delta;\Xi;\Gamma;\delta\vdash \mathbf{def}\,(x:B)\ \mathbf{in}\ [\![N':A]\!]}$$

The first premise is satisfied due to (6) and $\Delta \vdash B$ **ok**, which implies $\delta(B) = B$. The second premise also follows from $\Delta \vdash B$ **ok**, as it implies $\mathsf{ftv}(B) \subseteq \Delta$, meaning that all such variables are monomorphic.

- Case **let** $(x:A) = M'$ **in** N', where $M' \notin \mathsf{GVal}$:

  Analogous to the previous case, with $\Delta' = \emptyset$, $B' = B$. In particular, we do not need to apply SEM-FORALL.

□

Theorem 5.2 (Completeness of constraint generation with respect to typing relation; *theorem originally stated on page 109).*
Let $\Delta;\Gamma \vdash M$ **ok** *hold. Then* $\Delta;a;\Gamma;\delta \vdash [\![M : a]\!]$ *implies* $\Delta;\Gamma \vdash M : \delta(a)$.

Proof. We prove the following, slightly more general property by induction on M: If $\Delta;\Gamma \vdash M$ **ok** and $(\Delta,\Xi) \vdash A$ **ok** and $\Delta;\Xi;\Gamma;\delta \vdash [\![M : A]\!]$, then $\Delta;\Gamma \vdash M : \delta(A)$.

Note that by the implicit preconditions of $\Delta;\Xi;\Gamma;\delta \vdash [\![M : A]\!]$ we have $\Delta\#\Xi$ and $\Delta \vdash \Gamma$ **ok** and $\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$. The latter implies $\Delta \vdash \delta(A)$ **ok**.

We focus on the let cases.

- Case **let** $x = M'$ **in** N', $M' \in \mathsf{GVal}$:

  We have $\Delta;\Xi;\Gamma;\delta \vdash \mathbf{let}_\star\ x = \sqcap b.[\![M' : b]\!]$ **in** $[\![N' : A]\!]$. The derivation of this has the following form for some $B, \overline{a}, \Delta_\mathrm{m}, \Delta_\mathrm{o}, \delta_\mathrm{m}$ and δ':

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta,(\Xi,b),\Gamma,[\![M':b]\!],\Delta_\mathrm{m},\delta_\mathrm{m})\\ \Delta_\mathrm{o} = \mathsf{ftv}(\delta_\mathrm{m}(\Xi)) - \Delta \qquad \overline{a} = \mathsf{ftv}(\delta_\mathrm{m}(b)) - \Delta,\Delta_\mathrm{o}\\ \Delta\vdash\delta' : \Delta_\mathrm{o} \Rightarrow_\bullet \cdot \qquad B = \delta'(\delta_\mathrm{m}(b))\\ (\Delta,\overline{a});(\Xi,b);\Gamma;\delta[b\mapsto B]\vdash [\![M':b]\!] \qquad \Delta;\Xi;(\Gamma,x:\forall\overline{a}.B);\delta\vdash [\![N':A]\!]\end{array}}{\Delta;\Xi;\Gamma;\delta\vdash \mathbf{let}_\star\,x = \sqcap b.[\![M':b]\!]\ \mathbf{in}\ [\![N':A]\!]}$$

  By induction, this gives us $(\Delta,\overline{a});\Gamma \vdash M' : B$ and $\Delta;(\Gamma,x:\forall\overline{a}.B) \vdash N' : \delta(A)$. By $M' \in \mathsf{GVal}$, we have $\Delta \vdash M'$ and $\Delta \vdash \Gamma$. According to Lemma 5.2 this implies $\Delta;b;\Gamma \vdash [\![M' : b]\!]$ **ok** (i.e., b is the only free flexible variable of $[\![M' : b]\!]$).

This means that $[\![M' : b]\!]$ leaves all variables of Ξ entirely unconstrained. Therefore, we have that δ_m maps all of Ξ to fresh variables $\overline{c} \subseteq \Delta_\mathrm{m}$ and in particular $\overline{c} \# \mathsf{ftv}(\delta_\mathrm{m}(b))$. This implies $\Delta_\mathrm{o} \approx \overline{c}$ and $\overline{a} = \mathsf{ftv}(\delta_\mathrm{m}(b)) - \Delta, \Delta_\mathrm{o} = \mathsf{ftv}(\delta_\mathrm{m}(b)) - \Delta$ (i.e., removing Δ_o from the subtracted context has no effect) and $B = \delta_\mathrm{m}(b)$ (i.e., applying δ' to $\delta_\mathrm{m}(b)$ has no effect).

As a result, we have that we may strengthen $\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m})$ to $\mathsf{mostgen}(\Delta, b, \Gamma, [\![M' : b]\!], \overline{a}, [b \mapsto \delta_\mathrm{m}(b)])$. We may then apply Lemma 5.3 to the latter, yielding $\mathsf{principal}(\Delta, \Gamma, M', \overline{a}, B)$.

We can therefore derive the desired property $\Delta; \Gamma \vdash \textbf{let}\ x = M'\ \textbf{in}\ N' : \delta(A)$ as follows:

$$\frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(B) - \Delta \\ (\Delta, \overline{a}, M', B) \Updownarrow \forall \overline{a}.B \qquad (\Delta, \overline{a}); \Gamma \vdash M' : B \qquad \Delta; (\Gamma, x : \forall \overline{a}.B) \vdash N' : \delta(A) \\ \mathsf{principal}(\Delta, \Gamma, M', \overline{a}, B)\end{array}}{\Delta; \Gamma \vdash \textbf{let}\ x = M'\ \textbf{in}\ N' : \delta(A)}$$

- Case $\textbf{let}\ x = M'\ \textbf{in}\ N'$ if $M' \notin \mathsf{GVal}$:

  This case is largely analogous to the previous one. This time we have a derivation of the form:

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m}) \\ \Delta \vdash \delta' : \Delta_\mathrm{m} \Rightarrow_\bullet \cdot \qquad B = \delta'(\delta_\mathrm{m}(b)) \\ \Delta; (\Xi, b); \Gamma; \delta[b \mapsto B] \vdash [\![M' : b]\!] \qquad \Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]\end{array}}{\Delta; \Theta; \Gamma; \delta \vdash \textbf{let}_\bullet\ x = \sqcap b.[\![M' : b]\!]\ \textbf{in}\ [\![N' : A]\!]}$$

  Let $A' := \delta_\mathrm{m}(b)$ and $\overline{a} := \mathsf{ftv}(A') - \Delta$. By $\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m})$ we have $(\Delta, \Delta_\mathrm{m}); (\Xi, b); \Gamma; \delta_\mathrm{m} \vdash [\![M' : b]\!]$. By induction, this implies $(\Delta, \Delta_\mathrm{m}); \Gamma \vdash M' : A'$, which we can strengthen to $(\Delta, \overline{a}); \Gamma \vdash M' : A'$

  We obtain $\Delta; (\Gamma, x : B) \vdash N' : \delta(A)$ directly by applying the induction hypothesis to $\Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]$. Likewise, we obtain $\mathsf{principal}(\Delta, \Gamma, M', \overline{a}, A')$ using the same reasoning as in the previous case.

  Finally, we observe that $\Delta \vdash \delta' \restriction_{\overline{a}} : \overline{a} \Rightarrow_\bullet \cdot$ holds and $B = \delta' \restriction_{\overline{a}}(A')$. Therefore, we have $(\Delta, \overline{a}, M, A') \Updownarrow B$

Thus, we can derive the following:

$$\frac{\Delta,\overline{a};\Gamma \vdash M : A' \qquad \Delta;\Gamma,x : A \vdash N : \delta(A) \qquad \begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Delta \qquad (\Delta,\overline{a},M,A') \Updownarrow B \\ \mathsf{principal}(\Delta,\Gamma,M,\overline{a},A')\end{array}}{\Delta;\Gamma \vdash \textbf{let } x = M' \textbf{ in } N' : \delta(A)}$$

- Case $\textbf{let } (x : B) = M' \textbf{ in } N'$ if $M' \in \mathsf{GVal}$:

  Let $\overline{b}$ and H such that $B = \forall\overline{b}.H$.

  We then have $\Delta;\Xi;\Gamma;\delta \vdash (\forall\overline{b}.[\![M' : H]\!]) \wedge \textbf{def } (x : B) \textbf{ in } [\![N' : A]\!]$. The derivation tree of this contains derivations for $(\Delta,\overline{b});\Xi;\Gamma;\delta \vdash [\![M' : H]\!]$ and $\Delta;\Xi;(\Gamma,x : \delta(B));\delta \vdash [\![N' : A]\!]$.

  By $\Delta;\Gamma \vdash M$ **ok** we have $\Delta \vdash B$ (i.e., B contains no flexible variables). Therefore, $\delta(B) = B$ and $\delta(H) = H$. By induction, this then gives us $(\Delta,\overline{b});\Gamma \vdash M' : H$ and $\Delta;(\Gamma,x : B) \vdash N' : \delta(A)$.

  Hence, we can derive

$$\frac{(\overline{b},H) = \mathsf{split}(B,M') \qquad (\Delta,\overline{b});\Gamma \vdash M' : H \qquad \Delta;\Gamma,x : B \vdash N' : \delta(A)}{\Delta;\Gamma \vdash \textbf{let } (x : B) = M' \textbf{ in } N' : \delta(A)}$$

- Case $\textbf{let } (x : B) = M' \textbf{ in } N'$ if $M' \notin \mathsf{GVal}$:

  This case is similar to the previous one: We have $\Delta;\Xi;\Gamma;\delta \vdash [\![M' : B]\!]) \wedge \textbf{def } (x : B) \textbf{ in } [\![N' : A]\!]$ this time and $\delta(B) = B$ due to $\Delta;\Gamma \vdash M$ **ok**.

  We get $\Delta;\Gamma \vdash M' : B$ and $\Delta;(\Gamma,x : B) \vdash N' : \delta(A)$ by induction and can derive

$$\frac{(\cdot,B) = \mathsf{split}(B,M') \qquad \Delta;\Gamma \vdash M' : B \qquad \Delta;\Gamma,x : B \vdash N' : \delta(A)}{\Delta;\Gamma \vdash \textbf{let } (x : B) = M' \textbf{ in } N' : \delta(A)}$$

□

B.3 Correctness of constraint solver

We now show proofs of Theorems 5.3 and 5.4. We also show how the main correctness properties for our constraint solver, namely Conjecture 5.4, may be proved, assuming that Conjecture 5.1 holds.

We also prove three additional auxiliary lemmas directly related to the constraint solver.

Lemma B.4 (Stack machine steps preserve well-formedness of states)**.**
If $\vdash s$ **ok** *and* $s \rightarrow s'$ *then* $\vdash s'$ **ok**.

Proof. By case analysis over which stack machine rule was applied. In the case S-EQ, we require Lemmas A.9 and A.10 to show the idempotence and well-formedness of the substitution in the new state.

□

Lemma B.5 (Well-ordering on states)**.**
There exists a strict well-ordering $<$ *on the set* St *of stack machine states such that for all* $s, s' \in \mathsf{St}$ *with* $s \rightarrow s'$ *we have* $s' < s$.

Proof. First, we define the size of a constraint C, denoted $|C|$, s.t.

$$\begin{array}{rcl}
|\mathsf{true}| & = & 0 \\
|\mathsf{mono}(a)| & = & 1 \\
|A \sim B| & = & 1 \\
|\lceil x : A \rceil| & = & 2 \\
|x \preceq A| & = & 2 \\
|\exists a.C| & = & 1 + |C| \\
|\forall a.C| & = & 1 + |C| \\
|\,\mathbf{def}\,(x : A)\ \mathbf{in}\ C| & = & 1 + |C| \\
|C_1 \wedge C_2| & = & 1 + |C_1| + |C_2| \\
|\,\mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2| & = & 3 + |C_1| + |C_2|
\end{array}$$

Next, we define $\mathsf{insts}(C)$ to be the number of instantiation (sub-)constraints in C.

We now define the function $|\cdot|$ that maps states to elements of $\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$:

$$|(f_0 :: \cdots :: f_n, \Theta, \theta, C)| = \big(\mathsf{insts}(C),\ |F[C]|,\ |C|,\ \max\{i \mid 0 \leq i \leq n,\ f_i \text{ is an } \exists \text{ frame}\}\big)$$

We observe that the lexicographic ordering $<_{\text{lex}}$ on tuples from $\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$ constitutes a well-ordering on such tuples and we will show below that for each step $s \rightarrow s'$ we have that $|s'| <_{\text{lex}} |s|$ holds. However, the function $|\cdot|$ on states is surjective, which implies that defining $s' < s$ iff $|s'| <_{\text{lex}} |s|$ would *not* yield a total order on states. Hence, let $<_{\text{ord}}$ be some arbitrary strict well-order on states. We then define

$$s' < s \qquad \text{iff} \qquad |s'| <_{\text{lex}} |s| \text{ or } \big(|s'| = |s| \text{ and } s' <_{\text{ord}} s\big)$$

which is indeed a well-ordering.

It remains to show that each step of the stack machine produces a smaller state w.r.t. $|\cdot|$. Hence, assume $s \rightarrow s'$, where $s = (F, \Theta, \theta, C)$ and $s' = (F', \Theta', \theta', C')$.

- If the step is the result of applying the rule S-EQ, S-FREEZE, or S-MONO, then we have $F = F'$ and $|C'| < |C|$, yielding $|s'| <_{\text{lex}} |s|$ via the second component of the tuples.

- If the step is the result of applying S-INST, we have $\mathsf{insts}(C') = \mathsf{insts}(C) - 1$ and we have $|s'| <_{\text{lex}} |s|$ via the first component of the tuples.

- If the step is the result of applying the rule S-CONJPOP, S-FORALLPOP, or S-DEFPOP, we have $\mathsf{insts}(C) = \mathsf{insts}(C')$ and $|F'[C']| < |F[C]|$, yielding $|s'| <_{\text{lex}} |s|$ via the second component of the tuple.

- If the step is the result of applying the rule S-CONJPUSH, S-EXISTSPUSH, S-FORALLPUSH, S-DEFPUSH, or S-LETPUSH, we have $\mathsf{insts}(C) = \mathsf{insts}(C')$ and $F[C] = F'[C']$, but $|C'| < |C|$, yielding $|s'| <_{\text{lex}} |s|$ via the third component of the tuples.

- If S-EXISTSLOWER got applied, let $\overline{c}$ and $\overline{a}$ be defined as in the rule, $F = f_0 :: \ldots f_n$, and $l = |\overline{a}|$. Note that the rule imposes $l > 0$ and we have $C = C' = \mathsf{true}$.

  We observe that $\overline{c}$ is a subset of $\overline{a}$. If $|\overline{a}| > |\overline{c}|$, we have $|F'[C']| < |F[C]|$ because the set of frames of F' is a strict subset of the frames in F. We then obtain $|s'| <_{\text{lex}} |s|$ immediately via the second component of the tuples (the first component remains unchanged). Otherwise, we have that the two sets are equal. In that case we have $|F[C]| = |F'[C']|$ (as there is merely a reordering of stack frames happening) and $|C| = |C'| = 0$. The resulting stack F' is of the form $f'_0 :: \ldots f'_n$, where $f'_n = f_{n-l}$ (not an $\exists$ frame), and f'_{n-1} is an $\exists$ frame.

  Together, we have $|s| = (\mathsf{insts}(C), |F[C]|, 0, n)$, and $|s'| = (\mathsf{insts}(C'), |F[C]|, 0, n-1)$, and $\mathsf{insts}(C) = \mathsf{insts}(C')$, yielding $|s'| <_{\text{lex}} |s|$.

- If rule S-LETPOP⋆ was applied, we use the following reasoning: F is of the form $F_0 :: \mathsf{let}_\star x = \sqcap c.\square \,\mathsf{in}\, \hat{C} :: \exists\overline{a}$ and C is true. This yields

  $$\begin{aligned} |F[C]| &= |F_0[\mathbf{let}_\star\ x = \sqcap c.\exists\overline{a}.\mathsf{true}\ \mathbf{in}\ \hat{C}]| \\ &= |\mathbf{let}_\star\ x = \sqcap c.\mathsf{true}\ \mathbf{in}\ \mathsf{true}| + |\exists\overline{a}.\mathsf{true}| + |\hat{C}| + |F_0[\mathsf{true}]| \\ &= 3 + |\overline{a}| + |\hat{C}| + |F_0[\mathsf{true}]|, \end{aligned}$$

  Let $\overline{a''}$ be defined as in the rule. We have

  $$|F'[C']| = |F_0[\exists\overline{a''}.\,\mathbf{def}\ (x : A)\ \mathbf{in}\ \hat{C}] = 1 + |\overline{a''}| + |\hat{C}| + |F_0[\mathsf{true}]|$$

Proving $F'[C'] < F[C]$ is therefore equivalent to proving

$$\begin{aligned} & 1+|\overline{a''}|+|\hat{C}|+|F_0[\mathsf{true}]| < 3+|\overline{a}|+|\hat{C}|+|F_0[\mathsf{true}]| \\ \text{equiv.}\quad & \overline{a''} < 2+\overline{a}. \end{aligned}$$

We observe that $\overline{a''}$ is a strict subset of $(\overline{a},c)$ and hence $|\overline{a''}| < |\overline{a}|+2$, meaning that the inequality above holds.

We have $\mathsf{insts}(C) = \mathsf{insts}(C')$, and therefore $|s'| <_{\text{lex}} |s|$ via the second component of the tuples.

- The reasoning for rule S-LETPOP• is analogous to the previous case. The only change is that we need to observe that $(\overline{c},\overline{a''})$ is a subset of $(\overline{a},c)$.

□

Theorem 5.3 (Termination; *theorem originally stated on page 144).*
The constraint solver terminates when invoked on any well formed input state.

Proof. Follows immediately from Lemma B.5, which guarantees the absence of infinite sequences of steps. □

Theorem 5.4 (Progress; *theorem originally stated on page 145).*
Let $\vdash (F,\Theta,\theta,C)$ **ok** *and* $F[C] \neq \forall\Delta.\exists\Xi.\mathsf{true}$ *for all* Δ,Ξ. *Further, let* $\cdot;\cdot;\cdot;\emptyset \vdash F[C \wedge \mathcal{U}(\Theta,\theta)]$ *hold. Then there exists a state* s_1 *such that* $(F,\Theta,\Gamma,\theta,C) \to s_1$.

Proof. Let $s := (F,\Theta,\theta,C)$. By assumption, we have $\cdot;\cdot;\cdot;\emptyset \vdash F[C \wedge \mathcal{U}(\Theta,\theta)]$ **(1)**.
We first assume $C \neq \mathsf{true}$ and show that one of the rules in Figure 5.8 is applicable.

- Case $C_1 \wedge C_2$: Rule S-CONJPUSH is applicable.
- Case $A \sim B$: The left-hand-side of S-EQ matches. The derivation of (1) must contain a subderivation of the form

$$\frac{\delta(A) = \delta(B)}{\mathsf{rc}(F);\mathsf{fc}(F);\mathsf{tc}(F);\delta \vdash A \sim B}$$

for some δ, where $\mathsf{fc}(F) \approx \mathsf{ftv}(\Theta)$.

We also have $\mathsf{rc}(F);\mathsf{fc}(F);\mathsf{tc}(F);\delta \vdash\!\vdash \mathcal{U}(\Theta,\theta)$, which by Lemma B.2 means that δ is a refinement of θ (i.e., $\delta = \theta' \circ \theta$ for some θ' with $\mathsf{rc}(F) \vdash \theta' : \Theta \Rightarrow \cdot$).

Theorem 4.3 then guarantees that unification succeeds.

- Case $\lceil x : A \rceil$: The left-hand-side of S-INST matches. By (1) we have $x \in \mathsf{tc}(F)$, meaning that the rule succeeds.

- Case $x \preceq A$: Rule S-FREEZE is applicable (using the same argument to show $x \in \mathsf{tc}(F)$ as in the previous case).

- Case $\exists a.C'$: Rule S-EXISTSPUSH is applicable.

- Case $\forall a.C'$: Rule S-FORALLPUSH is applicable.

- Case **def** $(x : A)$ **in** C': The left-hand-side of S-DEFPUSH matches.

  The derivation of (1) must contain a sub-derivation of the form

$$\frac{\begin{array}{c}(\text{for all } a \in \mathsf{ftv}(A) - (\mathsf{rc}(F), \Delta') \mid (\mathsf{rc}(F), \Delta'); \mathsf{fc}(F); \delta(\mathsf{tc}(F)); \delta \vdash \mathsf{mono}(a) \\ (\mathsf{rc}(F), \Delta'); \mathsf{fc}(F); (\delta(\mathsf{tc}(F)), x : \delta A); \delta \vdash C'\end{array}}{(\mathsf{rc}(F), \Delta'); \mathsf{fc}(F); \delta(\mathsf{tc}(F)); \delta \vdash \textbf{def}\ (x : A)\ \textbf{in}\ C'}$$

  for some δ, where $C' = C \wedge \mathfrak{U}(\Theta, \theta)$ and $\mathsf{ftv}(\Theta) = \Xi$. Note that by $\vdash s\ \textbf{ok}$ we have $\mathsf{ftv}(A) \subseteq (\mathsf{rc}(F), \mathsf{fc}(F))$.

  According to Lemma B.2, we have $\delta = \theta' \circ \theta$ for some θ' with $(\mathsf{rc}(F), \Delta') \vdash \theta' : \Theta \Rightarrow \cdot$. Therefore, the monomorphism conditions imposed by S-DEFPUSH are satisfied.

- Case $\mathsf{mono}(a)$: Analogous to def case; the sub-derivation for $\mathsf{mono}(a)$ implies that the monomorphism conditions imposed by S-MONO are satisfied, making the rule applicable.

- Case $\textbf{let}_R\ x = \sqcap a.C_1\ \textbf{in}\ C_2$: Rule S-LETPUSH is applicable.

We now consider the case that C is true. Due the assumption about the shape of $F[C]$, we know that F is neither empty nor of the shape $\forall \Delta :: \exists \overline{a}$ for any $\overline{a}$.

We perform a case analysis on the topmost stack frame of F:

- Case $\square \wedge C_2$: Rule S-CONJPOP is applicable.

- Case **def** $(x : A)$: Rule S-DEFPOP is applicable.

- Case $\textbf{let}_R\ x = \sqcap a.C_1\ \textbf{in}\ C_2$: Rule S-LETPOP$\star$ or S-LETPOP$\bullet$ is applicable, where the sequence $\overline{a}$ mentioned in the rule's definition is empty. None of the respective rule's side conditions can fail.

- Case $\forall a$: Let F' be defined such that $F = F' :: \forall a$. By assumption s **ok** we have that the variables in Θ are exactly the variables bound by $\exists$ or **let** frames in F'. Suppose there exists $b \in \mathsf{ftv}(\Theta)$ such that $a \in \mathsf{ftv}(\theta(b))$, which would cause the rule to fail. Then there exists a frame f in F' that binds b. Let F_p be the (possibly empty) prefix of F' up to, but not including, frame f and let F_s be the (possibly empty) suffix from there (i.e., $F' = F_p :: f :: F_s$).

  We distinguish two sub-cases further:

  1. If $f = \exists b$ then the derivation of (1) must contain a sub-derivation of the following form:

  $$\frac{(\mathsf{rc}(F_p),\Delta');(\mathsf{fc}(F_p),b);\delta(\mathsf{tc}(F_p));\delta[b \mapsto A] \vdash C'}{(\mathsf{rc}(F_p),\Delta');\mathsf{fc}(F_p);\delta(\mathsf{tc}(F_p));\delta \vdash \exists b.C'}$$

     for some A, δ, and Δ', where $C' = F_s[C \wedge \mathfrak{U}(\Theta,\theta)]$.

     Note that $(\mathsf{rc}(F_p),\Delta');(\mathsf{fc}(F_p),b);\delta(\mathsf{tc}(F_p));\delta[b \mapsto A] \vdash C'$ implies $(\mathsf{rc}(F_p),$ $\Delta') \vdash A$ **ok (2)**. Further, due to the fact that F_s contains the frame $\forall a$, we have $a \notin \mathsf{rc}(F_p),\Delta'$ **(3)**.

     Likewise, there exists a subderivation showing $(\mathsf{rc}(F),\Delta'');\mathsf{fc}(F);\delta''(\mathsf{tc}(F));$ $\delta'' \vdash \mathfrak{U}(\Theta,\theta)$ for some Δ'' and δ'', where $\Delta'' \supseteq \Delta'$ and δ'' is an extension of $\delta[b \mapsto A]$. By Lemma B.2 we have that there exists θ' such that $\mathsf{rc}(F),\Delta'' \vdash \theta' : \Theta \Rightarrow \cdot$ and $\delta'' = (\theta' \circ \theta)$. Because δ'' is an extension of $\delta[b \mapsto A]$, this implies $A = \theta'(\theta(b))$.

     By $a \in \mathsf{rc}(F)$ we have $\theta'(a) = a$. Due to assumption $a \in \mathsf{ftv}(\theta(b))$ we then have $a \in \mathsf{ftv}(A)$ However, we have $a \notin \mathsf{btv}(F_p),\Delta'$ (3) and $\mathsf{rc}(F_p),\Delta' \vdash$ A **ok** (2), yielding the contradiction $a \notin \mathsf{ftv}(A)$.

  2. If f is of the the form $\mathbf{let}_\star\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$ then the derivation of (1) must contain a sub-derivation of the following form:

  $$\frac{\begin{array}{c}\dots\\(\mathsf{rc}(F_p),\Delta',\overline{a});(\mathsf{fc}(F_p),b);\delta(\mathsf{tc}(F_p));\delta[b \mapsto A] \vdash C_1\end{array}}{(\mathsf{rc}(F_p),\Delta');\mathsf{fc}(F_p);\delta(\mathsf{tc}(F_p));\delta \vdash \mathbf{let}_\star\ x = \sqcap b.C_1\ \mathbf{in}\ C_2}$$

     for some $A,\Delta',\overline{a}$ and δ, where $C_1 = F_s[C \wedge \mathfrak{U}(\Theta,\theta)]$. We have $(\mathsf{rc}(F_p),\Delta',\overline{a}) \vdash$ A **ok** and $a \notin \mathsf{rc}(F_p),\Delta',\overline{a}$ and may therefore obtain the same contradiction as in the previous case $f = \exists b$.

  3. The case $\mathbf{let}_\bullet\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$ is analogous.

- Case $\exists a$: If the topmost stack frames of F have the shape $\mathbf{let}_R\, x = \sqcap a.\square$ **in** $C' :: \exists \overline{b}$, then S-LETPOP$\star$ or S-LETPOP$\bullet$ is applicable, as discussed before. Otherwise, due to our assumption about the shape of $F[C]$, there exists a frame f in F that isn't an $\exists$ frame and we can apply S-EXISTSLOWER, which always succeeds.

$\square$

We now show how Conjecture 5.4, the main correctness property of the solver, may be proved, contingent on Conjecture 5.1. We state a slight variation of the former in Conjecture B.1 on the following page, more suitable for inductive reasoning, and subsequently show that it implies Conjecture 5.4.

We first formalise an additional well-formedness condition for whole states, which is required to state Conjecture B.1. Conjecture 5.4 only reasons about states of the simple form $(\cdot, \cdot, \emptyset, \forall \Delta.\exists \Xi.\, \mathbf{def}\ \Gamma\ \mathbf{in}\ C)$ (in particular: states with an empty stack), but Conjecture B.1 allows for more general forms of states.

While it is sufficient to require $\Delta \vdash^{\mathsf{ftv}} C\ \mathbf{ok}$ in Conjecture 5.4, we need a more involved property for Conjecture B.1 to represent that the $\vdash^{\mathsf{ftv}}$ condition imposed on the constraint components of states is preserved for all intermediate states encountered during solving. This effectively amounts to showing that whenever $\cdot \vdash^{\mathsf{ftv}} C_0\ \mathbf{ok}$ and $(\cdot, \cdot, \emptyset, C_0) \rightarrow^* (F_1, \Theta_1, \theta_1, C_1)$, then we have $\mathsf{rc}(F_1) \vdash^{\mathsf{ftv}} C_1\ \mathbf{ok}$ as well. Intuitively this holds due to the fact that whenever C_1 is a let constraint, it must have appeared as a sub-constraint of the original constraint C_0.

We need to state a condition such that for the initial state s_0 in Conjecture B.1 we have $\vdash^{\mathsf{ftv}} C'\ \mathbf{ok}$ for its constraint component C'. However, we additionally need to ensure that the latter condition is preserved once the solver takes a step from s_0 to a successor state. As this step may involve moving parts from the state's stack into its constraint component, this condition must involve reasoning about both the constraint *and* stack component of states.

To this end, we define an additional well-formedness relation on stacks, denoted $\vdash^{\mathsf{ftv}} F\ \mathbf{ok}$. Intuitively, it described that for each constraint C contained within a frame f of the stack, we have $\Delta \vdash^{\mathsf{ftv}} C\ \mathbf{ok}$, where Δ are the rigid variable bound by the prefix of F up to f. Here, constraints contained within a frame f refers to the constraints within a conjunction or let frame. The new relation is then formalised in Figure B.1 on the next page.

We then define the relation $\vdash^{\mathsf{ftv}} s\ \mathbf{ok}$ on states $s = (F, \Theta, \theta, C)$ such that $\vdash^{\mathsf{ftv}} s\ \mathbf{ok}$ holds

$\boxed{\vdash^{\text{ftv}} F\ \mathbf{ok}}$

$$\frac{}{\vdash^{\text{ftv}} \cdot\ \mathbf{ok}} \qquad \frac{\mathsf{rc}(F) \vdash^{\text{ftv}} C\ \mathbf{ok}}{\vdash^{\text{ftv}} F :: \Box \wedge C\ \mathbf{ok}} \qquad \frac{\vdash^{\text{ftv}} F\ \mathbf{ok}}{\vdash^{\text{ftv}} F :: \forall\, a\ \mathbf{ok}} \qquad \frac{\vdash^{\text{ftv}} F\ \mathbf{ok}}{\vdash^{\text{ftv}} F :: \exists\, a\ \mathbf{ok}}$$

$$\frac{\vdash^{\text{ftv}} F\ \mathbf{ok}}{\vdash^{\text{ftv}} F :: \mathbf{def}\ (x : A)\ \mathbf{ok}} \qquad \frac{\mathsf{rc}(F) \vdash^{\text{ftv}} C\ \mathbf{ok} \qquad \vdash^{\text{ftv}} F\ \mathbf{ok}}{\vdash^{\text{ftv}} F :: \mathbf{let}_R\, x = \sqcap a.\Box\ \mathsf{in}\ C\ \mathbf{ok}}$$

Figure B.1: Definition of $\vdash^{\text{ftv}} F\ \mathbf{ok}$

iff $\vdash^{\text{ftv}} F\ \mathbf{ok}$ and $\mathsf{rc}(F) \vdash^{\text{ftv}} C\ \mathbf{ok}$ hold.

Note that we could have combined these additional well-formedness relations $\vdash^{\text{ftv}} F\ \mathbf{ok}$ and $\vdash^{\text{ftv}} s\ \mathbf{ok}$ with the existing well-formedness relations on stacks and states, defined in Section 5.3.1.3, instead. We are only interested in stacks and states that satisfy these conditions. However, recall that for constraints the distinction between $\Delta;\Xi;\Gamma \vdash C\ \mathbf{ok}$ and $\Delta \vdash^{\text{ftv}} C\ \mathbf{ok}$ is strictly necessary. We thus choose to mirror the same distinction for stacks and states for the sake of consistency.

We may then show that this condition on states is indeed preserved during execution of the solver:

Lemma B.6.
If $\vdash^{\mathit{ftv}} s_0\ \mathbf{ok}$ *and* $s_0 \longrightarrow s_1$ *then* $\vdash^{\mathit{ftv}} s_1\ \mathbf{ok}$.

Proof. By case analysis on the applied rule from Figure 5.8. □

Recall that the following property is a slight variation of Conjecture 5.4; we use it in the contingent proof of the latter.

Conjecture B.1.
Let $s = (\forall\Delta :: \exists\overline{a} :: F, \Theta, \theta, C)$ *and* $\vdash s\ \mathbf{ok}$ *and* $\vdash^{\mathit{ftv}} s\ \mathbf{ok}$. *Then we have the following:*

$$\begin{array}{l} \Delta;\overline{a};\cdot;\delta \vdash F[C \wedge \mathfrak{U}(\Theta,\theta)] \\ \textit{iff} \\ \textit{there exist } \Theta', \theta'', \theta', \overline{b} \textit{ s.t.} \\ \quad s \longrightarrow^* (\forall\Delta :: \exists\,(\overline{a},\overline{b}), \Theta', \theta', \mathsf{true}) \textit{ and} \\ \quad \Delta \vdash \theta'' : \Theta' \Rightarrow \cdot \textit{ and} \\ \quad (\theta'' \circ \theta')\restriction_{\overline{a}} = \delta \end{array}$$

Proof (dependent on Conjecture 5.1). We show each direction individually:

$\Longrightarrow$ By transfinite induction on the well-ordering $<$ on stack machine states s whose existence is shown in Lemma B.5. Hence, we assume that the left-to-right direction of the lemma holds for all s' on the left of the $\to^*$ s.t. $s' < s$ and show that the left-to-right direction holds for s on the left of $\to^*$, too.

To this end, let $s = (\forall\Delta :: \exists\overline{a} :: F, \Theta, \theta, F, C)$ and we assume $\vdash s$ **ok (1)** as well as $\vdash^{\mathsf{ftv}} s$ **ok (2)** and $\Delta;\overline{a};\cdot;\delta \vdash F[C \wedge \mathfrak{U}(\Theta,\theta)]$ **(3)**.

We first consider the case that s is already a final state in the senses of this lemma, meaning that F is of the shape $\exists\overline{b}$ for some $\overline{b}$ and C is true. Further, we have $\theta = \theta'$ and $\Theta = \Theta'$, where $\mathsf{ftv}(\Theta) \approx \overline{a},\overline{b}$.

This makes (3) equivalent to $\Delta;\overline{a};\cdot;\delta \vdash \exists\overline{b}.\mathfrak{U}(\Theta,\theta)$. Applying Lemma B.2 then gives us the existence of an appropriate θ''.

We now consider the case where s is not a final state, i.e., we don't have $F[C] = \exists\overline{b}.\mathsf{true}$ for any $\overline{b}$. We observe that (3) implies $;\cdot;\cdot;\emptyset \vdash \forall\Delta :: \exists\overline{a} :: F[C \wedge \mathfrak{U}(\Theta,\theta)]$. This allows us to apply Theorem 5.4, showing that the machine can take a step from s to a new state s_1. We now show that s_1 is of the form $(\forall\Delta :: \exists\overline{a} :: F_1, \Theta_1, \theta_1, C_1)$ for some F_1, Θ_1, θ_1, and C_1:

If F is empty, then C must not be true. All stack machine rules applicable in this case preserve all existing stack frames. Otherwise, if F is not empty, we observe that the only rules of the stack machine that may replace more than the topmost stack frame are S-EXISTSLOWER, S-LETPOP$\bullet$, and S-LETPOP$\star$.

If S-EXISTSLOWER was applied, we observe that the only way for the variables $\overline{a}$ in the definition of the rule S-EXISTSLOWER not to be disjoint from the variables $\overline{a}$ in the statement of this lemma is if the stack of s is of the form $\forall\Delta :: \exists\overline{a} :: \exists\overline{b}$ for some $\overline{b}$, which violates the assumption about the shape of $F[C]$ above. Therefore, if S-EXISTSLOWER was applied, the bottom-most frames $\forall\Delta :: \exists\overline{a}$ of s remained unchanged. If S-LETPOP$\star$ or S-LETPOP$\bullet$ was applied, then F must contain a **let** frame and any stack frames below that in s (in particular, the frames $\forall\Delta :: \exists\overline{a}$) remain unchanged.

Therefore, the $\forall\Delta :: \exists\overline{a}$ frames at the bottom of s's stack are preserved by any rule possibly turning s into s_1. Using (1) and (2) and the fact that the lower stack frames of s_1 are $\forall\Delta :: \exists\overline{a}$, we may apply Conjecture 5.1 to the step $s \to s_1$, which gives us

$$\Delta;\overline{a};\cdot;\delta \vdash F[C \wedge \mathfrak{U}(\Theta,\theta)] \text{ iff } \Delta;\overline{a};\cdot;\delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)] \qquad \textbf{(4)}$$

By Lemma B.5, we further have $s_1 < s$ **(5)**. From Lemmas B.4 and B.6 we obtain $\vdash s_1$ **ok (6)** and $\vdash^{\text{ftv}} s_1$ **ok (7)**, respectively.

Combining (3) with (4) gives us $\Delta;\overline{a};\cdot;\delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)]$. This, together with (5), (6) and (7) allows us to apply the induction hypothesis to s_1. This gives us the existence of $\Theta',\theta'',\theta',\overline{a}$ such that the following holds.

$$(\forall\Delta :: \exists\overline{a} :: F_1,\Theta_1,\theta_1,C_1) \rightarrow^* (\forall\Delta :: \exists\,(\overline{a},\overline{b}),\Theta',\theta',\mathsf{true}) \qquad \textbf{(8)}$$

$$\Delta \vdash \theta'' : \Theta' \Rightarrow \cdot \qquad \textbf{(9)}$$

$$(\theta'' \circ \theta')\restriction_{\overline{a}} = \delta \qquad \textbf{(10)}$$

The step $s \rightarrow s_1$ extends (8) to $(\forall\Delta :: \exists\overline{a} :: F,\Theta,\theta,C) \rightarrow^* (\forall\Delta :: \exists\,(\overline{a},\overline{b}),\Theta',\theta',\mathsf{true})$ and (9) as well as (10) show us that θ'' has the desired properties.

$\Longleftarrow$ Let s be the state $(\forall\Delta :: \exists\overline{a} :: F,\Theta,\theta,C)$. We prove this direction by induction on the length n of the sequence $s \rightarrow^n (\forall\Delta :: \exists\,(\overline{a},\overline{b}),\Theta',\theta',\mathsf{true})$. By assumption, we also have $\Delta \vdash \theta'' : \Theta' \Rightarrow \cdot$ **(11)** and $(\theta'' \circ \theta')\restriction_{\overline{a}} = \delta$ **(12)**.

If $n = 0$ we have $\Theta = \Theta'$, $\theta = \theta'$, $C = \mathsf{true}$, and $F = \exists\overline{b}$. The property to prove simplifies to $\Delta;\overline{a};\cdot;\delta \vdash \exists\overline{b}.\mathfrak{U}(\Theta,\theta)$. This follows directly from applying Lemma B.2 to (11) and (12).

In the inductive step there exists some s_1 s.t.

$$s \rightarrow s_1 \rightarrow^* (\forall\Delta :: \exists(\overline{a},\overline{b}),\Theta',\theta',\mathsf{true})$$

We now assume that $F[C]$ is not of the form $\exists\overline{c}.\mathsf{true}$ for any $\overline{c}$ (otherwise, s would already be a final state in the sense of this lemma and we finish the proof directly using the $n = 0$ case above).

Therefore, using the same reasoning as in the $\Longrightarrow$ direction, we know that s_1 is of the form $(\forall\Delta :: \exists\overline{a} :: F_1,\Theta_1,\theta_1,C_1)$ for some F_1,Θ_1,θ_1, and C_1. According to Lemmas B.4 and B.6, we have $\vdash s_1$ **ok** and $\vdash^{\text{ftv}} s_1$ **ok**, respectively. We can therefore apply Conjecture 5.1 to this single step, yielding

$$\Delta;\overline{a};\cdot;\delta \vdash F[C \wedge \mathfrak{U}(\Theta,\theta)] \ \text{ iff } \ \Delta;\overline{a};\cdot;\delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)] \qquad \textbf{(13)}$$

We apply the induction hypothesis to the sequence $s_1 \rightarrow^* (\forall\Delta :: \exists(\overline{a},\overline{b}),\Theta',\theta',\mathsf{true})$, yielding $\Delta;\overline{a};\cdot;\delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1,\theta_1)]$ By (13), this gives us the desired property $\Delta;\overline{a};\cdot;\delta \vdash F[C \wedge \mathfrak{U}(\Theta,\theta)]$.

$\triangle$

Conjecture 5.4 (Correctness of the constraint solver; *conjecture originally stated on page 145).*
Let $\Delta \vdash \Gamma\ \mathbf{ok}$ *and* $\Delta;\Xi;\Gamma \vdash C\ \mathbf{ok}$ *and* $\Delta \vdash^{\mathit{ftv}} C\ \mathbf{ok}$. *Then we have*

$$\begin{array}{l}
\Delta;\Xi;\Gamma;\delta \vdash C \\
\textit{iff} \\
\textit{there exist } \Theta, \theta', \theta, \Xi' \textit{ s.t.} \\
\quad (\cdot, \cdot, \emptyset, \forall\Delta.\, \exists\Xi.\ \mathbf{def}\ \Gamma\ \mathbf{in}\ C) \to^* (\forall\Delta :: \exists\ (\Xi, \Xi'), \Theta, \theta, \mathsf{true}) \textit{ and} \\
\quad \Delta \vdash \theta' : \Theta \Rightarrow \cdot \textit{ and} \\
\quad (\theta' \circ \theta)\restriction_{\Xi} = \delta.
\end{array}$$

Proof (dependent on Conjecture B.1). Let $\overline{a}$ denote the variables in Ξ. Further, let $\theta_{\mathrm{a}} := [\overline{a \mapsto a}]$ and $\Theta_{\mathrm{a}} := (\overline{a : \star})$. We have

$$(\cdot, \cdot, \emptyset, \forall\Delta.\exists\overline{a}.\mathbf{def}\ \Gamma\ \mathbf{in}\ C) \ \to^* \ (\forall\Delta :: \exists\overline{a}, \Theta_{\mathrm{a}}, \theta_{\mathrm{a}}, \mathbf{def}\ \Gamma\ \mathbf{in}\ C)$$

after $|\Delta|$ applications of the rule S-FORALLPUSH and $|\overline{a}|$ applications of the rule S-EXISTSPUSH. Let the former state be defined as s, the latter one as s'. Here, due to $\Delta;\Theta;\Gamma \vdash C\ \mathbf{ok}$, we have $\vdash s'\ \mathbf{ok}$.

Therefore, for all $\hat{\Theta}, \hat{\theta}, \overline{b}$ we have

$$\begin{array}{rcl}
s' & \to^* & (\forall\Delta :: \exists\ (\overline{a}, \overline{b}), \hat{\Theta}, \hat{\theta}, \mathsf{true}) \\
 & \text{iff} & \\
s & \to^* & (\forall\Delta :: \exists\ (\overline{a}, \overline{b}), \hat{\Theta}, \hat{\theta}, \mathsf{true})
\end{array}$$

We observe that from assumption $\Delta \vdash^{\mathrm{ftv}} C\ \mathbf{ok}$ we immediately obtain $\vdash^{\mathrm{ftv}} s'$.

Now, let F be the empty stack. We then have

$$\begin{array}{lll}
 & \Delta;\Xi;\Gamma;\delta \vdash C & \\
\text{iff} & \Delta;\Xi;\cdot;\delta \vdash (\mathbf{def}\ \Gamma\ \mathbf{in}\ C) & (\text{by } \Delta \vdash \Gamma\ \mathbf{ok} : \\
 & & \ \text{all mono. conditions satisfied}) \\
\text{iff} & \Delta;\Xi;\cdot;\delta \vdash (\mathbf{def}\ \Gamma\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_{\mathrm{a}}, \theta_{\mathrm{a}}) & (\mathfrak{U}(\Theta_{\mathrm{a}}, \theta_{\mathrm{a}}) \text{ is equivalent to } \mathsf{true}) \\
\text{iff} & \Delta;\Xi;\cdot;\delta \vdash F[(\mathbf{def}\ \Gamma\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_{\mathrm{a}}, \theta_{\mathrm{a}})] & (F \text{ is empty})
\end{array}$$

Using this, the equivalence to prove then follows directly from Conjecture B.1. $\triangle$

B.4 Correctness of constraint-based type inference

We now show how we may prove Conjectures 5.5 and 5.6, which state the overall correctness of the constraint-based inference approach, assuming that Conjecture 5.4 holds.

Conjecture 5.5 (Constraint-based type-checking is sound; *conjecture originally stated on page 146).*
Let $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash M$ **ok** *and* $a\#\Delta$*. If* $(\cdot,\cdot,\emptyset,\forall\Delta.\,\exists a.\,\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \longrightarrow^* (\forall\Delta :: \exists\,(a,\overline{b}),\Theta,\theta,\mathsf{true})$ *and* $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$ *then* $\Delta;\Gamma \vdash M : (\theta' \circ \theta)(a)$.

Proof (dependent on Conjecture 5.4). Suppose $(\cdot,\cdot,\emptyset,\forall\Delta.\exists a.\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M:a]\!]) \longrightarrow^* (\forall\Delta :: \exists\,(a,\overline{b}),\Theta,\theta,\mathsf{true})$ and $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$. Let s refer to the first state of the sequence above and s' to its last state.

We apply Lemma 5.2, which gives us $\Delta;a;\Gamma \vdash [\![M : a]\!]$ **ok**. We therefore have $\vdash s$ **ok**. By Lemma B.4 we then have $\vdash s'$ **ok**, too, which implies $\Delta \vdash \Theta \Rightarrow \cdot$ and $\mathsf{ftv}(\Theta) = \overline{b},a$. We can therefore define δ as $\theta' \circ \theta\restriction_{\{a\}}$. By Lemma 5.7, we have $\Delta \vdash^{\mathsf{ftv}} [\![M : a]\!]$

Together, this allows us to apply Conjecture 5.4. We use the right-to-left direction of the theorem such that we need to show that the following properties hold:

$$(\cdot,\cdot,\emptyset,\forall\Delta.\,\exists a.\,\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \longrightarrow^* (\forall\Delta :: \exists\,(a,\overline{b}),\Theta,\theta,\mathsf{true})$$
$$\Delta \vdash \theta' : \Theta \Rightarrow \cdot$$
$$(\theta' \circ \theta)\restriction_a = \delta$$

The first two properties follow immediately by assumption, the third one holds by definition of δ. Therefore, the right-to-left direction of Conjecture 5.4 gives us $\Delta;a;\Gamma;\delta \vdash [\![M : a]\!]$. Theorem 5.2 then immediately yields $\Delta;\Gamma \vdash M : \delta(a)$. By definition of δ this is equivalent to the property to show. △

Conjecture 5.6 (Constraint-based type-checking is complete and most general; *conjecture originally stated on page 147).*
Let $a\#\Delta$*. If* $\Delta;\Gamma \vdash M : A$ *then there exist* $\Xi,\Theta,\theta,\ \delta$ *such that* $(\cdot,\cdot,\emptyset,\forall\Delta\,.\exists a\,.\,\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \longrightarrow^* (\forall\Delta :: \exists\,\Xi,\Theta,\theta,\mathsf{true})$ *and* $A = \delta(\theta(a))$.

Proof (dependent on Conjecture 5.4). We assume $\Delta;\Gamma \vdash M : A$, which implies $\Delta \vdash \Gamma$ and $\Delta;\Gamma \vdash M$ **ok**. This means that Theorem 5.1 gives us $\Delta;a;\Gamma;[a \mapsto A] \vdash [\![M : a]\!]$.

We may further apply Lemma 5.7, yielding $\Delta \vdash^{\mathsf{ftv}} [\![M : a]\!]$.

We may thus apply the left-to-right direction of Conjecture 5.4 (using $[\![M : a]\!]$ for C and $[a \mapsto A]$ for δ in the theorem's statement) which gives us the existence of $\Theta, \theta, \theta', \overline{c}$ such that the following conditions hold.

$$(\cdot, \cdot, \emptyset, \forall\Delta.\exists a.\,\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \rightarrow^{*} (\forall\Delta :: \exists(a, \overline{c}), \Theta, \theta, \mathsf{true}) \qquad \textbf{(1)}$$

$$\Delta \vdash \theta' : \Theta \Rightarrow \cdot \qquad \textbf{(2)}$$

$$(\theta' \circ \theta)\restriction_{\{a\}} = [a \mapsto A] \qquad \textbf{(3)}$$

Clearly, we have $(\theta' \circ \theta\restriction_{\{a\}})(a) = (\theta' \circ \theta)(a) = [a \mapsto A](a) = A$, which is the second property we need to show (choosing θ' for δ). By choosing $\Xi = (a, \overline{c})$, property (1) becomes the first property that we needed to show. △

Appendix C

Proofs for Chapter 6

This appendix contains those proof omitted from Chapter 6, specifically Section 6.4.1.

C.1 Proofs for Section 6.4.1

Lemma C.1.
Let $(\Delta^\bullet, \Theta) \vdash S\ \mathbf{ok}$ *and* $(\Delta^\bullet, \Theta) \vdash S'\ \mathbf{ok}$. *Then* $\mathcal{U}(\Delta, \Theta, S, S') = (\Theta', \theta)$ *implies that all types in the codomain of* θ *are syntactic monotypes.*

Proof. As with earlier proofs related to the unification algorithm, by induction on the number of recursive calls to $\mathcal{U}$. The only interesting case where a non-identity mapping is added to the resulting substitution is when a flexible variable a is unified with a type A. By assumption, A is either S or S', meaning that we add a mapping from a to a syntactic monotype. □

Lemma C.2.
Let M *be an ML term and let* $\Delta \vdash \Gamma\ \mathbf{ok}$ *hold, where* Γ *only contains ML type schemes. Then* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ *implies that* A *is a syntactic monotype and all types in the codomain of* θ *are syntactic monotypes.*

Proof. By structural induction on M, reasoning about the behaviour of infer in each case. The cases for plain variables and lambda abstractions are straightforward. In the case for term abstractions we rely on Lemma C.1. In the case for plain let bindings $\mathbf{let}\ x = M\ \mathbf{in}\ N$ we observe that the type we add for x to the environment is an ML type scheme, allowing us apply the induction hypothesis to N. □

Lemma 6.1 *(as originally stated on page 174).*
*Let M be an ML term and let $\Delta \vdash \Gamma$ **ok** hold, where Γ only contains ML type schemes. Then $\mathsf{principal}(\Delta,\Gamma,M,\Delta',A)$ implies that A is a monotype.*

Proof. Since M is an ML term, it cannot contain free type variables. The principality premise implies that M only contains free term variables from Γ. Altogether, this yields $\Delta;\Gamma \vdash M$ **ok**.

By completeness of inference (Theorem 4.6) $\mathsf{infer}(\Delta,\cdot,\Gamma,M)$ succeeds, returning some (Θ,θ,A'). By principality of inferred types (Theorem 4.7), $\mathsf{principal}(\Delta,\Gamma,M,$ $\mathsf{ftv}(\Theta),A')$ holds, and by uniqueness of principal types (Lemma 3.1) we have that A and A' are equal modulo a renaming of free type variables. Lemma C.2 then states that A' is a syntactic monotype, and hence A is, too. □

Lemma 6.2 *(as originally stated on page 175).*
*Let M be an ML term and let $\Delta \vdash \Gamma$ **ok** hold, where Γ only contains ML type schemes. Then $\Delta;\Gamma \vdash M : S$ implies that there exists a derivation of the latter judgement where every subterm of M is assigned a monomorphic type.*

Proof. By structural induction on the term under consideration. In the case for **let** $x = M$ **in** N, we rely on Lemma 6.1 to establish that the type used for M, namely the principal one, is monomorphic, and hence x receives an ML type scheme when typing N.

All other cases are straightforward, except when for term applications $M\,N$. The problem is that while we know by assumption that the return type of M is the monomorphic type S, the parameter of the function M (and hence the type of N) may use a polymorphic type. We show that we can, however, always construct a derivation for $\Delta;\Gamma \vdash M\,N : S$ that assigns a monomorphic parameter type to M.

To this end, we consider the individual steps taken by $\mathsf{infer}(\Delta,\cdot,\Gamma,M\,N)$, as defined in Figure 4.5 on page 86. Let $\theta_3' := \theta_3[a \mapsto B]$ and $\Delta' := \mathsf{ftv}(B) - \Delta$. We then have $\Delta' \subseteq \mathsf{ftv}(\Theta_3)$. By Lemma 4.4 and Theorems 4.2 and 4.5 we have $(\Delta,\Theta_3);\Gamma \vdash^{\mathrm{E}} M : \theta_3'(\theta_2(A'))$ and $(\Delta,\Theta_3);\Gamma \vdash^{\mathrm{E}} N : \theta_3'(A)$. Using Lemma 4.2, this gives us the following.

$$(\Delta,\mathsf{ftv}(\Theta_3));\Gamma \vdash M : \theta_3'(\theta_2(A')) \qquad \textbf{(1)}$$

$$(\Delta,\mathsf{ftv}(\Theta_3));\Gamma \vdash N : \theta_3'(A) \qquad \textbf{(2)}$$

By soundness of unification (Theorem 4.2) we have $\theta_3'(A \to b) = \theta_3'(A) \to B = \theta_3'(\theta_2(A'))$ **(3)**.

By principality of inferred types (Theorem 4.7) we have $\mathsf{principal}(\Delta, \Gamma, M\,N, \mathsf{ftv}(\Theta_3), B)$, which implies that there exists δ_p such that $\Delta \vdash \delta_\mathrm{p} : \mathsf{ftv}(\Theta_3) \Rightarrow_\star \cdot$ and $\delta_\mathrm{p}(B) = S$. Note that this implies that δ_p is a monomorphic instantiation when its domain is restricted to Δ'. We may now define δ with domain $\mathsf{ftv}(\Theta_3)$ such that $\delta(a) := \delta_\mathrm{p}(a)$ for all $a \in \Delta'$. For all other $a \in \Theta_3$, let $\delta(a)$ be an arbitrary type that is monomorphic in Δ, such as Unit. We observe that $\delta(B) = \delta_\mathrm{p}(B) = S$ and $\Delta \vdash \delta : \mathsf{ftv}(\Theta_3) \Rightarrow_\bullet \cdot$.

We can apply δ to (1) and (2), which together with (3) gives us $\Delta;\Gamma \vdash M : \delta(\theta'_3(A)) \to S$ and $\Delta;\Gamma \vdash N : \delta(\theta'_3(A))$, according to Lemma 4.1. Thus, we can ultimately construct the following derivation.

$$\frac{\Delta;\Gamma \vdash M : \delta(\theta'_3(A)) \to S \qquad \Delta;\Gamma \vdash N : \delta(\theta'_3(A))}{\Delta;\Gamma \vdash M\,N : S}$$

By Lemmas C.1 and C.2 we have that A', A and B are syntactic monotypes as well as that θ_1, θ_2 and θ'_3 only have syntactic monotypes in their codomains. Thus, we have that $\delta(\theta'_3(A))$ is monomorphic, too. □